aws

开发人员指南

AWS Lambda

AWS Lambda: 开发人员指南

Table of Contents

什么是 AWS Lambda？

您可以使用 AWS Lambda 运行代码而无需预置或管理服务器。

Lambda 在可用性高的计算基础设施上运行您的代码，执行计算资源的所有管理工作，其中包括服务器和操作系统维护、容量预置和弹性伸缩和记录。使用 Lambda，您只需在 Lambda 支持的一种语言运行时系统中提供代码。

您可以将代码组织到 Lambda 函数。Lambda 服务仅在需要时运行函数并自动扩展。您只需按使用的计算时间付费，代码未运行时不产生费用。有关更多信息，请参阅AWS Lambda 定价。

> **Tip**
>
> 要了解如何构建无服务器解决方案，请查看无服务器开发人员指南。

何时使用 Lambda

Lambda 是一种理想的计算服务，适用于需要快速纵向扩展并在不需要时缩减至零的应用程序场景。例如，您可以在以下情况中使用 Lambda：

- 文件处理：使用 Amazon Simple Storage Service（Amazon S3）在上传完毕后实时触发 Lambda 数据处理。
- 流处理：使用 Lambda 和 Amazon Kinesis 处理实时流数据，用于应用程序活动跟踪、交易订单处理、点击流分析、数据清理、日志筛选、索引编制、社交媒体分析、物联网（IoT）设备数据遥测和计量。
- Web 应用程序：将 Lambda 与其他 AWS 服务相结合，构建功能强大的 Web 应用程序，这些应用程序可自动纵向扩展和缩减，并跨多个数据中心以高可用配置运行。
- IoT 后端：使用 Lambda 构建无服务器后端以处理 Web、移动设备、IoT 和第三方 API 请求。
- 移动后端：使用 Lambda 和 Amazon API Gateway 构建后端以对 API 请求进行身份验证和处理 API 请求。使用 AWS Amplify 可轻松与 iOS、Android、Web 和 React Native 前端集成。

使用 Lambda 时，您只负责您的代码。Lambda 管理提供内存、CPU、网络和其他资源均衡的计算机群，以运行代码。由于 Lambda 管理这些资源，因此您无法在提供的运行时上登录计算实例或自定义操作系统。Lambda 代表您执行操作和管理活动，包括管理容量、监控并记录 Lambda 函数。

主要特征

以下关键功能可帮助您开发可扩展、安全且易于扩展的 Lambda 应用程序：

环境变量

使用环境变量来调整函数的行为，而无需更新代码。

版本

使用版本管理函数部署，例如，可将新函数用于 beta 测试，而这不会影响稳定生产版本的用户。

容器映像

使用 AWS 提供的基本映像或备用基本映像为 Lambda 函数创建容器映像，以便重复使用现有的容器工具或部署依赖大量依赖项的较大工作负载，例如机器学习。

图层

为库和其他依赖项打包，以减少部署存档的大小，并加快代码部署速度。

Lambda 扩展

通过监控、可观测性、安全性和监管工具增强 Lambda 函数。

函数 URL

向 Lambda 函数添加专用 HTTP（S）端点。

响应流式处理

配置 Lambda 函数 URL 以将响应负载从 Node.js 函数流式传输回客户端，以提高首字节时间（TTFB）性能或返回较大负载。

并发和扩展控制

对生产应用程序的扩展和响应能力进行精细控制。

代码签名

确认只有已获批准的开发人员才能发布 Lambda 函数中未经更改的可信代码

私有联网

为资源（如数据库、缓存实例或内部服务）创建专用网络。

文件系统访问

配置函数以将 Amazon Elastic File System（Amazon EFS）挂载到本地目录，以便函数代码能够在高并发下安全地访问和修改共享资源。

适用于 Java 的 Lambda SnapStart

将 Java 运行时系统的启动性能提高多达 10 倍，而无需支付额外费用，通常也无需更改函数代码。

创建第一个 Lambda 函数

要开始使用 Lambda，请使用 Lambda 控制台创建函数。您可以在几分钟的时间内创建和部署函数，并在控制台中对其进行测试。

在学习本教程时，您将学习一些基本的 Lambda 概念，例如如何使用 Lambda 事件对象将参数传递给函数。您还将学习如何从函数返回日志输出，以及如何在 CloudWatch Logs 中查看函数的调用日志。

为了简单起见，您可以使用 Python 或 Node.js 运行时系统创建函数。您可以使用这些解释性语言，在控制台的内置代码编辑器中直接编辑函数代码。如果使用 Java 和 C# 等编译语言，您需要在本地生成计算机上创建部署包并将其上传到 Lambda。要了解如何使用其他运行时系统将函数部署到 Lambda，请参阅 the section called “其他资源和后续步骤” 一节中的链接。

> **Tip**
>
> 要了解如何构建无服务器解决方案，请查看无服务器开发人员指南。

先决条件

注册 AWS 账户

如果您还没有 AWS 账户，请完成以下步骤来创建一个。

注册 AWS 账户

1. 打开 https://portal.aws.amazon.com/billing/signup。
2. 按照屏幕上的说明进行操作。

   在注册时，将接到一通电话，要求使用电话键盘输入一个验证码。

   当您注册 AWS 账户时，系统将会创建一个 AWS 账户根用户。根用户有权访问该账户中的所有 AWS 服务 和资源。作为安全最佳实践，请为用户分配管理访问权限，并且只使用根用户来执行需要根用户访问权限的任务。

注册过程完成后，AWS 会向您发送一封确认电子邮件。在任何时候，您都可以通过转至 https://aws.amazon.com/ 并选择我的账户来查看当前的账户活动并管理您的账户。

创建具有管理访问权限的用户

注册 AWS 账户 后，请保护好您的 AWS 账户根用户，启用 AWS IAM Identity Center，并创建一个管理用户，以避免使用根用户执行日常任务。

保护您的 AWS 账户根用户

1. 选择根用户并输入您的 AWS 账户电子邮件地址，以账户拥有者身份登录 AWS Management Console。在下一页上，输入您的密码。

   要获取使用根用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的以根用户身份登录。
2. 为您的根用户启用多重身份验证 (MFA)。

   有关说明，请参阅《IAM 用户指南》中的为 AWS 账户 根用户启用虚拟 MFA 设备（控制台）。

创建具有管理访问权限的用户

1. 启用 IAM Identity Center。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的启用 AWS IAM Identity Center。
2. 在 IAM Identity Center 中，为用户授予管理访问权限。

   有关如何使用 IAM Identity Center 目录 作为身份源的教程，请参阅《AWS IAM Identity Center 用户指南》中的使用默认的 IAM Identity Center 目录 配置用户访问权限。

以具有管理访问权限的用户身份登录

- 要使用您的 IAM Identity Center 用户身份登录，请使用您在创建 IAM Identity Center 用户时发送到您的电子邮件地址的登录网址。

  要获取使用 IAM Identity Center 用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的登录 AWS 访问门户。

将访问权限分配给其他用户

1. 在 IAM Identity Center 中，创建一个权限集，该权限集遵循应用最低权限的最佳做法。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的创建权限集。
2. 将用户分配到一个组，然后为该组分配单点登录访问权限。

有关说明，请参阅《AWS IAM Identity Center 用户指南》中的添加组。

使用控制台创建 Lambda 函数

在此示例中，函数采用一个 JSON 对象，其中包含两个标记有 `"length"` 和 `"width"` 的整数值。该函数将这些值相乘来计算区域，并将其作为 JSON 字符串返回。

函数还会输出计算区域及其 CloudWatch 日志组名称。在本教程的后面部分，您将学习使用 CloudWatch Logs 查看函数调用记录。

要创建函数，首先要使用控制台创建基本的 Hello world 函数。然后在下一步中，添加自己的函数代码。

使用控制台创建 Hello world Lambda 函数

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择 Create function（创建函数）。
3. 选择从头开始编写。
4. 在基本信息窗格中，为函数名称输入 **`myLambdaFunction`**。
5. 对于运行时系统，选择 Node.js 20.x 或 Python 3.12
6. 保留架构设置为 x86_64，然后选择创建函数。

Lambda 不仅创建了一个返回消息 `Hello from Lambda!` 的函数，还为函数创建了一个执行角色。执行角色是一个 AWS Identity and Access Management（IAM）角色，用于向 Lambda 函数授予访问 AWS 服务 和资源的权限。而 Lambda 为函数创建的角色用于授予写入 CloudWatch Logs 的基本权限。

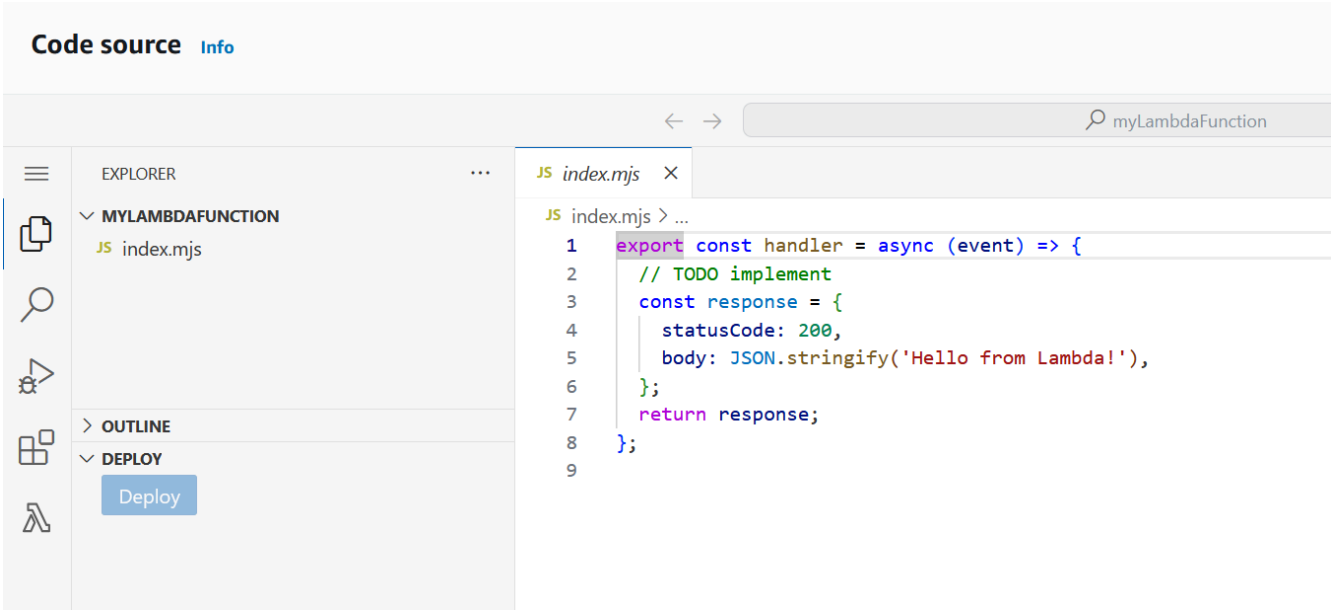
现在，您可以使用控制台的内置代码编辑器，将 Lambda 创建的 Hello world 代码替换为自己的函数代码。

Node.js

在控制台中修改代码

1. 选择节点选项卡。

在控制台的内置代码编辑器中，会显示 Lambda 创建的函数代码。如果代码编辑器中没有显示 index.mjs 选项卡，请在文件资源管理器中选择 index.mjs，如下图所示。



2. 将以下代码粘贴到 index.mjs 选项卡中，替换 Lambda 创建的代码。

```
export const handler = async (event, context) => {

  const length = event.length;
  const width = event.width;
  let area = calculateArea(length, width);
  console.log(`The area is ${area}`);

  console.log('CloudWatch log group: ', context.logGroupName);

  let data = {
    "area": area,
  };
    return JSON.stringify(data);

  function calculateArea(length, width) {
    return length * width;
  }
};
```

3. 在主侧栏中，展开部署部分，然后选择部署以更新函数的代码。当 Lambda 部署更改后，控制台会显示一个横幅，告知您函数已成功更新。

了解函数代码

在继续执行下一步之前，我们需要花些时间查看函数代码并了解一些关键的 Lambda 概念。

- Lambda 处理程序：

  Lambda 函数包含一个名为 `handler` 的 Node.js 函数。在 Node.js 中，Lambda 函数可以包含多个 Node.js 函数，但处理程序函数始终是代码的入口点。当调用函数时，Lambda 会运行此方法。

  当您使用控制台创建 Hello world 函数时，Lambda 会自动将函数的处理程序方法的名称设置为 `handler`。确保不要编辑此 Node.js 函数的名称。否则当您调用函数时，Lambda 无法运行代码。

  要了解有关 Node.js 中 Lambda 处理程序的更多信息，请参阅 the section called “处理程序”。

- Lambda 事件对象：

  函数 `handler` 采用 `event` 和 `context` 这两个参数。Lambda 中的事件是 JSON 格式的文档，其中包含要处理的函数数据。

  如果函数被其他 AWS 服务 调用，则事件对象会包含有关导致调用的事件的信息。例如，如果 Amazon Simple Storage Service（Amazon S3）存储桶在上传对象时调用函数，则事件会包含 Amazon S3 存储桶的名称和对象键。

  在此示例中，您将通过输入包含两个键值对的 JSON 格式文档，在控制台中创建事件。

- Lambda 上下文对象：

  函数采用的第二个参数是 `context`。Lambda 会自动将上下文对象传递给函数。上下文对象包含有关函数调用和执行环境的信息。

  您可以使用上下文对象输出有关函数调用的信息，以便进行监控。在此示例中，函数使用 `logGroupName` 参数来输出其 CloudWatch 日志组名称。

  要了解有关 Node.js 中 Lambda 上下文对象的更多信息，请参阅 the section called “上下文”。

- 登录 Lambda：

  借助 Node.js，您可以通过 `console.log` 和 `console.error` 等控制台方法将信息发送到函数日志。示例代码使用 `console.log` 语句输出计算区域以及函数的 CloudWatch 日志组名称。您还可以使用任何写入 `stdout` 或 `stderr` 的日志记录库。

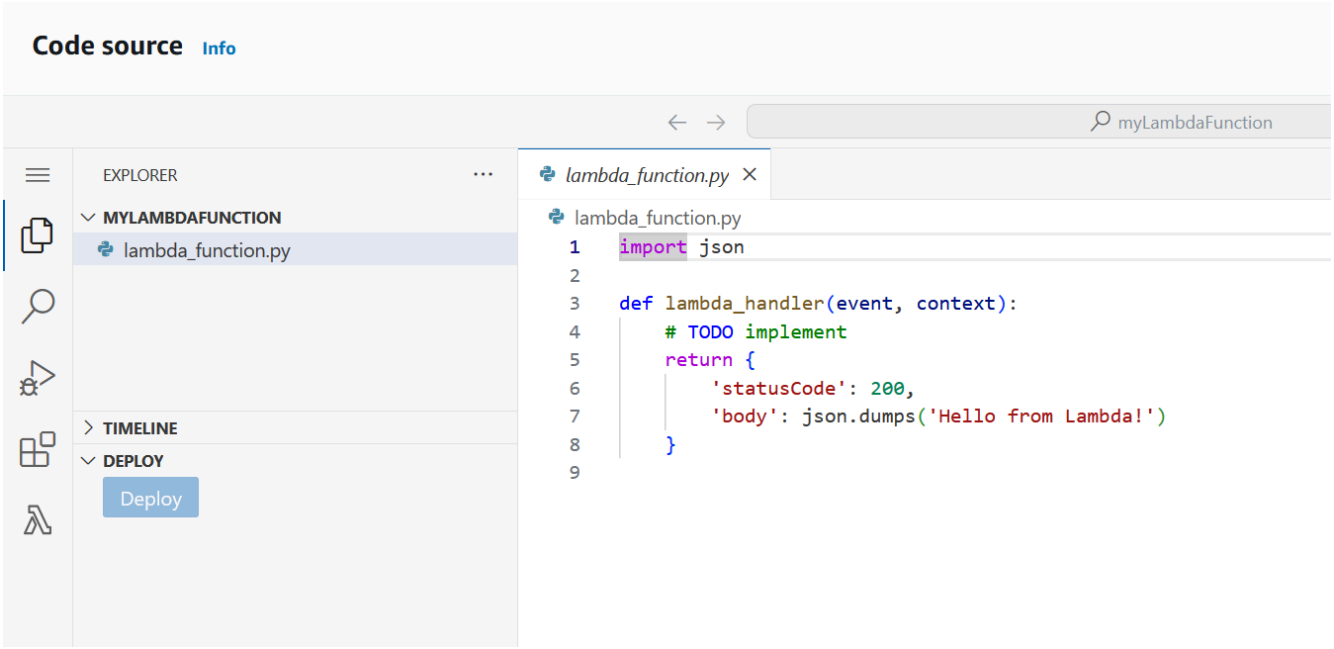
要了解更多信息，请参阅 the section called “日志记录”。要了解如何登录其他运行时系统，请参阅您感兴趣的运行时系统的“构建方式”页面。

Python

在控制台中修改代码

1. 选择节点选项卡。

   在控制台的内置代码编辑器中，会显示 Lambda 创建的函数代码。如果代码编辑器中没有显示 lambda_function.py 选项卡，请在文件资源管理器中选择 lambda_function.py，如下图所示。



2. 将以下代码粘贴到 lambda_function.py 选项卡中，替换 Lambda 创建的代码。

```
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

    # Get the length and width parameters from the event object. The
    # runtime converts the event object to a Python dictionary
    length = event['length']
```

```
    width = event['width']

    area = calculate_area(length, width)
    print(f"The area is {area}")

    logger.info(f"CloudWatch logs group: {context.log_group_name}")

    # return the calculated area as a JSON string
    data = {"area": area}
    return json.dumps(data)

def calculate_area(length, width):
    return length*width
```

3. 在主侧栏中，展开部署部分，然后选择部署以更新函数的代码。当 Lambda 部署更改后，控制台会显示一个横幅，告知您函数已成功更新。

了解函数代码

在继续执行下一步之前，我们需要花些时间查看函数代码并了解一些关键的 Lambda 概念。

- Lambda 处理程序：

  Lambda 函数包含一个名为 `lambda_handler` 的 Python 函数。在 Python 中，Lambda 函数可以包含多个 Python 函数，但处理程序函数始终是代码的入口点。当调用函数时，Lambda 会运行此方法。

  当您使用控制台创建 Hello world 函数时，Lambda 会自动将函数的处理程序方法的名称设置为 `lambda_handler`。确保不要编辑此 Python 函数的名称。否则当您调用函数时，Lambda 无法运行代码。

  要了解有关 Python 中 Lambda 处理程序的更多信息，请参阅 the section called “处理程序”。

- Lambda 事件对象：

  函数 `lambda_handler` 采用 `event` 和 `context` 这两个参数。Lambda 中的事件是 JSON 格式的文档，其中包含要处理的函数数据。

  如果函数被其他 AWS 服务 调用，则事件对象会包含有关导致调用的事件的信息。例如，如果 Amazon Simple Storage Service（Amazon S3）存储桶在上传对象时调用函数，则事件会包含 Amazon S3 存储桶的名称和对象键。

在此示例中，您将通过输入包含两个键值对的 JSON 格式文档，在控制台中创建事件。

- Lambda 上下文对象：

  函数采用的第二个参数是 `context`。Lambda 会自动将上下文对象传递给函数。上下文对象包含有关函数调用和执行环境的信息。

  您可以使用上下文对象输出有关函数调用的信息，以便进行监控。在此示例中，函数使用 `log_group_name` 参数来输出其 CloudWatch 日志组名称。

  要了解有关 Python 中 Lambda 上下文对象的更多信息，请参阅 the section called “上下文”。

- 登录 Lambda：

  借助 Python，您可以使用 `print` 语句或 Python 日志记录库，将信息发送到函数日志。为了说明捕获内容的差异，示例代码使用了这两种方法。在生产应用程序中，我们建议您使用日志记录库。

  要了解更多信息，请参阅 the section called “Python Lambda 函数日志记录和监控”。要了解如何登录其他运行时系统，请参阅您感兴趣的运行时系统的“构建方式”页面。

使用控制台调用 Lambda 函数

要使用 Lambda 控制台调用函数，首先要创建一个发送到函数的测试事件。该事件是包含两个键值对的 JSON 格式文档，其中键为 `"length"` 和 `"width"`。

创建测试事件

1. 在主侧栏中，展开测试事件部分，然后选择创建测试事件。
2. 在创建新测试事件选项卡中，为事件名称 输入 **`myTestEvent`**。
3. 在事件 JSON 面板中，通过粘贴以下内容来替换默认值：

   ```
   {
     "length": 6,
     "width": 7
   }
   ```

4. 选择保存。

现在，您可以测试函数，并使用 Lambda 控制台和 CloudWatch Logs 来查看函数的调用记录。

测试函数并在控制台中查看调用记录

- 在主侧栏的测试事件部分，选择测试事件旁边的运行图标。函数完成运行后，执行结果选项卡中将显示响应和函数日志。您将看到类似以下内容的结果：

  Node.js

```
Status: Succeeded
Test Event Name: myTestEvent

Response
"{\"area\":42}"

Function Logs
START RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Version: $LATEST
2023-08-31T23:39:45.313Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO The area is
 42
2023-08-31T23:39:45.331Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO CloudWatch
 log group:  /aws/lambda/myLambdaFunction
END RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a
REPORT RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Duration: 20.67 ms Billed
 Duration: 21 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration:
 163.87 ms

Request ID
5c012b0a-18f7-4805-b2f6-40912935034a
```

  Python

```
Status: Succeeded
Test Event Name: myTestEvent

Response
"{\"area\": 42}"

Function Logs
START RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Version: $LATEST
The area is 42
[INFO] 2023-08-31T23:43:26.428Z 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b CloudWatch
 logs group: /aws/lambda/myLambdaFunction
END RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
```

```
REPORT RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Duration: 1.42 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 39 MB Init Duration: 123.74
 ms

Request ID
2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
```

在此示例中，您通过控制台的测试功能调用代码。这意味着您可以直接在控制台中查看函数的执行结果。在控制台外部调用函数时，需要使用 CloudWatch Logs。

在 CloudWatch Logs 中查看函数的调用记录

1. 打开 CloudWatch 控制台的 Log groups page（日志组页面）。
2. 选择函数 (`/aws/lambda/myLambdaFunction`) 的日志组。这是函数输出到控制台的日志组名称。
3. 在日志流选项卡上，选择函数调用的日志流。

   您应该可以看到类似于如下所示的输出内容：

   Node.js

   ```
   INIT_START Runtime Version: nodejs:20.v13    Runtime Version ARN:
    arn:aws:lambda:us-
   west-2::runtime:e3aaabf6b92ef8755eaae2f4bfdcb7eb8c4536a5e044900570a42bdba7b869d9
   START RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20 Version: $LATEST
   2023-08-23T22:04:15.809Z    5c012b0a-18f7-4805-b2f6-40912935034a  INFO The area
    is 42
   2023-08-23T22:04:15.810Z    aba6c0fc-cf99-49d7-a77d-26d805dacd20  INFO
    CloudWatch log group:  /aws/lambda/myLambdaFunction
   END RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20
   REPORT RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20    Duration: 17.77 ms
    Billed Duration: 18 ms    Memory Size: 128 MB    Max Memory Used: 67 MB    Init
    Duration: 178.85 ms
   ```

   Python

   ```
   INIT_START Runtime Version: python:3.12.v16    Runtime Version ARN:
    arn:aws:lambda:us-
   west-2::runtime:ca202755c87b9ec2b58856efb7374b4f7b655a0ea3deb1d5acc9aee9e297b072
   START RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e Version: $LATEST
   ```

```
The area is 42
[INFO] 2023-09-01T00:05:22.464Z 9315ab6b-354a-486e-884a-2fb2972b7d84 CloudWatch
 logs group: /aws/lambda/myLambdaFunction
END RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e
REPORT RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e    Duration: 1.15 ms
 Billed Duration: 2 ms    Memory Size: 128 MB    Max Memory Used: 40 MB
```

清理

使用完示例函数后，请将其删除。您还可以删除存储函数日志的日志组以及控制台创建的执行角色。

要删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择函数。
3. 依次选择 Actions（操作）和 Delete（删除）。
4. 在 Delete function（删除函数）对话框中，输入 delete，然后选择 Delete（删除）。

删除日志组

1. 打开 CloudWatch 控制台的 Log groups page（日志组页面）。
2. 选择函数的日志组 (`/aws/lambda/my-function`)。
3. 依次选择 Actions（操作）和 Delete log group(s)（删除日志组）。
4. 在 Delete log group(s)（删除日志组）对话框中，选择 Delete（删除）。

删除执行角色

1. 打开 AWS Identity and Access Management (IAM) 控制台的 Roles page（角色页面）。
2. 选择函数的执行角色（例如 `myLambdaFunction-role-31exxmpl`）。
3. 选择删除。
4. 在 Delete role（删除角色）对话框中，输入角色名称，然后选择 Delete（删除）。

您可以使用 AWS CloudFormation 和 AWS Command Line Interface (AWS CLI) 自动创建和清理函数、日志组和角色。

其他资源和后续步骤

现在，您已经使用控制台创建并测试了一个简单的 Lambda 函数，请继续执行以下步骤：

- 了解如何向代码添加依赖项并使用 .zip 部署包进行部署。从以下链接中选择您感兴趣的语言。

  Node.js

  请参阅 the section called “部署 .zip 文件存档”。

  Typescript

  请参阅 the section called “部署 .zip 文件归档”

  Python

  请参阅 the section called “部署 .zip 文件归档”

  Ruby

  请参阅 the section called “部署 .zip 文件归档”

  Java

  请参阅 the section called “部署 .zip 文件存档”

  Go

  请参阅 the section called “部署 .zip 文件归档”

  C#

  请参阅 the section called “部署程序包”
- 执行使用 Amazon S3 触发器调用 Lambda 函数教程，了解如何将 Lambda 函数配置为由其他 AWS 服务 调用。
- 选择以下教程之一，查看将 Lambda 与其他 AWS 服务 结合使用的更复杂示例。
  - 利用 API Gateway 使用 Lambda：创建调用 Lambda 函数的 Amazon API Gateway REST API。
  - 使用 Lambda 函数访问 Amazon RDS 数据库：使用 Lambda 函数通过 RDS 代理将数据写入 Amazon Relational Database Service（Amazon RDS）。
  - 使用 Amazon S3 触发器创建缩略图：每次将图像文件上传到 Amazon S3 存储桶时，使用 Lambda 函数创建缩略图。

示例无服务器应用程序

以下示例提供了函数代码和基础设施即代码（IaC）模板，用于快速创建和部署可实现常见 Lambda 用例的无服务器应用程序。示例还包括代码示例，以及在部署应用程序后对其进行测试的说明。

对于每个示例应用程序，都提供了有关使用 AWS Management Console 手动创建和配置资源或借助 IaC 使用 AWS Serverless Application Model 部署资源的说明。按照控制台说明，了解有关为每个应用程序配置单个 AWS 资源的更多信息，或使用 AWS SAM 说明快速部署资源（类似于生产环境中的操作）。

您可以根据自己的用例修改所提供的函数代码和模板，从而将所提供的示例作为自己的无服务器应用程序的基础进行使用。

我们将继续创建新示例，因此请返回查看更多适用于常见 Lambda 用例的无服务器应用程序。

示例应用程序

- 示例无服务器文件处理应用程序

  创建无服务器应用程序，以便将对象上传到 Amazon S3 存储桶时自动执行文件处理任务。在此示例中，当上传 PDF 文件时，应用程序会加密该文件并将其保存到另一个 S3 存储桶。
- 计划的 cron 任务应用程序示例

  使用 cron 计划创建用于执行计划任务的应用程序。在此示例中，应用程序通过删除超过 12 个月的条目，来对 Amazon DynamoDB 表进行维护。

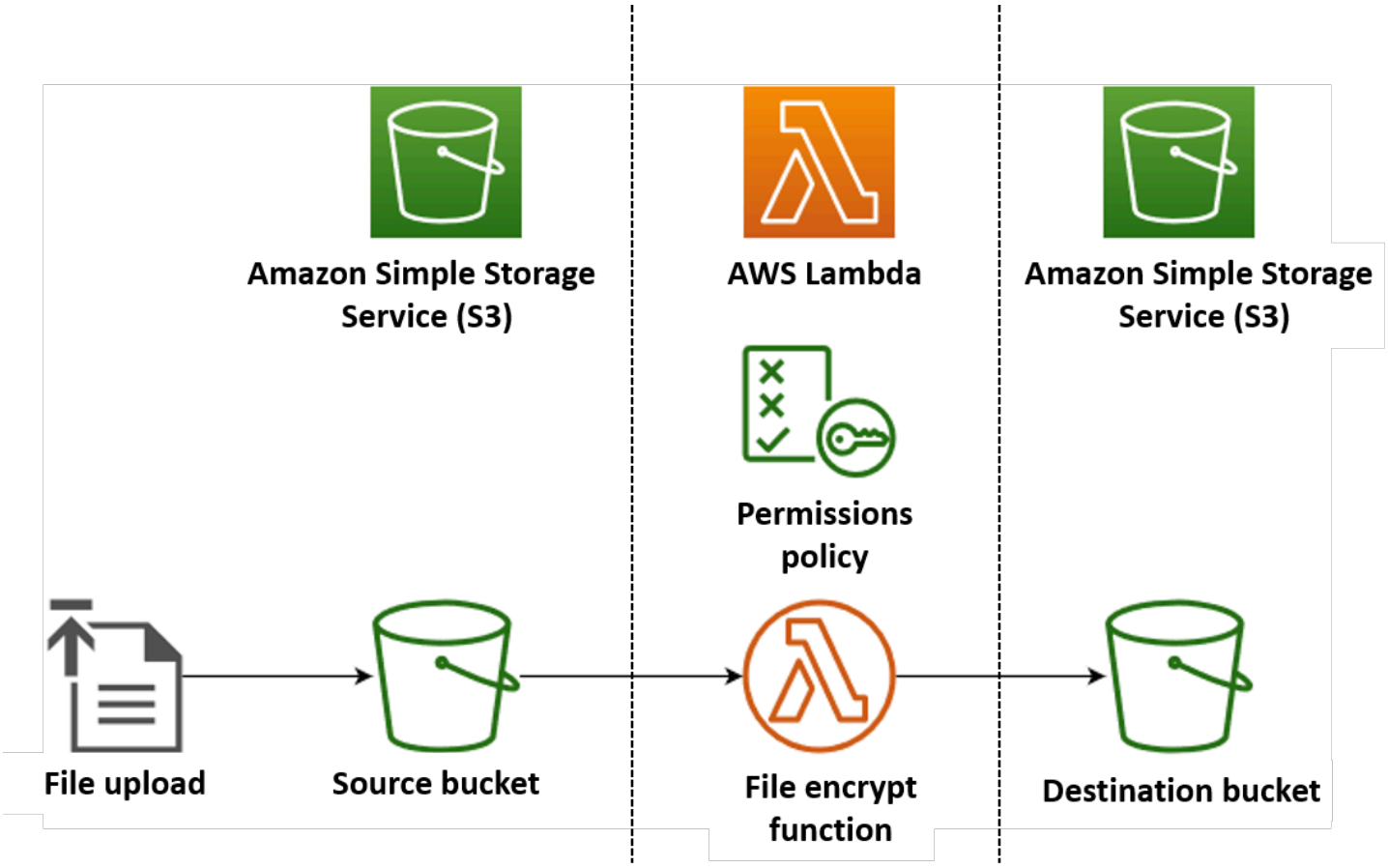
创建无服务器文件处理应用程序

Lambda 的一个常见用例是执行文件处理任务。例如，使用 Lambda 函数从 HTML 文件或图像自动创建 PDF 文件，或者在用户上传图像时创建缩略图。

在本示例中，当 PDF 文件上传至 Amazon Simple Storage Service（Amazon S3）存储桶时，创建好的应用程序会自动加密 PDF 文件。要实现此应用程序，您要创建以下资源：

- S3 存储桶，供用户上传 PDF 文件
- Python 中的 Lambda 函数，用于读取上传的文件并创建加密的、受密码保护的文件版本
- 第二个 S3 存储桶，供 Lambda 保存加密文件

您还可以创建一个 AWS Identity and Access Management（IAM）策略，来授予 Lambda 函数对 S3 存储桶执行读写操作的权限。



> **Tip**
>
> 若为 Lambda 全新用户，建议先阅读教程创建第一个函数，再创建此示例应用程序。

您可以使用 AWS Management Console 或 AWS Command Line Interface（AWS CLI）创建并配置资源来手动部署应用程序。您也可以使用 AWS Serverless Application Model（AWS SAM）来部署应用程序。AWS SAM 是基础设施即代码（IaC）工具。若借助 IaC，则不必手动创建资源，只需在代码中定义资源，就能自动部署这些资源。

若想在部署此示例应用程序之前，了解有关将 Lambda 与 IaC 结合使用的更多信息，请参阅基础设施即代码（IaC）。

先决条件

在创建示例应用程序之前，确保已安装好所需的命令行工具。

- AWS CLI

  您可以使用 AWS Management Console 或 AWS CLI 手动部署应用程序的资源。要使用 CLI，请按照《AWS Command Line Interface User Guide》中的安装说明进行安装。
- AWS SAM CLI

  若要使用 AWS SAM 部署示例应用程序，则需要同时安装 AWS CLI 和 AWS SAM CLI。要安装 AWS SAM CLI，请按照《AWS SAM User Guide》中的安装说明进行安装。
- pytest 模块

  部署应用程序后，您可以使用我们提供的自动化 Python 测试脚本对其进行测试。要使用此脚本，请运行以下命令，将 `pytest` 程序包安装到本地开发环境中：

  ```
  pip install pytest
  ```

要使用 AWS SAM 部署应用程序，还必须在生成计算机上安装 Docker。

下载示例应用程序文件

要创建并测试示例应用程序，请在项目目录中创建以下文件：

- `lambda_function.py` – 执行文件加密的 Lambda 函数的 Python 函数代码

- `requirements.txt` – 定义 Python 函数代码所需的依赖项的清单文件
- `template.yaml` – 可用于部署应用程序的 AWS SAM 模板
- `test_pdf_encrypt.py` – 可用于自动测试应用程序的测试脚本
- `pytest.ini` – 测试脚本的配置文件

展开以下各部分，查看代码，继而详细了解每个文件在创建和测试应用程序中所起的作用。要在本地计算机上创建文件，请复制并粘贴以下代码，或从 aws-lambda-developer-guideGitHub 存储库下载文件。

Python 函数代码

将以下代码复制并粘贴到名为 `lambda_function.py` 的文件。

```
from pypdf import PdfReader, PdfWriter
import uuid
import os
from urllib.parse import unquote_plus
import boto3

# Create the S3 client to download and upload objects from S3
s3_client = boto3.client('s3')

def lambda_handler(event, context):
    # Iterate over the S3 event object and get the key for all uploaded files
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = unquote_plus(record['s3']['object']['key']) # Decode the S3 object key to
 remove any URL-encoded characters
        download_path = f'/tmp/{uuid.uuid4()}.pdf' # Create a path in the Lambda tmp
 directory to save the file to
        upload_path = f'/tmp/converted-{uuid.uuid4()}.pdf' # Create another path to
 save the encrypted file to

        # If the file is a PDF, encrypt it and upload it to the destination S3 bucket
        if key.lower().endswith('.pdf'):
            s3_client.download_file(bucket, key, download_path)
            encrypt_pdf(download_path, upload_path)
            encrypted_key = add_encrypted_suffix(key)
            s3_client.upload_file(upload_path, f'{bucket}-encrypted', encrypted_key)

# Define the function to encrypt the PDF file with a password
```

```
def encrypt_pdf(file_path, encrypted_file_path):
    reader = PdfReader(file_path)
    writer = PdfWriter()

    for page in reader.pages:
        writer.add_page(page)

    # Add a password to the new PDF
    writer.encrypt("my-secret-password")

    # Save the new PDF to a file
    with open(encrypted_file_path, "wb") as file:
        writer.write(file)

# Define a function to add a suffix to the original filename after encryption
def add_encrypted_suffix(original_key):
    filename, extension = original_key.rsplit('.', 1)
    return f'{filename}_encrypted.{extension}'
```

> **Note**
>
> 在此示例代码中，加密文件 (`my-secret-password`) 的密码被硬编码到函数代码中。在生产应用程序中，切勿在函数代码中包含密码等敏感信息。使用 AWS Secrets Manager 安全地存储敏感参数。

Python 函数代码包含三个函数：一个是 Lambda 在调用函数时运行的处理程序函数，以及两个名为 `add_encrypted_suffix` 和 `encrypt_pdf` 的独立函数，处理程序调用它们来执行 PDF 加密。

在 Amazon S3 调用函数时，Lambda 会将一个 JSON 格式的事件参数传递给函数，该参数包含有关导致调用的事件详细信息。在本例中，此类信息包括 S3 存储桶的名称和上传文件的对象键。要了解有关 Amazon S3 事件对象格式的更多信息，请参阅the section called “S3”。

然后，函数使用 AWS SDK for Python (Boto3) 将事件对象中指定的 PDF 文件下载到其本地临时存储目录，再使用 `pypdf` 库对文件进行加密。

最后，函数使用 Boto3 SDK 将加密文件存储在 S3 目标存储桶中。

`requirements.txt` 清单文件

将以下代码复制并粘贴到名为 `requirements.txt` 的文件。

```
boto3
pypdf
```

在本示例中，函数代码只有两个不属于标准 Python 库的依赖项：一是适用于 Python 的 SDK（Boto3），二是函数用来执行 PDF 加密的 pypdf 程序包。

> **Note**
>
> 适用于 Python 的 SDK（Boto3）的一个版本作为 Lambda 运行时的一部分包含在内，因此无需将 Boto3 添加到函数的部署包中，代码即可运行。不过，为了完全控制函数依赖项并避免可能出现的版本不一致问题，Python 的最佳实践是将所有函数依赖项包含在函数的部署包中。请参阅the section called “Python 中的运行时系统依赖项”，了解更多信息。

AWS SAM 模板

将以下代码复制并粘贴到名为 `template.yaml` 的文件。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  EncryptPDFFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: EncryptPDF
      Architectures: [x86_64]
      CodeUri: ./
      Handler: lambda_function.lambda_handler
      Runtime: python3.12
      Timeout: 15
      MemorySize: 256
      LoggingConfig:
        LogFormat: JSON
      Policies:
        - AmazonS3FullAccess
      Events:
        S3Event:
          Type: S3
          Properties:
            Bucket: !Ref PDFSourceBucket
```

```
            Events: s3:ObjectCreated:*

  PDFSourceBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: EXAMPLE-BUCKET

  EncryptedPDFBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: EXAMPLE-BUCKET-encrypted
```

AWS SAM 模板定义了您为应用程序创建的资源。在本示例中，模板使用 `AWS::Serverless::Function` 类型定义一个 Lambda 函数，并使用 `AWS::S3::Bucket` 类型定义两个 S3 存储桶。模板中指定的存储桶名称是占位符。在使用 AWS SAM 部署应用程序之前，您需要编辑模板，使用符合 S3 存储桶命名规则的全局唯一名称重命名存储桶。the section called “使用 AWS SAM 来部署资源” 中将进一步介绍该步骤。

Lambda 函数资源的定义使用 `S3Event` 事件属性为函数配置触发器。只要在源存储桶中创建对象，该触发器就会调用函数。

函数定义还指定了要附加到函数执行角色的 AWS Identity and Access Management（IAM）策略。AWS 托管式策略 `AmazonS3FullAccess` 授予函数在 Amazon S3 中读取和写入对象所需的权限。

自动化测试脚本

将以下代码复制并粘贴到名为 `test_pdf_encrypt.py` 的文件。

```
import boto3
import json
import pytest
import time
import os

@pytest.fixture
def lambda_client():
    return boto3.client('lambda')

@pytest.fixture
def s3_client():
    return boto3.client('s3')
```

```
@pytest.fixture
def logs_client():
    return boto3.client('logs')

@pytest.fixture(scope='session')
def cleanup():
    # Create a new S3 client for cleanup
    s3_client = boto3.client('s3')

    yield
    # Cleanup code will be executed after all tests have finished

    # Delete test.pdf from the source bucket
    source_bucket = 'EXAMPLE-BUCKET'
    source_file_key = 'test.pdf'
    s3_client.delete_object(Bucket=source_bucket, Key=source_file_key)
    print(f"\nDeleted {source_file_key} from {source_bucket}")

    # Delete test_encrypted.pdf from the destination bucket
    destination_bucket = 'EXAMPLE-BUCKET-encrypted'
    destination_file_key = 'test_encrypted.pdf'
    s3_client.delete_object(Bucket=destination_bucket, Key=destination_file_key)
    print(f"Deleted {destination_file_key} from {destination_bucket}")


@pytest.mark.order(1)
def test_source_bucket_available(s3_client):
    s3_bucket_name = 'EXAMPLE-BUCKET'
    file_name = 'test.pdf'
    file_path = os.path.join(os.path.dirname(__file__), file_name)

    file_uploaded = False
    try:
        s3_client.upload_file(file_path, s3_bucket_name, file_name)
        file_uploaded = True
    except:
        print("Error: couldn't upload file")

    assert file_uploaded, "Could not upload file to S3 bucket"



@pytest.mark.order(2)
def test_lambda_invoked(logs_client):
```

```
    # Wait for a few seconds to make sure the logs are available
    time.sleep(5)

    # Get the latest log stream for the specified log group
    log_streams = logs_client.describe_log_streams(
        logGroupName='/aws/lambda/EncryptPDF',
        orderBy='LastEventTime',
        descending=True,
        limit=1
    )

    latest_log_stream_name = log_streams['logStreams'][0]['logStreamName']

    # Retrieve the log events from the latest log stream
    log_events = logs_client.get_log_events(
        logGroupName='/aws/lambda/EncryptPDF',
        logStreamName=latest_log_stream_name
    )

    success_found = False
    for event in log_events['events']:
        message = json.loads(event['message'])
        status = message.get('record', {}).get('status')
        if status == 'success':
            success_found = True
            break

    assert success_found, "Lambda function execution did not report 'success' status in
 logs."

@pytest.mark.order(3)
def test_encrypted_file_in_bucket(s3_client):
    # Specify the destination S3 bucket and the expected converted file key
    destination_bucket = 'EXAMPLE-BUCKET-encrypted'
    converted_file_key = 'test_encrypted.pdf'

    try:
        # Attempt to retrieve the metadata of the converted file from the destination
 S3 bucket
        s3_client.head_object(Bucket=destination_bucket, Key=converted_file_key)
    except s3_client.exceptions.ClientError as e:
        # If the file is not found, the test will fail
```

```
        pytest.fail(f"Converted file '{converted_file_key}' not found in the
 destination bucket: {str(e)}")

def test_cleanup(cleanup):
    # This test uses the cleanup fixture and will be executed last
    pass
```

自动化测试脚本将执行三个测试函数来确认应用程序运行是否正确：

- 测试 `test_source_bucket_available` 通过将测试 PDF 文件上传到存储桶来确认源存储桶是否已成功创建。
- 测试 `test_lambda_invoked` 通过询问函数的最新 CloudWatch Logs 日志流，来确认上传测试文件时，Lambda 函数是否已运行并报告成功。
- 测试 `test_encrypted_file_in_bucket` 确认目标存储桶是否包含加密 `test_encrypted.pdf` 文件。

待这些测试运行完毕后，脚本会执行额外的清理步骤，以便从源存储桶和目标存储桶中删除 `test.pdf` 和 `test_encrypted.pdf` 文件。

与 AWS SAM 模板一样，此文件中指定的存储桶名称是占位符。在运行测试前，必须使用应用程序的真实存储桶名称编辑此文件。the section called “使用自动化脚本测试应用程序” 中将进一步介绍此步骤

测试脚本配置文件

将以下代码复制并粘贴到名为 `pytest.ini` 的文件。

```
[pytest]
markers =
    order: specify test execution order
```

这是为了指定 `test_pdf_encrypt.py` 脚本中测试运行的顺序。

部署应用程序

您可以手动或使用 AWS SAM 自动创建并部署此示例应用程序的资源。在生产环境中，建议使用 AWS SAM 之类的 IaC 工具来快速、可重复地部署整个无服务器应用程序，无需采用手动流程。

在本示例中，可按照控制台或 AWS CLI 说明了解如何单独配置每种 AWS 资源，也可以直接跳至 the section called “使用 AWS SAM 来部署资源”，使用几个 CLI 命令快速部署应用程序。

手动部署资源

要手动部署应用程序，请执行以下步骤：

- 创建源和目标 Amazon S3 存储桶
- 创建一个 Lambda 函数，用于加密 PDF 文件并将加密版文件保存到 S3 存储桶
- 配置一个 Lambda 触发器，该触发器将在对象上传到源存储桶时调用函数

按照以下段落中的说明创建并配置资源。

创建两个 S3 存储桶

先创建两个 S3 存储桶。第一个是源存储桶，供您向其上传 PDF 文件。第二个是目标存储桶，供 Lambda 保存调用函数时加密的文件。

Console

创建 S3 存储桶（控制台）

1. 打开 Amazon S3 控制台的存储桶页面。
2. 选择 Create bucket（创建存储桶）。
3. 在 General configuration（常规配置）下，执行以下操作：
   a. 对于存储桶名称，输入符合 Amazon S3 存储桶命名规则的全局唯一名称。存储桶名称只能由小写字母、数字、句点（.）和连字符（-）组成。
   b. 对于 AWS 区域，请选择最接近您地理位置的 AWS 区域。在部署过程的后面部分，您必须在同个 AWS 区域 中创建 Lambda 函数，因此请记下您选择的区域。
4. 将所有其他选项设置为默认值并选择创建存储桶。
5. 重复步骤 1 到 4 以创建自己的目标存储桶。在存储桶名称中输入 **`SOURCEBUCKET-encrypted`**，其中 **`SOURCEBUCKET`** 是您刚刚创建的源存储桶的名称。

AWS CLI

创建 Amazon S3 存储桶（AWS CLI）

1. 运行以下 CLI 命令来创建自己的源存储桶。您为存储桶选择的名称必须具有全局唯一性，并遵守 Amazon S3 存储桶命名规则。名称只能由小写字母、数字、句点（.）和连字符（-）组成。对于 `region` 和 `LocationConstraint`，请选择最接近您地理位置的 AWS 区域。

```
aws s3api create-bucket --bucket SOURCEBUCKET --region us-west-2 \
--create-bucket-configuration LocationConstraint=us-west-2
```

在本教程的后面部分，您必须在与源存储桶相同的 AWS 区域 中创建 Lambda 函数，因此请记下您选择的区域。

2. 运行以下命令来创建自己的目标存储桶。对于存储桶名称，必须使用 **SOURCEBUCKET-encrypted**，其中 **SOURCEBUCKET** 是您在步骤 1 中创建的源存储桶名称。对于 `region` 和 `LocationConstraint`，请选择与用于创建源存储桶时相同的 AWS 区域。

```
aws s3api create-bucket --bucket SOURCEBUCKET-encrypted --region us-west-2 \
--create-bucket-configuration LocationConstraint=us-west-2
```

创建执行角色（仅限 AWS CLI）

执行角色是一个 IAM 角色，用于向 Lambda 函数授予访问 AWS 服务 和资源的权限。使用 Lambda 控制台创建函数时，Lambda 会自动创建一个执行角色。只有选择使用 AWS CLI 来部署应用程序时，才需要手动创建角色。要授予函数对 Amazon S3 的读取和写入权限，必须附加 AWS 托管式策略 `AmazonS3FullAccess`。

Console

只有选择使用 AWS CLI 来部署应用程序时，才需要执行此步骤。

AWS CLI

创建执行角色并附加 **AmazonS3FullAccess** 托管式策略（AWS CLI）

1. 将下列 JSON 保存在名为 `trust-policy.json` 的文件中。此信任策略允许 Lambda 通过向服务主体 `lambda.amazonaws.com` 授予调用 AWS Security Token Service（AWS STS）`AssumeRole` 操作的权限来使用该角色的权限。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
```

```
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. 在保存 JSON 信任策略文档的目录中，运行以下 CLI 命令来创建执行角色。

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. 要附加 AmazonS3FullAccess 托管式策略，请运行下列 CLI 命令。

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::aws:policy/AmazonS3FullAccess
```

创建函数部署包

要创建函数，您需要创建包含函数代码和所有依赖项的部署包。对于此应用程序，函数代码使用单独的库来加密 PDF。

创建部署包

1. 导航到包含之前创建的或从 GitHub 下载的 lambda_function.py 和 requirements.txt 文件的项目目录，然后创建一个名为 package 的新目录。
2. 运行以下命令，将 requirements.txt 文件中指定的依赖项安装到 package 目录中。

```
pip install -r requirements.txt --target ./package/
```

3. 创建一个包含应用程序代码及其依赖项的 .zip 文件。在 Linux 或 MacOS 中，从命令行界面运行以下命令。

```
cd package
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

在 Windows 中，使用您首选的压缩工具来创建 lambda_function.zip 文件。确保您的 lambda_function.py 文件和包含依赖项的文件夹都位于.zip 文件的根目录下。

您也可以使用 Python 虚拟环境创建部署包。请参阅 将 .zip 文件归档用于 Python Lambda 函数。

创建 Lambda 函数

现在，您可以使用在上一步中创建的部署包来部署 Lambda 函数。

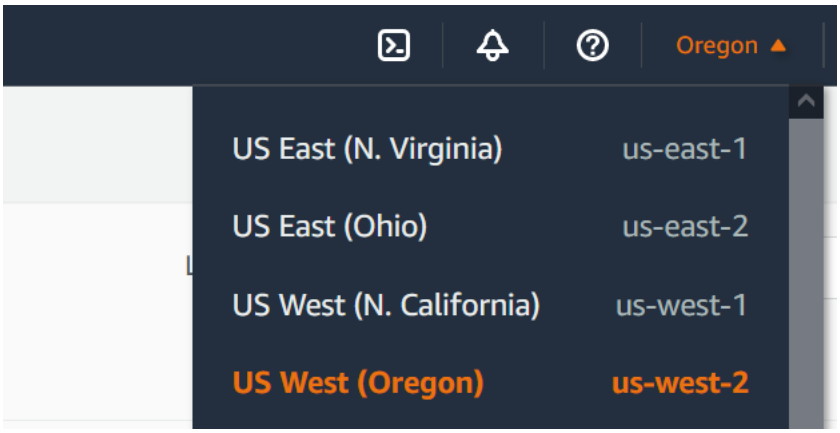
Console

创建函数（控制台）

要使用控制台创建 Lambda 函数，首先要创建包含一些“Hello world”代码的基本函数。然后，通过上传在上一步中创建的 .zip 文件，将此代码替换为自己的函数代码。

为确保加密大型 PDF 文件时函数不会超时，必须配置该函数的内存和超时设置。您还要将函数的日志格式设置为 JSON。使用提供的测试脚本时，必须配置 JSON 格式的日志，以便其可以从 CloudWatch Logs 中读取函数的调用状态，从而确认是否成功调用。

1. 打开 Lamba 控制台的函数页面。
2. 确保您在创建 S3 存储桶所在的同一 AWS 区域 内操作。您可以使用屏幕顶部的下拉列表更改区域。



3. 选择 Create function (创建函数)。
4. 选择从头开始创作。
5. 在基本信息中，执行以下操作：
   a. 对于 Function name（函数名称），请输入 **`EncryptPDF`**。
   b. 对于运行时，选择 Python 3.12。
   c. 对于架构，选择 x86_64。
6. 选择 Create function (创建函数)。

上传函数代码（控制台）

1. 在代码源窗格中，选择上传自。
2. 选择 .zip 文件。
3. 选择上传。
4. 在文件选择器中，选择 .zip 文件，然后选择打开。
5. 选择保存。

配置函数内存和超时（控制台）

1. 选择函数的配置选项卡。
2. 在常规配置窗格中，选择编辑。
3. 将内存设置为 256 MB，并将超时设置为 15 秒。
4. 选择保存。

配置日志格式（控制台）

1. 选择函数的配置选项卡。
2. 选择监控和操作工具。
3. 在日志记录配置窗格中，选择编辑。
4. 对于日志记录配置，选择 JSON。
5. 选择保存。

AWS CLI

创建函数（AWS CLI）

- 从包含 `lambda_function.zip` 文件的目录中运行以下命令。对于 `region` 参数，将 `us-west-2` 替换为创建 S3 存储桶时使用的区域。

```
aws lambda create-function --function-name EncryptPDF \
--zip-file fileb://lambda_function.zip --handler lambda_function.lambda_handler
 \
--runtime python3.12 --timeout 15 --memory-size 256 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-west-2 \
```

```
--logging-config LogFormat=JSON
```

配置 Amazon S3 触发器来调用函数

为了在将文件上传到源存储桶时运行 Lambda 函数，您需要为函数配置触发器。您可以使用控制台或 AWS CLI 配置 Amazon S3 触发器。

> **Important**
>
> 此程序将 S3 存储桶配置为每次在该存储桶中创建对象时调用您的函数。请确保仅在源存储桶上配置。如果您的 Lambda 函数在调用此函数的同一个存储桶中创建对象，则可以在循环中持续调用您的函数。这可能会导致您的 AWS 账户 产生额外费用。

Console

配置 Amazon S3 触发器（控制台）

1. 打开 Lambda 控制台的函数页面，然后选择函数 (`EncryptPDF`)。
2. 选择添加触发器。
3. 选择 S3。
4. 在存储桶下，选择自己的源存储桶。
5. 在事件类型下，选择所有对象创建事件。
6. 在递归调用下，选中复选框以确认知晓不建议使用相同的 S3 存储桶用于输入和输出。您可以阅读 Serverless Land 中的 Recursive patterns that cause run-away Lambda functions，进一步了解 Lambda 中的递归调用模式。
7. 选择添加。

   在您使用 Lambda 控制台创建触发器时，Lambda 会自动创建基于资源的策略，授予您选择的服务调用函数的权限。

AWS CLI

配置 Amazon S3 触发器（AWS CLI）

1. 为了在添加文件时让 Amazon S3 源存储桶能调用函数，您首先需要使用基于资源的策略为函数配置权限。基于资源的策略声明授予其他 AWS 服务 调用您函数的权限。要授予 Amazon

S3 调用函数的权限，请运行以下 CLI 命令。请务必将 source-account 参数替换为您的 AWS 账户 ID 并使用自己的源存储桶名称。

```
aws lambda add-permission --function-name EncryptPDF \
--principal s3.amazonaws.com --statement-id s3invoke --action
 "lambda:InvokeFunction" \
--source-arn arn:aws:s3:::SOURCEBUCKET \
--source-account 123456789012
```

您使用此命令定义的策略允许 Amazon S3 仅在源存储桶上执行操作时调用函数。

> **Note**
>
> 虽然 S3 存储桶名称具有全局唯一性，但在使用基于资源的策略时，最佳做法是指定存储桶必须属于您的账户。这是因为，如果删除一个存储桶，则另一个 AWS 账户 账户可能会创建具有相同 Amazon 资源名称（ARN）的存储桶。

2. 将下列 JSON 保存在名为 notification.json 的文件中。在应用到您的源存储桶时，此 JSON 会将存储桶配置为在每次添加新对象时向 Lambda 函数发送通知。将 Lambda 函数 ARN 中的 AWS 账户 号码和 AWS 区域 替换为您自己的账号和区域。

```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "EncryptPDFEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:EncryptPDF",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. 运行以下 CLI 命令，将您创建的 JSON 文件中的通知设置应用到源存储桶。将 SOURCEBUCKET 替换为源存储桶的名称。

```
aws s3api put-bucket-notification-configuration --bucket SOURCEBUCKET \
--notification-configuration file://notification.json
```

要了解有关 put-bucket-notification-configuration 命令和 notification-configuration 选项的更多信息，请参阅 AWS CLI 命令参考中的 put-bucket-notification-configuration。

使用 AWS SAM 来部署资源

要使用 AWS SAM CLI 来部署示例应用程序，请执行以下步骤。

请确保已安装最新版本的 CLI，并且生成计算机上安装了 Docker。

1. 编辑 template.yaml 文件来指定 S3 存储桶的名称。S3 存储桶必须具有符合 S3 存储桶命名规则的全局唯一名称。

   将存储桶名称 EXAMPLE-BUCKET 替换为所选名称，该名称必须由小写字母、数字、点（.）和连字符（-）组成。对于目标存储桶，将 EXAMPLE-BUCKET-encrypted 替换为 <source-bucket-name>-encrypted，其中 <source-bucket> 是您为源存储桶选择的名称。

2. 在保存 template.yaml、lambda_function.py 和 requirements.txt 文件的目录中，运行以下命令。

   ```
   sam build --use-container
   ```

   此命令会收集应用程序的构建构件，并将其以适当的格式放置在适当的位置进行部署。指定 --use-container 选项会在类似 Lambda 的 Docker 容器中构建函数。我们在这里使用它，您无需在本地计算机上安装 Python 3.12 即可进行构建。

   在构建过程中，AWS SAM 会在以模板中的 CodeUri 属性指定的位置中查找 Lambda 函数代码。在本例中，我们将当前目录指定为位置（./）。

   如果存在 requirements.txt 文件，则 AWS SAM 使用该文件来收集指定的依赖项。默认情况下，AWS SAM 会创建包含函数代码和依赖项的 .zip 部署包。您也可以选择使用 PackageType 属性将函数部署为容器映像。

3. 要部署应用程序并创建 AWS SAM 模板中指定的 Lambda 和 Amazon S3 资源，请运行以下命令。

   ```
   sam deploy --guided
   ```

使用 `--guided` 标志意味着 AWS SAM 将向您显示提示，以指导您完成部署过程。对于此部署，请按 Enter 接受默认选项。

在部署过程中，AWS SAM 将在 AWS 账户 中创建以下资源：

- 一个名为 `sam-app` 的 AWS CloudFormation 堆栈
- 一个名为 `EncryptPDF` 的 Lambda 函数
- 两个 S3 存储桶，其名称为编辑 `template.yaml` AWS SAM 模板文件时选择的名称
- 一个函数的 IAM 执行角色，名称格式为 `sam-app-EncryptPDFFunctionRole-2qGaapHFWOQ8`

AWS SAM 创建完资源后，您将看到以下消息：

```
Successfully created/updated stack - sam-app in us-west-2
```

测试应用程序

要测试应用程序，必须将 PDF 文件上传到源存储桶，还要确认 Lambda 已在目标存储桶中创建了加密版文件。在本示例中，您可以使用控制台或 AWS CLI 对此进行手动测试，也可以使用提供的测试脚本进行自动测试。

对于生产应用程序，可以使用单元测试等传统的测试方法和技术，来确认 Lambda 函数代码是否正常运行。最佳实践也是执行与所提供测试脚本中的测试类似的测试，这些测试使用真实的基于云的资源执行集成测试。在云端中执行集成测试可确认基础架构是否已正确部署，并确认事件是否按预期在不同的服务之间流动。要了解更多信息，请参阅 测试无服务器函数。

手动测试应用程序

您可以将 PDF 文件添加到 Amazon S3 源存储桶来手动测试函数。将文件添加到源存储桶时，应自动调用 Lambda 函数，并应在目标存储桶中存储加密版文件。

Console

通过上传文件来测试应用程序（控制台）

1. 要将 PDF 文件上传到 S3 存储桶，请执行以下操作：

   a. 打开 Amazon S3 控制台的存储桶页面，然后选择您的源存储桶。

b. 选择上传。

c. 选择添加文件，然后使用文件选择器选择要上传的 PDF 文件。

d. 选择打开，然后选择上传。

2. 执行以下操作，验证 Lambda 是否已将加密版 PDF 文件保存在目标存储桶中：

a. 导航回 Amazon S3 控制台的存储桶页面，然后选择您的目标存储桶。

b. 在对象窗格中，现在应该可以到一个名称格式为 `filename_encrypted.pdf` 的文件（其中 `filename.pdf` 是已上传到源存储桶的文件的名称）。要下载加密的 PDF，请选择所需文件，然后选择下载。

c. 确认是否可以使用 Lambda 函数保护的密码 (`my-secret-password`) 打开下载的文件。

AWS CLI

通过上传文件来测试应用程序（AWS CLI）

1. 在包含要上传的 PDF 文件的目录中，运行以下 CLI 命令。将 `--bucket` 参数替换为源存储桶的名称。对于 `--key` 和 `--body` 参数，请使用测试文件的文件名。

```
aws s3api put-object --bucket SOURCEBUCKET --key test.pdf --body ./test.pdf
```

2. 验证函数是否已创建加密版文件并已保存到目标 S3 存储桶中。运行以下 CLI 命令，将 `SOURCEBUCKET-encrypted` 替换为自己的目标存储桶的名称。

```
aws s3api list-objects-v2 --bucket SOURCEBUCKET-encrypted
```

如果函数成功运行，您将看到类似于以下内容的输出。目标存储桶应包含名称格式为 *<your_test_file>*`_encrypted.pdf` 的文件，其中 <your_test_file> 是已上传文件的名称。

```
{
    "Contents": [
        {
            "Key": "test_encrypted.pdf",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        }
```

```
    ]
}
```

3. 要下载 Lambda 保存在目标存储桶中的文件，请运行以下 CLI 命令。将 --bucket 参数替换为目标存储桶的名称。对于 --key 参数，请使用文件名 *<your_test_file>*_encrypted.pdf，其中 <your_test_file> 是已上传的测试文件的名称。

```
aws s3api get-object --bucket SOURCEBUCKET-encrypted --key test_encrypted.pdf
 my_encrypted_file.pdf
```

该命令会将文件下载到当前目录并将其另存为 my_encrypted_file.pdf。

4. 确认是否可以使用 Lambda 函数保护的密码 (my-secret-password) 打开下载的文件。

使用自动化脚本测试应用程序

要使用提供的测试脚本测试应用程序，请先确保已在本地环境中安装了 pytest 模块。可以通过运行以下命令安装 pytest：

```
pip install pytest
```

您还需要编辑 test_pdf_encrypt.py 文件中的代码，将以占位符表示的存储桶名称分别替换为 Amazon S3 源存储桶和目标存储桶的名称。对 test_pdf_encrypt.py 进行以下更改：

- 在 test_source_bucket_available 函数中，将 EXAMPLE-BUCKET 替换为源存储桶的名称。
- 在 test_encrypted_file_in_bucket 函数中，将 EXAMPLE-BUCKET-encrypted 替换为 <source-bucket>-encrypted，其中 <source-bucket> 是源存储桶的名称。
- 在 cleanup 函数中，将 EXAMPLE-BUCKET 替换为源存储桶的名称，并将 EXAMPLE-BUCKET-encrypted 替换为 #source-bucket>-encrypted，其中 <source-bucket> 是源存储桶的名称。

要运行测试，请执行以下操作：

- 将名为 test.pdf 的 PDF 文件保存在包含 test_pdf_encrypt.py 和 pytest.ini 文件的目录中。
- 打开终端或 Shell 程序，从包含测试文件的目录中运行以下命令。

```
pytest -s -v
```

测试完成后，输出应与以下内容类似：

```
============================================================= test session starts
 ==============================================================
platform linux -- Python 3.12.2, pytest-7.2.2, pluggy-1.0.0 -- /usr/bin/python3
cachedir: .pytest_cache
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/
pdf_encrypt_app/.hypothesis/examples')
Test order randomisation NOT enabled. Enable with --random-order or --random-order-
bucket=<bucket_type>
rootdir: /home/pdf_encrypt_app, configfile: pytest.ini
plugins: anyio-3.7.1, hypothesis-6.70.0, localserver-0.7.1, random-order-1.1.0
collected 4 items

test_pdf_encrypt.py::test_source_bucket_available PASSED
test_pdf_encrypt.py::test_lambda_invoked PASSED
test_pdf_encrypt.py::test_encrypted_file_in_bucket PASSED
test_pdf_encrypt.py::test_cleanup PASSED
Deleted test.pdf from EXAMPLE-BUCKET
Deleted test_encrypted.pdf from EXAMPLE-BUCKET-encrypted


============================================================== 4 passed in 7.32s
 ==============================================================
```

后续步骤

您已经创建了示例应用程序，现在可以在所提供代码的基础上创建其他类型的文件处理应用程序。修改 `lambda_function.py` 文件中的代码，为用例实现文件处理逻辑。

许多常见的文件处理用例都涉及图像处理。使用 Python 时，pillow 等常用的图像处理库通常包含 C 或 C++ 组件。为了确保函数的部署包与 Lambda 执行环境兼容，请务必使用正确的源分发二进制文件。

使用 AWS SAM 来部署资源时，必须额外采取一些步骤，以便在部署包中包含正确的源分发。由于 AWS SAM 不会为与生成计算机不同的平台安装依赖项，因此，如果生成机器使用与 Lambda 执行环境不同的操作系统或架构，则在 `requirements.txt` 文件中指定正确的源分发（`.whl` 文件）将不适用。所以，应执行以下操作之一：

- 运行 `sam build` 时使用 `--use-container` 选项。指定此选项时，AWS SAM 会下载与 Lambda 执行环境兼容的容器基础映像，并使用该映像在 Docker 容器中构建函数的部署包。要了解更多信息，请参阅 Building a Lambda function inside of a provided container。
- 使用正确的源分发二进制文件自行构建函数的 .zip 部署包，并将 .zip 文件保存在 AWS SAM 模板中指定为 `CodeUri` 的目录中。要了解有关使用二进制分发文件为 Python 构建.zip 部署包的更多信息，请参阅the section called “创建含依赖项的 .zip 部署包”和the section called “使用原生库创建 .zip 部署包”。

创建用于执行定期数据库维护的应用程序

您可以使用 AWS Lambda 来替换自动系统备份、文件转换和维护任务等计划进程。在此示例中，您将创建无服务器应用程序，该应用程序通过删除旧条目对 DynamoDB 表执行定期维护。该应用程序按照 cron 计划，使用 EventBridge 调度器调用 Lambda 函数。调用时，该函数会在表中查询早于一年的项目，然后将其删除。该函数会将每个已删除的项目记录在 CloudWatch Logs 中。

要实现此示例，请先创建 DynamoDB 表，并在其中填充部分测试数据以供您进行函数查询。然后，创建具有 EventBridge 调度器触发器和 IAM 执行角色的 Python Lambda 函数，该执行角色可授予函数读取和删除表中项目的权限。

> **Tip**
>
> 若您为 Lambda 全新用户，建议先阅读教程（创建第一个函数），再创建此示例应用程序。

您可以使用 AWS Management Console 创建并配置资源来手动部署应用程序。您也可以使用 AWS Serverless Application Model（AWS SAM）来部署应用程序。AWS SAM 是基础设施即代码（IaC）工具。若借助 IaC，则不必手动创建资源，只需在代码中定义资源，就能自动部署这些资源。

若想在部署此示例应用程序之前，了解有关将 Lambda 与 IaC 结合使用的更多信息，请参阅基础设施即代码（IaC）。

先决条件

在创建示例应用程序之前，确保已安装好所需的命令行工具和程序。

- Python

  为了填充您为测试应用程序而创建的 DynamoDB 表，此示例使用 Python 脚本和 CSV 文件将数据写入表中。确保在计算机上已安装 Python 3.8 或更高版本。

- AWS SAM CLI

  若要使用 AWS SAM 创建 DynamoDB 表和部署示例应用程序，则需要同时安装 AWS SAM CLI。请按照《AWS SAM User Guide》中的 installation instructions 操作。

- AWS CLI

  要使用提供的 Python 脚本填充测试表，您需要安装并配置 AWS CLI。由于脚本使用 AWS SDK for Python (Boto3)，它需要访问您的 AWS Identity and Access Management（IAM）证书。您还需要安装 AWS CLI 才能使用 AWS SAM 部署资源。要安装 CLI，请按照《AWS Command Line Interface User Guide》中的 installation instructions 操作。

- Docker

  要使用 AWS SAM 部署应用程序，还必须在生成计算机上安装 Docker。按照 Docker 文档网站上的 Install Docker Engine 中的说明操作。

下载示例应用程序文件

要创建示例数据库和用于定期维护的应用程序，请在项目目录中创建以下文件：

数据库文件示例

- `template.yaml`：可用于创建 DynamoDB 表的 AWS SAM 模板
- `sample_data.csv`：包含要加载到表中的示例数据的 CSV 文件
- `load_sample_data.py`：用于将 CSV 文件中的数据写入表中的 Python 脚本

用于定期维护的应用程序文件

- `lambda_function.py`：执行数据库维护的 Lambda 函数的 Python 函数代码
- `requirements.txt` – 定义 Python 函数代码所需的依赖项的清单文件
- `template.yaml` – 可用于部署应用程序的 AWS SAM 模板

测试文件

- `test_app.py`：Python 脚本，用于扫描表并通过输出所有超过一年的记录来确认函数是否成功运行

展开以下各部分，查看代码，继而详细了解每个文件在创建和测试应用程序中所起的作用。要在本地计算机上创建文件，请复制并粘贴以下代码。

AWS SAM 模板（DynamoDB 表示例）

将以下代码复制并粘贴到名为 `template.yaml` 的文件。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM Template for DynamoDB Table with Order_number as Partition Key and
 Date as Sort Key

Resources:
  MyDynamoDBTable:
    Type: AWS::DynamoDB::Table
    DeletionPolicy: Retain
    UpdateReplacePolicy: Retain
    Properties:
      TableName: MyOrderTable
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: Order_number
          AttributeType: S
        - AttributeName: Date
          AttributeType: S
      KeySchema:
        - AttributeName: Order_number
          KeyType: HASH
        - AttributeName: Date
          KeyType: RANGE
      SSESpecification:
        SSEEnabled: true
      GlobalSecondaryIndexes:
        - IndexName: Date-index
          KeySchema:
            - AttributeName: Date
              KeyType: HASH
          Projection:
            ProjectionType: ALL
      PointInTimeRecoverySpecification:
        PointInTimeRecoveryEnabled: true
```

```
Outputs:
  TableName:
    Description: DynamoDB Table Name
    Value: !Ref MyDynamoDBTable
  TableArn:
    Description: DynamoDB Table ARN
    Value: !GetAtt MyDynamoDBTable.Arn
```

> **Note**
>
> AWS SAM 模板使用的标准命名约定为 `template.yaml`。在此示例中，您有两个模板文件：其中一个用于创建示例数据库，另一个用于创建应用程序本身。将它们保存在项目文件夹中的单独子目录中。

此 AWS SAM 模板定义了您为测试应用程序而创建的 DynamoDB 表资源。该表使用的主键为 `Order_number`，排序键为 `Date`。为了让 Lambda 函数直接按日期查找项目，我们还定义了名为 `Date-index` 的全局二级索引。

要了解有关使用该 `AWS::DynamoDB::Table` 资源创建和配置 DynamoDB 表的更多信息，请参阅《AWS CloudFormation User Guide》中的 AWS::DynamoDB::Table。

数据库数据文件示例

将以下代码复制并粘贴到名为 `sample_data.csv` 的文件。

```
Date,Order_number,CustomerName,ProductID,Quantity,TotalAmount
2023-09-01,ORD001,Alejandro Rosalez,PROD123,2,199.98
2023-09-01,ORD002,Akua Mansa,PROD456,1,49.99
2023-09-02,ORD003,Ana Carolina Silva,PROD789,3,149.97
2023-09-03,ORD004,Arnav Desai,PROD123,1,99.99
2023-10-01,ORD005,Carlos Salazar,PROD456,2,99.98
2023-10-02,ORD006,Diego Ramirez,PROD789,1,49.99
2023-10-03,ORD007,Efua Owusu,PROD123,4,399.96
2023-10-04,ORD008,John Stiles,PROD456,2,99.98
2023-10-05,ORD009,Jorge Souza,PROD789,3,149.97
2023-10-06,ORD010,Kwaku Mensah,PROD123,1,99.99
2023-11-01,ORD011,Li Juan,PROD456,5,249.95
2023-11-02,ORD012,Marcia Oliveria,PROD789,2,99.98
2023-11-03,ORD013,Maria Garcia,PROD123,3,299.97
2023-11-04,ORD014,Martha Rivera,PROD456,1,49.99
```

```
2023-11-05,ORD015,Mary Major,PROD789,4,199.96
2023-12-01,ORD016,Mateo Jackson,PROD123,2,199.99
2023-12-02,ORD017,Nikki Wolf,PROD456,3,149.97
2023-12-03,ORD018,Pat Candella,PROD789,1,49.99
2023-12-04,ORD019,Paulo Santos,PROD123,5,499.95
2023-12-05,ORD020,Richard Roe,PROD456,2,99.98
2024-01-01,ORD021,Saanvi Sarkar,PROD789,3,149.97
2024-01-02,ORD022,Shirley Rodriguez,PROD123,1,99.99
2024-01-03,ORD023,Sofia Martinez,PROD456,4,199.96
2024-01-04,ORD024,Terry Whitlock,PROD789,2,99.98
2024-01-05,ORD025,Wang Xiulan,PROD123,3,299.97
```

此文件包含部分测试数据示例，供以标准逗号分隔值（CSV）格式填充 DynamoDB 表。

用于加载示例数据的 Python 脚本

将以下代码复制并粘贴到名为 `load_sample_data.py` 的文件。

```
import boto3
import csv
from decimal import Decimal

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('MyOrderTable')
print("DDB client initialized.")

def load_data_from_csv(filename):
    with open(filename, 'r') as file:
        csv_reader = csv.DictReader(file)
        for row in csv_reader:
            item = {
                'Order_number': row['Order_number'],
                'Date': row['Date'],
                'CustomerName': row['CustomerName'],
                'ProductID': row['ProductID'],
                'Quantity': int(row['Quantity']),
                'TotalAmount': Decimal(str(row['TotalAmount']))
            }
            table.put_item(Item=item)
            print(f"Added item: {item['Order_number']} - {item['Date']}")

if __name__ == "__main__":
    load_data_from_csv('sample_data.csv')
```

```
    print("Data loading completed.")
```

此 Python 脚本首先使用 AWS SDK for Python (Boto3) 创建与 DynamoDB 表的连接。然后，它将迭代示例数据 CSV 文件中的各行，从该行创建项目，然后使用 boto3 SDK 将该项目写入 DynamoDB 表。

Python 函数代码

将以下代码复制并粘贴到名为 `lambda_function.py` 的文件。

```
import boto3
from datetime import datetime, timedelta
from boto3.dynamodb.conditions import Key, Attr
import logging

logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    # Initialize the DynamoDB client
    dynamodb = boto3.resource('dynamodb')

    # Specify the table name
    table_name = 'MyOrderTable'
    table = dynamodb.Table(table_name)

    # Get today's date
    today = datetime.now()

    # Calculate the date one year ago
    one_year_ago = (today - timedelta(days=365)).strftime('%Y-%m-%d')

    # Scan the table using a global secondary index
    response = table.scan(
        IndexName='Date-index',
        FilterExpression='#date < :one_year_ago',
        ExpressionAttributeNames={
            '#date': 'Date'
        },
        ExpressionAttributeValues={
            ':one_year_ago': one_year_ago
        }
    )
```

```
     # Delete old items
    with table.batch_writer() as batch:
        for item in response['Items']:
            Order_number = item['Order_number']
            batch.delete_item(
                Key={
                    'Order_number': Order_number,
                    'Date': item['Date']
                }
            )
            logger.info(f'deleted order number {Order_number}')

    # Check if there are more items to scan
    while 'LastEvaluatedKey' in response:
        response = table.scan(
            IndexName='DateIndex',
            FilterExpression='#date < :one_year_ago',
            ExpressionAttributeNames={
                '#date': 'Date'
            },
            ExpressionAttributeValues={
                ':one_year_ago': one_year_ago
            },
            ExclusiveStartKey=response['LastEvaluatedKey']
        )

        # Delete old items
        with table.batch_writer() as batch:
            for item in response['Items']:
                batch.delete_item(
                    Key={
                        'Order_number': item['Order_number'],
                        'Date': item['Date']
                    }
                )

    return {
        'statusCode': 200,
        'body': 'Cleanup completed successfully'
    }
```

Python 函数代码包含 Lambda 在调用函数时运行的处理程序函数 (`lambda_handler`)。

当 EventBridge 调度器调用函数时，它会使用 AWS SDK for Python (Boto3) 创建与要执行计划维护任务的 DynamoDB 表的连接。然后，它使用 Python `datetime` 库计算一年前的日期，然后扫描表中是否有早于该日期的项目，并将其删除。

请注意，来自 DynamoDB 查询和扫描操作的响应大小限制为最大 1 MB。如果响应大于 1 MB，则 DynamoDB 会对数据进行分页，并在响应中返回 `LastEvaluatedKey` 元素。为确保我们的函数将处理表中的所有记录，我们检查此密钥是否存在，并从上次评估的位置继续执行表扫描，直到扫描完整个表。

`requirements.txt` 清单文件

将以下代码复制并粘贴到名为 `requirements.txt` 的文件。

```
boto3
```

在本示例中，函数代码只有一个不属于标准 Python 库的依赖项：即适用于 Python 的 SDK（Boto3），供函数用于扫描和删除来自 DynamoDB 表的项目。

> **Note**
>
> 适用于 Python 的 SDK（Boto3）的一个版本作为 Lambda 运行时的一部分包含在内，因此无需将 Boto3 添加到函数的部署包中，代码即可运行。不过，为了完全控制函数依赖项并避免可能出现的版本不一致问题，Python 的最佳实践是将所有函数依赖项包含在函数的部署包中。请参阅the section called “Python 中的运行时系统依赖项”，了解更多信息。

AWS SAM 模板（用于定期维护的应用程序）

将以下代码复制并粘贴到名为 `template.yaml` 的文件。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM Template for Lambda function and EventBridge Scheduler rule

Resources:
  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: ScheduledDBMaintenance
      CodeUri: ./
      Handler: lambda_function.lambda_handler
```

```
      Runtime: python3.11
      Architectures:
        - x86_64
      Events:
        ScheduleEvent:
          Type: ScheduleV2
          Properties:
            ScheduleExpression: cron(0 3 1 * ? *)
            Description: Run on the first day of every month at 03:00 AM
      Policies:
        - CloudWatchLogsFullAccess
        - Statement:
            - Effect: Allow
              Action:
                - dynamodb:Scan
                - dynamodb:BatchWriteItem
              Resource: !Sub 'arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/
MyOrderTable'

  LambdaLogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub /aws/lambda/${MyLambdaFunction}
      RetentionInDays: 30

Outputs:
  LambdaFunctionName:
    Description: Lambda Function Name
    Value: !Ref MyLambdaFunction
  LambdaFunctionArn:
    Description: Lambda Function ARN
    Value: !GetAtt MyLambdaFunction.Arn
```

> **Note**
>
> AWS SAM 模板使用的标准命名约定为 `template.yaml`。在此示例中，您有两个模板文件：其中一个用于创建示例数据库，另一个用于创建应用程序本身。将它们保存在项目文件夹中的单独子目录中。

此 AWS SAM 模板用于定义应用程序的资源。我们使用 `AWS::Serverless::Function` 资源定义 Lambda 函数。用于调用 Lambda 函数的 EventBridge 调度器计划和触发的创建都使用了此资源的

Events 属性，类型为 ScheduleV2。要了解有关在 AWS SAM 模板中定义 EventBridge 调度器的更多信息，请参阅《AWS Serverless Application Model Developer Guide》中的 ScheduleV2。

除了 Lambda 函数和 EventBridge 调度器计划外，我们还为函数定义了 CloudWatch 日志组，用于向其发送已删除项目的记录。

测试脚本

将以下代码复制并粘贴到名为 test_app.py 的文件。

```
import boto3
from datetime import datetime, timedelta
import json

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')

# Specify your table name
table_name = 'YourTableName'
table = dynamodb.Table(table_name)

# Get the current date
current_date = datetime.now()

# Calculate the date one year ago
one_year_ago = current_date - timedelta(days=365)

# Convert the date to string format (assuming the date in DynamoDB is stored as a
 string)
one_year_ago_str = one_year_ago.strftime('%Y-%m-%d')

# Scan the table
response = table.scan(
    FilterExpression='#date < :one_year_ago',
    ExpressionAttributeNames={
        '#date': 'Date'
    },
    ExpressionAttributeValues={
        ':one_year_ago': one_year_ago_str
    }
)

# Process the results
old_records = response['Items']
```

```
# Continue scanning if we have more items (pagination)
while 'LastEvaluatedKey' in response:
    response = table.scan(
        FilterExpression='#date < :one_year_ago',
        ExpressionAttributeNames={
            '#date': 'Date'
        },
        ExpressionAttributeValues={
            ':one_year_ago': one_year_ago_str
        },
        ExclusiveStartKey=response['LastEvaluatedKey']
    )
    old_records.extend(response['Items'])

for record in old_records:
    print(json.dumps(record))

# The total number of old records should be zero.
print(f"Total number of old records: {len(old_records)}")
```

此测试脚本使用 AWS SDK for Python (Boto3) 创建与 DynamoDB 表的连接，并扫描超过一年的项目。为确认 Lambda 函数是否成功运行，在测试结束时，该函数会打印表中仍存在超过一年的记录数。如果 Lambda 函数成功运行，则表中的旧记录数应为零。

创建和填充示例 DynamoDB 表

要测试用于定期维护的应用程序，您需要先创建 DynamoDB 表，然后在其中填充部分示例数据。您可以使用 AWS Management Console 手动创建表，也可以使用 AWS SAM 创建表。我们建议您使用 AWS SAM，通过使用 AWS CLI 命令快速创建和配置表。

Console

创建 DynamoDB 表

1. 打开 DynamoDB 控制台中 Tables page（表页面）。
2. 选择创建表。
3. 通过执行以下操作创建表：
    a. 在表详细信息下，在表名称中输入 **`MyOrderTable`**。
    b. 对于分区键，输入 **`Order_number`**，并将数据类型设置为字符串。

c. 对于排序键，输入 **Date**，并将类型设置为字符串。

d. 将表设置设置为默认设置，然后选择创建表。

4. 当表创建完成且其状态显示为有效时，请执行以下操作来创建全局二级索引（GSI）。应用程序将使用此 GSI 直接按日期搜索项目，以确定要删除的内容。

a. 从表列表中选择 MyOrderTable。

b. 选择索引选项卡。

c. 在全局二级索引下，选择创建索引。

d. 在索引详细信息下，对于分区键，输入 **Date**，并将数据类型设置为字符串。

e. 对于 Index name (索引名称)，输入 **Date-index**。

f. 将所有其他参数设置为其默认值，滚动到页面底部，然后选择创建索引。

AWS SAM

创建 DynamoDB 表

1. 导航到您为 DynamoDB 表保存 `template.yaml` 文件的文件夹。请注意，此示例使用两个 `template.yaml` 文件。确保将它们保存在单独的子文件夹中，并且您位于包含模板的正确文件夹中，以便创建 DynamoDB 表。

2. 运行以下命令。

```
sam build
```

此命令会收集您想要部署资源的构建构件，并将其以适当的格式放置在适当的位置进行部署。

3. 要创建 `template.yaml` 文件中指定的 DynamoDB 资源，请运行以下命令。

```
sam deploy --guided
```

使用 `--guided` 标志意味着 AWS SAM 将向您显示提示，以指导您完成部署过程。对于此部署，输入 **cron-app-test-db** 的 `Stack name`，然后使用 Enter 接受所有其他选项的默认值。

当 AWS SAM 创建完 DynamoDB 资源后，您将看到以下消息。

```
Successfully created/updated stack - cron-app-test-db in us-west-2
```

4. 此外，您还可以通过打开 DynamoDB 控制台的表页面来确认 DynamoDB 表已创建。您应该会看到名为 `MyOrderTable` 的表。

创建表后，接下来要添加部分示例数据来测试应用程序。您之前下载的 CSV 文件 (`sample_data.csv`) 包含许多示例条目，包括订单号、日期以及客户和订单信息。使用提供的 python 脚本 (`load_sample_data.py`) 将此数据添加到表中。

将示例数据添加到表中

1. 导航到含有 `sample_data.csv` 和 `load_sample_data.py` 文件的目录。如果这些文件位于不同的目录中，请进行移动，将文件保存在相同的位置。
2. 通过运行以下命令，创建 Python 虚拟环境，以运行脚本。我们建议您使用虚拟环境，因为在接下来的步骤中，您需要安装 AWS SDK for Python (Boto3)。

   ```
   python -m venv venv
   ```

3. 通过运行以下命令激活虚拟环境。

   ```
   source venv/bin/activate
   ```

4. 通过运行以下命令，将 SDK for Python（Boto3）安装到虚拟环境中。该脚本使用此库连接到 DynamoDB 表并添加项目。

   ```
   pip install boto3
   ```

5. 通过运行以下命令，运行脚本以填充表。

   ```
   python load_sample_data.py
   ```

   如果脚本成功运行，则它应在加载项目和报告 `Data loading completed` 时将每个项目打印到控制台。

6. 通过运行以下命令停用虚拟环境。

   ```
   deactivate
   ```

7. 您可以执行以下操作，以验证数据是否已加载到 DynamoDB 表：

   a. 打开 DynamoDB 控制台中的浏览项目页面，然后选择表 (`MyOrderTable`)。

b. 在返回的项目窗格中，您应该会看到脚本添加到表中 CSV 文件中的 25 个项目。

创建用于定期维护的应用程序

您可以使用 AWS Management Console 或通过使用 AWS SAM 创建并部署此示例应用程序的资源。在生产环境中，建议使用 AWS SAM 等基础设施即代码（IaC）工具来重复部署无服务器应用程序，无需采用手动流程。

在本示例中，可按照控制台或说明了解如何单独配置每种 AWS 资源，或按照 AWS SAM 说明，使用 AWS CLI 命令快速部署应用程序。

Console

使用 AWS Management Console 创建函数

首先，创建包含基本起始代码的函数。然后，通过将此代码替换为自己的函数代码，方法是将代码直接复制并粘贴到 Lambda 代码编辑器中，或者将代码作为 `.zip` 包上传。对于此任务，建议您只需复制并粘贴代码即可。

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择 Create function (创建函数)。
3. 选择从头开始创作。
4. 在基本信息中，执行以下操作：
   a. 对于 Function name（函数名称），请输入 **`ScheduledDBMaintenance`**。
   b. 对于运行时，请选择最新的 Python 版本。
   c. 对于架构，选择 x86_64。
5. 选择 Create function (创建函数)。
6. 创建函数后，您可以使用提供的函数代码配置函数。
   a. 在代码源窗格中，将 Lambda 创建的 Hello world 代码替换为之前保存的 `lambda_function.py` 文件中的 Python 函数代码。
   b. 展开主侧栏中的部署部分，然后选择部署。

配置函数内存和超时（控制台）

1. 选择函数的配置选项卡。

2. 在常规配置窗格中，选择编辑。
3. 将内存设置为 256 MB，并将超时设置为 15 秒。如果您正在处理包含许多记录的大型表，例如在生产环境中，则可以考虑将超时设置为更大数值。此举会让函数有更多时间扫描和清理数据库。
4. 选择保存。

配置日志格式（控制台）

您可以将 Lambda 函数配置为以非结构化文本或 JSON 格式输出日志。我们建议将 JSON 格式用于日志，以简化搜索和筛选日志数据的流程。要了解有关 Lambda 日志配置选项的更多信息，请参阅the section called “配置函数日志”。

1. 选择函数的配置选项卡。
2. 选择监控和操作工具。
3. 在日志记录配置窗格中，选择编辑。
4. 对于日志记录配置，选择 JSON。
5. 选择保存。

要设置 IAM 权限

要向函数授予读取和删除 DynamoDB 项目所需的权限，您需要向函数的执行角色添加来定义必要权限的策略。

1. 打开配置选项卡，然后从左侧导航栏中选择权限。
2. 在执行角色下，选择角色名称。
3. 在 IAM 控制台中，选择添加权限，然后选择创建内联策略。
4. 使用 JSON 编辑器并输入以下策略：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:Scan",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
```

```
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/MyOrderTable"
        }
    ]
}
```

5. 将策略命名为 **DynamoDBCleanupPolicy**，然后创建策略。

将 EventBridge 调度器设置为触发器（控制台）

1. 打开 EventBridge 控制台。
2. 在左侧导航窗格中，选择调度器部分下的调度器。
3. 选择创建计划。
4. 通过执行以下操作配置计划：

   a. 在计划名称下，输入计划的名称（例如：**DynamoDBCleanupSchedule**）。

   b. 在计划模式下，选择定期计划。

   c. 对于计划类型，将默认值保留为基于 Cron 的计划，然后输入以下计划详细信息：

      - 分钟：**0**
      - 小时：**3**
      - 日：**1**
      - 月：*****
      - 星期：**?**
      - 年：*****

      评估后，此 cron 表达式会在每月第 1 天上午 03:00 运行。

   d. 对于灵活时间窗口下，选择关闭。
5. 选择下一步。
6. 通过执行以下操作，为 Lambda 函数配置触发器：

   a. 在目标详细信息窗格中，将目标 API 设置为模板化目标，然后选择 AWS Lambda 调用。

   b. 在调用下，从下拉列表中选择 Lambda 函数 (ScheduledDBMaintenance)。

   c. 将有效载荷留空，然后选择下一步。

d. 向下滚动到权限，然后选择为此计划创建新角色。使用控制台创建新 EventBridge 调度器计划时，EventBridge 调度器会创建新策略，该策略具有调用函数所需的必要权限。有关管理计划权限的更多信息，请参阅《EventBridge Scheduler User Guide》中的 Cron-based schedules。

e. 选择下一步。

7. 查看设置，然后选择创建计划以完成计划和 Lambda 触发器的创建。

AWS SAM

要部署应用程序，请使用 AWS SAM

1. 导航到您为应用程序保存 `template.yaml` 文件的文件夹。请注意，此示例使用两个 `template.yaml` 文件。确保将它们保存在单独的子文件夹中，并且您位于包含模板的正确文件夹中，以便创建应用程序。
2. 将之前下载的 `lambda_function.py` 和 `requirements.txt` 文件复制到同一个文件夹。AWS SAM 模板中指定的代码位置是 `./`，即当前位置。尝试部署应用程序时，AWS SAM 将在此文件夹中搜索 Lambda 函数代码。
3. 运行以下命令。

```
sam build --use-container
```

此命令会收集您想要部署资源的构建构件，并将其以适当的格式放置在适当的位置进行部署。指定 `--use-container` 选项会在类似 Lambda 的 Docker 容器中构建函数。我们在这里使用它，您无需在本地计算机上安装 Python 3.12 即可进行构建。

4. 要创建 `template.yaml` 文件中指定的 Lambda 和 EventBridge 调度器资源，请运行以下命令。

```
sam deploy --guided
```

使用 `--guided` 标志意味着 AWS SAM 将向您显示提示，以指导您完成部署过程。对于此部署，输入 **`cron-maintenance-app`** 的 `Stack name`，然后使用 Enter 接受所有其他选项的默认值。

当 AWS SAM 创建完 Lambda 和 EventBridge 调度器资源后，您将看到以下消息。

```
Successfully created/updated stack - cron-maintenance-app in us-west-2
```

5. 此外，您还可以通过打开 Lambda 控制台的函数页面，来确认系统已创建 Lambda 函数。您应该会看到名为 ScheduledDBMaintenance 的函数。

测试应用程序

要测试计划是否正确触发了函数，以及函数是否正确清理了数据库中的记录，您可以对计划进行临时修改，设置为在特定时间运行一次。然后，您可以再次运行 sam deploy，以将定期计划重置为每月运行一次。

使用 AWS Management Console 运行应用程序。

1. 导航回到 EventBridge 调度器控制台页面。
2. 选择计划，然后选择编辑。
3. 在计划模式部分的定期下，选择一次性计划。
4. 将调用时间设置为距现在起几分钟后，查看设置，然后选择保存。

在计划运行并调用其目标之后，您可以运行 test_app.py 脚本来验证函数是否从 DynamoDB 表中成功删除了所有旧记录。

使用 Python 脚本验证是否已删除旧记录

1. 在命令行窗口中，导航到保存 test_app.py 的文件夹。
2. 运行 脚本。

```
python test_app.py
```

如果成功删除，您将会看到以下输出。

```
Total number of old records: 0
```

后续步骤

现在，您可以修改 EventBridge 调度器以满足特定应用程序要求。EventBridge 调度器支持以下计划表达式：cron、速率和一次性计划。

有关 EventBridge 调度器计划表达式的更多信息，请参阅《EventBridge Scheduler User Guide》中的 Schedule types。

了解关键的 Lambda 概念

Lambda 是一种事件驱动的计算服务，它运行函数实例来处理事件。您可以使用 Lambda API 来直接调用函数，也可以配置 AWS 服务或资源来调用函数。

以下各节介绍了 Lambda 函数的一些关键概念、事件驱动的编程模型和函数运行的环境。

概念

- 函数
- 触发器
- 事件
- 执行环境
- 部署程序包
- 运行时
- 层
- 并发
- Qualifier
- 目标位置

函数

函数是一种资源，您可以对其调用以在 Lambda 中运行您的代码。函数所具有的代码可以处理您传递给函数或其他 AWS 服务发送给函数的事件。

触发器

触发器是调用 Lambda 函数的资源或配置。触发器包括可配置为调用函数的AWS服务以及事件源映射。事件源映射是 Lambda 中的一种资源，它从流或队列中读取项目并调用函数。有关更多信息，请参阅了解 Lambda 函数调用方法 和使用来自其 AWS 他服务的事件调用 Lambda。

事件

事件是 JSON 格式的文档，其中包含要处理的 Lambda 函数的数据。运行时将事件转换为一个对象，并将该对象传递给函数代码。在调用函数时，可以确定事件的结构和内容。

Example 自定义事件 – 天气数据

```
{
  "TemperatureK": 281,
  "WindKmh": -3,
  "HumidityPct": 0.55,
  "PressureHPa": 1020
}
```

当 AWS 服务调用您的函数时，该服务会定义事件的形状。

Example 服务事件 – Amazon SNS 通知

```
{
  "Records": [
    {
      "Sns": {
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        ...
```

有关 AWS 服务中的事件的更多信息，请参阅 使用来自其 AWS 他服务的事件调用 Lambda。

执行环境

执行环境为您的 Lambda 函数提供一个安全和隔离的运行时环境。执行环境管理运行函数所需的进程和其他资源。执行环境为函数以及与函数关联的任何扩展提供生命周期支持。

有关更多信息，请参阅 了解 Lambda 执行环境生命周期。

部署程序包

您可以使用部署程序包来部署 Lambda 函数代码。Lambda 支持两种类型的部署程序包：

- 包含函数代码及其依赖项的 .zip 文件存档。Lambda 为函数提供操作系统和运行时。有关更多信息，请参阅 将 Lambda 函数部署为 .zip 文件归档。
- 与 Open Container Initiative (OCI) 规范兼容的容器映像。将函数代码和依赖项添加到映像中。还必须包含操作系统和 Lambda 运行时。有关更多信息，请参阅 使用容器映像创建 Lambda 函数。

运行时

运行时提供在执行环境中运行的语言特定环境。运行时在 Lambda 与函数之间中继调用事件、上下文信息和响应。您可以使用 Lambda 提供的运行时，或构建您自己的运行时。如果要将代码打包为 .zip 文件存档，则必须将您的函数配置为使用与编程语言匹配的运行时。对于容器映像，映像构建时会包括运行时。

有关更多信息，请参阅 Lambda 运行时。

层

Lambda 层是可以包含其他代码或其他内容的 .zip 文件归档。层可以包含库、自定义运行时、数据或配置文件。

层提供了一种方便的方法来打包库和其他可与 Lambda 函数搭配使用的依赖项。使用层可以缩小上传的部署存档的大小，并加快代码的部署速度。层还可促进代码共享和责任分离，以便您可以更快地迭代编写业务逻辑。

每个函数最多可以包含五个层。层计入标准 Lambda 部署大小配额。在函数中包含图层时，内容将提取到执行环境中的 `/opt` 目录中。

默认情况下，您创建的层是 AWS 账户私有的。您可以选择与其他账户共享层，也可以选择将层设为公有。如果您的函数使用其他账户发布的层，则函数可以在删除层版本后或在撤消访问该层的权限后继续使用该层版本。但是，您无法使用已删除的层版本创建新函数或更新函数。

作为容器映像部署的函数不使用层。相反，在构建映像时，您可以将首选运行时、库和其他依赖项打包到容器映像。

有关更多信息，请参阅 Lambda 层。

并发

并发性 是您的函数在任何给定时间所服务的请求的数目。在调用函数时，Lambda 会预配置其实例以处理事件。当函数代码完成运行时，它会处理另一个请求。如果当仍在处理请求时再次调用函数，则预配置另一个实例，从而增加该函数的并发性。

并发性受 AWS 区域级别配额的约束。您还可以配置各个函数来限制其并发性，或使得它们达到特定级别的并发性。有关更多信息，请参阅 为函数配置预留并发。

Qualifier

当您调用或查看某个函数时，可以包含限定符来指定版本或别名。版本 是具有数字限定符的函数代码和配置的不可变快照。例如：`my-function:1`。别名 是版本的指针，您可以更新该版本以映射到其他版本，或者在两个版本之间拆分流量。例如：`my-function:BLUE`。可以将版本和别名一起使用，为客户端提供用于调用您的函数的稳定接口。

有关更多信息，请参阅 管理 Lambda 函数版本。

目标位置

目标是 Lambda 可以从异步调用发送事件的一种 AWS 资源。您可以为处理失败的事件配置目标。某些服务还支持为处理成功的事件配置目标。

有关更多信息，请参阅添加目标。

将 Lambda 与基础设施即代码（IaC）结合使用

Lambda 函数很少单独运行。相反，它们通常与数据库、队列和存储等其他资源一起都是无服务器应用程序的组成部分。借助基础设施即代码（IaC），可以自动执行部署流程，从而快速、可重复地部署和更新整个无服务器应用程序，涵盖许多独立的 AWS 资源。这种方法可以加快开发周期，简化配置管理，并确保每次都以相同的方式部署资源。

适用于 Lambda 的 IaC 工具

AWS CloudFormation

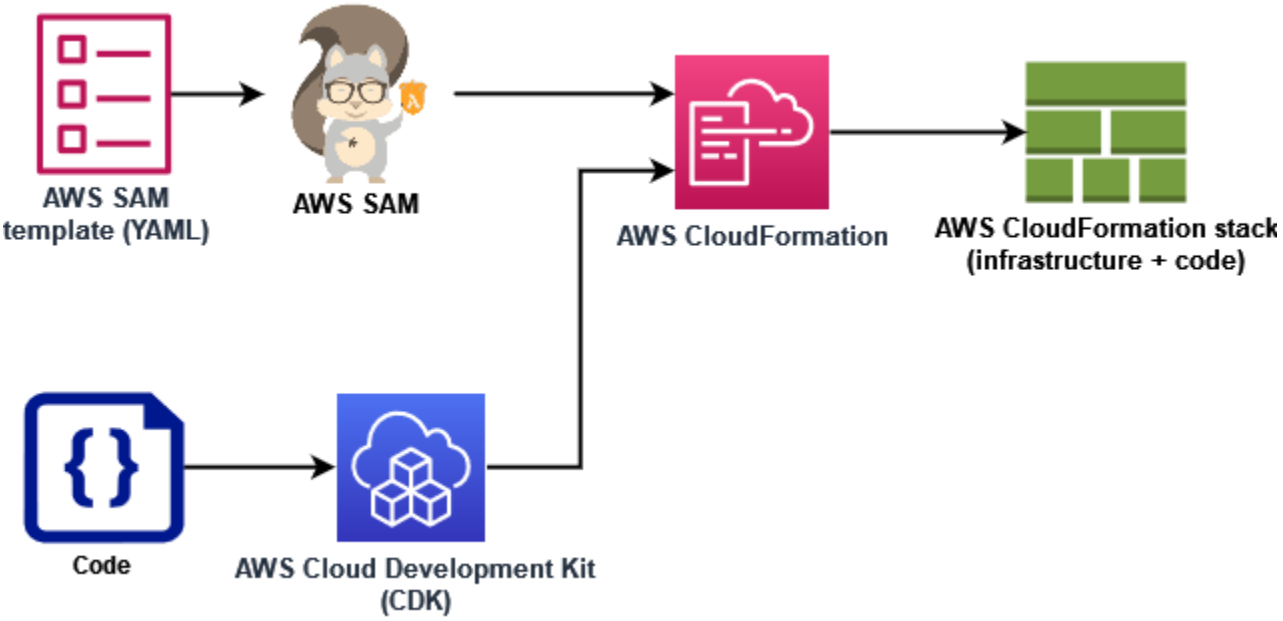
CloudFormation 是来自 AWS 的基础 IaC 服务。可以使用 YAML 或 JSON 模板，对整个 AWS 基础设施（包括 Lambda 函数）进行建模和预置。CloudFormation 可以处理创建、更新和删除 AWS 资源的复杂工作。

AWS Serverless Application Model (AWS SAM)

AWS SAM 是一个开源框架，构建在 CloudFormation 之上。它提供了用于定义无服务器应用程序的简化语法。使用 AWS SAM 模板，只需几行 YAML 即可快速预置 Lambda 函数、API、数据库和事件源。

AWS Cloud Development Kit (AWS CDK)

CDK 是 IaC 的代码优先方法。可以使用 TypeScript、JavaScript、Python、Java、C#/Net 或 Go 来定义基于 Lambda 的架构。选择首选语言，并使用参数、条件、循环、组合和继承等编程元素来定义基础设施的预期结果。然后，CDK 会生成底层 CloudFormation 模板用于部署。有关如何配合使用 Lambda 和 CDK 的示例，请参阅 使用 AWS CDK 部署 Lambda 函数。



AWS 还提供了一项名为 AWS 基础设施编辑器 的服务，可使用简单图形界面开发 IaC 模板。凭借基础设施编辑器，可以通过在可视画布中拖动、分组和连接 AWS 服务来设计应用程序架构。然后，基础设施编辑器会根据您的设计创建 AWS SAM 模板或 AWS CloudFormation 模板，供您用于部署应用程序。

在以下 the section called “IaC for Lambda 入门” 部分中，您将使用基础设施编辑器，根据现有 Lambda 函数为无服务器应用程序开发模板。

IaC for Lambda 入门

在本教程中，您可以通过使用现有 Lambda 函数创建 AWS SAM 模板，然后通过添加其他 AWS 资源在基础设施编辑器中构建无服务器应用程序，从而开始将 IaC 与 Lambda 结合使用。

在学习本教程时，您将学习一些基本概念，例如如何在 AWS SAM 中指定 AWS 资源。您还将了解如何使用基础设施编辑器来构建可使用 AWS SAM 或 AWS CloudFormation 部署的无服务器应用程序。

要完成本教程，请执行以下步骤：

- 创建示例 Lambda 函数
- 使用 Lambda 控制台查看函数的 AWS SAM 模板
- 将函数的配置导出到 AWS 基础设施编辑器 并根据函数的配置设计一个简单的无服务器应用程序
- 保存更新后的 AWS SAM 模板，可作为部署无服务器应用程序的基础

先决条件

在本教程中，您将使用基础设施编辑器的本地同步功能，将模板和代码文件保存到本地生成计算机。要使用此功能，您需要一个支持文件系统访问 API 的浏览器，它允许 Web 应用程序在本地文件系统中读取、写入和保存文件。我们建议使用 Google Chrome 或 Microsoft Edge。有关文件系统访问 API 的更多信息，请参阅 What is the File System Access API?

创建 Lambda 函数

在第一步中，您会创建 Lambda 函数以用于完成本教程的其余部分。为了简单起见，您可以使用 Lambda 控制台通过 Python 3.11 运行时系统创建基本的“Hello world”函数。

使用控制台创建“Hello world”Lambda 函数

1. 打开 Lambda 控制台。
2. 选择 Create function（创建函数）。
3. 选择从头开始创作，然后在基本信息中输入 **LambdaIaCDemo** 作为函数名称。
4. 对于运行时系统，选择 Python 3.11。
5. 选择 Create function (创建函数)。

查看函数的 AWS SAM 模板

在将函数配置导出到基础设施编辑器之前，请使用 Lambda 控制台将函数的当前配置作为 AWS SAM 模板查看。按照本节中的步骤操作，您将了解 AWS SAM 模板的剖析以及如何定义诸如 Lambda 函数之类的资源以开始指定无服务器应用程序。

查看函数的 AWS SAM 模板

1. 打开 Lamba 控制台的函数页面。
2. 选择您刚刚创建的函数（`LambdaIaCDemo`）。
3. 在函数概述窗格中，选择模板。

   您将看到的不是表示函数配置的图表，而是函数的 AWS SAM 模板。该模板应该如下所示。

```
# This AWS SAM template has been generated from your function's
# configuration. If your function has one or more triggers, note
# that the AWS resources associated with these triggers aren't fully
# specified in this template and include placeholder values.Open this template
```

```
# in AWS Application Composer or your favorite IDE and modify
# it to specify a serverless application with other AWS resources.
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        Statement:
          - Effect: Allow
            Action:
              - logs:CreateLogGroup
            Resource: arn:aws:logs:us-east-1:123456789012:*
          - Effect: Allow
            Action:
              - logs:CreateLogStream
              - logs:PutLogEvents
            Resource:
              - >-
                arn:aws:logs:us-east-1:123456789012:log-group:/aws/lambda/
LambdaIaCDemo:*
```

我们需要花些时间查看函数的 YAML 模板并理解一些关键概念。

模板以声明 `Transform: AWS::Serverless-2016-10-31` 开头。此声明是必需的，因为 AWS SAM 模板在幕后要通过 AWS CloudFormation 部署。使用 `Transform` 语句将模板标识为 AWS SAM 模板文件。

`Transform` 声明之后是 `Resources` 部分。在这里可以定义要使用 AWS SAM 模板部署的 AWS 资源。AWS SAM 模板可以包含 AWS SAM 资源和 AWS CloudFormation 资源的组合。这是因为在部署过程中，AWS SAM 模板会扩展为 AWS CloudFormation 模板，因此任何有效的 AWS CloudFormation 语法都可以添加到 AWS SAM 模板中。

目前，模板的 `Resources` 部分中只定义了一个资源，即您的 Lambda 函数 `LambdaIaCDemo`。要向 AWS SAM 模板添加 Lambda 函数，请使用 `AWS::Serverless::Function` 资源类型。Lambda 函数资源的 `Properties` 定义函数的运行时系统、函数处理程序和其他配置选项。此处还定义了 AWS SAM 应该用于部署函数的函数源代码的路径。要了解有关 AWS SAM 中Lambda 函数资源的更多信息，请参阅《AWS SAM 开发人员指南》中的 AWS::Serverless::Function。

除了函数属性和配置外，该模板还指定了函数的 AWS Identity and Access Management（IAM）policy。此策略授予函数向 Amazon CloudWatch Logs 写入日志的权限。当您在 Lambda 控制台中创建函数时，Lambda 会自动将此策略附加到您的函数。要了解有关为 AWS SAM 模板中的函数指定 IAM policy 的更多信息，请参阅 AWS SAM 开发者指南中 AWS::Serverless::Function 页面上的 `policies` 属性。

要了解有关 AWS SAM 模板结构的更多信息，请参阅AWS SAM 模板剖析。

使用 AWS 基础设施编辑器 设计无服务器应用程序

要从函数的 AWS SAM 模板开始构建简单的无服务器应用程序，您可以将函数配置导出到基础设施编辑器，并激活基础设施编辑器的本地同步模式。本地同步会自动将您的函数代码和 AWS SAM 模板保存到本地生成计算机，并在您在基础设施编辑器中添加其他 AWS 资源时让保存的模板保持同步。

将函数导出到基础设施编辑器

1. 在函数概述窗格中，选择导出到应用程序编辑器。

   为了将函数的配置和代码导出到基础设施编辑器，Lambda 在您的账户中创建了一个 Amazon S3 存储桶来临时存储此数据。
2. 在对话框中，选择确认并创建项目以接受此存储桶的默认名称，并将函数的配置和代码导出到基础设施编辑器。
3. （可选）要为 Lambda 创建的 Amazon S3 存储桶选择其他名称，请输入新名称并选择确认并创建项目。Amazon S3 存储桶的名称必须全局唯一，并遵守存储桶命名规则。

选择确认并创建项目，将打开基础设施编辑器控制台。在画布上，您将看到 Lambda 函数。

4. 从菜单下拉列表中，选择激活本地同步。
5. 在打开的对话框中选择选择文件夹，然后在本地生成计算机上选择一个文件夹。
6. 选择激活以激活本地同步。

要将函数导出到基础设施编辑器，您需要拥有使用特定 API 操作的权限。如果您无法导出函数，请查看 the section called “所需的权限” 并确保您拥有所需的权限。

> **Note**
>
> 标准 Amazon S3 定价适用于您将函数导出到基础设施编辑器时由 Lambda 创建的存储桶。由 Lambda 放入存储桶的对象会在 10 天后自动删除，但是 Lambda 不会删除存储桶本身。为避免给您的 AWS 账户 增加额外费用，请在将函数导出到基础设施编辑器后，按照删除存储桶中的说明进行操作。有关 Lambda 创建的 Amazon S3 存储桶的更多信息，请参阅 the section called “基础设施编辑器”。

在基础设施编辑器中设计您的无服务器应用程序

激活本地同步后，您在基础设施编辑器中所做的更改将反映在本地生成计算机上保存的 AWS SAM 模板中。现在，您可以将其他 AWS 资源拖放到基础设施编辑器画布上来构建应用程序。在此示例中，您添加了一个 Amazon SQS 简单队列作为 Lambda 函数的触发器，并添加了一个 DynamoDB 表供该函数写入数据。

1. 通过执行以下操作，将 Amazon SQS 触发器添加到 Lambda 函数：
   a. 在资源选项板的搜索字段中输入 **SQS**。
   b. 将 SQS 队列资源拖到画布上，然后将其放置在 Lambda 函数的左侧。
   c. 选择详细信息，然后在逻辑 ID 中输入 **LambdaIaCQueue**。
   d. 选择保存。
   e. 单击 SQS 队列卡上的订阅端口，然后将其拖到 Lambda 函数卡上的左侧端口，即可连接您的 Amazon SQS 和 Lambda 资源。两个资源之间出现一条线，表示连接成功。基础设施编辑器还会在画布底部显示一条消息，表明两个资源已连接成功。
2. 通过执行以下操作，为您的 Lambda 函数添加一个 Amazon DynamoDB 表，供其写入数据：
   a. 在资源选项板的搜索字段中输入 **DynamoDB**。

b. 将 DynamoDB 表资源拖到画布上，然后将其放置在 Lambda 函数的右侧。

c. 选择详细信息，然后在逻辑 ID 中输入 **LambdaIaCTable**。

d. 选择保存。

e. 单击 Lambda 函数卡的右侧端口，然后将其拖动到 DynamoDB 卡的左侧端口，即可将 DynamoDB 表连接到您的 Lambda 函数。

现在，您已经添加了这些额外资源，让我们看看基础设施编辑器创建的更新后的 AWS SAM 模板。

查看更新后的 AWS SAM 模板

- 在基础设施编辑器画布上，选择模板以从画布视图切换到模板视图。

现在，您的 AWS SAM 模板应包含以下额外资源和属性：

- 具有标识符 `LambdaIaCQueue` 的 Amazon SQS 队列

```
LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
```

使用基础设施编辑器添加 Amazon SQS 队列时，基础设施编辑器会设置 `MessageRetentionPeriod` 属性。您也可以通过在 SQS 队列卡上选择详细信息并选中或取消选中 Fifo 队列来设置 `FifoQueue` 属性。

要为队列设置其他属性，您可以手动编辑模板以添加这些属性。要了解有关 `AWS::SQS::Queue` 资源及其可用属性的更多信息，请参阅《AWS CloudFormation 用户指南》中的 AWS:: SQS::Queue。

- Lambda 函数定义中的一个 `Events` 属性，用于将 Amazon SQS 队列指定为函数的触发器

```
Events:
  LambdaIaCQueue:
    Type: SQS
    Properties:
      Queue: !GetAtt LambdaIaCQueue.Arn
      BatchSize: 1
```

Events 属性由一个事件类型和一组取决于类型的属性组成。要了解您可以配置用于触发 Lambda 函数的不同 AWS 服务 以及可以设置的属性，请参阅 AWS SAM 开发者指南中的 EventSource。

- 带有标识符 LambdaIaCTable 的 DynamoDB 表

```
LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

使用基础设施编辑器添加 DynamoDB 表时，可以通过在 DynamoDB 表卡上选择详细信息并编辑键值来设置表的键。基础设施编辑器还为许多其他属性设置默认值，包括 BillingMode 和 StreamViewType。

要详细了解这些属性以及可以添加到 AWS SAM 模板中的其他属性，请参阅《AWS CloudFormation 用户指南》中的 AWS::DynamoDB::Table。

- 一项新的 IAM policy，允许您的函数对您添加的 DynamoDB 表执行 CRUD 操作。

```
Policies:
...
  - DynamoDBCrudPolicy:
    TableName: !Ref LambdaIaCTable
```

最终的 AWS SAM 模板应该如下所示。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
```

```
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - logs:CreateLogGroup
              Resource: arn:aws:logs:us-east-1:594035263019:*
            - Effect: Allow
              Action:
                - logs:CreateLogStream
                - logs:PutLogEvents
              Resource:
                - arn:aws:logs:us-east-1:594035263019:log-group:/aws/lambda/
LambdaIaCDemo:*
        - DynamoDBCrudPolicy:
            TableName: !Ref LambdaIaCTable
      Events:
        LambdaIaCQueue:
          Type: SQS
          Properties:
            Queue: !GetAtt LambdaIaCQueue.Arn
            BatchSize: 1
      Environment:
        Variables:
          LAMBDAIACTABLE_TABLE_NAME: !Ref LambdaIaCTable
          LAMBDAIACTABLE_TABLE_ARN: !GetAtt LambdaIaCTable.Arn
  LambdaIaCQueue:
```

```
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
  LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

使用 AWS SAM（可选）部署您的无服务器应用程序

如果要使用 AWS SAM 来利用刚刚在基础设施编辑器中创建的模板部署无服务器应用程序，则需要先安装 AWS SAM CLI。为此，请遵循安装 AWS SAM CLI 中的说明。

在部署应用程序之前，您还需要更新基础设施编辑器随模板一起保存的函数代码。目前，基础设施编辑器保存的 `lambda_function.py` 文件仅包含您在创建函数时 Lambda 提供的基本“Hello world”代码。

要更新函数代码，请复制以下代码并将其粘贴到由基础设施编辑器保存到本地生成计算机的 `lambda_function.py` 文件中。激活“本地同步”模式时，您指定了基础设施编辑器要将此文件保存到的目录。

此代码接受的键值对来自您在基础设施编辑器中创建的 Amazon SQS 队列的消息。如果键和值都是字符串，则代码会使用它们向您的模板中定义的 DynamoDB 表写入项目。

更新了 Python 函数代码

```
import boto3
import os
import json

# define the DynamoDB table that Lambda will connect to
tablename = os.environ['LAMBDAIACTABLE_TABLE_NAME']

# create the DynamoDB resource
```

```
dynamo = boto3.client('dynamodb')

def lambda_handler(event, context):
    # get the message out of the SQS event
    message = event['Records'][0]['body']
    data = json.loads(message)
    # write event data to DDB table
    if check_message_format(data):
        key = next(iter(data))
        value = data[key]
        dynamo.put_item(
            TableName=tablename,
            Item={
                'id': {'S': key},
                'Value': {'S': value}
            }
        )
    else:
        raise ValueError("Input data not in the correct format")

# check that the event object contains a single key value
# pair that can be written to the database
def check_message_format(message):
    if len(message) != 1:
        return False

    key, value = next(iter(message.items()))

    if not (isinstance(key, str) and isinstance(value, str)):
        return False

    else:
        return True
```

部署无服务器应用程序

要使用 AWS SAM CLI 部署应用程序，请执行以下步骤。为了正确构建和部署函数，必须在您的生成计算机和 PATH 上安装 Python 3.11 版本。

1. 在基础设施编辑器保存 `template.yaml` 和 `lambda_function.py` 文件的目录运行以下命令。

```
sam build
```

此命令会收集应用程序的构建构件，并将其以适当的格式放置在适当的位置进行部署。

2. 要部署您的应用程序并创建 AWS SAM 模板中指定的 Lambda、Amazon SQS 和 DynamoDB 资源，请运行以下命令。

```
sam deploy --guided
```

使用 `--guided` 标志意味着 AWS SAM 将向您显示提示，以指导您完成部署过程。对于此部署，请按 Enter 接受默认选项。

在部署过程中，AWS SAM 将在 AWS 账户 中创建以下资源：

- 一个名为 `sam-app` 的 AWS CloudFormation 堆栈
- 名称格式为 `sam-app-LambdaIaCDemo-`*`99VXPpYQVv1M`* 的 Lambda 函数
- 名称格式为 `sam-app-LambdaIaCQueue-`*`xL87VeKsGiIo`* 的 Amazon SQS 队列
- 名称格式为 `sam-app-LambdaIaCTable-`*`CN0S66C0VLNV`* 的 DynamoDB 表

AWS SAM 还会创建必要的 IAM 角色和策略，以便您的 Lambda 函数可以读取来自 Amazon SQS 队列的消息并对 DynamoDB 表执行 CRUD 操作。

测试已部署的应用程序（可选）

要确认您的无服务器应用程序已正确部署，请向您的 Amazon SQS 队列发送一条包含密钥值对的消息，并检查 Lambda 是否使用这些值将项目写入您的 DynamoDB 表。

测试您的无服务器应用程序

1. 打开 Amazon SQS 控制台的队列页面，然后选择 AWS SAM 通过您的模板创建的队列。名称的格式为 `sam-app-LambdaIaCQueue-`*`xL87VeKsGiIo`*。
2. 选择发送和接收消息，然后将以下 JSON 粘贴到发送消息部分的消息正文中。

```
{
    "myKey": "myValue"
}
```

3. 选择 Send message（发送消息）。

将您的消息发送到队列会导致 Lambda 通过 AWS SAM 模板中定义的事件源映射调用您的函数。要确认 Lambda 已按预期调用您的函数，请确认项目已被添加到您的 DynamoDB 表。

4. 打开 DynamoDB 控制台中的表页面，然后选择您的表。名称的格式为 `sam-app-LambdaIaCTable-CN0S66C0VLNV`。
5. 选择 Explore table items（浏览表项目）。在返回的项目窗格中，您会看到一个带 id `myKey` 和值 `myValue` 的项目。

使用 AWS CDK 部署 Lambda 函数

AWS Cloud Development Kit (AWS CDK) 是一个基础设施即代码（IaC）框架，可以让您使用所选编程语言来定义 AWS 云基础设施。要定义您自己的云基础设施，请首先编写一个包含一个或多个堆栈的应用程序（使用 CDK 支持的一种语言）。然后，将其合成为 AWS CloudFormation 模板并将您的资源部署到 AWS 账户。按照本主题中的步骤部署可从 Amazon API Gateway 端点返回事件的 Lambda 函数。

CDK 中包含的 AWS 构造库提供可用于对 AWS 服务 提供的资源进行建模的模块。对于常用的服务，该库提供具有智能默认值和最佳实践的精选构造。只需几行代码，就可以使用 aws_lambda 模块来定义您的函数和支持资源。

先决条件

在开始本教程之前，通过运行以下命令安装 AWS CDK。

```
npm install -g aws-cdk
```

第 1 步：设置您的 AWS CDK 项目

为您的新 AWS CDK 应用程序创建目录并初始化项目。

JavaScript

```
mkdir hello-lambda
cd hello-lambda
cdk init --language javascript
```

TypeScript

```
mkdir hello-lambda
```

```
cd hello-lambda
cdk init --language typescript
```

Python

```
mkdir hello-lambda
cd hello-lambda
cdk init --language python
```

项目启动后，请激活项目的虚拟环境并安装 AWS CDK 的基线依赖关系。

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
mkdir hello-lambda
cd hello-lambda
cdk init --language java
```

将此 Maven 项目导入到 Java 集成式开发环境（IDE）中。例如，在 Eclipse 中，依次选择文件、导入、Maven、现有的 Maven 项目。

C#

```
mkdir hello-lambda
cd hello-lambda
cdk init --language csharp
```

> **Note**
>
> AWS CDK 应用程序模板使用项目目录的名称来生成源文件和类的名称。在此示例中，该目录名为 `hello-lambda`。如果您使用其他项目目录名称，则您的应用将与这些说明不匹配。

AWS CDK v2 在名为 `aws-cdk-lib` 的单个程序包中包含适用于所有 AWS 服务 的稳定构造。当您初始化该项目时，此程序包作为依赖项进行安装。使用某些编程语言时，会在您首次构建项目时安装该程序包。

步骤 2：定义 AWS CDK 堆栈

CDK 堆栈是一个或多个构造的集合，用于定义 AWS 资源。每个 CDK 堆栈代表您的 CDK 应用程序中的一个 AWS CloudFormation 堆栈。

要定义您的 CDK 堆栈，请按照首选编程语言的说明操作。此堆栈定义以下内容：

- 函数的逻辑名称：`MyFunction`
- 在 `code` 属性中指定的函数代码的位置。有关更多信息，请参阅《AWS Cloud Development Kit (AWS CDK) API Reference》中的 Handler code。
- REST API 的逻辑名称：`HelloApi`
- API Gateway 端点的逻辑名称：`ApiGwEndpoint`

请注意，本教程中的所有 CDK 堆栈都将 Node.js 运行时用于 Lambda 函数。可以为 CDK 堆栈和 Lambda 函数使用不同的编程语言，以利用每种语言的优势。例如，可以将 TypeScript 用于 CDK 堆栈，以利用静态输入为基础设施代码带来的好处。可以将 JavaScript 用于 Lambda 函数，以利用动态输入语言的灵活性和快速开发优势。

JavaScript

打开 `lib/hello-lambda-stack.js` 文件并将相应内容替换为以下内容。

```
const { Stack } = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const apigw = require('aws-cdk-lib/aws-apigateway');

class HelloLambdaStack extends Stack {
  /**
   *
   * @param {Construct} scope
   * @param {string} id
   * @param {StackProps=} props
   */
  constructor(scope, id, props) {
    super(scope, id, props);
    const fn = new lambda.Function(this, 'MyFunction', {
      code: lambda.Code.asset('lib/lambda-handler'),
      runtime: lambda.Runtime.NODEJS_LATEST,
      handler: 'index.handler'
    });
```

```
    const endpoint = new apigw.LambdaRestApi(this, 'MyEndpoint', {
      handler: fn,
      restApiName: "HelloApi"
    });

  }
}

module.exports = { HelloLambdaStack }
```

TypeScript

打开 lib/hello-lambda-stack.ts 文件并将相应内容替换为以下内容。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as apigw from "aws-cdk-lib/aws-apigateway";
import * as lambda from "aws-cdk-lib/aws-lambda";
import * as path from 'node:path';

export class HelloLambdaStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps){
    super(scope, id, props)
    const fn = new lambda.Function(this, 'MyFunction', {
      runtime: lambda.Runtime.NODEJS_LATEST,
      handler: 'index.handler',
      code: lambda.Code.fromAsset(path.join(__dirname, 'lambda-handler')),
    });

    const endpoint = new apigw.LambdaRestApi(this, `ApiGwEndpoint`, {
      handler: fn,
      restApiName: `HelloApi`,
    });

  }
}
```

Python

打开 /hello-lambda/hello_lambda/hello_lambda_stack.py 文件并将相应内容替换为以下内容。

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigw,
    aws_lambda as _lambda
)
from constructs import Construct

class HelloLambdaStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        fn = _lambda.Function(
            self,
            "MyFunction",
            runtime=_lambda.Runtime.NODEJS_LATEST,
            handler="index.handler",
            code=_lambda.Code.from_asset("lib/lambda-handler")
        )

        endpoint = apigw.LambdaRestApi(
            self,
            "ApiGwEndpoint",
            handler=fn,
            rest_api_name="HelloApi"
        )
```

Java

打开 /hello-lambda/src/main/java/com/myorg/HelloLambdaStack.java 文件并将相应内容替换为以下内容。

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.apigateway.LambdaRestApi;
import software.amazon.awscdk.services.lambda.Function;

public class HelloLambdaStack extends Stack {
    public HelloLambdaStack(final Construct scope, final String id) {
        this(scope, id, null);
```

```
    }

    public HelloLambdaStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Function hello = Function.Builder.create(this, "MyFunction")

 .runtime(software.amazon.awscdk.services.lambda.Runtime.NODEJS_LATEST)

 .code(software.amazon.awscdk.services.lambda.Code.fromAsset("lib/lambda-handler"))
                            .handler("index.handler")
                            .build();

        LambdaRestApi api = LambdaRestApi.Builder.create(this, "ApiGwEndpoint")
                                .restApiName("HelloApi")
                                .handler(hello)
                                .build();

    }
}
```

C#

打开 src/HelloLambda/HelloLambdaStack.cs 文件并将相应内容替换为以下内容。

```
using Amazon.CDK;
using Amazon.CDK.AWS.APIGateway;
using Amazon.CDK.AWS.Lambda;
using Constructs;

namespace HelloLambda
{
    public class HelloLambdaStack : Stack
    {
        internal HelloLambdaStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var fn = new Function(this, "MyFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_LATEST,
                Code = Code.FromAsset("lib/lambda-handler"),
                Handler = "index.handler"
            });
```

```
            var api = new LambdaRestApi(this, "ApiGwEndpoint", new
 LambdaRestApiProps
            {
                Handler = fn
            });
        }
    }
}
```

步骤 3：创建 Lambda 函数代码

1. 从项目的根目录 (hello-lambda)，创建 Lambda 函数代码的 /lib/lambda-handler 目录。此目录在 AWS CDK 堆栈的 code 属性中指定。
2. 在 /lib/lambda-handler 目录中，创建名为 index.js 的新文件。将以下代码粘贴到该文件中。该函数从 API 请求中提取特定属性，并将其作为 JSON 响应返回。

```
exports.handler = async (event) => {
  // Extract specific properties from the event object
  const { resource, path, httpMethod, headers, queryStringParameters, body } =
 event;
  const response = {
    resource,
    path,
    httpMethod,
    headers,
    queryStringParameters,
    body,
  };
  return {
    body: JSON.stringify(response, null, 2),
    statusCode: 200,
  };
};
```

步骤 4：部署 AWS CDK 堆栈

1. 从项目的根目录运行 cdk synth 命令：

```
cdk synth
```

此命令合成 CDK 堆栈中的 AWS CloudFormation 模板。该模板是一个约 400 行的 YAML 文件，类似于以下内容。

> **Note**
>
> 如果出现以下错误，请确保您位于项目目录的根目录中。
>
> ```
> --app is required either in command-line, in cdk.json or in ~/.cdk.json
> ```

Example AWS CloudFormation 模板

```
Resources:
  MyFunctionServiceRole3C357FF2:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
          - Action: sts:AssumeRole
            Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
        Version: "2012-10-17"
      ManagedPolicyArns:
        - Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - :iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Metadata:
      aws:cdk:path: HelloLambdaStack/MyFunction/ServiceRole/Resource
  MyFunction1BAA52E7:
    Type: AWS::Lambda::Function
    Properties:
      Code:
        S3Bucket:
          Fn::Sub: cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}
```

```
        S3Key:
 ab1111111cd32708dc4b83e81a21c296d607ff2cdef00f1d7f48338782f92l3901.zip
      Handler: index.handler
      Role:
        Fn::GetAtt:
          - MyFunctionServiceRole3C357FF2
          - Arn
      Runtime: nodejs20.x
      ...
```

2. 运行 cdk deploy 命令：

```
cdk deploy
```

等待资源创建完成。最终输出包括 API Gateway 端点的 URL。例如：

```
Outputs:
HelloLambdaStack.ApiGwEndpoint77F417B1 = https://abcd1234.execute-api.us-
east-1.amazonaws.com/prod/
```

步骤 5：测试函数

要调用 Lambda 函数，请复制 API Gateway 端点，并将其粘贴到 Web 浏览器中或运行 `curl` 命令：

```
curl -s https://abcd1234.execute-api.us-east-1.amazonaws.com/prod/
```

响应是原始事件对象中选定属性的 JSON 表示，其中包含有关向 API Gateway 端点发出的请求的信息。例如：

```
{
  "resource": "/",
  "path": "/",
  "httpMethod": "GET",
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7",
    "Accept-Encoding": "gzip, deflate, br, zstd",
    "Accept-Language": "en-US,en;q=0.9",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Desktop-Viewer": "true",
```

```
    "CloudFront-Is-Mobile-Viewer": "false",
    "CloudFront-Is-SmartTV-Viewer": "false",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Viewer-ASN": "16509",
    "CloudFront-Viewer-Country": "US",
    "Host": "abcd1234.execute-api.us-east-1.amazonaws.com",
     ...
```

步骤 6：清除资源

API Gateway 端点可公开访问。为防止意外收费，请运行 cdk destroy 命令，删除堆栈和所有关联资源。

```
cdk destroy
```

后续步骤

有关使用所选语言编写 AWS CDK 应用程序的信息，请参阅以下内容：

TypeScript

- 在 TypeScript 中使用 AWS CDK

JavaScript

- 在 JavaScript 中使用 AWS CDK

Python

- 在 Python 中使用 AWS CDK

Java

- 在 Java 中使用 AWS CDK

C#

- 在 C# 中使用 AWS CDK

Go

- 在 Go 中使用 AWS CDK

了解 Lambda 编程模型

Lambda 提供了对所有运行时都通用的编程模型。该编程模型定义了代码和 Lambda 系统之间的接口。您可以通过在函数配置中定义处理程序来告知 Lambda 您函数的入口点。运行时将对象（例如函数名和请求 ID）传递给包含调用事件以及上下文的处理程序。

在处理程序完成第一个事件的处理后，运行时会向处理程序发送另一个事件。函数的类保留在内存中，因此，可以重用初始化代码中在处理程序方法外部声明的客户端和变量。要节省后续事件的处理时间，请在初始化期间创建可重用的资源，如 AWS 开发工具包客户端。在初始化后，函数的每个实例都可以处理数千个请求。

此外，您的函数有权访问 `/tmp` 目录中的本地存储。冻结执行环境时，目录内容会保留，同时提供可用于多次调用的暂时性缓存。有关更多信息，请参阅 Lambda 执行环境。

在启用AWS X-Ray跟踪时，运行时会单独记录初始化和执行的子分段。

运行时捕获来自函数的日志记录输出，并将它发送到 Amazon CloudWatch Logs。除了记录函数的输出外，运行时还会在函数调用开始和结束时记录条目。这包括具有请求 ID、计费持续时间、初始化持续时间和其他详细信息的报告日志。如果函数引发一个错误，则运行时会将该错误返回到调用程序。

> **Note**
>
> 日志记录受 CloudWatch Logs 配额的限制。由于节流或在某些情况下函数实例停止，日志数据可能会丢失。

Lambda 通过在需求增加时运行其他实例并在需求减少时停止实例来扩展您的函数。此模型会导致应用程序架构发生变化，例如：

- 除非另有说明，否则传入请求可能会不按次序处理或同时处理。
- 不依赖长期存在的函数实例，而是将应用程序的状态存储在其他位置。
- 虽然可使用本地存储和类级别对象来提高性能，但应将部署包的大小和传输到执行环境中的数据量保持在最小值。

有关您常用的编程语言中的编程模式的实际操作介绍，请参阅以下章节。

- 使用 Node.js 构建 Lambda 函数
- 使用 Python 构建 Lambda 函数

- 使用 Ruby 构建 Lambda 函数
- 使用 Java 构建 Lambda 函数
- 使用 Go 构建 Lambda 函数
- 使用 C# 构建 Lambda 函数
- 使用 PowerShell 构建 Lambda 函数

Lambda 运行时

Lambda 通过使用运行时支持多种语言。运行时系统提供特定于语言的环境，用于在 Lambda 与函数之间中继调用事件、上下文信息和响应。您可以使用 Lambda 提供的运行时，或构建您自己的运行时。

每个主要编程语言版本都有单独的运行时，并具有唯一的运行时标识符，例如 `nodejs20.x` 或 `python3.12`。要将某个函数配置为使用新的主要语言版本，您需要更改运行时系统标识符。由于 AWS Lambda 无法保证主要版本之间的向后兼容性，因此该操作应由客户发起。

对于定义为容器映像的函数，您可以在创建容器映像时选择运行时系统和 Linux 发行版。要更改运行时，您需要创建一个新的容器映像。

要将 .zip 文件存档作为部署程序包，您需要在创建函数时选择运行时。要更改运行时，您可以更新函数的配置。运行时与其中一个 Amazon Linux 发行版配对。底层执行环境提供了您可通过函数代码访问的额外的库和环境变量。

Lambda 调用执行环境中的函数。执行环境提供管理运行函数所需的运行时和其他资源的安全、隔离的运行时环境。Lambda 会重新使用上一个调用的执行环境（如果可用），其也可以创建新的执行环境。

要在 Lambda 中使用 Go 或 Rust 等其他语言，请使用仅限操作系统的运行时系统。Lambda 执行环境提供运行时接口来获取调用事件并发送响应。您可以通过将自定义运行时系统与函数代码一起实施来部署其他语言，也可以在层中部署它们。

支持的运行时

下表列出了支持的 Lambda 运行时系统和预计的弃用日期。弃用运行时系统后，您仍然可以在有限的时间内创建和更新函数。有关更多信息，请参阅 the section called “弃用后的运行时系统使用”。该表提供了当前预测的运行时系统弃用日期。这些日期仅供规划之用，可能会发生变化。

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
Node.js 20	`nodejs20.x`	Amazon Linux 2023	未计划	未计划	未计划
Node.js 18	`nodejs18.x`	Amazon Linux 2	2025 年 7 月 31 日	2025 年 9 月 1 日	2025 年 10 月 1 日

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
Python 3.12	python3.12	Amazon Linux 2023	未计划	未计划	未计划
Python 3.11	python3.11	Amazon Linux 2	未计划	未计划	未计划
Python 3.10	python3.10	Amazon Linux 2	未计划	未计划	未计划
Python 3.9	python3.9	Amazon Linux 2	未计划	未计划	未计划
Java 21	java21	Amazon Linux 2023	未计划	未计划	未计划
Java 17	java17	Amazon Linux 2	未计划	未计划	未计划
Java 11	java11	Amazon Linux 2	未计划	未计划	未计划
Java 8	java8.al2	Amazon Linux 2	未计划	未计划	未计划
.NET 8	dotnet8	Amazon Linux 2023	未计划	未计划	未计划
.NET 6	dotnet6	Amazon Linux 2	2024 年 12 月 20 日	2025 年 2 月 28 日	2025 年 3 月 31 日
Ruby 3.3	ruby3.3	Amazon Linux 2023	未计划	未计划	未计划
Ruby 3.2	ruby3.2	Amazon Linux 2	未计划	未计划	未计划
仅限操作系统的运行时系统	provided.al2023	Amazon Linux 2023	未计划	未计划	未计划

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
仅限操作系统的运行时系统	`provided.al2`	Amazon Linux 2	未计划	未计划	未计划

> **Note**
>
> 对于新区域，Lambda 将不支持设置为在未来 6 个月内弃用的运行时系统。

Lambda 通过补丁和对次要版本发布的支持使托管运行时系统及其对应的容器基础映像保持最新。有关更多信息，请参阅 Lambda 运行时更新。

弃用 Go 1.x 运行时系统后，Lambda 继续支持 Go 编程语言。有关更多信息，请参阅 AWS Compute Blog 上的 Migrating AWS Lambda functions from the Go1.x runtime to the custom runtime on Amazon Linux 2。

受支持的所有 Lambda 运行时系统均支持 x86_64 和 arm64 架构。

新的运行时系统版本

只有在新语言版本的发布周期达到长期支持（LTS）阶段时，Lambda 才会为其提供托管运行时系统。例如，在 Node.js release cycle 中，当发布进入 Active LTS 阶段时。

在版本进入长期支持阶段之前，其仍处于开发阶段，仍可能发生重大变化。默认情况下，Lambda 会自动应用运行时系统更新，因此对运行时系统版本进行重大更改可能会导致函数无法按预期运行。

Lambda 不为未计划发布 LTS 的语言版本提供托管运行时系统。

下表列出了即将推出的 Lambda 运行时系统的目标发布月份。这些日期可能会发生变化，仅供参考。

- Python 3.13 - 2024 年 11 月
- Node.js 22 - 2024 年 11 月
- Ruby 3.4 – 2025 年 3 月
- Java 25 – 2025 年 10 月
- Python 3.14 – 2025 年 11 月
- Node.js 24 – 2025 年 11 月

- .NET 10 – 2025 年 12 月

运行时弃用策略

适用于 .zip 文件存档的 Lambda 运行时 是围绕不断进行维护和安全更新的操作系统、编程语言和软件库的组合构建的。Lambda 的标准弃用策略是，当运行时系统的任何主要组件的社区长期支持（LTS）结束且安全更新不再可用时，就会弃用该运行时系统。大多数情况是因为语言运行时系统，但是在某些情况下，由于操作系统（OS）的 LTS 结束，也可能会弃用运行时系统。

当运行时被弃用后，AWS 可能不再对该运行时系统应用安全补丁或更新，使用该运行时系统的函数将不再有资格获得技术支持。此类已弃用的运行时系统按“原样”提供，不提供任何担保，并且可能包含漏洞、错误、缺陷或其他漏洞。

要了解有关管理运行时系统升级和弃用的更多信息，请参阅以下部分和 AWS Compute Blog 上的 Managing AWS Lambda runtime upgrades。

> **Important**
>
> Lambda 偶尔会将 Lambda 运行时系统所支持的语言版本的支持终止日期延后一段有限的时间。在此期间，Lambda 仅对运行时系统操作系统应用安全补丁。Lambda 在编程语言运行时系统的支持到期后不会对其应用安全补丁。

责任共担模式

Lambda 负责为所有受支持的托管运行时和容器基础映像整理和发布安全更新。默认情况下，Lambda 将这些更新自动应用于使用托管式运行时的函数。如果默认的自动运行时更新设置已更改，则请参阅运行时管理控件责任共担模式。对于使用容器映像部署的函数，您负责从最新的基础映像中重建函数的容器映像并对其进行重新部署。

当运行时被弃用时，Lambda 更新托管式运行时和容器基础映像的责任即告终止。您将负责升级函数以使用支持的运行时或基础映像。

在所有情况下，您都有责任将更新应用于函数代码，包括其依赖项。下表汇总了您在责任共担模式下的责任。

运行时生命周期阶段	Lambda 的责任	您的责任
支持的托管式运行时	提供定期的运行时更新，包括安全补丁和其他更新。 默认情况下自动应用运行时更新（有关非默认行为，请参阅 the section called “运行时更新模式”）。	更新您的函数代码，包括依赖项，以解决任何安全漏洞。
支持的容器映像	通过安全补丁和其他更新提供对容器基础映像的定期更新。	更新您的函数代码，包括依赖项，以解决任何安全漏洞。 使用最新的基础映像定期重新构建和重新部署您的容器映像。
托管式运行时即将弃用	在运行时弃用之前，通过文档、AWS Health Dashboard、电子邮件和 Trusted Advisor 通知客户。 运行时更新的责任在弃用时结束。	监控 Lambda 文档、AWS Health Dashboard、电子邮件或 Trusted Advisor 以了解运行时弃用信息。 在弃用之前的运行时之前，将函数升级到支持的运行时。
容器映像即将弃用	使用容器映像的函数不可用弃用通知。 容器基础映像更新的责任在弃用时结束。	在弃用之前的映像之前，请注意弃用计划并将函数升级到支持的基础映像。

弃用后的运行时系统使用

当运行时被弃用后，AWS 可能不再对该运行时系统应用安全补丁或更新，使用该运行时系统的函数将不再有资格获得技术支持。此类已弃用的运行时系统按“原样”提供，不提供任何担保，并且可能包含漏洞、错误、缺陷或其他漏洞。使用已弃用运行时的函数也可能会出现性能下降或证书过期等其他问题，进而导致函数无法正常运行。

在运行时系统弃用后的至少 30 天内，您仍然可以使用该运行时创建新的 Lambda 函数。从弃用 30 天后，Lambda 会开始阻止新函数的创建。

在运行时系统弃用后的至少 60 天内，您仍然可以更新现有函数的函数代码和配置。从弃用 60 天后，Lambda 会开始阻止更新现有函数的函数代码和配置。

> **Note**
>
> 对于某些运行时系统，AWS 会将块函数创建日期和块函数更新日期推迟到弃用后通常的 30 天和 60 天之后。AWS 根据客户反馈进行了此项更改，让您有更多时间升级函数。请参阅 the section called “支持的运行时” 和 the section called “已弃用的运行时” 中的表格，查看运行时系统的相关日期。

弃用运行时系统后，可以更新函数以无限期地使用较新的支持运行时系统。在生产环境中应用运行时更改之前，您应该测试您的函数是否可以与新的运行时结合使用，因为在 60 天期限过后，您将无法恢复到已弃用的运行时。建议使用函数版本和别名，以便通过回滚来实现安全部署。

请注意，可以继续创建和更新函数的确切时长并不固定。此期限可能因每次弃用而异，也可能因不同 AWS 区域 而有所不同。本页第一部分的“支持的运行时”表中提供了阻止函数创建和更新的标称日期。在本表提供的日期之前，Lambda 不会开始阻止函数创建或更新。

弃用运行时系统后，可以无限期地继续调用函数。但是，AWS 强烈建议您将函数迁移到受支持的运行时系统，以便函数继续接收安全补丁程序并保持获得技术支持的资格。

接收运行时系统弃用通知

当运行时系统接近其弃用日期时，如果您 AWS 账户 中的任何函数使用该运行时，Lambda 会向您发送电子邮件提醒。通知还显示在 AWS Health Dashboard 和 AWS Trusted Advisor 中。

- 接收电子邮件通知：

  Lambda 会在运行时系统被弃用前至少 180 天向您发送电子邮件提醒。这封电子邮件将列出使用运行时系统的所有函数的 $LATEST 版本。要查看受影响函数版本的完整列表，请使用 Trusted Advisor 或查看 the section called “按运行时获取有关函数的相关数据”。

  Lambda 会向您 AWS 账户 的主要账户联系人发送电子邮件通知。有关查看或更新账户中电子邮件地址的信息，请参阅《AWS 一般参考》中的 Updating contact information。
- 通过 AWS Health Dashboard 接收通知：

AWS Health Dashboard 将在运行时系统被弃用前至少 180 天显示通知。通知显示在您的账户运行状况页面的其他通知下。通知的受影响资源选项卡列出了使用运行时系统的所有函数的 $LATEST 版本。

> **Note**
>
> 要查看受影响函数版本的最新完整列表，请使用 Trusted Advisor 或查看 the section called "按运行时获取有关函数的相关数据"。

AWS Health Dashboard 通知将在受影响的运行时系统被弃用 90 天后过期。

- 使用 AWS Trusted Advisor

  Trusted Advisor 将在运行时系统被弃用前 180 天显示通知。通知显示在安全页面上。受影响函数的列表显示在使用弃用运行时系统的 AWS Lambda 函数下。此函数列表显示 $LATEST 和已发布的版本，并会自动更新以反映函数的当前状态。

  您可以在 Trusted Advisor 控制台的首选项页面中的 Trusted Advisor 打开每周电子邮件通知。

已弃用的运行时

已经终止对下列运行时的支持：

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
Python 3.8	`python3.8`	Amazon Linux 2	2024 年 10 月 14 日	2025 年 2 月 28 日	2025 年 3 月 31 日
Node.js 16	`nodejs16.x`	Amazon Linux 2	2024 年 6 月 12 日	2025 年 2 月 28 日	2025 年 3 月 31 日
.NET 7（仅限容器）	`dotnet7`	Amazon Linux 2	2024 年 5 月 14 日	不适用	不适用
Java 8	`java8`	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2025 年 2 月 28 日

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
Go 1.x	go1.x	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2025 年 2 月 28 日
仅限操作系统的运行时系统	provided	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2025 年 2 月 28 日
Ruby 2.7	ruby2.7	Amazon Linux 2	2023 年 12 月 7 日	2024 年 1 月 9 日	2025 年 2 月 28 日
Node.js 14	nodejs14.x	Amazon Linux 2	2023 年 12 月 4 日	2024 年 1 月 9 日	2025 年 2 月 28 日
Python 3.7	python3.7	Amazon Linux	2023 年 12 月 4 日	2024 年 1 月 9 日	2025 年 2 月 28 日
.NET Core 3.1	dotnetcore3.1	Amazon Linux 2	2023 年 4 月 3 日	2023 年 4 月 3 日	2023 年 5 月 3 日
Node.js 12	nodejs12.x	Amazon Linux 2	2023 年 3 月 31 日	2023 年 3 月 31 日	2023 年 4 月 30 日
Python 3.6	python3.6	Amazon Linux	2022 年 7 月 18 日	2022 年 7 月 18 日	2022 年 8 月 29 日
.NET 5（仅限容器）	dotnet5.0	Amazon Linux 2	2022 年 5 月 10 日	不适用	不适用
.NET Core 2.1	dotnetcore2.1	Amazon Linux	2022 年 1 月 5 日	2022 年 1 月 5 日	2022 年 4 月 13 日
Node.js 10	nodejs10.x	Amazon Linux 2	2021 年 7 月 30 日	2021 年 7 月 30 日	2022 年 2 月 14 日
Ruby 2.5	ruby2.5	Amazon Linux	2021 年 7 月 30 日	2021 年 7 月 30 日	2022 年 3 月 31 日
Python 2.7	python2.7	Amazon Linux	2021 年 7 月 15 日	2021 年 7 月 15 日	2022 年 5 月 30 日

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
Node.js 8.10	`nodejs8.10`	Amazon Linux	2020 年 3 月 6 日	不适用	2020 年 3 月 6 日
Node.js 4.3	`nodejs4.3`	Amazon Linux	2020 年 3 月 5 日	不适用	2020 年 3 月 5 日
Node.js 4.3 边缘	`nodejs4.3-edge`	Amazon Linux	2020 年 3 月 5 日	不适用	2019 年 4 月 30 日
Node.js 6.10	`nodejs6.10`	Amazon Linux	2019 年 8 月 12 日	2019 年 8 月 12 日	不适用
.NET Core 1.0	`dotnetcore1.0`	Amazon Linux	2019 年 6 月 27 日	不适用	2019 年 7 月 30 日
.NET Core 2.0	`dotnetcore2.0`	Amazon Linux	2019 年 5 月 30 日	不适用	2019 年 5 月 30 日
Node.js 0.10	`nodejs`	Amazon Linux	不适用	不适用	2016 年 10 月 31 日

在大多数情况下，语言版本或操作系统的使用结束期限预先可知。以下链接提供了 Lambda 支持作为托管运行时系统使用的每种语言的终止使用计划。

语言和框架支持政策

- Node.js – github.com
- Python – devguide.python.org
- Ruby – www.ruby-lang.org
- Java – www.oracle.com 和 Corretto 常见问题
- Go – golang.org
- .NET – dotnet.microsoft.com

了解 Lambda 如何管理运行时版本更新

Lambda 通过安全更新、错误修复、新功能、性能增强和对次要发行版的支持，使每个托管运行时保持更新。这些运行时更新作为运行时版本发布。Lambda 通过将函数从早期运行时版本迁移到新的运行时版本，来对函数应用运行时更新。

对于使用托管式运行时的函数，默认情况下，Lambda 会自动应用运行时更新。借助自动运行时更新，Lambda 免去了修补运行时版本的操作负担。对于大多数客户来说，自动更新是正确的选择。您可以通过配置运行时管理设置来更改此默认行为。

Lambda 还将每个新的运行时版本作为容器映像发布。要更新基于容器的函数的运行时版本，您必须从更新后的基本映像创建一个新的容器映像并重新部署函数。

每个运行时版本都与版本号和 ARN（Amazon 资源名称）相关联。运行时版本号使用 Lambda 定义的编号方案，独立于编程语言使用的版本号。运行时版本 ARN 是每个运行时版本的唯一标识符。您可以在 Lambda 控制台或函数日志 `INIT_START` 行中查看函数当前运行时版本的 ARN。

不应将运行时版本与运行时标识符混淆。每个运行时都有唯一的运行时标识符，例如 `python3.12` 或 `nodejs20.x`。它们对应于每个主要的编程语言版本。运行时版本描述单个运行时的补丁版本。

> **Note**
>
> 相同运行时版本号的 ARN 可能因 AWS 区域 和 CPU 架构而异。

主题

运行时更新模式

Lambda 致力于提供与现有函数向后兼容的运行时更新。但是，与软件修补一样，在极少数情况下，运行时更新会对现有函数产生负面影响。例如，安全性补丁可能会暴露现有函数的潜在问题，而该问题取决于先前的不安全行为。在运行时版本不兼容的极少数情况下，Lambda 运行时管理控件有助于减少对工作负载造成任何有风险的影响。对于每个函数版本（$LATEST 或已发布版本），您可以选择以下运行时更新模式之一：

- 自动（默认）– 通过 两阶段运行时版本推出，自动更新到最新的安全运行时版本。我们建议大多数客户使用此模式，以便您始终受益于运行时更新。
- 函数更新 – 当您更新函数时，系统将更新到最新的安全运行时版本。当您更新函数时，Lambda 会将函数的运行时更新为最新的安全运行时版本。这种方法可将运行时更新与函数部署同步，这样您就可以控制 Lambda 应用运行时更新的时间。使用此模式，您可以尽早检测和缓解少数运行时更新不兼容问题。使用此模式时，您必须定期更新函数以保持最新的函数运行时。
- 手动 – 手动更新您的运行时版本。您需要在函数配置中指定运行时版本。该函数将无限期使用此运行时版本。在极少数情况下，新的运行时版本与现有函数不兼容，您可以使用此模式将函数回滚到早期运行时版本。不建议使用 Manual（手动）模式来尝试实现跨部署的运行时一致性。有关更多信息，请参阅 回滚 Lambda 运行时版本。

将运行时更新应用于函数的责任因您选择的运行时更新模式而异。有关更多信息，请参阅 了解 Lambda 运行时管理的责任共担模式。

两阶段运行时版本推出

Lambda 按照以下顺序推出新的运行时版本：

1. 在第一阶段，每当您创建或更新函数时，Lambda 都会应用新的运行时版本。当您调用 UpdateFunctionCode 或 UpdateFunctionConfiguration API 操作时，函数会更新。
2. 在第二阶段，Lambda 会更新任何使用 Auto（自动）运行时更新模式且尚未更新到新运行时版本的函数。

推出过程的总持续时间因多种因素而异，例如运行时更新中包含的任何安全性补丁的严重性。

如果您正在积极开发和部署函数，您很可能会在第一阶段接受新的运行时版本。这可以使运行时更新与函数更新同步。在极少数情况下，最新的运行时版本会对应用程序造成负面影响，而这种方法可让您及时采取纠正措施。未处于积极开发阶段的函数在第二阶段仍能获得自动运行时更新的操作优势。

这种方法不影响设置为 Function update（函数更新）或 Manual（手动）模式的函数。使用 Function update（函数更新）模式的函数只有在创建或更新时才会接收最新的运行时更新。使用 Manual（手动）模式的函数不接收运行时更新。

Lambda 以渐进、滚动的方式跨 AWS 区域 发布新的运行时版本。如果您的函数设置为 Auto（自动）或 Function update（函数更新）模式，则同时部署到不同区域或在同一区域不同时间部署的函数可能会采用不同的运行时版本。需要在其环境中保证运行时系统版本一致性的客户应使用容器映像部署其 Lambda 函数。手动模式旨在作为临时缓解措施，以便在出现运行时与您的函数不兼容的极少数情况下回滚运行时版本。

配置 Lambda 运行时管理设置

您可以使用 Lambda 控制台或 AWS Command Line Interface（AWS CLI）配置运行时管理设置。

> **Note**
>
> 您可以为每个函数版本单独配置运行时管理设置。

配置 Lambda 更新运行时版本的方式（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数的名称。
3. 在 Code（代码）选项卡上的 Runtime settings（运行时设置）下，选择 Edit runtime management configuration（编辑运行时管理配置）。
4. 在 Runtime management configuration（运行时管理配置）下，选择以下选项之一：

   - 要让函数自动更新到最新的运行时版本，请选择 Auto（自动）。
   - 要在更改函数后将函数更新到最新的运行时版本，请选择 Function update（函数更新）。
   - 要仅在更改运行时版本 ARN 后将函数更新到最新的运行时版本，请选择 Manual（手动）。您可以在 Runtime management configuration（运行时管理配置）下找到运行时版本 ARN。您也可以在函数日志 `INIT_START` 行中找到 ARN。

   有关这些选项的更多信息，请参阅运行时更新模式。
5. 选择保存。

配置 Lambda 更新运行时版本的方式（AWS CLI）

要为函数配置运行时管理，请运行 put-runtime-management-config AWS CLI 命令。使用 Manual 模式时，还必须提供运行时版本 ARN。

```
aws lambda put-runtime-management-config \
  --function-name my-function \
  --update-runtime-on Manual \
  --runtime-version-arn arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1
```

您应该可以看到类似于如下所示的输出内容：

```
{
  "UpdateRuntimeOn": "Manual",
  "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
  "RuntimeVersionArn": "arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1"
}
```

回滚 Lambda 运行时版本

在极少数情况下，新的运行时版本与现有函数不兼容，您可以将其运行时版本回滚到早期版本。这样，可以使应用程序保持正常运行并最大限度地减少中断，从而在返回到最新的运行时版本之前有时间解决不兼容问题。

Lambda 不会对您可以使用任何特定运行时版本的时间施加限制。但是，我们强烈建议您尽快更新到最新的运行时版本，以便从最新的安全性补丁、性能改进和功能中受益。Lambda 提供回滚到早期运行时版本的选项，仅作为出现运行时更新兼容性问题的极少数情况下的临时缓解措施。如果长时间使用早期运行时版本的函数，最终可能会出现性能下降或证书过期等问题，进而导致函数无法正常运行。

您可以通过以下方式回滚运行时版本：

- 使用手动运行时更新模式
- 使用已发布的函数版本

有关更多信息，请参阅 AWS 计算博客上的 推出 AWS Lambda 运行时管理控件。

使用 Manual（手动）运行时更新模式回滚运行时版本

如果您使用的是 Auto（自动）运行时版本更新模式，或者 $LATEST 运行时版本，则可以使用 Manual（手动）模式回滚运行时版本。对于要回滚的函数版本，将运行时版本更新模式更改为

Manual（手动），并指定先前运行时版本的 ARN。有关查找先前运行时版本的 ARN 的更多信息，请参阅 识别 Lambda 运行时版本更改。

> **Note**
>
> 如果函数的 $LATEST 版本配置为使用 Manual（手动）模式，则无法更改函数使用的 CPU 架构或运行时版本。要进行这些更改，必须更改为 Auto（自动）或 Function update（函数更新）模式。

使用已发布的函数版本回滚运行时版本

已发布的函数版本是您对其进行创建时，$LATEST 函数代码和配置的不可变快照。在 Auto（自动）模式下，Lambda 会在运行时版本推出的第二阶段自动更新已发布函数版本的运行时版本。在 Function update（函数更新）模式下，Lambda 不会更新已发布函数版本的运行时版本。

因此，使用 Function update（函数更新）模式的已发布函数版本会创建函数代码、配置和运行时版本的静态快照。通过将 Function update（函数更新）模式与函数版本结合使用，您可以将运行时更新与部署同步。您还可以通过将流量重定向到较早发布的函数版本，来协调代码、配置和运行时版本的回滚。您可以将此方法集成到持续集成和持续交付（CI/CD）中，以便在运行时更新不兼容的极少数情况下实现全自动回滚。使用此方法时，必须定期更新函数并发布新的函数版本以获取最新的运行时更新。有关更多信息，请参阅 了解 Lambda 运行时管理的责任共担模式。

识别 Lambda 运行时版本更改

运行时版本号和 ARN 记录在 INIT_START 日志行中，Lambda 每次创建新的执行环境时都会将其发送到 CloudWatch Logs。由于执行环境对所有函数调用使用相同的运行时系统版本，因此 Lambda 仅在执行初始化阶段时才会发送 INIT_START 日志行。Lambda 不会针对每次函数调用发送此日志行。Lambda 将日志行发送到 CloudWatch Logs，但不会在控制台中显示。

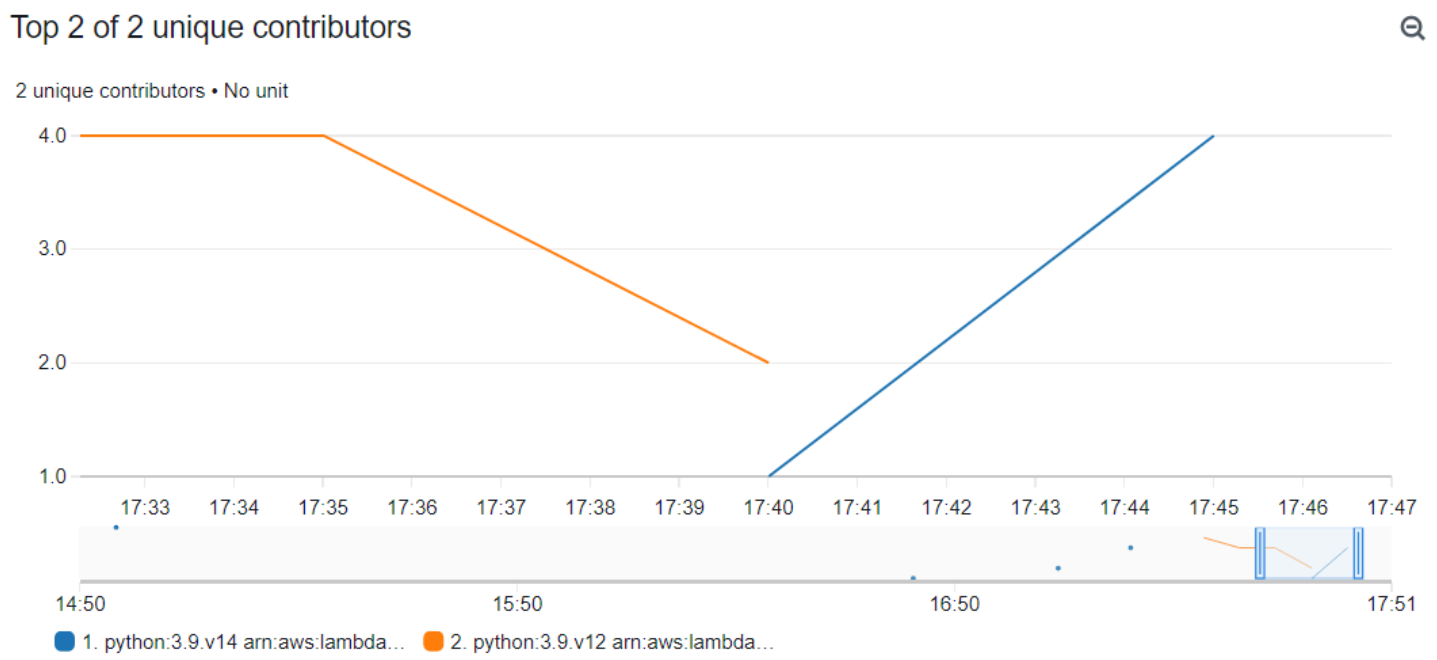
Example 示例 INIT_START 日志行

```
INIT_START Runtime Version: python:3.9.v14    Runtime Version ARN: arn:aws:lambda:eu-
south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e6I
```

您可以使用 Amazon CloudWatch Contributor Insights 来识别运行时版本之间的转换，而不是直接使用日志。以下规则计算每个 INIT_START 日志行中不同的运行时版本。要使用该规则，请将示例日志组名称 /aws/lambda/* 替换为函数或函数组的相应前缀。

```
{
  "Schema": {
    "Name": "CloudWatchLogRule",
    "Version": 1
  },
  "AggregateOn": "Count",
  "Contribution": {
    "Filters": [
      {
        "Match": "eventType",
        "In": [
          "INIT_START"
        ]
      }
    ],
    "Keys": [
      "runtimeVersion",
      "runtimeVersionArn"
    ]
  },
  "LogFormat": "CLF",
  "LogGroupNames": [
    "/aws/lambda/*"
  ],
  "Fields": {
    "1": "eventType",
    "4": "runtimeVersion",
    "8": "runtimeVersionArn"
  }
}
```

以下 CloudWatch Contributor Insights 报告显示了上述规则捕获的运行时版本转换示例。橙线表示早期运行时版本（python:3.9.v12）的执行环境初始化，蓝线表示新运行时版本（python:3.9.v14）的执行环境初始化。



了解 Lambda 运行时管理的责任共担模式

Lambda 负责为所有受支持的托管运行时和容器映像整理和发布安全更新。更新现有函数以使用最新运行时版本的责任因所使用的运行时更新模式而异。

Lambda 负责将运行时更新应用于配置为使用 Auto（自动）运行时更新模式的所有函数。

对于使用 Function update（函数更新）运行时更新模式配置的函数，您负责定期更新函数。Lambda 负责在您进行这些更新时应用运行时更新。如果您不更新函数，则 Lambda 不会更新运行时。如果您不经常更新函数，我们强烈建议您将其配置为自动运行时更新，以便其继续接收安全更新。

对于配置为使用 Manual（手动）运行时更新模式的函数，您需要负责更新函数以使用最新的运行时版本。强烈建议您仅使用此模式来回滚运行时版本，以此作为在出现运行时更新不兼容问题的极少数情况下的临时缓解措施。我们还建议您尽快切换为 Auto（自动）模式，以最大限度地减少未修补函数的时间。

如果您使用容器映像部署函数，则 Lambda 负责发布更新的基本映像。在这种情况下，您负责从最新的基本映像中重建函数的容器映像并对其进行重新部署。

下表对此进行了总结：

Deployment mode (部署模式)	Lambda 的责任	客户责任
托管运行时，Auto（自动）模式	发布包含最新补丁的新运行时版本。 将运行时补丁应用于现有函数。	如果出现运行时更新兼容性问题的极少数情况，请回滚到先前的运行时版本。
托管运行时，Function update（函数更新）模式	发布包含最新补丁的新运行时版本。	定期更新函数以获取最新的运行时版本。 如果您不经常更新函数，请将函数切换为 Auto（自动）模式。 如果出现运行时更新兼容性问题的极少数情况，请回滚到先前的运行时版本。
托管运行时，Manual（手动）模式	发布包含最新补丁的新运行时版本。	只有在出现运行时更新兼容性问题的极少数情况下，才能使用此模式进行临时运行时回滚。 尽快将函数切换为 Auto（自动）或 Function update（函数更新）模式和最新的运行时版本。
容器映像	发布包含最新补丁的新容器映像。	使用最新的容器基本映像定期重新部署函数以获取最新的补丁。

有关与 AWS 责任共担的更多信息，请参阅责任共担模式。

控制高合规性应用程序的 Lambda 运行时更新权限

Lambda 客户通常依赖自动运行时更新来满足修补要求。如果您的应用程序需要严格遵守修补即时性要求，则您可能需要限制对早期运行时版本的使用。您可以使用 AWS Identity and Access Management（IAM）拒绝 AWS 账户中的用户访问 PutRuntimeManagementConfig API 操作，以此限制 Lambda 的运行时管理控件。此操作用于为函数选择运行时更新模式。拒绝访问此操作会导致所有函数默认为 Auto（自动）模式。您可以使用服务控制策略（SCP）在整个组织中应用此限制。如果您必须将函数回滚到早期运行时版本，可根据具体情况授予策略例外。

检索使用已弃用运行时的 Lambda 函数的相关数据

Lambda 运行时即将弃用时，Lambda 会通过电子邮件予以提醒，并在 AWS Health Dashboard 和 Trusted Advisor 中提供通知。这些电子邮件和通知将列出使用运行时的函数的 $LATEST 版本。要列出所有使用特定运行时的函数版本，可以利用 AWS Command Line Interface（AWS CLI）或一种 AWS SDK。

如果有大量函数使用即将弃用的运行时，还可以借助 AWS CLI 或 AWS SDK 来确定优先更新最常调用函数的顺序。

有关如何使用 AWS CLI 和 AWS SDK 收集使用特定运行时的函数的相关数据，请参阅以下各节的内容。

列出使用特定运行时的函数版本

要使用 AWS CLI 列出使用特定运行时的所有函数版本，请运行以下命令。将 `RUNTIME_IDENTIFIER` 替换为要弃用的运行时系统名称，然后选择自己的 AWS 区域 名称。要仅列出 $LATEST 函数版本，请在命令中省略 `--function-version ALL`。

```
aws lambda list-functions --function-version ALL --region us-east-1 --output text --
query "Functions[?Runtime=='RUNTIME_IDENTIFIER'].FunctionArn"
```

> **Tip**
>
> 该示例命令列出了特定 AWS 账户 在 `us-east-1` 区域中的函数。您需要为您的账户拥有函数的每个区域以及每个 AWS 账户 重复此命令。

您还可以使用一种 AWS SDK 列出使用特定运行时的函数。以下示例代码使用 V3 AWS SDK for JavaScript 和 AWS SDK for Python (Boto3) 来返回使用特定运行时的函数的函数 ARN 列表。该示例代码还会返回列出的各函数的 CloudWatch 日志组。您可以使用该日志组来查找该函数的上次调用日期。有关更多信息，请参阅下文的the section called “识别最常用和最近调用的函数”小节。

Node.js

Example 用于列出使用特定运行时的函数的 JavaScript 代码

```
import { LambdaClient, ListFunctionsCommand } from "@aws-sdk/client-lambda";
```

```
const lambdaClient = new LambdaClient();

const command = new ListFunctionsCommand({
    FunctionVersion: "ALL",
    MaxItems: 50
});
const response = await lambdaClient.send(command);

for (const f of response.Functions){
    if (f.Runtime == '<your_runtime>'){ // Use the runtime id, e.g. 'nodejs18.x' or
 'python3.9'
        console.log(f.FunctionArn);
        // get the CloudWatch log group of the function to
        // use later for finding the last invocation date
        console.log(f.LoggingConfig.LogGroup);
    }
}
// If your account has more functions than the specified
// MaxItems, use the returned pagination token in the
// next request with the 'Marker' parameter
if ('NextMarker' in response){
    let paginationToken = response.NextMarker;
  }
```

Python

Example 用于列出使用特定运行时的函数的 Python 代码

```
import boto3
from botocore.exceptions import ClientError

def list_lambda_functions(target_runtime):

    lambda_client = boto3.client('lambda')

    response = lambda_client.list_functions(
        FunctionVersion='ALL',
        MaxItems=50
    )
    if not response['Functions']:
            print("No Lambda functions found")
    else:
        for function in response['Functions']:
```

```
            if function['PackageType']=='Zip' and function['Runtime'] ==
 target_runtime:
                print(function['FunctionArn'])
                # Print the CloudWatch log group of the function
                # to use later for finding last invocation date
                print(function['LoggingConfig']['LogGroup'])

    if 'NextMarker' in response:
       pagination_token = response['NextMarker']

if __name__ == "__main__":
    # Replace python3.12 with the appropriate runtime ID for your Lambda functions
    list_lambda_functions('python3.12')
```

要了解有关使用 AWS SDK 通过 ListFunctions 操作列出函数的更多信息，请参阅您首选编程语言的 SDK 文档。

您还可以使用 AWS Config 高级查询功能列出使用受影响运行时系统的所有函数。此查询仅返回函数的 $LATEST 版本，但您可以使用单个命令聚合查询以列出所有区域和多个 AWS 账户 中的函数。要了解更多信息，请参阅《AWS Config 开发人员指南》中的查询 AWS Auto Scaling 资源的当前配置状态。

识别最常用和最近调用的函数

如果 AWS 账户 包含的函数使用即将弃用的运行时，则可能需要优先更新经常调用的函数或最近调用的函数。

如果只有几个函数，则可以使用 CloudWatch Logs 控制台通过查看函数的日志流来收集这些信息。有关更多信息，请参阅 View log data sent to CloudWatch Logs。

要查看最近调用函数的次数，您还可以利用 Lambda 控制台中显示的 CloudWatch 指标信息。要查看此信息，请执行以下操作：

1. 打开 Lamba 控制台的函数页面。
2. 选择您想要查看调用统计信息的函数。
3. 选择监控选项卡。
4. 使用日期范围选择器设置要查看统计数据的时间段。最近调用显示在调用窗格中。

对于有大量函数的账户，借助 AWS CLI 或一种 AWS SDK，通过 DescribeLogStreams 和 GetMetricStatistics API 操作以编程方式收集这些数据可能更高效。

以下示例提供了使用 V3 AWS SDK for JavaScript 和 AWS SDK for Python (Boto3) 的代码片段，用于识别特定函数的上次调用日期，并确定过去 14 天内特定函数的调用次数。

Node.js

Example 用于查找函数上次调用时间的 JavaScript 代码

```
import { CloudWatchLogsClient, DescribeLogStreamsCommand } from "@aws-sdk/client-
cloudwatch-logs";
const cloudWatchLogsClient = new CloudWatchLogsClient();
const command = new DescribeLogStreamsCommand({
    logGroupName: '<your_log_group_name>',
    orderBy: 'LastEventTime',
    descending: true,
    limit: 1
});
try {
    const response = await cloudWatchLogsClient.send(command);
    const lastEventTimestamp = response.logStreams.length > 0 ?
        response.logStreams[0].lastEventTimestamp : null;
    // Convert the UNIX timestamp to a human-readable format for display
    const date = new Date(lastEventTimestamp).toLocaleDateString();
    const time = new Date(lastEventTimestamp).toLocaleTimeString();
    console.log(`${date} ${time}`);

} catch (e){
    console.error('Log group not found.')
}
```

Python

Example 用于查找函数上次调用时间的 Python 代码

```
import boto3
from datetime import datetime

cloudwatch_logs_client  = boto3.client('logs')

response = cloudwatch_logs_client.describe_log_streams(
    logGroupName='<your_log_group_name>',
    orderBy='LastEventTime',
    descending=True,
```

```
    limit=1
)

try:
    if len(response['logStreams']) > 0:
        last_event_timestamp = response['logStreams'][0]['lastEventTimestamp']
        print(datetime.fromtimestamp(last_event_timestamp/1000)) # Convert timestamp
 from ms to seconds
    else:
        last_event_timestamp = None
except:
    print('Log group not found')
```

> **Tip**
>
> 您可以使用 ListFunctions API 操作来查找函数的日志组名称。有关如何执行此操作的示例，请参阅the section called “列出使用特定运行时的函数版本”中的代码。

Node.js

Example 用于查找过去 14 天内调用次数的 JavaScript 代码

```
import { CloudWatchClient, GetMetricStatisticsCommand } from "@aws-sdk/client-
cloudwatch";
const cloudWatchClient = new CloudWatchClient();
const command = new GetMetricStatisticsCommand({
    Namespace: 'AWS/Lambda',
    MetricName: 'Invocations',
    StartTime: new Date(Date.now()-86400*1000*14), // 14 days ago
    EndTime: new Date(Date.now()),
    Period: 86400 * 14, // 14 days.
    Statistics: ['Sum'],
    Dimensions: [{
        Name: 'FunctionName',
        Value: '<your_function_name>'
    }]
});
const response = await cloudWatchClient.send(command);
const invokesInLast14Days = response.Datapoints.length > 0 ?
    response.Datapoints[0].Sum : 0;
```

```
console.log('Number of invocations: ' + invokesInLast14Days);
```

Python

Example 用于查找过去 14 天内调用次数的 Python 代码

```
import boto3
from datetime import datetime, timedelta

cloudwatch_client = boto3.client('cloudwatch')

response = cloudwatch_client.get_metric_statistics(
    Namespace='AWS/Lambda',
    MetricName='Invocations',
    Dimensions=[
        {
            'Name': 'FunctionName',
            'Value': '<your_function_name>'
        },
    ],
    StartTime=datetime.now() - timedelta(days=14),
    EndTime=datetime.now(),
    Period=86400 * 14, # 14 days
    Statistics=[
        'Sum'
    ]
)

if len(response['Datapoints']) > 0:
    invokes_in_last_14_days = int(response['Datapoints'][0]['Sum'])
else:
    invokes_in_last_14_days = 0

print(f'Number of invocations: {invokes_in_last_14_days}')
```

修改运行时环境

您可以使用内部扩展来修改运行时进程。内部扩展不是单独的进程 – 其作为运行时进程的一部分运行。

Lambda 提供特定语言的环境变量，您可以设置此变量，向运行时添加选项和工具。Lambda 还提供包装程序脚本，此脚本允许 Lambda 将运行时启动委托给脚本。您可以创建包装脚本来自定义运行时启动行为。

特定于语言的环境变量

Lambda 支持仅配置方法，以便在函数初始化期间通过以下特定语言的环境变量预加载代码：

- `JAVA_TOOL_OPTIONS` – 在 Java 上，Lambda 支持此环境变量以在 Lambda 中设置其他命令行变量。此环境变量允许您指定工具的初始化，特别是使用 `agentlib` 或 `javaagent` 选项启动本机或 Java 编程语言代理。有关更多信息，请参阅JAVA_TOOL_OPTIONS环境变量。
- `NODE_OPTIONS` – 在 Node.js 运行时系统中可用。
- `DOTNET_STARTUP_HOOKS` – 在 .NET Core 3.1 及更高版本中，此环境变量指定了 Lambda 可以使用的程序集 (dll) 的路径。

使用特定于语言的环境变量是设置启动属性的首选方法。

包装脚本

您可以创建一个包装程序脚本，自定义 Lambda 函数的运行时启动行为。使用包装脚本，您可以设置无法通过特定于语言的环境变量设置的配置参数。

> **Note**
>
> 如果包装脚本未成功启动运行时进程，则调用可能会失败。

所有本机 Lambda 运行时系统都支持包装脚本。仅限操作系统的运行时系统（`provided` 运行时系统系列）不支持包装脚本。

当您为函数使用包装程序脚本时，Lambda 会使用您的脚本启动运行时。Lambda 会向脚本发送解释器的路径以及标准运行时启动的所有原始参数。您的脚本可以扩展或转换程序的启动行为。例如，脚本可以注入和更改参数、设置环境变量或捕获指标、错误和其他诊断信息。

您可以通过将 AWS_LAMBDA_EXEC_WRAPPER 环境变量的值设置为可执行二进制文件或脚本的文件系统路径来指定脚本。

示例：使用 Python 3.8 创建并使用包装脚本

在以下示例中，您创建一个包装脚本，以使用 -X importtime 选项启动 Python 解释器。当您运行该函数时，Lambda 会生成一个日志条目，显示每次导入的导入持续时间。

使用 Python 3.8 创建并使用包装脚本

1. 要创建包装脚本，请将以下代码粘贴到名为 importtime_wrapper 的文件中：

```
#!/bin/bash

# the path to the interpreter and all of the originally intended arguments
args=("$@")

# the extra options to pass to the interpreter
extra_args=("-X" "importtime")

# insert the extra options
args=("${args[@]:0:$#-1}" "${extra_args[@]}" "${args[@]: -1}")

# start the runtime with the extra options
exec "${args[@]}"
```

2. 要授予脚本可执行文件权限，请从命令行输入 chmod +x importtime_wrapper。
3. 将脚本部署为 Lambda 层。
4. 使用 Lambda 控制台创建 Lambda 函数。
   a. 打开 Lambda 控制台。
   b. 选择 Create function（创建函数）。
   c. 在基本信息 下，对于函数名称，输入 **wrapper-test-function**。
   d. 对于运行时，选择 Python 3.8。
   e. 选择 Create function (创建函数)。
5. 将层添加到函数。
   a. 选择您的函数，然后选择 Code (代码)（如果尚未选择）。

b. 选择 Add a layer。

c. 在选择层下，选择您之前创建的兼容层的名称和版本。

d. 选择添加。

6. 将代码和环境变量添加到您的函数中。

a. 在函数代码编辑器中，粘贴以下函数代码：

```
import json

  def lambda_handler(event, context):
      # TODO implement
      return {
          'statusCode': 200,
          'body': json.dumps('Hello from Lambda!')
      }
```

b. 选择 Save（保存）。

c. 在 Environment variables (环境变量) 下，选择 Edit (编辑)。

d. 选择 Add environment variable (添加环境变量)。

e. 对于 Key（键），输入 `AWS_LAMBDA_EXEC_WRAPPER`。

f. 对于 Value（值），输入 `/opt/importtime_wrapper`。

g. 选择 Save（保存）。

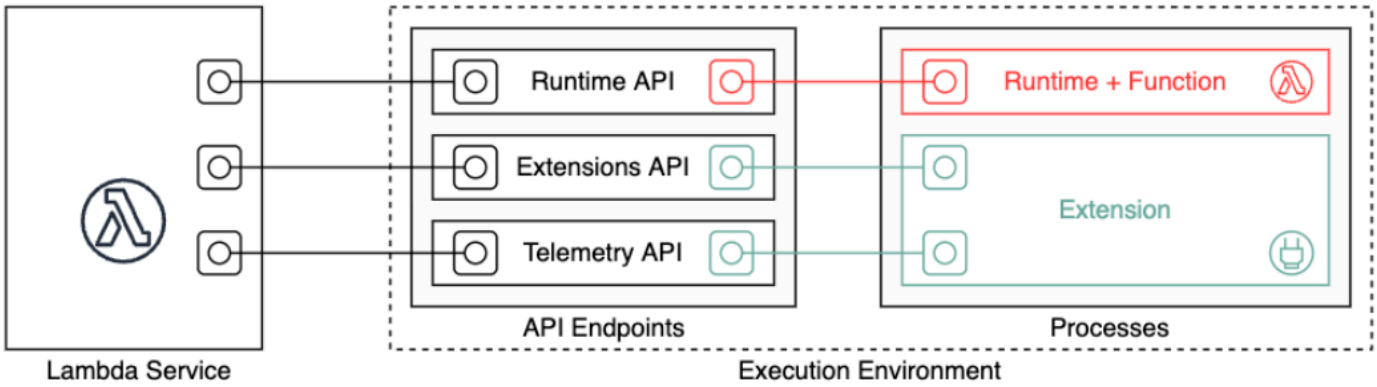
7. 要运行该函数，请选择 Test（测试）。

由于您的包装脚本使用 `-X importtime` 选项启动了 Python 解释器，因此日志会显示每次导入所需的时间。例如：

```
...
2020-06-30T18:48:46.780+01:00 import time: 213 | 213 | simplejson
2020-06-30T18:48:46.780+01:00 import time: 50 | 263 | simplejson.raw_json
...
```

将 Lambda 运行时 API 用于自定义运行时

AWS Lambda 提供了用于自定义运行时的 HTTP API，接收来自 Lambda 的调用事件并在 Lambda 执行环境中发送回响应数据。本节包含 Lambda 运行时 API 的 API 参考。



运行时 API 版本 2018-06-01 的 OpenAPI 规范在 runtime-api.zip 中提供

为了创建 API 请求 URL，运行时会从 `AWS_LAMBDA_RUNTIME_API` 环境变量获取 API 终端节点，添加 API 版本，然后添加所需的资源路径。

Example 请求

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next"
```

API 方法

- 下一个调用
- 调用响应
- 初始化错误
- 调用错误

下一个调用

路径 – `/runtime/invocation/next`

方法 – GET

运行时将此消息发送到 Lambda 以请求调用事件。响应主体包含来自调用的有效负载，该负载是包含来自函数触发器的事件数据的 JSON 文档。响应标头包含有关调用的额外数据。

响应标头

- Lambda-Runtime-Aws-Request-Id – 请求 ID，用于标识触发了函数调用的请求。

  例如：8476a536-e9f4-11e8-9739-2dfe598c3fcd。
- Lambda-Runtime-Deadline-Ms – 函数超时的日期（Unix 时间形式，以毫秒为单位）。

  例如：1542409706888。
- Lambda-Runtime-Invoked-Function-Arn – Lambda 函数的 ARN、版本或在调用中指定的别名。

  例如：arn:aws:lambda:us-east-2:123456789012:function:custom-runtime。
- Lambda-Runtime-Trace-Id – AWS X-Ray跟踪标头。

  例如：Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1。
- Lambda-Runtime-Client-Context – 对于来自 AWS Mobile SDK 的调用，为有关客户端应用程序和设备的数据。
- Lambda-Runtime-Cognito-Identity – 对于来自 AWS Mobile SDK 的调用，为有关 Amazon Cognito 身份提供商的数据。

不要对 GET 请求设置超时，因为响应可能会延迟。在 Lambda 引导运行时和运行时返回事件之间，运行时进程可能会冻结几秒钟。

请求 ID 用于跟踪 Lambda 中的调用。使用它可在您发送响应时指定调用。

跟踪标头包含跟踪 ID、父 ID 和采样决策。如果对请求进行采样，则由 Lambda 或某个上游服务对请求进行采样。运行时应设置具有标头的值的 _X_AMZN_TRACE_ID。X-Ray 开发工具包会读取此项以获取 ID 并确定是否要跟踪请求。

调用响应

路径 – /runtime/invocation/*AwsRequestId*/response

方法 – POST

在函数运行完成后，运行时会将调用响应发送到 Lambda。对于同步调用，Lambda 会将响应发送到客户端。

Example 成功请求

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "SUCCESS"
```

初始化错误

如果函数返回错误或运行时在初始化期间遇到错误，则运行时将使用此方法向 Lambda 报告错误。

路径 – `/runtime/init/error`

方法 – POST

标头

`Lambda-Runtime-Function-Error-Type` – 运行时遇到的错误类型。必需：否

此标头由字符串值组成。Lambda 接受任何字符串，但建议您使用 <category.reason> 格式。例如：

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound
- Runtime.ConfigInvalid
- Runtime.UnknownReason

主体参数

`ErrorRequest` – 有关错误的其他信息。必需：否

此字段是具有以下结构的 JSON 对象：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

请注意，Lambda 接受任何 `errorType` 值。

以下示例显示了 Lambda 函数错误消息，其中函数无法解析调用中提供的事件数据。

Example 函数错误

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

响应正文参数

- StatusResponse – 字符串。状态信息，随 202 个响应代码一起发送。
- ErrorResponse – 其他错误信息，随错误响应代码一起发送。ErrorResponse 包含错误类型和错误消息。

响应代码

- 202 – 已接受
- 403 – 禁止访问
- 500 – 容器错误。不可恢复状态。运行时应立即退出。

Example 初始化错误请求

```
ERROR="{\"errorMessage\" : \"Failed to load function.\", \"errorType\" :
 \"InvalidFunctionException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/init/error" -d "$ERROR" --
header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

调用错误

如果函数返回错误或运行时遇到错误，则运行时将使用此方法向 Lambda 报告错误。

路径 – /runtime/invocation/*AwsRequestId*/error

方法 – POST

标头

Lambda-Runtime-Function-Error-Type – 运行时遇到的错误类型。必需：否

此标头由字符串值组成。Lambda 接受任何字符串，但建议您使用 <category.reason> 格式。例如：

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound
- Runtime.ConfigInvalid
- Runtime.UnknownReason

主体参数

ErrorRequest – 有关错误的其他信息。必需：否

此字段是具有以下结构的 JSON 对象：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

请注意，Lambda 接受任何 errorType 值。

以下示例显示了 Lambda 函数错误消息，其中函数无法解析调用中提供的事件数据。

Example 函数错误

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

响应正文参数

- StatusResponse – 字符串。状态信息，随 202 个响应代码一起发送。
- ErrorResponse – 其他错误信息，随错误响应代码一起发送。ErrorResponse 包含错误类型和错误消息。

响应代码

- 202 – 已接受
- 400 – 错误请求

- 403 – 禁止访问
- 500 – 容器错误。不可恢复状态。运行时应立即退出。

Example 错误请求

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
ERROR="{\"errorMessage\" : \"Error parsing event data.\", \"errorType\" :
 \"InvalidEventDataException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/error"
 -d "$ERROR" --header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

何时使用 Lambda 仅限操作系统的运行时

Lambda 为 Java、Python、Node.js、.NET 和 Ruby 提供托管运行时系统。要使用无法作为托管运行时系统使用的编程语言创建 Lambda 函数，请使用仅限操作系统的运行时系统（provided 运行时系统系列）。仅限操作系统的运行时系统有三种主要用例：

- 本机提前（AOT）编译：Go、Rust 和 C++ 等语言本机编译为可执行的二进制文件，不需要专用语言运行时系统。这些语言只需要一个可以运行编译后二进制文件的操作系统环境。您还可以使用 Lambda 的仅限操作系统的运行时系统来部署使用.NET 本机 AOT 和 Java GraalVM 本机编译的二进制文件。

  您必须在二进制文件中包含运行时系统接口客户端。运行时系统接口客户端调用 将 Lambda 运行时 API 用于自定义运行时 来检索函数调用，然后调用您的函数处理程序。Lambda 为 Go、.NET 本机 AOT、C++ 和 Rust（实验）提供了运行时系统接口客户端。

  您的二进制文件必须在 Linux 环境下编译，且与您计划用于函数的指令集架构（x86_64 或 arm64）相同。
- 第三方运行时系统：您可以使用现成的运行时来运行 Lambda 函数，例如适用于 PHP 的 Bref 或适用于 Swift 的 SwiftAWS Lambda 运行时系统。
- 自定义运行时系统：您可以为 Lambda 未提供托管运行时系统的语言或语言版本（例如 Node.js 19）构建自己的运行时系统。有关更多信息，请参阅 构建 AWS Lambda 的自定义运行时系统。对于仅限操作系统的运行时系统，该用例最不常见。

Lambda 支持以下仅限操作系统的运行时系统：

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
仅限操作系统的运行时系统	`provided.al2023`	Amazon Linux 2023	未计划	未计划	未计划
仅限操作系统的运行时系统	`provided.al2`	Amazon Linux 2	未计划	未计划	未计划

与 Amazon Linux 2 相比，Amazon Linux 2023 (`provided.al2023`) 运行时系统具有多项优势，包括较小的部署占用空间和 `glibc` 等更新版本的库。

provided.al2023 运行时系统使用 dnf 而不是 yum 作为程序包管理器，后者是 Amazon Linux 2 中的默认程序包管理器。有关 provided.al2023 和 provided.al2 之间区别的更多信息，请参阅 AWS 计算博客上的 AWS Lambda 的 Amazon Linux 2023 运行时系统简介。

构建 AWS Lambda 的自定义运行时系统

您可以采用任何编程语言实施 AWS Lambda 运行时。运行时是一个程序，调用函数时，改程序将运行 Lambda 函数的处理程序方法。该运行时系统可包含在函数的部署包中，也可以分布在层中。创建 Lambda 函数时，请选择仅限操作系统的运行时系统（provided 运行时系统系列）。

> **Note**
>
> 创建自定义运行时系统是一个高级用例。如果您想了解有关编译为本机二进制文件或使用第三方现成运行时系统的信息，请参阅 何时使用 Lambda 仅限操作系统的运行时。

有关自定义运行时系统部署过程的演练，请参阅 教程：构建自定义运行时系统。您还可在 GitHub 的 awslabs/aws-lambda-cpp 上了解以 C++ 实现的自定义运行时。

主题

要求

自定义运行时系统必须完成某些初始化和处理任务。运行时系统会运行函数的设置代码、从环境变量读取处理程序名称，以及从 Lambda 运行时系统 API 读取调用事件。运行时会将事件数据传递到函数处理程序，并将来自处理程序的响应发布回 Lambda。

初始化任务

初始化任务将对函数的每个实例运行一次以准备用于处理调用的环境。

- 检索设置 – 读取环境变量以获取有关函数和环境的详细信息。
  - _HANDLER – 处理程序的位置（来自函数的配置）。标准格式为 *file*.*method*，其中 file 是没有表达式的文件的名称，method 是在文件中定义的方法或函数的名称。
  - LAMBDA_TASK_ROOT – 包含函数代码的目录。

- AWS_LAMBDA_RUNTIME_API – 运行时 API 的主机和端口。

  有关可用变量的完整列表，请参阅 定义运行时环境变量。

- 初始化函数 – 加载处理程序文件并运行它包含的任何全局或静态代码。函数应该创建静态资源（如开发工具包客户端和数据库连接）一次，然后将它们重复用于多个调用。
- 处理错误 – 如果出现错误，请调用初始化错误 API 并立即退出。

计入收费的执行时间和超时的初始化计数。当执行触发您的函数的新实例的初始化时，您可以在日志和 AWS X-Ray 跟踪中看到初始化时间。

Example 日志

```
REPORT RequestId: f8ac1208... Init Duration: 48.26 ms   Duration: 237.17 ms   Billed
 Duration: 300 ms   Memory Size: 128 MB   Max Memory Used: 26 MB
```

处理任务

其运行时，运行时将使用 Lambda 运行时界面来管理传入事件和报告错误。完成初始化任务后，运行时将在一个循环中处理传入事件。在运行时代码中，按顺序执行下面的步骤。

- 获取事件 – 调用下一个调用 API 来获取下一个事件。响应正文包含事件数据。响应标头包含请求 ID 和其他信息。
- 传播跟踪标头 – 从 API 响应中的 Lambda-Runtime-Trace-Id 标头获取 X-Ray 跟踪标头。使用相同的值在本地设置 _X_AMZN_TRACE_ID 环境变量。X-Ray SDK 使用此值在服务之间连接追踪数据。
- 创建上下文对象 – 使用来自 API 响应中的环境变量和标头的上下文信息创建一个对象。
- 调用函数处理程序 – 将事件和上下文对象传递到处理程序。
- 处理响应 – 调用调用响应 API 以发布来自处理程序的响应。
- 处理错误 – 如果出现错误，请调用调用错误 API。
- 清理 – 释放未使用的资源，将数据发送到其他服务，或在获取下一个事件之前执行其他任务。

Entrypoint

自定义运行时的入口点是一个名为 bootstrap 的可执行文件。引导文件可以是运行时，也可以调用创建运行时的另一个文件。如果部署包的根目录不包含名为 bootstrap 的文件，则 Lambda 将在

函数的层中查找该文件。如果 bootstrap 文件不存在或不是可执行文件，则函数将在调用后返回 Runtime.InvalidEntrypoint 错误。

以下是一个 bootstrap 文件示例，该文件使用捆绑版本的 Node.js 在名为 runtime.js 的单独文件中运行 JavaScript 运行时系统。

Example bootstrap

```
#!/bin/sh
    cd $LAMBDA_TASK_ROOT
    ./node-v11.1.0-linux-x64/bin/node runtime.js
```

在自定义运行时中实施响应流

对于响应流式处理函数，response 和 error 端点的行为略经修改，允许运行时系统将部分响应流式传输到客户端并以区块形式返回负载。有关特定行为的更多信息，请参阅以下内容：

- /runtime/invocation/AwsRequestId/response – 从运行时系统传播 Content-Type 标头以发送到客户端。Lambda 通过 HTTP/1.1 分块传输编码，以区块形式返回响应负载。响应流的最大大小为 20MiB。要将响应流式传输到 Lambda，运行时系统必须：
  - 将 Lambda-Runtime-Function-Response-Mode HTTP 标头设置为 streaming。
  - 将 Transfer-Encoding 标头设置为 chunked。
  - 编写符合 HTTP/1.1 分块传输编码规范的响应。
  - 成功编写响应后，关闭底层连接。
- /runtime/invocation/AwsRequestId/error – 运行时系统可以使用此端点向 Lambda 报告函数或运行时系统错误，Lambda 也接受 Transfer-Encoding 标头。只能在运行时开始发送调用响应之前调用此端点。
- 使用 /runtime/invocation/AwsRequestId/response 中的错误后缀报告中游错误 – 要报告运行时系统开始编写调用响应后发生的错误，运行时系统可以选择附加名为 Lambda-Runtime-Function-Error-Type 和 Lambda-Runtime-Function-Error-Body 的 HTTP 尾随标头。Lambda 将此视为成功响应，并将运行时系统提供的错误元数据转发给客户端。

> **Note**
>
> 要附加尾随标头，运行时必须在 HTTP 请求的开头设置 Trailer 标头值。这是 HTTP/1.1 分块传输编码规范的要求。

- `Lambda-Runtime-Function-Error-Type` – 运行时系统遇到的错误类型。此标头由字符串值组成。Lambda 接受任何字符串，但建议您使用 *<category.reason>* 格式。例如，`Runtime.APIKeyNotFound`。
- `Lambda-Runtime-Function-Error-Body` – 有关错误的 Base64 编码信息。

教程：构建自定义运行时系统

在本教程中，您将创建一个具有自定义运行时的 Lambda 函数。首先，您在函数的部署程序包中包含运行时。然后，您将其迁移到一个您独立于函数管理的层。最后，您通过更新运行时层的基于资源的权限策略来将运行时层与全球共享。

先决条件

本教程假设您对 Lambda 基本操作和 Lambda 控制台有一定了解。如果您还没有了解，请按照 使用控制台创建 Lambda 函数 中的说明创建您的第一个 Lambda 函数。

要完成以下步骤，您需要 AWS CLI 版本 2。在单独的数据块中列出了命令和预期输出：

```
aws --version
```

您应看到以下输出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

对于长命令，使用转义字符 (\) 将命令拆分为多行。

在 Linux 和 macOS 中，可使用您首选的 shell 和程序包管理器。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常与 Lambda 一起使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。本指南中的示例 CLI 命令使用 Linux 格式。如果您使用的是 Windows CLI，则必须重新格式化包含内联 JSON 文档的命令。

您需要一个 IAM 角色来创建 Lambda 函数。该角色需要权限方可将日志发送到 CloudWatch Logs 并访问您的函数使用的AWS服务。如果您没有函数开发的角色，请立即创建一个。

创建执行角色

1. 在 IAM 控制台中，打开 Roles（角色）页面。
2. 选择创建角色。
3. 创建具有以下属性的角色。

   - Trusted entity（可信任的实体）– Lambda。
   - Permissions（权限）– AWSLambdaBasicExecutionRole。
   - Role name（角色名称）– **lambda-role**。

   AWSLambdaBasicExecutionRole 策略具有函数将日志写入 CloudWatch Logs 所需的权限。

创建函数

使用自定义运行时创建 Lambda 函数。此示例包含两个文件：一个运行时系统 `bootstrap` 文件和一个函数处理程序。两个文件都在 Bash 中实施。

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir runtime-tutorial
cd runtime-tutorial
```

2. 创建名为 `bootstrap` 的新文件。这是自定义运行时系统。

   Example bootstrap

```
#!/bin/sh

set -euo pipefail

# Initialization - load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

# Processing
while true
do
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")
```

```
  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

运行时将从部署程序包加载函数脚本。它使用两个变量来查找脚本。LAMBDA_TASK_ROOT 向它告知在何处提取程序包，_HANDLER 包含脚本的名称。

在运行时系统加载函数脚本之后，它会使用运行时系统 API 从 Lambda 检索调用事件，并将事件传递到处理程序，然后将响应发送回 Lambda。为了获取请求 ID，运行时会将来自 API 响应的标头保存到临时文件，并从该文件读取 Lambda-Runtime-Aws-Request-Id 标头。

> **Note**
>
> 运行时还具有其他职责（包括错误处理），并向处理程序提供上下文信息。有关详细信息，请参阅要求。

3. 为函数创建脚本。以下示例脚本将定义一个处理程序函数，该函数将选取事件数据，将该数据记录到 stderr，然后返回它。

   Example function.sh

```
function handler () {
  EVENT_DATA=$1
  echo "$EVENT_DATA" 1>&2;
  RESPONSE="Echoing request: '$EVENT_DATA'"

  echo $RESPONSE
}
```

   runtime-tutorial 目录现在应如下所示：

```
runtime-tutorial
```

```
# bootstrap
# function.sh
```

4. 使文件可执行并将其添加到 .zip 文件存档。这就是部署包。

```
chmod 755 function.sh bootstrap
zip function.zip function.sh bootstrap
```

5. 创建名为 `bash-runtime` 的函数。对于 `--role`，请输入您的 Lambda 执行角色的 ARN。

```
aws lambda create-function --function-name bash-runtime \
--zip-file fileb://function.zip --handler function.handler --runtime
 provided.al2023 \
--role arn:aws:iam::123456789012:role/lambda-role
```

6. 调用函数。

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

应出现如下响应：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

7. 验证响应。

```
cat response.txt
```

应出现如下响应：

```
Echoing request: '{"text":"Hello"}'
```

创建层

要将运行时代码与函数代码分开，请创建一个仅包含运行时的层。层可让您单独开发函数的各个依赖项，而且，通过对多个函数使用相同的层，还可以减少存储使用。有关更多信息，请参阅 使用层管理 Lambda 依赖项。

1. 创建包含 bootstrap 文件的 .zip 文件。

```
zip runtime.zip bootstrap
```

2. 使用 publish-layer-version 命令创建层。

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

这将创建第一个版本的层。

更新函数

要在函数中使用运行时系统层，请将函数配置为使用该层，并从函数中删除运行时代码。

1. 更新函数配置以拉入到层。

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1
```

这会将运行时添加到 /opt 目录中的函数中。为确保 Lambda 使用层中的运行时系统，您必须从函数的部署包中移除 boostrap，如接下来的两个步骤所示。

2. 创建包含函数代码的 .zip 文件。

```
zip function-only.zip function.sh
```

3. 更新函数代码以仅包含处理程序脚本。

```
aws lambda update-function-code --function-name bash-runtime --zip-file fileb://
function-only.zip
```

4. 调用函数以确认它适用于运行时系统层。

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

应出现如下响应：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

5. 验证响应。

```
cat response.txt
```

应出现如下响应：

```
Echoing request: '{"text":"Hello"}'
```

更新运行时

1. 要记录有关执行环境的信息，请更新运行时脚本以输出环境变量。

Example bootstrap

```
#!/bin/sh

set -euo pipefail

# Configure runtime to output environment variables
echo "##  Environment variables:"
env

# Load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"
```

```
# Processing
while true
do
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

2. 创建包含新版本 `bootstrap` 文件的 .zip 文件。

```
zip runtime.zip bootstrap
```

3. 创建新版本的 `bash-runtime` 层。

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

4. 配置函数以使用新版本的层。

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:2
```

共享层

要与其他 AWS 账户 共享层，请在该层的基于资源的策略中添加跨账户权限语句。执行 add-layer-version-permission 命令，并将账户 ID 指定为 `principal`。在每个语句中，您可以向 AWS Organizations 中的单个账户、所有账户或组织授予权限。

以下示例向账户 111122223333 授予访问 `bash-runtime` 层版本 2 的权限。

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
  --output text
```

您应该可以看到类似于如下所示的输出内容：

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

权限仅适用于单个层版本。每次创建新的层版本时都需重复此过程。

清理

删除每个版本的层。

```
aws lambda delete-layer-version --layer-name bash-runtime --version-number 1
aws lambda delete-layer-version --layer-name bash-runtime --version-number 2
```

由于函数包含对版本 2 的层的引用，因此该层仍然存在于 Lambda 中。函数可继续工作，但无法再被配置为使用删除的版本。如果您修改了函数的层的列表，则必须指定新版本或忽略已删除的层。

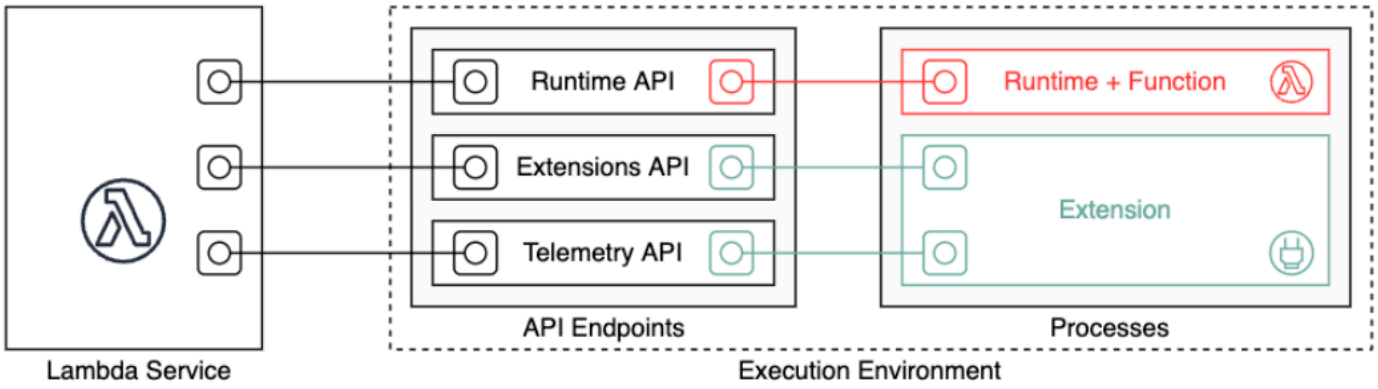
使用 delete-function 命令删除函数。

```
aws lambda delete-function --function-name bash-runtime
```

了解 Lambda 执行环境生命周期

Lambda 在执行环境中调用您的函数，该环境提供一个安全和隔离的运行时环境。执行环境管理运行函数所需的资源。执行环境为函数的运行时以及与函数关联的任何外部扩展提供生命周期支持。

函数的运行时使用运行时 API 与 Lambda 进行通信。扩展使用扩展 API 与 Lambda 进行通信。扩展还可借助遥测 API，从该函数接收日志消息与其他遥测数据。



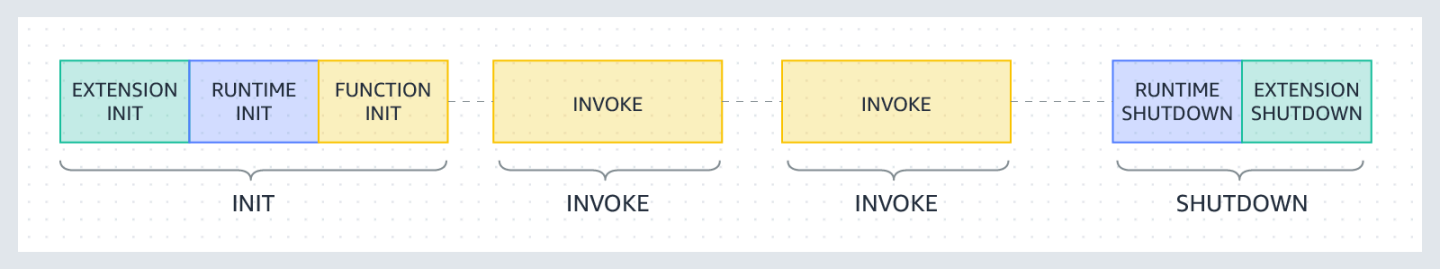
创建 Lambda 函数时，您需要指定配置信息，例如可用内存量和函数允许的最长执行时间。Lambda 使用此信息设置执行环境。

函数的运行时和每个外部扩展都是在执行环境中运行的进程。权限、资源、凭证和环境变量在函数和扩展之间共享。

主题

- Lambda 执行环境生命周期
- 在函数中实现无状态性

Lambda 执行环境生命周期



每个阶段都以 Lambda 发送到运行时和所有注册的扩展的事件开始。运行时和每个扩展通过发送 Next API 请求来指示完成。当运行时和每个扩展完成且没有挂起的事件时，Lambda 会冻结执行环境。

主题

Init 阶段

在 Init 阶段，Lambda 执行三项任务：

- 开启所有扩展 (Extension init)
- 引导运行时 (Runtime init)
- 运行函数的静态代码 (Function init)
- 运行任何 beforeCheckpoint 运行时挂钩（仅限 Lambda SnapStart）

当运行时和所有扩展通过发送 Init API 请求表明它们已准备就绪时，Next 阶段结束。Init 阶段限制为 10 秒。如果所有三个任务都未在 10 秒内完成，Lambda 在第一个函数调用时使用配置的函数超时值重试 Init 阶段。

激活 Lambda SnapStart 后，在您发布一个函数版本时会发生 Init 阶段。Lambda 保存初始化的执行环境的内存和磁盘状态的快照，永久保存加密快照并对其进行缓存以实现低延迟访问。如果您具有 beforeCheckpoint 运行时挂钩，则该代码将在 Init 阶段结束时运行。

> **Note**
>
> 10 秒超时不适用于使用预调配并发或 SnapStart 的函数。对于预调配并发和 SnapStart 函数，初始化代码最长可能会运行 15 分钟。时间限制为 130 秒或配置的函数超时（最大 900 秒），以较高者为准。

使用预置并发时，Lambda 会在您为函数配置 PC 设置时初始化执行环境。Lambda 还确保初始化的执行环境在调用之前始终可用。您会发现函数的调用和初始化阶段之间存在时间差。根据函数的运行时系统和内存配置，在初始化的执行环境中首次调用时可能会发生一些延迟变化。

对于使用按需并发的函数，Lambda 可能会在调用请求之前偶尔初始化执行环境。发生这种情况时，您可能会观察到函数的初始化和调用阶段之间存在时间差。我们建议您不要依赖此行为。

在 Init 阶段失败

如果函数在 `Init` 阶段崩溃或超时，Lambda 会在日志中发出错误信息。`INIT_REPORT`

Example — INIT_REPORT 超时日志

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: timeout
```

Example — INIT_REPORT 扩展失败日志

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: error Error Type:
 Extension.Crash
```

如果 `Init` 阶段成功，除非激活 SnapStart，否则 Lambda 不会发出 `INIT_REPORT` 日志。SnapStart 函数始终会发出 `INIT_REPORT`。有关更多信息，请参阅 监控 Lambda SnapStart。

还原阶段（仅限 Lambda SnapStart）

当您首次调用 SnapStart 函数时，随着该函数的扩展，Lambda 会从永久保存的快照中恢复新的执行环境，而不是从头开始初始化函数。如果您有 `afterRestore()` 运行时挂钩，则代码将在 `Restore` 阶段结束时运行。`afterRestore()` 运行时挂钩执行期间将产生费用。必须加载运行时（JVM），并且 `afterRestore()` 运行时挂钩必须在超时限制（10 秒）内完成。否则，您将收到 SnapStartTimeoutException。`Restore` 阶段完成后，Lambda 将调用函数处理程序（调用阶段）。

在 Restore 阶段失败

如果 `Restore` 阶段失败，Lambda 会在 `RESTORE_REPORT` 日志中发出错误信息。

Example — RESTORE_REPORT 超时日志

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: timeout
```

Example — RESTORE_REPORT 运行时系统钩子失败日志

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: error Error Type: Runtime.ExitError
```

有关 RESTORE_REPORT 日志的更多信息，请参阅 监控 Lambda SnapStart。

调用阶段

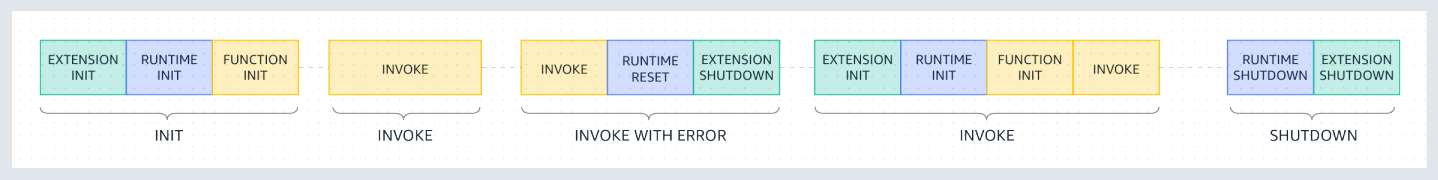
当调用 Lambda 函数以响应 Next API 请求时，Lambda 向运行时和每个扩展发送一个 Invoke 事件。

函数的超时设置限制了整个 Invoke 阶段的持续时间。例如，如果将函数超时设置为 360 秒，则该函数和所有扩展都需要在 360 秒内完成。请注意，没有独立的调用后阶段。持续时间是所有调用时间（运行时 + 扩展）的总和，直到函数和所有扩展完成执行之后才计算。

调用阶段在运行时之后结束，所有扩展都通过发送 Next API 表示它们已完成。

在调用阶段失败

如果 Lambda 函数在 Invoke 阶段崩溃或超时，Lambda 会重置执行环境。下图演示了调用失败时的 Lambda 执行环境行为：



在上图中：

- 第一个阶段是 INIT 阶段，运行没有错误。
- 第二个阶段是 INVOKE 阶段，运行没有错误。
- 假设您的函数在某个时点遇到调用失败的问题（例如函数超时或运行时错误）。标签为 INVOKE WITH ERROR 的第三个阶段演示了这种情况。出现这种情况时，Lambda 服务会执行重置。重置的行为类似于 Shutdown 事件。首先，Lambda 会关闭运行时，然后向每个注册的外部扩展发送一个 Shutdown 事件。该事件包括关闭的原因。如果此环境用于新调用，则 Lambda 会将扩展和运行时与下一次调用一起重新初始化。

  请注意，Lambda 重置不会在下一个初始化阶段之前清除 /tmp 目录内容。这种行为与常规关闭阶段一致。

> **Note**
>
> AWS 目前正在实施对 Lambda 服务的更改。由于这些更改，您可能会看到 AWS 账户 中不同 Lambda 函数发出的系统日志消息和跟踪分段的结构和内容之间存在细微差异。
> 如果您的函数的系统日志配置设置为纯文本，则当您的函数遇到调用失败时，此更改会影响在 CloudWatch Logs 中捕获的日志消息。以下示例显示了新旧格式的日志输出。
> 这些更改将在未来几周内实施，除中国和 GovCloud 区域外，所有 AWS 区域 的函数都将过渡到使用新格式的日志消息和跟踪分段。

Example CloudWatch Logs 日志输出（运行时或扩展崩溃）（旧样式）

```
START RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Version: $LATEST
RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Error: Runtime exited without
 providing a reason
Runtime.ExitError
END RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1
REPORT RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Duration: 933.59 ms Billed
 Duration: 934 ms Memory Size: 128 MB Max Memory Used: 9 MB
```

Example CloudWatch Logs 日志输出（函数超时）（旧样式）

```
START RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21 Version: $LATEST
2024-03-04T17:22:38.033Z b70435cc-261c-4438-b9b6-efe4c8f04b21 Task timed out after
 3.00 seconds
END RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21
REPORT RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21 Duration: 3004.92 ms Billed
 Duration: 3000 ms Memory Size: 128 MB Max Memory Used: 33 MB Init Duration: 111.23
 ms
```

CloudWatch 日志的新格式在 REPORT 行中包含一个附加 `status` 字段。在运行时或扩展崩溃的情况下，REPORT 行还包含一个字段 `ErrorType`。

Example CloudWatch Logs 日志输出（运行时或扩展崩溃）（新样式）

```
START RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd Version: $LATEST
END RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd
```

```
REPORT RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd Duration: 133.61 ms Billed
 Duration: 133 ms Memory Size: 128 MB Max Memory Used: 31 MB Init Duration: 80.00
 ms Status: error Error Type: Runtime.ExitError
```

Example CloudWatch Logs 日志输出（函数超时）（新样式）

```
START RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda Version: $LATEST
END RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda
REPORT RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda Duration: 3016.78 ms Billed
 Duration: 3016 ms Memory Size: 128 MB Max Memory Used: 31 MB Init Duration: 84.00
 ms Status: timeout
```

- 第四个阶段是调用失败后立即进入的 INVOKE 阶段。在这里，Lambda 通过重新运行 INIT 阶段重新初始化环境。此情况称为隐藏初始化。出现隐藏初始化时，Lambda 不会在 CloudWatch Logs 中显式报告额外的 INIT 阶段。相反，您可能会注意到 REPORT 行中的持续时间包括一个额外的 INIT 持续时间 + INVOKE 持续时间。例如，假设您在 CloudWatch 中看到以下日志：

  ```
  2022-12-20T01:00:00.000-08:00 START RequestId: XXX Version: $LATEST
  2022-12-20T01:00:02.500-08:00 END RequestId: XXX
  2022-12-20T01:00:02.500-08:00 REPORT RequestId: XXX Duration: 3022.91 ms
  Billed Duration: 3000 ms Memory Size: 512 MB Max Memory Used: 157 MB
  ```

  在此例中，REPORT 和 START 时间戳的间隔为 2.5 秒。这与报告的持续时间（3022.91 毫秒）不一致，因为它没有考虑 Lambda 执行的额外 INIT（隐藏初始化）。在此例中，您可以推断出实际的 INVOKE 阶段用时 2.5 秒。

  要更深入地了解这种行为，您可以使用 使用遥测 API 访问扩展的实时遥测数据。每当在调用阶段之间出现隐藏初始化时，Telemetry API 都会发出 `INIT_START`、`INIT_RUNTIME_DONE`、`INIT_REPORT` 事件以及 `phase=invoke`。
- 第五个阶段是 SHUTDOWN 阶段，该阶段运行没有错误。

关闭阶段

若 Lambda 即将关闭运行时，它会向每个已注册的外部扩展发送一个 Shutdown 事件。扩展可以使用此时间执行最终清理任务。Shutdown 事件是对 Next API 请求的响应。

持续时间：整个 Shutdown 阶段的上限为 2 秒。如果运行时或任何扩展没有响应，则 Lambda 会通过一个信号 (SIGKILL) 终止它。

在函数和所有扩展完成后，Lambda 维护执行环境一段时间，以预期另一个函数调用。但是，Lambda 每隔几个小时就会终止执行环境，以便进行运行时更新和维护，即使是连续调用的函数亦不例外。您不应假设执行环境将无限期持续。有关更多信息，请参阅 在函数中实现无状态性。

当再次调用该函数时，Lambda 会解冻环境以便重复使用。重复使用执行环境会产生以下影响：

- 在该函数的处理程序方法的外部声明的对象保持已初始化的状态，再次调用函数时提供额外的优化功能。例如，如果您的 Lambda 函数建立数据库连接，而不是重新建立连接，则在后续调用中使用原始连接。建议您在代码中添加逻辑，以便在创建新连接之前检查是否存在连接。
- 每个执行环境都在 `/tmp` 目录中提供 512MB 到 10240MB 之间的磁盘空间（以 1MB 递增）。冻结执行环境时，目录内容会保留，同时提供可用于多次调用的暂时性缓存。您可以添加额外的代码来检查缓存中是否有您存储的数据。有关部署大小限制的更多信息，请参阅Lambda 配额：
- 如果 Lambda 重复使用执行环境，则由 Lambda 函数启动但在函数结束时未完成的后台进程或回调将继续执行。确保代码中的任何后台进程或回调在代码退出前已完成。

在函数中实现无状态性

在编写 Lambda 函数代码时，应视执行环境无状态，即假定执行环境仅在单次调用中存在。Lambda 每隔几个小时就会终止执行环境，以便进行运行时更新和维护，即使是连续调用的函数亦不例外。请在函数启动时初始化任何要求的状态（例如，从 Amazon DynamoDB 表中获取购物车）。在退出之前，请将永久数据更改提交给持久存储，如 Amazon Simple Storage Service（Amazon S3）、DynamoDB 或 Amazon Simple Queue Service（Amazon SQS）。避免依赖跨多个调用的现有数据结构、临时文件或状态，例如计数器或聚合。这样可以确保函数能够独立处理每次调用。

配置 AWS Lambda 函数

了解如何使用 Lambda API 或控制台配置 Lambda 函数的核心功能和选项。

内存

了解如何及何时增加函数内存。

临时存储

了解如何以及何时增加函数的临时存储容量。

超时

了解如何以及何时增加函数的超时值。

环境变量

您可以使您的函数代码可移植，并通过环境变量将密钥存储在函数配置中，从而将该密钥存放在代码之外。

出站联网

您可以将 Lambda 函数与 Amazon VPC 中的 AWS 资源结合使用。通过将函数连接到 VPC，您可以访问私有子网中的资源，例如关系数据库和缓存。

入站联网

您可以使用此接口 VPC 端点来调用您的 Lambda 函数，无需跨越公有 Internet。

文件系统

您可以使用 Lambda 函数将 Amazon EFS 挂载到本地目录。文件系统允许您的函数代码在高并发下安全地访问和修改共享资源。

别名

您可以对客户端进行配置，以便可以使用别名调用特定的 Lambda 函数版本，而不是更新客户端。

版本

通过发布一个函数版本，您可以将代码和配置存储为一个单独的资源，该资源无法更改。

标签

使用标签启用基于属性的访问权限控制（ABAC）、组织 Lambda 函数并使用 AWS Cost Explorer 或 AWS 账单和成本管理服务筛选和生成有关函数的报告。

响应流式处理

配置 Lambda 函数 URL 以将响应负载流式传输回客户端。响应流式处理可通过提高首字节时间（TTFB）性能，使延迟敏感型应用程序受益。这是因为您可以在部分响应可用时将其发送回客户端。此外，您可以使用响应流式处理来构建返回较大负载的函数。

将 Lambda 函数部署为 .zip 文件归档

创建 Lambda 函数时，您可将函数代码打包到部署程序包中。Lambda 支持两种类型的部署程序包：容器镜像和 .zip 文件归档。创建函数的工作流取决于部署包类型。使用 the section called “容器映像” 控制台创建定义为容器镜像的函数。

您可使用 Lambda 控制台和 Lambda API 创建定义为 .zip 文件归档的 Lambda 函数。此外，您还可上载更新的 .zip 文件更改函数代码。

> **Note**
>
> 您无法更改现有函数的部署包类型（.zip 或容器映像）。例如，您无法将容器映像函数转换为使用 .zip 文件归档。您必须创建新函数。

主题

创建函数

创建定义为 .zip 文件归档的 Lambda 函数时，请选择代码模板、语言版本及函数的执行角色。您可以在 Lambda 创建函数后添加函数代码。

创建函数

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择 Create function（创建函数）。
3. 选择 Author from scratch（从头开始创作）或者 Use a blueprint（使用蓝图）创建函数。
4. 在 Basic information（基本信息）中，执行以下操作：

a. 对于 Function name（函数名称），输入函数名称。函数名称的长度限制为 64 个字符。

b. 对于Runtime（运行时），请选择函数使用的语言版本。

c. （可选）对于架构，选择要用于函数的指令集架构。默认架构为 x86_64。为您的函数构建部署包时，请确保它与此指令集架构兼容。

5. （可选）在 Permissions（权限）下，展开 Change default execution role（更改默认执行角色）。您可以使用现有角色，也可以创建一个执行角色。

6. （可选）展开 Advanced settings（高级设置）。您可以为函数选择代码签名配置。您还可以为要访问的函数配置 (Amazon VPC)。

7. 选择 Create function（创建函数）。

Lambda 将创建新函数。现在，您可以使用控制台添加函数代码并配置其他函数参数和特性。有关代码部署说明，请参阅函数使用的运行时的处理程序页面。

Node.js

使用 .zip 文件归档部署 Node.js Lambda 函数

Python

将 .zip 文件归档用于 Python Lambda 函数

Ruby

使用 .zip 文件归档部署 Ruby Lambda 函数

Java

使用 .zip 或 JAR 文件归档部署 Java Lambda 函数

Go

使用 .zip 文件归档部署 Go Lambda 函数

C#

使用 .zip 文件归档构建和部署 C# Lambda 函数

PowerShell

使用 .zip 文件归档部署 PowerShell Lambda 函数

使用控制台代码编辑器

控制台将使用单个源文件创建一个 Lambda 函数。对于脚本语言，您可以在内置代码编辑器中编辑此文件并添加更多文件。要保存您的更改，请选择 Save（保存）。然后，要运行代码，请选择 Test（测试）。

保存函数代码时，Lambda 控制台会创建一个 .zip 文件归档部署包。在控制台外部开发函数代码时（使用 IDE），您需要创建部署程序包将代码上载到 Lambda 函数。

更新函数代码

对于脚本语言（Node.js、Python 及 Ruby），您可以在嵌入式编辑器中编辑函数代码。如果代码大于 3MB，或者如果需要添加库，或对于编辑器不支持的语言（Java、Go、C#），您必须将函数代码以 .zip 归档上载。如果 .zip 文件归档小于 50 MB，则可以从本地计算机上传 .zip 文件归档。如果文件大于 50MB，请将文件从 Simple Storage Service (Amazon S3) 存储桶上载到函数。

将函数代码以 .zip 归档上载

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择要更新的函数，然后选择 Code（代码）选项卡。
3. 在 Code source（代码源）下，选择 Upload from（上载自）。
4. 选择 .zip file (.zip 文件)，然后选择 Upload file (上载文件)。
   - 在文件选择器中，选择新映像版本，然后依次选择 Open (打开)、Save (保存)。
5. （步骤 4 的替代方案）选择 Amazon S3 location（Simple Storage Service (Amazon S3) 位置）。
   - 在文本框中，输入 .zip 文件归档的 S3 链接 URL，然后选择 Save（保存）。

更改运行时

如果您更新函数配置以使用新的运行时版本，则可能需要更新函数代码才能与新的运行时版本兼容。如果您将函数配置更新为使用其他运行时，则必须提供与运行时和架构兼容的新函数代码。有关如何为函数代码创建部署包的说明，请参阅函数使用的运行时的处理程序页面。

Node.js 20、Python 3.12、Java 21、.NET 8、Ruby 3.3 及更高版本的基础映像都基于 Amazon Linux 2023 最小容器映像。早期的基础映像使用 Amazon Linux 2。与 Amazon Linux 2 相比，AL2023 具有

多项优势，包括较小的部署占用空间以及 glibc 等更新版本的库。有关更多信息，请参阅 AWS 计算博客上的AWS Lambda 的 Amazon Linux 2023 运行时系统简介。

更改运行时

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择要更新的函数，然后选择 Code（代码）选项卡。
3. 向下滚动至位于代码编辑器下方的 Runtime settings（运行时设置）部分。
4. 选择编辑。
    a. 对于 Runtime（运行时），请选择运行时标识符。
    b. 对于 Handler（处理程序），请为您的函数指定文件名和处理程序。
    c. 对于架构，选择要用于您的函数的指令集架构。
5. 选择保存。

更改架构

在更改指令集架构之前，您需要确保函数的代码与目标架构兼容。

如果您使用 Node.js、Python 或 Ruby 并在嵌入式编辑器中编辑函数代码，则现有代码可以在不修改的情况下运行。

但是，如果您使用 .zip 文件归档部署包提供函数代码，则必须准备一个新的 .zip 文件归档，针对目标运行时和指令集架构正确编译和构建此归档。有关说明，请参阅函数运行时的处理程序页面。

更改指令集架构

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择要更新的函数，然后选择 Code（代码）选项卡。
3. 在 Runtime settings（运行时设置）中，选择 Edit（编辑）。
4. 对于架构，选择要用于您的函数的指令集架构。
5. 选择保存。

使用 Lambda API

要创建和配置使用 .zip 文件归档的函数，请使用以下 API 操作：

- CreateFunction
- UpdateFunctionCode
- UpdateFunctionConfiguration

AWS CloudFormation

您可以使用 AWS CloudFormation 创建使用 .zip 文件归档的 Lambda 函数。在 AWS CloudFormation 模板中，`AWS::Lambda::Function` 资源将指定 Lambda 函数。有关 `AWS::Lambda::Function` 资源中属性的描述，请参阅 AWS CloudFormation 用户指南中的 AWS::Lambda::Function。

在 `AWS::Lambda::Function` 资源中，设置以下属性以创建定义为 .zip 文件归档的函数：

- AWS::Lambda::Function
  - PackageType – 设置为 `Zip`。
  - 代码 – 在 `S3Bucket` 和 `S3Key` 字段中输入 Amazon S3 存储桶名称和 .zip 文件名。对于 Node.js 或 Python，您可以提供 Lambda 函数的内联源代码。
  - 运行时 – 设置运行时值。
  - 架构 - 将架构值设置为 `arm64` 以使用 AWS Graviton2 处理器。默认情况下，架构值为 `x86_64`。

使用容器映像创建 Lambda 函数

您的 AWS Lambda 函数代码由脚本或编译的程序及其依赖项组成。您可以使用部署程序包将函数代码部署到 Lambda。Lambda 支持两种类型的部署程序包：容器镜像和 .zip 文件归档。

有三种方法可以为 Lambda 函数构建容器映像：

- 使用 Lambda 的 AWS 基本映像

  AWS 基本映像会预加载一个语言运行时系统、一个用于管理 Lambda 和函数代码之间交互的运行时系统接口客户端，以及一个用于本地测试的运行时系统接口仿真器。

- 使用 AWS 仅限操作系统的基础镜像

  AWS 仅限操作系统的运行时系统包含 Amazon Linux 发行版和运行时系统接口模拟器。这些镜像通常用于为编译语言（例如 Go 和 Rust）以及 Lambda 未提供基础映像的语言或语言版本（例如 Node.js 19）创建容器镜像。您也可以使用仅限操作系统的基础映像来实施自定义运行时系统。要使映像与 Lambda 兼容，则必须在映像中包含适用于您的语言的运行时系统接口客户端。

- 使用非 AWS 基本映像

  您还可以使用其他容器注册表的备用基本映像，例如 Alpine Linux 或 Debian。您还可以使用您的组织创建的自定义映像。要使映像与 Lambda 兼容，则必须在映像中包含适用于您的语言的运行时系统接口客户端。

> **Tip**
>
> 要缩短 Lambda 容器函数激活所需的时间，请参阅 Docker 文档中的使用多阶段构建。要构建高效的容器映像，请遵循编写 Dockerfiles 的最佳实践。

要从容器映像创建 Lambda 函数，请在本地构建映像，然后将其上传到 Amazon Elastic Container Registry（Amazon ECR）存储库。然后，在您创建该函数时指定存储库 URI。Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

> **Note**
>
> 对于容器映像，Lambda 不支持 Amazon ECR FIPS 端点。如果您的存储库 URI 包含 `ecr-fips`，则表示您使用的是 FIPS 端点。示例：`111122223333.dkr.ecr-fips.us-east-1.amazonaws.com`。

本页面介绍了创建与 Lambda 兼容的容器映像的基本映像类型和要求。

> **Note**
>
> 您无法更改现有函数的部署包类型（.zip 或容器映像）。例如，您无法将容器映像函数转换为使用 .zip 文件归档。您必须创建新函数。

主题

要求

安装 AWS CLI 版本 2 和 Docker CLI。此外，请注意以下要求：

- 容器映像必须实施 将 Lambda 运行时 API 用于自定义运行时。AWS开源运行时接口客户端实施 API。您可以将运行时接口客户端添加到首选基本映像中以使其与 Lambda 兼容。
- 容器映像必须能够在只读文件系统上运行。您的函数代码可以访问具有介于 512 MB 至 10,240 MB（以 1 MB 为增量）的存储空间的可写 `/tmp` 目录。
- 默认 Lambda 用户必须能够读取运行函数代码所需的所有文件。Lambda 通过定义具有最低权限的默认 Linux 用户来遵循安全最佳实践。这意味着您无需在 Dockerfile 中指定 USER。验证您的应用程序代码是否不依赖于其他 Linux 用户被限制运行的文件。

- Lambda 仅支持基于 Linux 的容器映像。
- Lambda 提供多架构基础映像。但是，您为函数构建的映像必须仅针对其中一个架构。Lambda 不支持使用多架构容器映像的函数。

使用 Lambda 的 AWS 基本映像

您可以使用 Lambda 的其中一个 AWS 基本映像，为函数代码构建容器映像。基本镜像预加载了语言运行时和在 Lambda 上运行容器镜像所需的其他组件。将函数代码和依赖项添加到基本镜像中，然后将其打包为容器镜像。

AWS定期为 Lambda 的AWS基本映像提供更新。如果 Dockerfile 在 FROM 属性中包含映像名称，则 Docker 客户端将从 Amazon ECR 存储库中提取最新版本的映像。要使用更新后的基本映像，必须重建容器映像并更新函数代码。

Node.js 20、Python 3.12、Java 21、.NET 8、Ruby 3.3 及更高版本的基础映像都基于 Amazon Linux 2023 最小容器映像。早期的基础映像使用 Amazon Linux 2。与 Amazon Linux 2 相比，AL2023 具有多项优势，包括较小的部署占用空间以及 `glibc` 等更新版本的库。

基于 AL2023 的映像使用 `microdnf`（符号链接为 `dnf`）作为软件包管理器，而不是 Amazon Linux 2 中的默认软件包管理器 `yum`。`microdnf` 是 `dnf` 的独立实现。有关基于 AL2023 的映像中已包含软件包的列表，请参阅 Comparing packages installed on Amazon Linux 2023 Container Images 中的 Minimal Container 列。有关 AL2023 和 Amazon Linux 2 之间区别的更多信息，请参阅 AWS 计算博客上的 Introducing the Amazon Linux 2023 runtime for AWS Lambda。

> **Note**
>
> 要在本地运行基于 AL2023 的映像，包括使用 AWS Serverless Application Model（AWS SAM），您必须使用 Docker 版本 20.10.10 或更高版本。

要使用 AWS 基本映像构建容器映像，请选择您的首选语言的说明：

- Node.js
- TypeScript（使用 Node.js 基本映像）
- Python
- Java
- Go

- .NET
- Ruby

使用 AWS 仅限操作系统的基础镜像

AWS 仅限操作系统的运行时系统包含 Amazon Linux 发行版和运行时系统接口模拟器。这些镜像通常用于为编译语言（例如 Go 和 Rust）以及 Lambda 未提供基础映像的语言或语言版本（例如 Node.js 19）创建容器镜像。您也可以使用仅限操作系统的基础映像来实施自定义运行时系统。要使映像与 Lambda 兼容，则必须在映像中包含适用于您的语言的运行时系统接口客户端。

标签	运行时	操作系统	Dockerfile	淘汰
al2023	仅限操作系统的运行时系统	Amazon Linux 2023	GitHub 上用于仅限操作系统的运行时系统的 Dockerfile	未计划
al2	仅限操作系统的运行时系统	Amazon Linux 2	GitHub 上用于仅限操作系统的运行时系统的 Dockerfile	未计划

Amazon Elastic Container Registry Public Gallery：gallery.ecr.aws/lambda/provided

使用非 AWS 基本映像

Lambda 支持符合以下映像清单格式之一的任何映像：

- Docker Image Manifest V2，Schema 2（与 Docker 版本 1.10 和更新版本配合使用）
- Open Container Initiative (OCI) 规范（v1.0.0 和更高版本）

Lambda 支持的最大未压缩图像大小为 10GB，包括所有层。

> **Note**
>
> 要使映像与 Lambda 兼容，则必须在映像中包含适用于您的语言的运行时系统接口客户端。

运行时接口客户端

如果使用仅限操作系统的基础映像或者备用基础映像，则必须在映像中包括运行时系统接口客户端。该运行时系统接口客户端必须扩展 将 Lambda 运行时 API 用于自定义运行时，它管理 Lambda 与函数代码之间的交互。AWS 为以下语言提供开源运行时系统接口客户端：

- Node.js
- Python
- Java
- .NET
- Go
- Ruby
- Rust – Rust 运行时系统客户端是实验性程序包。它随时可能更改，并且仅用于评估目的。

如果您使用的语言没有 AWS 提供的运行时系统接口客户端，则必须创建您自己的运行时系统接口客户端。

Amazon ECR 权限

在从容器映像创建 Lambda 函数之前，您必须在本地构建映像并将其上传到 Amazon ECR 存储库。在您创建该函数时，指定 Amazon ECR 存储库 URI。

确保创建函数的用户或角色的权限包含 `GetRepositoryPolicy` 和 `SetRepositoryPolicy`。

例如，使用 IAM 控制台创建具有以下策略的角色：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "ecr:SetRepositoryPolicy",
        "ecr:GetRepositoryPolicy"
      ],
      "Resource": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world"
    }
  ]
```

```
}
```

Amazon ECR 存储库策略

要将相同账户中的函数用作 Amazon ECR 中的容器映像，您可以向 Amazon ECR 存储库策略添加 `ecr:BatchGetImage` 和 `ecr:GetDownloadUrlForLayer` 权限。以下示例显示最小策略：

```
{
        "Sid": "LambdaECRImageRetrievalPolicy",
        "Effect": "Allow",
        "Principal": {
          "Service": "lambda.amazonaws.com"
        },
        "Action": [
          "ecr:BatchGetImage",
          "ecr:GetDownloadUrlForLayer"
        ]
    }
```

有关更多 Amazon ECR 存储库权限的信息，请参阅《Amazon Elastic Container Registry 用户指南》中的私有存储库策略。

如果 Amazon ECR 存储库中不包含这些权限，Lambda 会为 `ecr:BatchGetImage` 和 `ecr:GetDownloadUrlForLayer` 添加容器镜像存储库权限。仅当 Lambda 的主要调用具有 `ecr:getRepositoryPolicy` 和 `ecr:setRepositoryPolicy` 权限时，Lambda 才能添加这些权限。

要查看或编辑您的 Amazon ECR 存储库权限，请参阅《Amazon Elastic Container Registry 用户指南》中的设置私有存储库策略声明。

Amazon ECR 跨账户权限

相同区域中的不同账户可以创建一个使用您的账户拥有的容器镜像的函数。在以下示例中，您的 Amazon ECR 存储库权限策略需要以下语句，才能向账号 123456789012 授予访问权限。

- CrossAccountPermission – 允许账号 123456789012 创建和更新使用此 ECR 存储库中的镜像的 Lambda 函数。
- LambdaECRImageCrossAccountRetrievalPolicy – 如果在延迟期内未调用 Lambda，则 Lambda 最终会将函数的状态设置为不活动。此语句是必需的，以便 Lambda 可以检索容器镜像，代表 123456789012 拥有的函数进行优化和缓存。

Example – 添加对存储库的跨账户权限

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CrossAccountPermission",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:root"
      }
    },
    {
      "Sid": "LambdaECRImageCrossAccountRetrievalPolicy",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Condition": {
        "StringLike": {
          "aws:sourceARN": "arn:aws:lambda:us-east-1:123456789012:function:*"
        }
      }
    }
  ]
}
```

要授予对多个账户的访问权限，您可以在 `CrossAccountPermission` 策略中将该账户 ID 添加到主体列表，并在 `LambdaECRImageCrossAccountRetrievalPolicy` 中将该账户 ID 添加到条件评估列表。

如果您在 AWS Organization 中使用多个账户，我们建议您在 ECR 权限策略中枚举每个账户 ID。此方法遵循在 IAM policy 中设置窄权限的 AWS 安全最佳实践。

除了 Lambda 权限外，创建函数的用户或角色还必须拥有 BatchGetImage 和 GetDownloadUrlForLayer 权限。

函数周期

上传新的或更新的容器镜像后，Lambda 会在函数可以处理调用之前优化镜像。优化过程可能需要几秒钟的时间。函数一直处于 Pending 状态，直到过程完成。然后函数将转换为 Active 状态。在状态为 Pending 时，您可以调用该函数，但对该函数的其他操作会失败。映像更新过程中发生的调用将运行上一个映像中的代码。

如果函数在数周内未被调用，则 Lambda 回收其优化版本，函数将转换为 Inactive 状态。要重新激活函数，必须调用它。Lambda 拒绝第一次调用，函数进入 Pending 状态，直到 Lambda 重新优化镜像。然后函数返回到 Active 状态。

Lambda 定期从 Amazon ECR 存储库获取关联的容器映像。如果相应的容器镜像不再存在于 Amazon ECR 上或权限已撤消，则函数将进入 Failed 状态，并且 Lambda 将对任何函数调用返回失败。

您可以使用 Lambda API 获取函数状态的相关信息。有关更多信息，请参阅 Lambda 函数状态。

配置 Lambda 函数内存

Lambda 根据配置的内存量按比例分配 CPU 功率。内存是在运行时可用于 Lambda 函数的内存量。请使用内存设置增加分配给函数的内存和 CPU 处理能力。您可以以 1MB 的增量将内存配置在 128MB 与 10240MB 之间。大小为 1769 MB 时，函数相当于一个 vCPU（每秒一个 vCPU 秒的积分）的处理能力。

本页介绍如何以及何时更新 Lambda 函数的内存设置。

Sections

确定 Lambda 函数的适当内存设置

内存是控制函数性能的主要杠杆。默认设置 128MB 是可能的最低设置。建议您仅将 128MB 用于简单的 Lambda 函数，例如用于转换事件并将其路由到其他 AWS 服务的函数。更高的内存分配可以提高函数的性能，这些函数使用导入的库、Lambda 层、Amazon Simple Storage Service（Amazon S3）或 Amazon Elastic File System（Amazon EFS）。按比例增加更多内存会增加 CPU 的容量，从而提高可用的总体计算能力。如果函数受限于 CPU、网络或内存，则增加内存设置可以显著提高其性能。

要为您的函数查找适合的内存配置，建议使用开源 AWS Lambda 功率调谐项目。此工具使用 AWS Step Functions 在不同的内存分配下运行 Lambda 函数的多个并行版本并衡量性能。输入函数在您的 AWS 账户中运行，以执行实时 HTTP 调用和 SDK 交互，从而衡量实时制作场景中可能的性能。您也可以实施 CI/CD 流程，以使用此工具自动衡量所部署的新函数的性能。

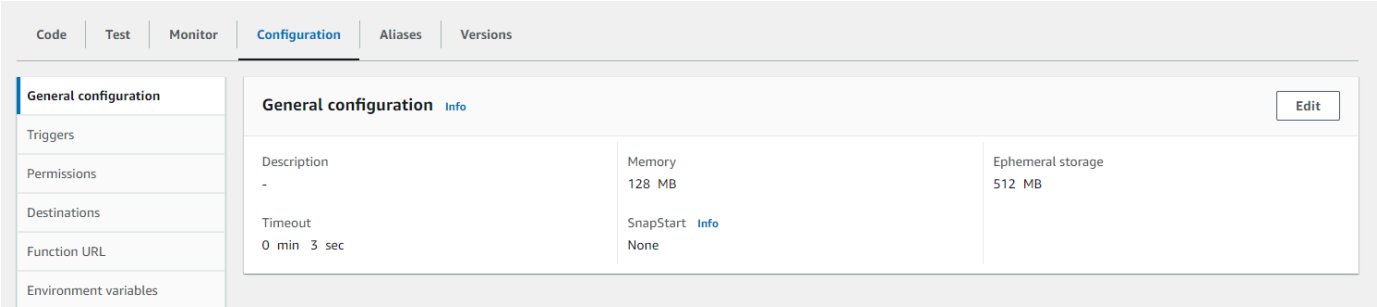
配置函数内存（控制台）

您可以在 Lambda 控制台中配置函数内存。

要更新函数内存

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择函数。

3. 选择配置选项卡，然后选择常规配置。



4. 在常规配置下，选择编辑。
5. 对于内存，设置一个从 128MB 到 10240MB 的值。
6. 选择保存。

配置函数内存（AWS CLI）

您可以使用 update-function-configuration 命令来配置函数的内存。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --memory-size 1024
```

配置函数内存（AWS SAM）

您可以使用 AWS Serverless Application Model 来配置函数的内存。更新 `template.yaml` 文件中的 MemorySize 属性，然后运行 sam deploy。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 1024
```

```
    # Other function properties...
```

接受函数内存推荐（控制台）

如果您在 AWS Identity and Access Management (IAM) 中拥有管理员权限，则可以选择加入以接收来自 AWS Compute Optimizer 的 Lambda 函数内存设置推荐。有关选择加入您账户或企业的内存推荐的说明，请参阅 AWS Compute Optimizer 用户指南中的选择加入您的账户。

> **Note**
>
> Compute Optimizer 只支持使用 x86_64 架构的函数。

当您选择加入并且 Lambda 函数符合 Compute Optimizer 要求时，您可以在常规配置中的 Lambda 控制台的 Compute Optimizer 中查看和接受函数内存推荐。

为 Lambda 函数配置短暂存储

Lambda 为 /tmp 目录中的函数提供短暂存储。此存储是临时的，并且对于每个执行环境都是独一无二的。您可以使用短暂存储设置来控制分配给函数的短暂存储量。您可以以 1MB 的增量将短暂存储配置在 512MB 与 10240MB 之间。存储在 /tmp 中的所有数据都是使用由 AWS 管理密钥进行静态加密。

本页介绍常见应用场景以及如何更新 Lambda 函数的短暂存储。

Sections

增加短暂存储的常见应用场景

以下是几个可从增加短暂存储中受益的常见应用场景：

- 提取-转换-加载（ETL）作业：当您的代码执行中间计算或下载其他资源以完成处理时，增加短暂存储。更多的临时空间允许在 Lambda 函数中运行更复杂的 ETL 作业。
- 机器学习（ML）推理：许多推理任务依赖于大型参考数据文件，包括库和模型。有了更多短暂存储，您可以从 Amazon Simple Storage Service（Amazon S3）将更大的模型下载到 /tmp 并将其用于处理。
- 数据处理：对于从 Amazon S3 下载对象以响应 S3 事件的工作负载，更多的 /tmp 空间使得无需使用内存中处理即可处理较大的对象。创建 PDF 或处理媒体的工作负载也将从更多短暂的存储中获益。
- 图形处理：图像处理是基于 Lambda 的应用程序的常见应用场景。对于处理大型 TIFF 文件或卫星图像的工作负载，更多的短暂存储可以更轻松地在 Lambda 中使用库和执行计算。

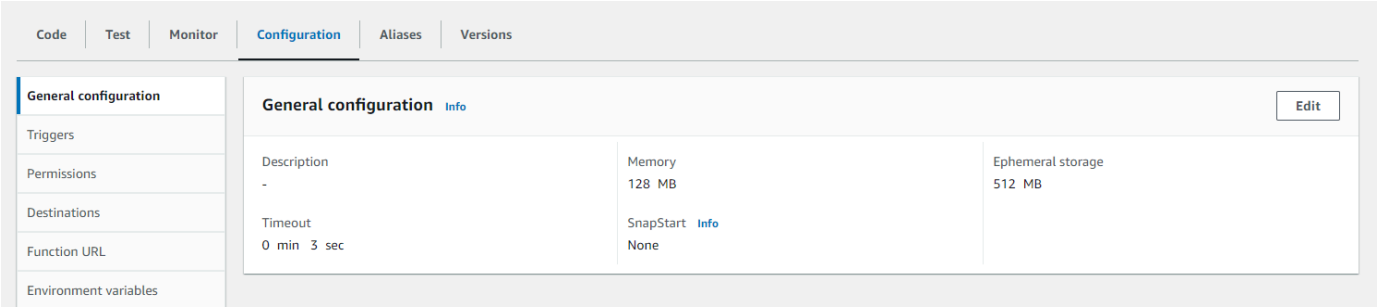
配置短暂存储（控制台）

您可以在 Lambda 控制台中配置临时存储。

要修改函数的临时存储

1. 打开 Lamba 控制台的 Functions（函数）页面。

2. 选择函数。
3. 选择配置选项卡，然后选择常规配置。



4. 在常规配置下，选择编辑。
5. 对于短暂存储，以 1MB 的增量将值设置在 512MB 与 10240MB 之间。
6. 选择保存。

配置短暂存储（AWS CLI）

您可以使用 update-function-configuration 命令来配置短暂存储。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --ephemeral-storage '{"Size": 1024}'
```

配置短暂存储（AWS SAM）

您可以使用 AWS Serverless Application Model 为您的函数配置短暂存储。更新 `template.yaml` 文件中的 EphemeralStorage 属性，然后运行 sam deploy。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
```

```
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs20.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      # Other function properties...
```

为 Lambda 函数选择和配置指令集架构

Lambda 函数的指令集架构决定了 Lambda 用于运行该函数的计算机处理器类型。Lambda 提供了多种指令集架构以供选择：

- arm64 - 64 位 ARM 架构，适用于 AWS Graviton2 处理器。
- x86_64 - 64 位 x86 架构，适用于基于 x86 的处理器。

> **Note**
>
> arm64 架构在大多数 AWS 区域 中都可用。有关更多信息，请参阅AWS Lambda 定价。在内存价格表中，选择 Arm Price（Arm 定价）选项卡，然后打开 Region（区域）下拉列表，查看哪些 AWS 区域 支持使用 Lambda 的 arm64。
> 有关如何使用 arm64 架构创建函数的示例，请参阅 AWS Graviton2 处理器提供支持的 AWS Lambda 函数。

主题

使用 arm64 架构的优势

与在 x86_64 架构上运行的同等函数相比，使用 arm64 架构的 Lambda 函数（AWS Graviton2 处理器）价格更低廉且性能更出色。可考虑将 arm64 用于计算密集型应用程序，例如高性能计算、视频编码和模拟工作负载。

Graviton2 CPU 使用 Neoverse N1 内核，并支持 Armv8.2（包括 CRC 和加密扩展）以及其他几个架构扩展。

Graviton2 通过为每个 vCPU 提供更大的二级缓存来缩短内存读取时间，从而提高了 Web 和移动后端、微服务和数据处理系统的延迟性能。Graviton2 还提供了改进的加密性能，并支持可改善基于 CPU 的机器学习推断延迟的指令集。

有关 AWS Graviton2 的更多信息，请参阅 AWS Graviton 处理器。

迁移到 arm64 架构的要求

当您选择要迁移到 arm64 架构的 Lambda 函数时，为确保顺利迁移，请确保函数满足以下要求：

- 部署包只包含您控制的开源组件和源代码，从而您可以对迁移进行任何必要的更新。
- 如果函数代码包含第三方依赖项，则每个库或软件包都提供 arm64 版本。

函数代码与 arm64 架构的兼容性

您的 Lambda 函数代码必须与函数的指令集架构兼容。在将函数迁移到 arm64 架构之前，请注意有关当前函数代码的以下几点：

- 如果您使用嵌入式代码编辑器添加函数代码，那么您的代码可能不作任何修改地在任一架构上运行
- 如果您上传了函数代码，则必须上传与目标架构兼容的新代码。
- 如果您的函数使用层，则必须检查每个层以确保它与新架构兼容。如果层不兼容，请编辑函数以将当前层版本替换为兼容的层版本。
- 如果您的函数使用 Lambda 扩展，则必须检查每个扩展以确保它与新架构兼容。
- 如果您的函数使用容器映像部署包类型，则必须创建与该函数架构兼容的新容器映像。

如何迁移到 arm64 架构

要将 Lambda 函数迁移到 arm64 架构，我们建议执行以下步骤：

1. 为应用程序或工作负载构建依赖项列表。常见的依赖项包括：
   - 该函数使用的所有库和软件包。
   - 用于构建、部署和测试函数的工具，例如编译器、测试套件、持续集成和持续交付 (CI/CD) 管道、调配工具和脚本。
   - 您用于监控生产中函数的 Lambda 扩展和第三方工具。
2. 对于每个依赖项，请检查版本，然后检查 arm64 版本是否可用。
3. 构建环境以迁移应用程序
4. 引导启动应用程序。
5. 测试和调试应用程序。

6. 测试 arm64 函数的性能。将此性能与 x86_64 版本进行比较。
7. 更新您的基础设施管道以支持 arm64 Lambda 函数。
8. 将部署投入生产阶段。

   例如，使用别名路由配置在函数的 x86 和 arm64 版本之间拆分流量，并比较性能和延迟。

有关如何为 arm64 架构创建代码环境的更多信息，包括 Java、Go、.NET 和 Python 的语言特定信息，请参阅AWS Graviton 入门 GitHub 存储库。

配置指令集架构

您可以使用 Lambda 控制台、AWS 开发工具包、AWS Command Line Interface (AWS CLI) 或 AWS CloudFormation 为新的和现有 Lambda 函数配置指令集架构。按照以下步骤从控制台对现有 Lambda 函数的指令集架构进行更改。

1. 打开 Lamba 控制台的函数页面。
2. 选择要为其配置指令集架构的函数的名称。
3. 在主代码选项卡上，对于运行时设置部分，选择编辑。
4. 对于架构，选择要用于您的函数的指令集架构。
5. 选择保存。

配置 Lambda 函数超时

在超时之前，Lambda 会在设定的时间内运行您的代码。超时是 Lambda 函数可以运行的最大时间量（以秒为单位）。此设置的默认值为 3 秒，但您可以按照 1 秒增量调整此值，最大值为 900 秒（15 分钟）。

本页介绍如何以及何时更新 Lambda 函数的超时设置。

Sections

确定 Lambda 函数的适当超时值

如果超时值接近函数的平均持续时间，则该函数意外超时的风险较高。函数的持续时间可能因数据传输和处理量以及与该函数交互的任何服务的延迟而不同。导致超时的一些常见原因包括：

- Amazon Simple Storage Service（Amazon S3）下载量大于平均水平或需要的时间更长。
- 一个函数向另一项服务发出请求，这需要更长的时间才能响应。
- 提供给函数的参数要求函数具有更高的计算复杂度，这会导致调用花费更长的时间。

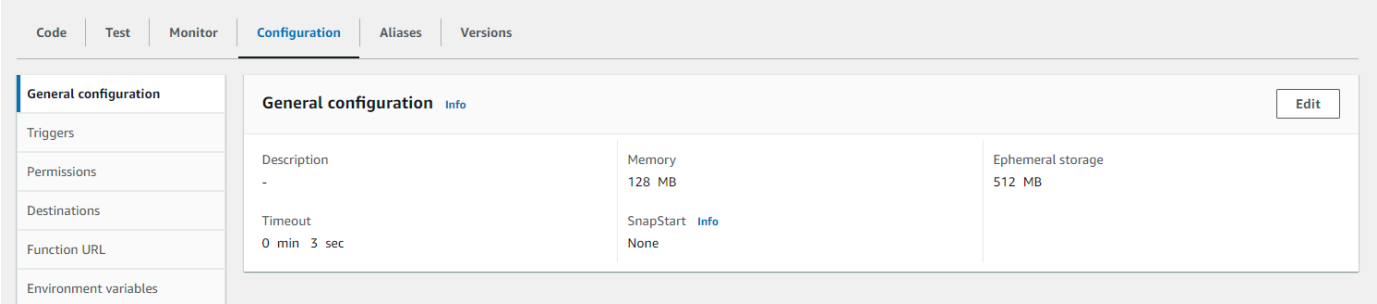
在测试应用程序时，请确保您的测试准确反映数据的大小和数量以及真实的参数值。为方便起见，测试通常使用少量样本，但您应该在您的工作负载合理预期值的上限使用数据集。

配置超时（控制台）

您可以在 Lambda 控制台中配置函数的超时时间。

修改函数的超时时间

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择配置选项卡，然后选择常规配置。



4. 在常规配置下，选择编辑。
5. 对于超时，设置一个介于 1 到 900 秒（15 分钟）之间的值。
6. 选择保存。

配置超时（AWS CLI）

您可以使用 update-function-configuration 命令来配置超时值，以秒为单位。以下示例命令将函数超时增加到 120 秒（2 分钟）。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --timeout 120
```

配置超时（AWS SAM）

您可以使用 AWS Serverless Application Model 来配置函数的超时值。更新 `template.yaml` 文件中的 Timeout 属性，然后运行 sam deploy。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
```

```
      Timeout: 120
      # Other function properties...
```

使用 Lambda 环境变量配置代码中的值

您可以使用环境变量来调整函数的行为，而无需更新代码。环境变量是存储在函数的版本特定配置中的一对字符串。Lambda 运行时使环境变量可用于您的代码，并设置其他环境变量，这些变量包含有关函数和调用请求的信息。

> **Note**
>
> 为了提高安全性，建议使用 AWS Secrets Manager 而非环境变量来存储数据库凭证和其他敏感信息，例如 API 密钥或授权令牌。有关更多信息，请参阅使用 AWS Secrets Manager 创建和管理密钥。

环境变量不会在调用函数之前评估。您定义的任何值都将被视为文字字符串，且不会被展开。在函数代码中执行变量估算。

您可以使用 Lambda 控制台、AWS Command Line Interface（AWS CLI）、AWS Serverless Application Model（AWS SAM）或使用 AWS SDK 在 Lambda 中配置环境变量。

Console

您可以在函数的未发布版本上定义环境变量。发布一个版本时，会锁定该版本的环境变量以及其他特定于版本的配置设置。

您可以通过定义键和值为函数创建环境变量。您的函数使用键名来检索环境变量的值。

在 Lambda 控制台中设置环境变量

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration（配置），然后选择 Environment variables（环境变量）。
4. 在 Environment variables（环境变量）下，选择 Edit（编辑）。
5. 选择 Add environment variable (添加环境变量)。
6. 输入密钥和值。

   要求

   - 密钥以字母开头，并且至少为两个字符。
   - 键仅包含字母、数字和下划线字符 (_)。

- Lambda 不会保留密钥。
- 所有环境变量的总大小不超过 4 KB。

7. 选择 Save（保存）。

在控制台代码编辑器中生成环境变量列表

您可以在 Lambda 代码编辑器中生成环境变量列表。这是在编码时快速引用环境变量的方法。

1. 选择节点选项卡。
2. 选择环境变量选项卡。
3. 依次选择工具、显示环境变量。

在控制台代码编辑器中列出时，环境变量会保持加密状态。如果您为传输中的加密启用了加密帮助程序，这些设置则会保持不变。有关更多信息，请参阅 保护 Lambda 环境变量。

环境变量列表是只读的，仅在 Lambda 控制台上可用。当您下载函数的.zip 文件存档时，此文件并不包括在内；您也无法通过上传此文件来添加环境变量。

AWS CLI

以下示例在名为 `my-function` 的函数上设置两个环境变量。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --environment "Variables={BUCKET=amzn-s3-demo-bucket,KEY=file.txt}"
```

使用 `update-function-configuration` 命令应用环境变量时，会替换 `Variables` 结构的整个内容。要在添加新环境变量时保留现有环境变量，请在请求中包含所有现有值。

要获取当前配置，请使用 `get-function-configuration` 命令。

```
aws lambda get-function-configuration \
  --function-name my-function
```

您应看到以下输出：

```
{
    "FunctionName": "my-function",
```

```
    "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::111122223333:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    "RevisionId": "0894d3c1-2a3d-4d48-bf7f-abade99f3c15",
    ...
}
```

您可以将 `get-function-configuration` 输出中的修订版 ID 作为参数传递给 `update-function-configuration`。此举可确保这些值在您读取配置和更新配置之间不会发生变化。

要配置函数的加密密钥，请设置 KMSKeyARN 选项。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --kms-key-arn arn:aws:kms:us-east-2:111122223333:key/055efbb4-xmpl-4336-ba9c-538c7d31f599
```

AWS SAM

您可以使用 AWS Serverless Application Model 来为您的函数配置环境变量。更新 `template.yaml` 文件中的 Environment 和 Variables 属性，然后运行 sam deploy。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs18.x
```

```
    Architectures:
      - x86_64
    EphemeralStorage:
      Size: 10240
    Environment:
      Variables:
        BUCKET: amzn-s3-demo-bucket
        KEY: file.txt
    # Other function properties...
```

AWS SDKs

要使用 AWS SDK 管理环境变量，请使用以下 API 操作。

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

要了解更多信息，请参阅您首选编程语言的 AWS SDK 文档。

定义运行时环境变量

Lambda 运行时会在初始化过程中设置多个环境变量。大多数环境变量提供有关函数或运行时的信息。这些环境变量的键是预留 的，无法在函数配置中设置。

预留环境变量

- _HANDLER – 函数上配置的处理程序位置。
- _X_AMZN_TRACE_ID – X-Ray 跟踪标头。此环境变量会随着每次调用发生变化。
  - 此环境变量不是为仅限操作系统的运行时系统（provided 运行时系统系列）定义的。您可以使用来自 下一个调用 的 Lambda-Runtime-Trace-Id 响应标头设置用于自定义运行时的 _X_AMZN_TRACE_ID。
  - 对于 Java 运行时系统版本 17 及更高版本，不使用此环境变量。相反，Lambda 将把跟踪信息存储在 com.amazonaws.xray.traceHeader 系统属性中。
- AWS_DEFAULT_REGION – 执行 Lambda 函数的默认 AWS 区域。
- AWS_REGION – 执行 Lambda 函数的 AWS 区域。如果定义了该值，该值将会覆盖 AWS_DEFAULT_REGION。

  - 有关在 AWS SDK 中使用 AWS 区域 环境变量的更多信息，请参阅《AWS SDK 和工具参考指南》中的 AWS 区域。
- AWS_EXECUTION_ENV – 运行时标识符，前缀为 AWS_Lambda_（例如 AWS_Lambda_java8）。此环境变量不是为仅限操作系统的运行时系统（provided 运行时系统系列）定义的。
- AWS_LAMBDA_FUNCTION_NAME – 函数的名称。
- AWS_LAMBDA_FUNCTION_MEMORY_SIZE – 对函数可用的内存量（以 MB 为单位）。
- AWS_LAMBDA_FUNCTION_VERSION – 要执行的函数的版本。
- AWS_LAMBDA_INITIALIZATION_TYPE – 函数的初始化类型，即 on-demand、provisioned-concurrency 或 snap-start。有关信息，请参阅配置预置并发或 使用 Lambda SnapStart 提高启动性能。
- AWS_LAMBDA_LOG_GROUP_NAME、AWS_LAMBDA_LOG_STREAM_NAME – Amazon CloudWatch Logs 组和函数的流名称。AWS_LAMBDA_LOG_GROUP_NAME 和 AWS_LAMBDA_LOG_STREAM_NAME 环境变量在 Lambda SnapStart 函数中不可用。
- AWS_ACCESS_KEY、AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEYAWS_SESSION_TOKEN – 从函数的执行角色中获取的访问密钥。
- AWS_LAMBDA_RUNTIME_API – （自定义运行时）运行时 API 的主机和端口。
- LAMBDA_TASK_ROOT – Lambda 函数代码的路径。
- LAMBDA_RUNTIME_DIR – 运行时库的路径。

以下附加环境变量并非预留，可以在函数配置中扩展。

非预留环境变量

- LANG – 运行时的区域设置 (en_US.UTF-8)。
- PATH – 执行路径 (/usr/local/bin:/usr/bin/:/bin:/opt/bin)。
- LD_LIBRARY_PATH – 系统库路径 (/var/lang/lib:/lib64:/usr/lib64:$LAMBDA_RUNTIME_DIR:$LAMBDA_RUNTIME_DIR/lib:$LAMBDA_TASK_ROOT:$LAMBDA_TASK_ROOT/lib:/opt/lib)。
- NODE_PATH – (Node.js) Node.js 库路径 (/opt/nodejs/node12/node_modules/:/opt/nodejs/node_modules:$LAMBDA_RUNTIME_DIR/node_modules)。
- PYTHONPATH – （Python 2.7、3.6、3.8）Python 库路径 ($LAMBDA_RUNTIME_DIR)。
- GEM_PATH – （Ruby）Ruby 库路径 ($LAMBDA_TASK_ROOT/vendor/bundle/ruby/2.5.0:/opt/ruby/gems/2.5.0)。

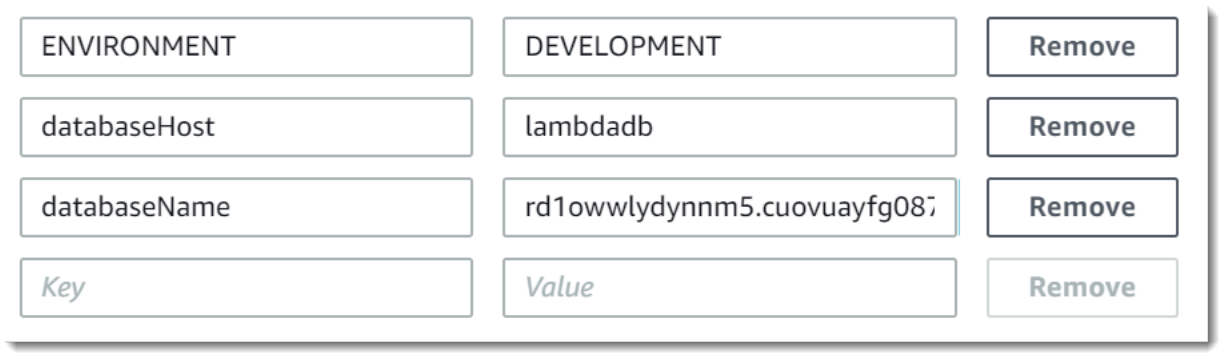
- AWS_XRAY_CONTEXT_MISSING – 对于 X-Ray 跟踪，Lambda 会将其设置为 LOG_ERROR，以避免从 X-Ray 开发工具包引发运行时错误。
- AWS_XRAY_DAEMON_ADDRESS – 对于 X-Ray 跟踪，X-Ray 进程守护程序的 IP 地址和端口。
- AWS_LAMBDA_DOTNET_PREJIT：对于 .NET 6 和 NET 7 运行时系统，请将此变量设置为启用或禁用 .NET 特定的运行时系统优化。值包括 always、never 和 provisioned-concurrency。有关更多信息，请参阅 为函数配置预置并发。
- TZ – 环境的时区 (:UTC)。执行环境使用 NTP 同步系统时钟。

显示的示例值反映了最新的运行时。特定变量或其值是否存在会因早先的运行时而异。

环境变量的示例场景

您可以使用环境变量来自定义测试环境和生产环境中的函数行为。例如，您可以创建两个具有相同代码但不同配置的函数。一个函数连接到测试数据库，另一个函数连接到生产数据库。在这种情况下，您可以使用环境变量向函数传递数据库的主机名和其他连接详细信息。

以下示例说明如何将数据库主机和数据库名称定义为环境变量。



如果希望测试环境生成比生产环境更多的调试信息，可以设置环境变量来配置测试环境使用更详细的日志记录或跟踪。

保护 Lambda 环境变量

为了保护环境变量，您可以使用服务器端加密来保护静态数据，使用客户端加密来保护传输中数据。

> **Note**
>
> 为了提高数据库的安全性，建议您使用 AWS Secrets Manager 而不是环境变量来存储数据库凭证。有关更多信息，请参阅 将 AWS Lambda 与 Amazon RDS 结合使用。

静态安全

Lambda 始终使用 AWS KMS key 提供服务器端静态加密。默认情况下，Lambda 使用 AWS 托管式密钥。如果此默认行为适合您的工作流，您无需设置任何其他内容。Lambda 将在账户中创建 AWS 托管式密钥，并为您管理其权限。AWS不会向您收取使用此密钥的费用。

如果您愿意，可以提供 AWS KMS 客户托管式密钥。这样做可能是为了控制 KMS 密钥的轮换，或者是为了满足组织管理 KMS 密钥的要求。当您使用客户托管式密钥时，只有您账户中有权访问 KMS 密钥的用户才能查看或管理函数上的环境变量。

客户托管式密钥产生标准 AWS KMS 费用。有关更多信息，请参阅AWS Key Management Service 定价。

传输过程中的安全

为了提高安全性，您可以为传输中加密启用帮助程序，这样可以确保环境变量在客户端加密，以便在传输过程中提供保护。

为环境变量配置加密

1. 使用 AWS Key Management Service（AWS KMS）创建任意客户托管式密钥，供 Lambda 用于服务器端和客户端加密。有关更多信息，请参阅 AWS Key Management Service 开发人员指南中的创建密钥。
2. 使用 Lambda 控制台，导航到 Edit environment variables (编辑环境变量) 页面。
   a. 打开 Lamba 控制台的 Functions（函数）页面。
   b. 选择函数。
   c. 选择 Configuration (配置)，然后从左侧导航栏选择 Environment variables (环境变量)。
   d. 在 Environment variables (环境变量) 部分中，选择 Edit (编辑)。
   e. 展开 Encryption configuration (加密配置)。
3. （可选）启用控制台加密帮助程序，以使用客户端加密来保护传输中数据。
   a. 在 Encryption in transit (传输中加密) 下，选择 Enable helpers for encryption in transit (为传输中加密启用帮助程序)。
   b. 对于要为其启用控制台加密帮助程序的每个环境变量，选择环境变量旁的 Encrypt（加密）。
   c. 在要传输中加密的 AWS KMS key 下，选择在此过程开始时创建的客户托管式密钥。
   d. 选择 Execution role policy (执行角色策略)并复制策略。此策略授予函数执行角色解密环境变量的权限。

保存此策略以便在此过程的最后一步中使用。

e. 将代码添加到解密环境变量的函数中。选择解密密钥代码段来查看示例。

4. （可选）为静态加密指定您的客户托管密钥。

a. 选择 Use a customer master key (使用客户主密钥)。

b. 选择在此过程开始时创建的客户托管式密钥。

5. 选择 Save（保存）。

6. 设置权限。

如果您使用带有服务器端加密的客户托管密钥，请将权限授予您希望能查看或管理函数上的环境变量的任何用户或角色。有关更多信息，请参阅 管理服务器端加密 KMS 密钥的权限。

如果您为传输中的安全性启用客户端加密，您的函数需要调用 `kms:Decrypt` API 操作的权限。将之前在此过程中保存的策略添加到函数的执行角色。

管理服务器端加密 KMS 密钥的权限

您的用户或函数的执行角色不需要 AWS KMS 权限，即可使用默认加密密钥。要使用客户托管式密钥，您需要具有使用密钥的权限。Lambda 使用您的权限在密钥上创建授权。Lambda 可将其用于加密。

- `kms:ListAliases` – 在 Lambda 控制台中查看密钥。
- `kms:CreateGrant`、`kms:Encrypt` – 在函数上配置客户托管式密钥。
- `kms:Decrypt` – 查看和管理使用客户托管式密钥加密的环境变量。

您可以从您的 AWS 账户 或从密钥的基于资源的权限策略获取这些权限。`ListAliases` 由 Lambda 的托管策略提供。密钥策略将剩余权限授予密钥用户组中的用户。

没有 `Decrypt` 权限的用户仍然可以管理函数，但无法在 Lambda 控制台中查看或管理环境变量。要防止用户查看环境变量，请向用户的权限添加一条语句，该语句拒绝访问默认密钥、客户托管式密钥或所有密钥。

Example IAM policy – 按密钥 ARN 拒绝访问

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Deny",
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-2:111122223333:key/3be10e2d-xmpl-4be4-
bc9d-0405a71945cc"
        }
    ]
}
```

有关托管密钥权限的详细信息，请参阅《AWS Key Management Service 开发人员指南》中的使用 AWS KMS 的密钥策略。

检索 Lambda 环境变量

要检索函数代码中的环境变量，请使用编程语言的标准方法。

Node.js

```
let region = process.env.AWS_REGION
```

Python

```
import os
  region = os.environ['AWS_REGION']
```

> **Note**
>
> 在某些情况下，您可能需要使用以下格式：
>
> ```
> region = os.environ.get('AWS_REGION')
> ```

Ruby

```
region = ENV["AWS_REGION"]
```

Java

```
String region = System.getenv("AWS_REGION");
```

Go

```
var region = os.Getenv("AWS_REGION")
```

C#

```
string region = Environment.GetEnvironmentVariable("AWS_REGION");
```

PowerShell

```
$region = $env:AWS_REGION
```

Lambda 通过静态加密来安全地存储环境变量。您可以配置 Lambda 以使用不同的加密密钥、在客户端加密环境变量值或使用 AWS Secrets Manager 在 AWS CloudFormation 模板中设置环境变量。

授予 Lambda 函数访问 Amazon VPC 中资源的权限

使用 Amazon Virtual Private Cloud（Amazon VPC），您可以在自己的 AWS 账户中创建私有网络，以用于托管 Amazon Elastic Compute Cloud（Amazon EC2）实例、Amazon Relational Database Service（Amazon RDS）实例和 Amazon ElastiCache 实例等资源。您可以通过包含相关资源的私有子网将函数附加到 VPC，从而向您的 Lambda 函数授予访问在 Amazon VPC 中托管的资源的权限。按照以下各节中的说明，通过 Lambda 控制台、AWS Command Line Interface（AWS CLI）或 AWS SAM 将 Lambda 函数附加到 Amazon VPC。

> **Note**
>
> 每个 Lambda 函数都在由 Lambda 服务拥有和管理的 VPC 内运行。这些 VPC 由 Lambda 自动维护，对客户不可见。配置您的函数以访问 Amazon VPC 中的其他 AWS 资源，不会影响在其中运行函数的由 Lambda 托管的 VPC。

Sections

所需的 IAM 权限

要将 Lambda 函数附加到您的 AWS 账户中的 Amazon VPC，Lambda 需要具有创建和管理网络接口的权限，以向您的函数授予访问该 VPC 中资源的权限。

Lambda 创建的网络接口被称为 Hyperplane 弹性网络接口，简称 Hyperplane ENI。要了解有关这些网络接口的更多信息，请参阅 the section called “了解 Hyperplane 弹性网络接口（ENI）”。

您可以通过将 AWS 托管式策略 AWSLambdaVPCAccessExecutionRole 附加到函数的执行角色，从而向函数授予所需的权限。当您在 Lambda 控制台中创建新函数并将其附加到 VPC 时，Lambda 会自动为您添加此权限策略。

如果您希望创建自己的 IAM 权限策略，请务必添加以下所有权限：

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces：仅当所有资源 (`"Resource": "*"`) 都允许执行此操作时，此操作才有效。
- ec2:DescribeSubnets
- ec2:DeleteNetworkInterface：如果您没有在执行角色中为 DeleteNetworkInterface 指定资源 ID，则函数可能无法访问 VPC。请指定唯一的资源 ID，或包含所有资源 ID，例如 `"Resource": "arn:aws:ec2:us-west-2:123456789012:*/*"`。
- ec2:AssignPrivateIpAddresses
- ec2:UnassignPrivateIpAddresses

请注意，函数的角色需要这些权限只是为了创建网络接口，而不是调用您的函数。将函数附加到 Amazon VPC 后，即使您从函数的执行角色中移除了这些权限，您仍然可以成功调用该函数。

要将您的函数附加到 VPC，Lambda 还需要使用您的 IAM 用户角色验证网络资源。确保您的用户角色具有以下 IAM 权限：

- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcs

> **Note**
>
> Lambda 服务使用授予函数执行角色的 Amazon EC2 权限，将函数附加到 VPC。不过，您还会隐式向函数的代码授予这些权限。这意味着函数代码能够进行这些 Amazon EC2 API 调用。有关遵循安全性最佳实践的建议，请参阅 the section called “安全最佳实操”。

将 Lambda 函数附加到您的 AWS 账户 中的 Amazon VPC

通过 Lambda 控制台、AWS CLI 或 AWS SAM 将您的函数附加到您的 AWS 账户中的 Amazon VPC。如果使用 AWS CLI 或 AWS SAM，或者使用 Lambda 控制台将现有的函数附加到 VPC，请确保函数的执行角色具有上一节中列出的必要权限。

Lambda 函数无法直接连接到具有专用实例租赁的 VPC。要连接到专用 VPC 中的资源，请使其与具有默认租赁的第二个 VPC 对等。

Lambda console

在创建函数时将函数附加到 Amazon VPC

1. 打开 Lambda 控制台的函数页面，然后选择创建函数。
2. 在基本信息下的函数名称中输入函数的名称。
3. 通过执行以下操作配置函数的 VPC 设置：
   a. 展开 Advanced settings（高级设置）。
   b. 选择启用 VPC，然后选择您希望附加该函数的 VPC。
   c. （可选）要允许出站 IPv6 流量，请选择允许双堆栈子网的 IPv6 流量。
   d. 选择要为其创建网络接口的子网和安全组。如果您已选择允许双堆栈子网的 IPv6 流量，则所有选定的子网都必须具有 IPv4 CIDR 块和 IPv6 CIDR 块。

   > **Note**
   >
   > 要访问私有资源，请将函数连接到私有子网。如果函数需要互联网访问权限，请参阅the section called “VPC 函数的互联网访问权限”。将函数连接到公有子网不会授予其 Internet 访问权限或公有 IP 地址。

4. 选择 Create function（创建函数）。

将现有函数附加到 Amazon VPC

1. 打开 Lambda 控制台的“函数”页面，然后选择函数。
2. 选择配置选项卡，然后选择 VPC。
3. 选择编辑。
4. 在 VPC 下，选择要附加您的函数的 Amazon VPC。

5. （可选）要允许出站 IPv6 流量，请选择允许双堆栈子网的 IPv6 流量。
6. 选择要为其创建网络接口的子网和安全组。如果您已选择允许双堆栈子网的 IPv6 流量，则所有选定的子网都必须具有 IPv4 CIDR 块和 IPv6 CIDR 块。

> **Note**
>
> 要访问私有资源，请将函数连接到私有子网。如果函数需要互联网访问权限，请参阅the section called “VPC 函数的互联网访问权限”。将函数连接到公有子网不会授予其 Internet 访问权限或公有 IP 地址。

7. 选择保存。

AWS CLI

在创建函数时将函数附加到 Amazon VPC

- 要创建 Lambda 函数并将其附加到 VPC，请运行以下 CLI `create-function` 命令。

```
aws lambda create-function --function-name my-function \
--runtime nodejs20.x --handler index.js --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/lambda-role \
--vpc-config
 Ipv6AllowedForDualStack=true,SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a03
```

指定您自己的子网和安全组，并根据您的应用场景将 `Ipv6AllowedForDualStack` 设置为 `true` 或 `false`。

将现有函数附加到 Amazon VPC

- 要将现有的 Lambda 函数附加到 VPC，请运行以下 CLI `update-function-configuration` 命令。

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config Ipv6AllowedForDualStack=true,
 SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-085912
```

从 VPC 分离函数

- 要从 VPC 分离函数，请使用空白的子网和安全组列表运行以下 `update-function-configuration` CLI 命令。

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

AWS SAM

将函数附加到 VPC

- 要将 Lambda 函数附加到 Amazon VPC，请将 `VpcConfig` 属性添加到函数定义中，如以下示例模板所示。有关此属性的更多信息，请参阅《AWS CloudFormation 用户指南》中的 AWS::Lambda::Function VpcConfig（将 AWS SAM `VpcConfig` 属性直接传递给 AWS CloudFormation `AWS::Lambda::Function` 资源的 `VpcConfig` 属性）。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./lambda_function/
      Handler: lambda_function.handler
      Runtime: python3.12
      VpcConfig:
        SecurityGroupIds:
          - !Ref MySecurityGroup
        SubnetIds:
          - !Ref MySubnet1
          - !Ref MySubnet2
      Policies:
        - AWSLambdaVPCAccessExecutionRole

  MySecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Security group for Lambda function
      VpcId: !Ref MyVPC
```

```
MySubnet1:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref MyVPC
    CidrBlock: 10.0.1.0/24

MySubnet2:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref MyVPC
    CidrBlock: 10.0.2.0/24

MyVPC:
  Type: AWS::EC2::VPC
  Properties:
    CidrBlock: 10.0.0.0/16
```

有关在 AWS SAM 中配置 VPC 的更多信息，请参阅《AWS CloudFormation 用户指南》中的 AWS::EC2::VPC。

连接到 VPC 时的互联网访问权限

默认情况下，Lambda 函数可以访问公共互联网。当您将函数附加到 VPC 时，该函数只能访问该 VPC 内可用的资源。要使您的函数能够访问互联网，您还需要将 VPC 配置为可以访问互联网。要了解更多信息，请参阅 the section called “VPC 函数的互联网访问权限”。

IPv6 支持

您的函数可以通过 IPv6 连接到双堆栈 VPC 子网中的资源。默认情况下，此选项处于关闭状态。要允许出站 IPv6 流量，请使用控制台或使用带有 create-function 或 update-function-configuration 命令的 `--vpc-config Ipv6AllowedForDualStack=true`选项。

> **Note**
>
> 要在 VPC 中允许出站 IPv6 流量，连接到该函数的所有子网都必须是双堆栈子网。Lambda 不支持 VPC 中针对仅限 IPv6 的子网的出站 IPv6 连接，不支持针对未连接到 VPC 的函数的出站 IPv6 连接，也不支持使用 VPC 端点（AWS PrivateLink）的入站 IPv6 连接。

您可以更新函数代码以通过 IPv6 显式连接到子网资源。以下 Python 示例会打开一个套接字并连接到 IPv6 服务器。

Example ：连接到 IPv6 服务器

```
def connect_to_server(event, context):
    server_address = event['host']
    server_port = event['port']
    message = event['message']
    run_connect_to_server(server_address, server_port, message)

def run_connect_to_server(server_address, server_port, message):
    sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM, 0)
    try:
        # Send data
        sock.connect((server_address, int(server_port), 0, 0))
        sock.sendall(message.encode())
        BUFF_SIZE = 4096
        data = b''
        while True:
            segment = sock.recv(BUFF_SIZE)
            data += segment
            # Either 0 or end of data
            if len(segment) < BUFF_SIZE:
                break
        return data
    finally:
        sock.close()
```

将 Lambda 与 Amazon VPC 结合使用的最佳实践

为确保您的 Lambda VPC 配置符合最佳实践指南，请遵循以下各节中的建议。

安全最佳实操

要将您的 Lambda 函数附加到 VPC，您需要向函数的执行角色授予一些 Amazon EC2 权限。在创建函数用来访问 VPC 中资源的网络接口时将需要这些权限。不过，还会向函数的代码隐式授予这些权限。这意味着函数代码具有进行这些 Amazon EC2 API 调用的权限。

为遵循最低权限原则，请在函数的执行角色中添加与如下例类似的拒绝策略。此策略可防止您的函数调用 Lambda 服务用于将函数附加到 VPC 的 Amazon EC2 API。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                 "ec2:CreateNetworkInterface",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeSubnets",
                 "ec2:DetachNetworkInterface",
                 "ec2:AssignPrivateIpAddresses",
                 "ec2:UnassignPrivateIpAddresses"
            ],
            "Resource": [ "*" ],
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": [
                        "arn:aws:lambda:us-west-2:123456789012:function:my_function"
                    ]
                }
            }
        }
    ]
}
```

AWS 提供了安全组和网络访问控制列表（ACL）功能来增强 VPC 中的安全性。安全组可以控制您的资源的入站和出站流量，网络 ACL 可以控制您的子网的入站和出站流量。安全组为大多数子网提供足够的访问控制。如果需要为 VPC 增加额外安全保护，您可以使用网络 ACL。有关使用 Amazon VPC 时的安全最佳实践的一般指南，请参阅《Amazon Virtual Private Cloud 用户指南》中的 VPC 的安全最佳实践。

性能最佳实践

当您将函数附加到 VPC 时，Lambda 会检查是否有可用来连接的可用网络资源（Hyperplane ENI）。Hyperplane ENI 与特定的安全组和 VPC 子网组合相关联。将一个函数附加到 VPC 后，如果在附加其他函数时指定相同的子网和安全组，这将意味着 Lambda 可以共享网络资源，无需创建新的 Hyperplane ENI。有关 Hyperplane ENI 及其生命周期的更多信息，请参阅 the section called “了解 Hyperplane 弹性网络接口（ENI）”。

了解 Hyperplane 弹性网络接口（ENI）

Hyperplane ENI 是一种托管式资源，在您的 Lambda 函数和您希望函数连接到的资源之间充当网络接口。当您将函数附加到 VPC 时，Lambda 服务会自动创建和管理这些 ENI。

Hyperplane ENI 对您并不直接可见，因此您无需对其进行配置或管理。不过，了解其工作原理有助您在将函数附加到 VPC 时了解其行为。

首次使用特定的子网和安全组组合将函数附加到 VPC 时，Lambda 将创建一个 Hyperplane ENI。您账户中使用相同子网和安全组组合的其他函数也可以使用该 ENI。Lambda 会尽可能重复使用现有的 ENI 来优化资源利用率并减少新 ENI 的创建。每个 Hyperplane ENI 最多支持 65,000 个连接/端口。如果连接数超过此限制，Lambda 会根据网络流量和并发要求自动扩展 ENI 的数量。

对于新函数，当 Lambda 创建 Hyperplane ENI 时，您的函数仍处于“待处理”状态，因此无法调用。只有在 Hyperplane ENI 准备就绪后，您的函数才会变为“活动”状态，这可能需要几分钟。对于现有函数，您无法执行以该函数为目标的其他操作（例如创建版本或更新函数的代码），但可以继续调用该函数的早期版本。

> **Note**
>
> 如果 Lambda 函数持续 30 天保持空闲状态，Lambda 将会回收任何未使用的 Hyperplane ENI，并将函数状态设置为空闲。新调用尝试将会失败，并且函数会重新进入“待处理”状态，直到 Lambda 完成 Hyperplane ENI 创建或分配。有关 Lambda 函数状态的更多信息，请参阅 the section called “函数状态”。

将 IAM 条件键用于 VPC 设置

您可以将特定于 Lambda 的条件键用于 VPC 设置，从而为您的 Lambda 函数提供额外的权限控制。例如，您可以要求组织中的所有函数都连接到 VPC。您还可以指定函数的用户可以使用和不能使用的子网和安全组。

Lambda 在 IAM policy 中支持以下条件键：

- lambda:VpcIds – 允许或拒绝一个或多个 VPC。
- lambda:SubnetIds – 允许或拒绝一个或多个子网。
- lambda:SecurityGroupIds – 允许或拒绝一个或多个安全组。

Lambda API 操作 CreateFunction 和 UpdateFunctionConfiguration 支持这些条件键。有关在 IAM policy 中使用条件键的更多信息，请参阅《IAM 用户指南》中的IAM JSON policy 元素：条件。

> **Tip**
>
> 如果您的函数已包含来自前一个 API 请求的 VPC 配置，则可以发送不带 VPC 配置的 `UpdateFunctionConfiguration` 请求。

带有用于 VPC 设置的条件键的示例策略

以下示例演示如何将条件键用于 VPC 设置。创建具有所需限制的策略语句后，为目标用户或角色附加策略语句。

确保用户仅部署与 VPC 连接的函数

要确保所有用户仅部署与 VPC 连接的函数，您可以拒绝不包含有效 VPC ID 的函数创建和更新操作。

请注意，VPC ID 不是 `CreateFunction` 或 `UpdateFunctionConfiguration` 请求的输入参数。Lambda 根据子网和安全组参数检索 VPC ID 值。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceVPCFunction",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "Null": {
           "lambda:VpcIds": "true"
        }
      }
    }
  ]
}
```

拒绝用户访问特定的 VPC、子网或安全组

要拒绝用户访问特定 VPC，请使用 StringEquals 检查 lambda:VpcIds 条件的值。以下示例拒绝用户访问 vpc-1 和 vpc-2。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceOutOfVPC",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
            "lambda:VpcIds": ["vpc-1", "vpc-2"]
        }
      }
    }
```

要拒绝用户访问特定子网，请使用 StringEquals 检查 lambda:SubnetIds 条件的值。以下示例拒绝用户访问 subnet-1 和 subnet-2。

```
{
      "Sid": "EnforceOutOfSubnet",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

要拒绝用户访问特定安全组，请使用 StringEquals 检查 lambda:SecurityGroupIds 条件的值。以下示例拒绝用户访问 sg-1 和 sg-2。

```
{
      "Sid": "EnforceOutOfSecurityGroups",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

允许用户使用特定 VPC 设置创建和更新函数

要允许用户访问特定的 VPC，请使用 StringEquals 检查 lambda:VpcIds 条件的值。以下示例允许用户访问 vpc-1 和 vpc-2。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceStayInSpecificVpc",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
            "lambda:VpcIds": ["vpc-1", "vpc-2"]
        }
      }
    }
```

要允许用户访问特定子网，请使用 StringEquals 检查 lambda:SubnetIds 条件的值。以下示例允许用户访问 subnet-1 和 subnet-2。

```
{
      "Sid": "EnforceStayInSpecificSubnets",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

要允许用户访问特定的安全组， 请使用 StringEquals 检查 lambda:SecurityGroupIds 条件的值。以下示例允许用户访问 sg-1 和 sg-2。

```
{
      "Sid": "EnforceStayInSpecificSecurityGroup",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

VPC 教程

在以下教程中，您会将 Lambda 函数连接到 VPC 中的资源。

- 教程：使用 Lambda 函数，以便在 Amazon VPC 中访问 Amazon RDS
- 教程：配置 Lambda 函数，以便在 Amazon VPC 中访问 Amazon ElastiCache

授予 Lambda 函数访问其他账户中 Amazon VPC 中资源的权限

可以为 AWS Lambda 函数提供权限来访问由其他账户管理的 Amazon Virtual Private Cloud 中 Amazon VPC 中的资源，而无需将任一 VPC 暴露在互联网中。此访问模式支持使用 AWS 与其他组织共享数据。使用此访问模式，可以在 VPC 之间共享数据，安全性和性能都比通过互联网共享更高。将 Lambda 函数配置为使用 Amazon VPC 对等连接来访问这些资源。

> **Warning**
>
> 如果您允许在账户或 VPC 之间进行访问，请检查您的计划是否符合管理这些账户的相应组织的安全要求。按照本文档中的说明进行操作，将影响资源的安全状况。

在本教程中，将使用 IPv4 通过对等连接将两个账户连接在一起。配置的 Lambda 函数尚未连接到 Amazon VPC。需配置 DNS 解析以将函数连接到不提供静态 IP 的资源。要使这些说明适应其他对等互连场景，请参阅《VPC Peering Guide》。

先决条件

要让 Lambda 函数访问其他账户中的资源，您必须具有：

- 一个 Lambda 函数，配置为使用您的资源进行身份验证，然后从中读取。
- 其他账户中的资源，例如 Amazon RDS 集群，可通过 Amazon VPC 获得。
- Lambda 函数账户和资源账户的凭证。如果您无权使用您的资源账户，请联系授权用户准备好该账户。
- 用于创建和更新 VPC（以及支持的 Amazon VPC 资源）以与 Lambda 函数关联的权限。
- 用于更新 Lambda 函数的执行角色和 VPC 配置的权限。
- 用于在 Lambda 函数的账户中创建 VPC 对等连接的权限。
- 用于在资源账户中接受 VPC 对等连接的权限。
- 用于更新资源的 VPC（以及支持的 Amazon VPC 资源）的配置的权限。
- 用于调用 Lambda 函数的权限。

在函数账户中创建 Amazon VPC

在 Lambda 函数账户中创建 Amazon VPC、子网、路由表和安全组。

使用控制台创建 VPC、子网和其他 VPC 资源

1. 通过以下网址打开 Amazon VPC 控制台：https://console.aws.amazon.com/vpc/。
2. 在控制面板上，选择创建 VPC。
3. 对于 IPv4 CIDR 块，请提供私有 CIDR 块。CIDR 块不得与资源 VPC 中的块重叠。不要选择资源 VPC 用于为资源分配 IP 的块，也不要选择资源 VPC 中已在路由表中定义的块。有关定义相应的 CIDR 块的更多信息，请参阅 VPC CIDR 块。
4. 选择自定义可用区。
5. 选择与您的资源相同的可用区。
6. 对于公有子网数量，选择 0。
7. 对于 VPC endpoints（VPC 端点），选择 None（无）。
8. 选择创建 VPC。

将 VPC 权限授予函数的执行角色

将 AWSLambdaVPCAccessExecutionRole 附加到函数的执行角色以允许其连接到 VPC。

将 VPC 权限授予函数的执行角色

1. 打开 Lamba 控制台的函数页面。
2. 选择函数的名称。
3. 选择配置。
4. 选择权限。
5. 在角色名称下，选择执行角色。
6. 在权限策略部分中，选择添加权限。
7. 在下拉列表中选择附加策略。
8. 在搜索框中，输入 AWSLambdaVPCAccessExecutionRole。
9. 选中策略名称左侧的复选框。
10. 选择添加权限。

将函数附加到 Amazon VPC

1. 打开 Lamba 控制台的函数页面。

2. 选择函数的名称。
3. 选择配置选项卡，然后选择 VPC。
4. 选择编辑。
5. 在 VPC 下方，选择您的 VPC。
6. 在子网下方，选择您的子网。
7. 在安全组下方，选择 VPC 的默认安全组。
8. 选择保存。

创建 VPC 对等连接请求

创建从函数 VPC（请求者 VPC）到资源 VPC（接受方 VPC）的 VPC 对等连接请求。

请求从函数 VPC 进行 VPC 对等连接

1. 打开 https://console.aws.amazon.com/vpc/。
2. 在导航窗格中，选择 Peering Connections（对等连接）。
3. 选择 Create Peering Connection（创建对等连接）。
4. 对于 VPC ID（请求者），请选择函数的 VPC。
5. 对于 账户 ID，输入资源账户的 ID。
6. 对于 VPC ID（接受者），输入资源的 VPC。

准备好资源账户

要创建对等连接并准备资源的 VPC 以使用该连接，请使用具有先决条件中列出的权限的角色登录资源账户。根据账户安全保障方式，登录步骤可能会有所不同。有关如何登录 AWS 账户的更多信息，请参阅《AWS Sign-in User Guide》。在资源账户中，执行以下过程。

接受 VPC 对等连接请求

1. 打开 https://console.aws.amazon.com/vpc/。
2. 在导航窗格中，选择 Peering Connections（对等连接）。
3. 选择待处理的 VPC 对等连接（状态为“待接受”）。
4. 选择操作。
5. 从下拉列表中选择接受请求。

6. 当系统提示进行确认时，选择接受请求。
7. 选择立即修改我的路由表，以向 VPC 的主路由表添加路由，从而确保您可以通过对等连接收发流量。

检查路由表，了解资源的 VPC。根据资源 VPC 的设置方式，Amazon VPC 生成的路由可能无法建立连接。检查 VPC 的新路由和现有配置之间是否存在冲突。有关问题排查的更多信息，请参阅《Amazon Virtual Private Cloud VPC 对等连接指南》中的对 VPC 对等连接进行问题排查。

更新资源的安全组

1. 打开 https://console.aws.amazon.com/vpc/。
2. 在导航窗格中，选择 Security Groups（安全组）。
3. 为资源选择安全组。
4. 选择操作。
5. 从下拉列表中选择编辑入站规则。
6. 选择 添加规则。
7. 对于来源，请输入函数的账户 ID 和安全组 ID，用正斜线分隔（例如 111122223333/sg-1a2b3c4d）。
8. 选择编辑出站规则。
9. 检查出站流量是否受到限制。默认 VPC 设置允许所有出站流量。如果出站流量受到限制，请继续执行下一步。
10. 选择 添加规则。
11. 对于目的地，请输入函数的账户 ID 和安全组 ID，用正斜线分隔（例如 111122223333/sg-1a2b3c4d）。
12. 选择保存规则。

实现对对等连接的 DNS 解析

1. 打开 https://console.aws.amazon.com/vpc/。
2. 在导航窗格中，选择 Peering Connections（对等连接）。
3. 选择对等连接。
4. 选择操作。
5. 选择编辑 DNS 设置。

6. 在接受方 DNS 解析下方，选择允许请求者 VPC 将接受方 VPC 主机的 DNS 解析为私有 IP。
7. 选择 Save changes（保存更改）。

更新函数账户中的 VPC 配置

登录函数账户，然后更新 VPC 配置。

为 VPC 对等连接添加路由

1. 打开 https://console.aws.amazon.com/vpc/。
2. 在导航窗格中，选择 Route tables（路由表）。
3. 选中与函数关联之子网的路由表名称旁边的复选框。
4. 选择操作。
5. 选择 Edit routes (编辑路由)。
6. 选择 Add route（添加路由）。
7. 对于目的地，输入资源 VPC 的 CIDR 块。
8. 对于目标，选择 VPC 对等连接。
9. 选择 Save changes（保存更改）。

有关在更新路由表时可能遇到的注意事项的更多信息，请参阅为 VPC 对等连接更新路由表。

更新 Lambda 函数的安全组

1. 打开 https://console.aws.amazon.com/vpc/。
2. 在导航窗格中，选择 Security Groups（安全组）。
3. 选择操作。
4. 选择编辑入站规则。
5. 选择添加规则。
6. 对于来源，请输入资源的账户 ID 和安全组 ID，用正斜线分隔（例如 111122223333/sg-1a2b3c4d）。
7. 选择保存规则。

实现对对等连接的 DNS 解析

1. 打开 https://console.aws.amazon.com/vpc/。
2. 在导航窗格中，选择 Peering Connections（对等连接）。
3. 选择对等连接。
4. 选择操作。
5. 选择编辑 DNS 设置。
6. 在请求者 DNS 解析下方，选择允许接受方 VPC 将请求者 VPC 主机的 DNS 解析为私有 IP。
7. 选择 Save changes（保存更改）。

测试函数

创建测试事件并检查函数的输出

1. 在代码源窗格中，选择测试。
2. 选择创建新事件。
3. 在事件 JSON 面板中，将默认值替换为适合 Lambda 函数的输入。
4. 选择 调用。
5. 在执行结果选项卡中，确认响应包含预期输出。

此外，可以检查函数的日志，以验证日志是否符合预期。

在 CloudWatch Logs 中查看函数的调用记录

1. 选择监控选项卡。
2. 选择查看 CloudWatch 日志。
3. 在日志流选项卡上，选择函数调用的日志流。
4. 确认日志符合预期。

为连接到 VPC 的 Lambda 函数启用互联网访问权限

默认情况下，Lambda 函数在可访问互联网的 Lambda 托管 VPC 中运行。要访问账户中 VPC 内的资源，可以向函数添加 VPC 配置。此举会将该函数限制为仅针对该 VPC 内的资源，除非该 VPC 可以访问互联网。本页内容会介绍如何为连接到 VPC 的 Lambda 函数提供互联网访问权限。

尚无 VPC

创建 VPC

创建 VPC 工作流程可创建 Lambda 函数从私有子网访问公有互联网所需的所有 VPC 资源，包括子网、NAT 网关、互联网网关和路由表条目。

创建 VPC

1. 通过以下网址打开 Amazon VPC 控制台：https://console.aws.amazon.com/vpc/。
2. 在控制面板上，选择创建 VPC。
3. 对于要创建的资源，选择 VPC 等。
4. 配置 VPC
   a. 对于 Name tag auto-generation（名称标签自动生成），为 VPC 输入名称。
   b. 对于 IPv4 CIDR 块，您可以保留默认建议，也可以输入应用程序或网络所需的 CIDR 块。
   c. 如果应用程序使用 IPv6 地址进行通信，则选择 IPv6 CIDR 块、Amazon 提供的 IPv6 CIDR 块。
5. 配置子网
   a. 对于可用区数量，选择 2。为了获得高可用性，建议至少选择两个 AZ。
   b. 对于Number of public subnets（公有子网数量），选择 2。
   c. 对于 Number of private subnets（私有子网数量），选择 2。
   d. 您可以保留公有子网的默认 CIDR 块，也可以展开自定义子网 CIDR 块并输入 CIDR 块。有关更多信息，请参阅子网 CIDR 块。
6. 对于 NAT 网关，选择每个可用区 1 个以提高故障恢复能力。
7. 对于仅限出口的互联网网关，如果选择了包含 IPv6 CIDR 块，则选择是。
8. 对于 VPC 端点，请保留默认值（S3 网关）。选择此项不会产生任何费用。有关更多信息，请参阅适用于 Amazon S3 的 VPC 端点类型。
9. 对于 DNS 选项，保留默认设置。

10. 选择创建 VPC。

配置 Lambda 函数

在创建函数时配置 VPC

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择 Create function（创建函数）。
3. 在基本信息下的函数名称中输入函数的名称。
4. 展开 Advanced settings（高级设置）。
5. 选择启用 VPC，再选择一个 VPC。
6. （可选）要允许出站 IPv6 流量，请选择允许双堆栈子网的 IPv6 流量。
7. 对于子网，选择全部私有子网。私有子网可以通过 NAT 网关访问互联网。将函数连接到公有子网并不会授予其互联网访问权限。

> **Note**
>
> 如果您已选择允许双堆栈子网的 IPv6 流量，则所有选定的子网都必须具有 IPv4 CIDR 块和 IPv6 CIDR 块。

8. 对于安全组，选择一个允许出站流量的安全组。
9. 选择 Create function (创建函数)。

Lambda 使用 AWSLambdaVPCAccessExecutionRole AWS 托管策略自动创建执行角色。只有在为 VPC 配置创建弹性网络接口时需要此策略中的权限，调用函数时并不需要。要应用最低权限，可以在创建函数和 VPC 配置之后，从执行角色中删除 AWSLambdaVPCAccessExecutionRole 策略。有关更多信息，请参阅 所需的 IAM 权限。

为现有函数配置 VPC

要将 VPC 配置添加到现有函数，该函数的执行角色必须具有创建和管理弹性网络接口的权限。AWSLambdaVPCAccessExecutionRole AWS 托管策略包含这些必要权限。要应用最低权限，可以在创建 VPC 配置之后，从执行角色中删除 AWSLambdaVPCAccessExecutionRole 策略。

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。

3. 选择配置选项卡，然后选择 VPC。
4. 在 VPC 下，选择 Edit（编辑）。
5. 选择 VPC。
6. （可选）要允许出站 IPv6 流量，请选择允许双堆栈子网的 IPv6 流量。
7. 对于子网，选择全部私有子网。私有子网可以通过 NAT 网关访问互联网。将函数连接到公有子网并不会授予其互联网访问权限。

> **Note**
>
> 如果您已选择允许双堆栈子网的 IPv6 流量，则所有选定的子网都必须具有 IPv4 CIDR 块和 IPv6 CIDR 块。

8. 对于安全组，选择一个允许出站流量的安全组。
9. 选择保存。

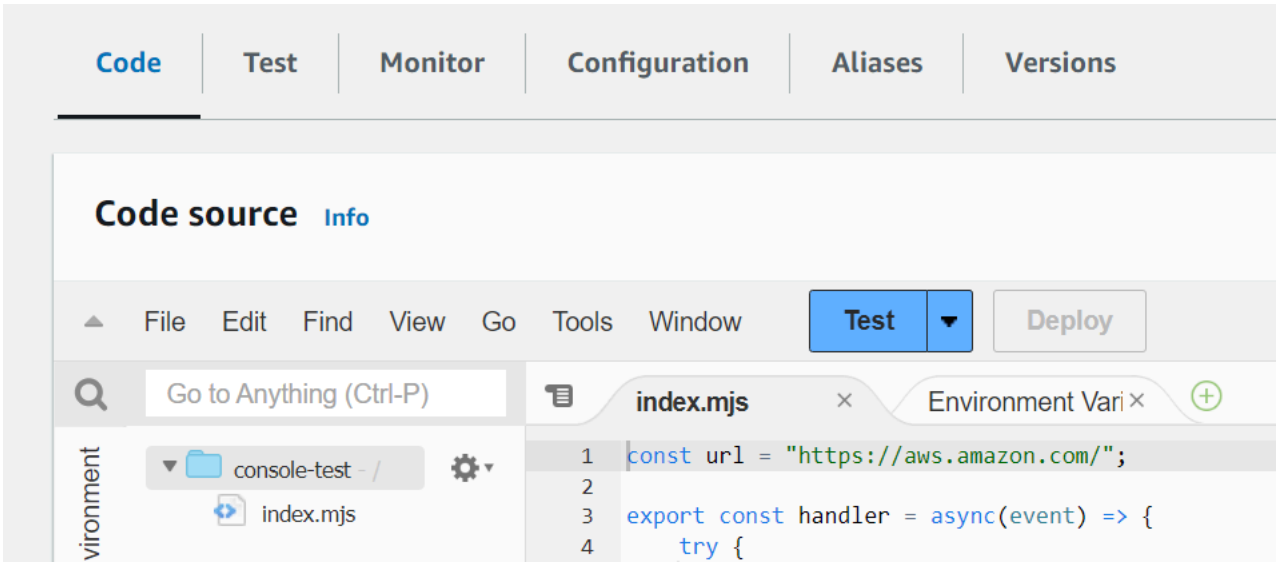
测试此函数

使用以下示例代码，确认连接到 VPC 的函数是否可以访问公有互联网。如果成功，代码将返回 `200` 状态代码。如果失败，函数将超时。

Node.js

此示例使用 `fetch`，它支持 `nodejs18.x` 和更高版本的运行时系统。

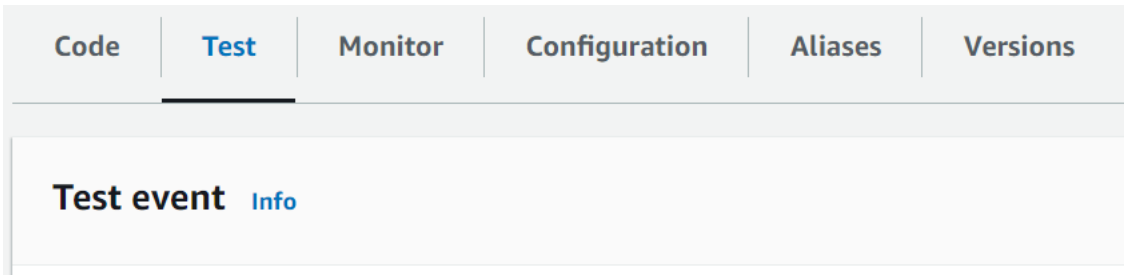
1. 在 Lambda 控制台的代码源窗格中，将以下代码粘贴到 index.mjs 文件中。该函数向公有端点发出 HTTP GET 请求，再返回 HTTP 响应代码来测试该函数是否可以访问公有互联网。



Example 示例：包含 async/await 的 HTTP 请求

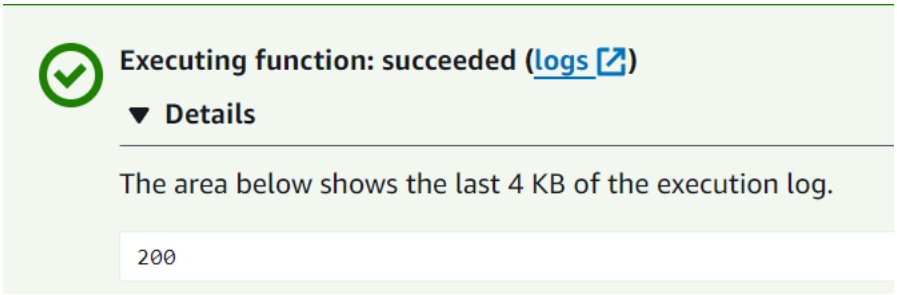
```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        // fetch is available with Node.js 18 and later runtimes
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

2. 选择部署。
3. 选择测试选项卡。



4. 选择测试。

5. 该函数返回 200 状态代码。此结果表示该函数具有出站互联网访问权限。

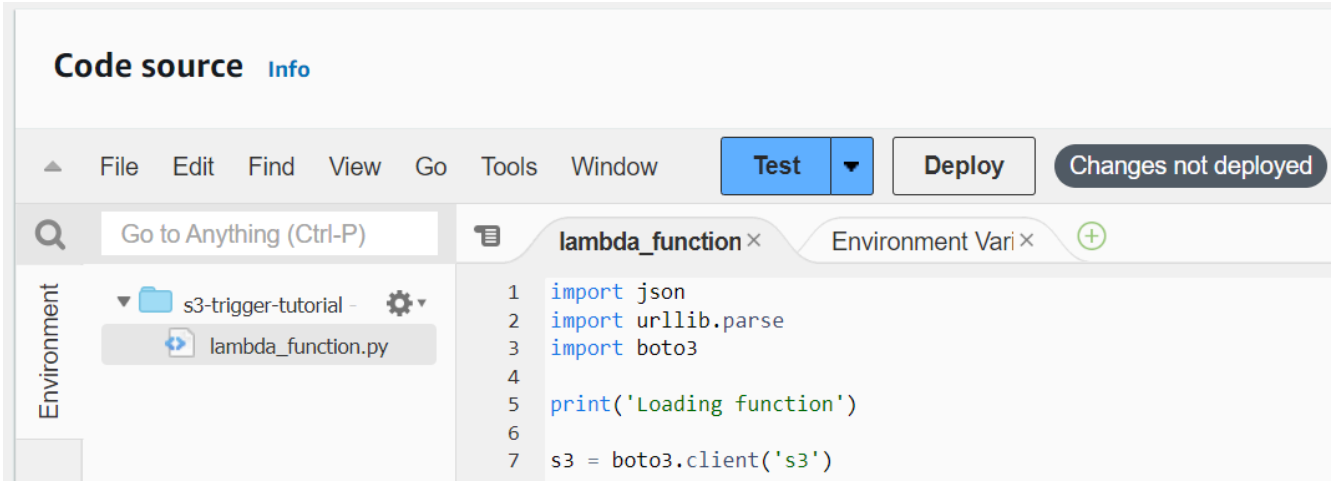


如果该函数无法访问公有互联网，则会收到如下错误消息：

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Python

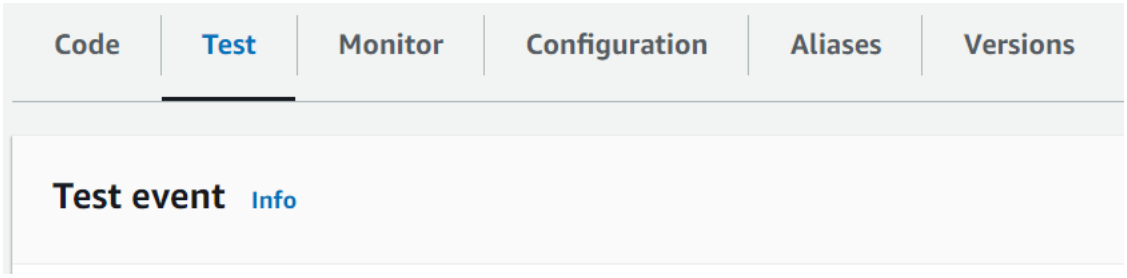
1. 在 Lambda 控制台的代码源窗格中，将以下代码粘贴到 ambda_function.py 文件中。该函数向公有端点发出 HTTP GET 请求，再返回 HTTP 响应代码来测试该函数是否可以访问公有互联网。



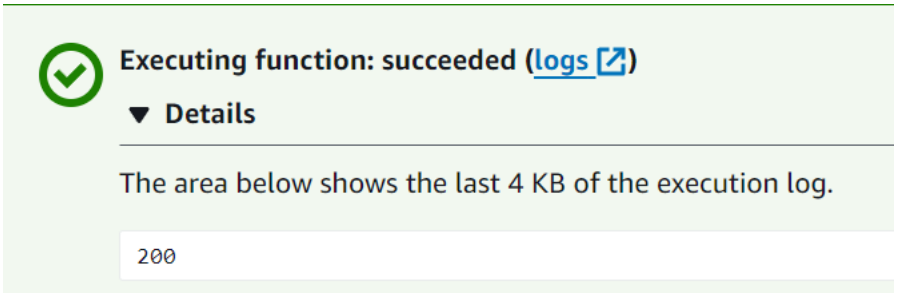

```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
```

```
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

2. 选择部署。
3. 选择测试选项卡。



4. 选择测试。
5. 该函数返回 200 状态代码。此结果表示该函数具有出站互联网访问权限。



如果该函数无法访问公有互联网，则会收到如下错误消息：

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

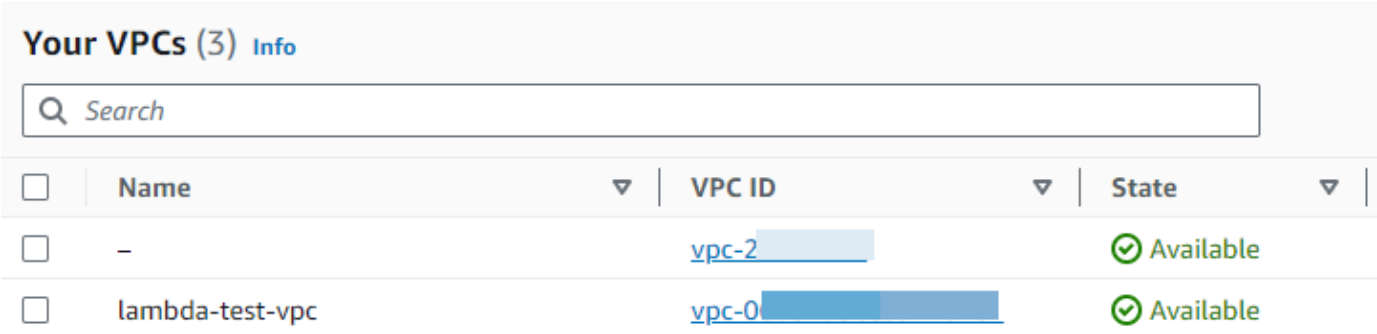
已有 VPC

如果已有 VPC，但需要为 Lambda 函数配置公有互联网访问权限，请按照以下步骤操作。此过程假设 VPC 至少有两个子网。如果没有两个子网，请参阅《Amazon VPC 用户指南》中的创建子网。

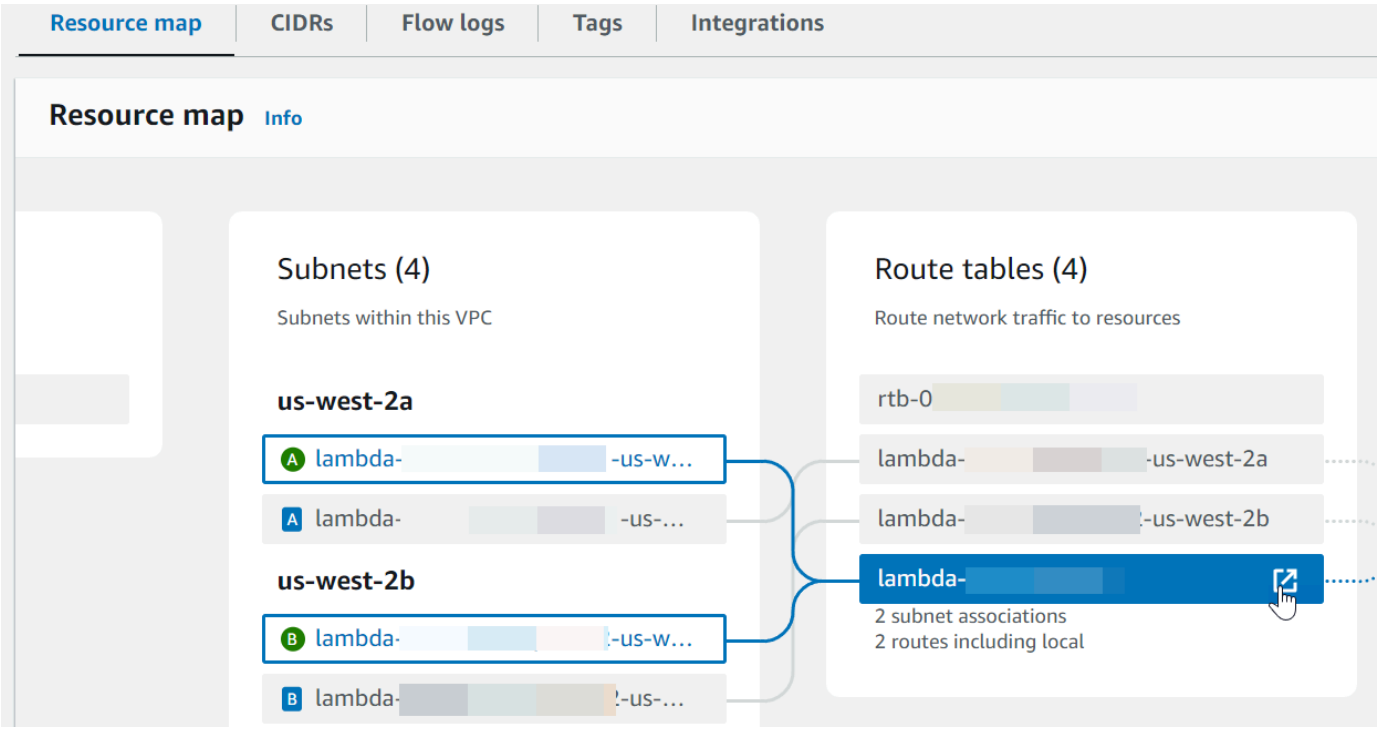
验证路由表配置

1. 通过以下网址打开 Amazon VPC 控制台：https://console.aws.amazon.com/vpc/。

2. 选择 VPC ID。



3. 向下滚动到资源映射部分。请注意路由表映射。打开映射到子网的每个路由表。

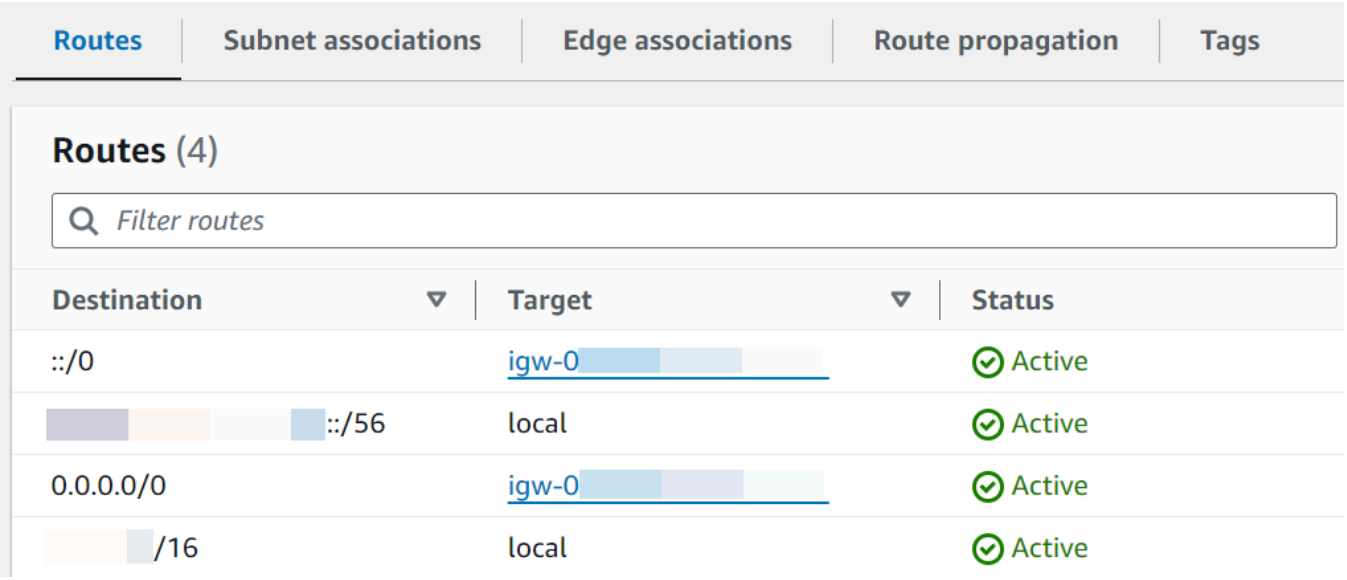


4. 向下滚动到路由选项卡。查看路由，判断是否满足以下情况之一。这些要求中的每一项都必须由单独的路由表来满足。

   - 将面向互联网的流量（IPv4 的路由为 `0.0.0.0/0`，IPv6 的路由为 `::/0`）路由到互联网网关 (`igw-xxxxxxxxxx`)。这表示与路由表关联的子网是公有子网。

> **Note**
>
> 如果子网没有 IPv6 CIDR 块，则只能看到 IPv4 的路由 (`0.0.0.0/0`)。

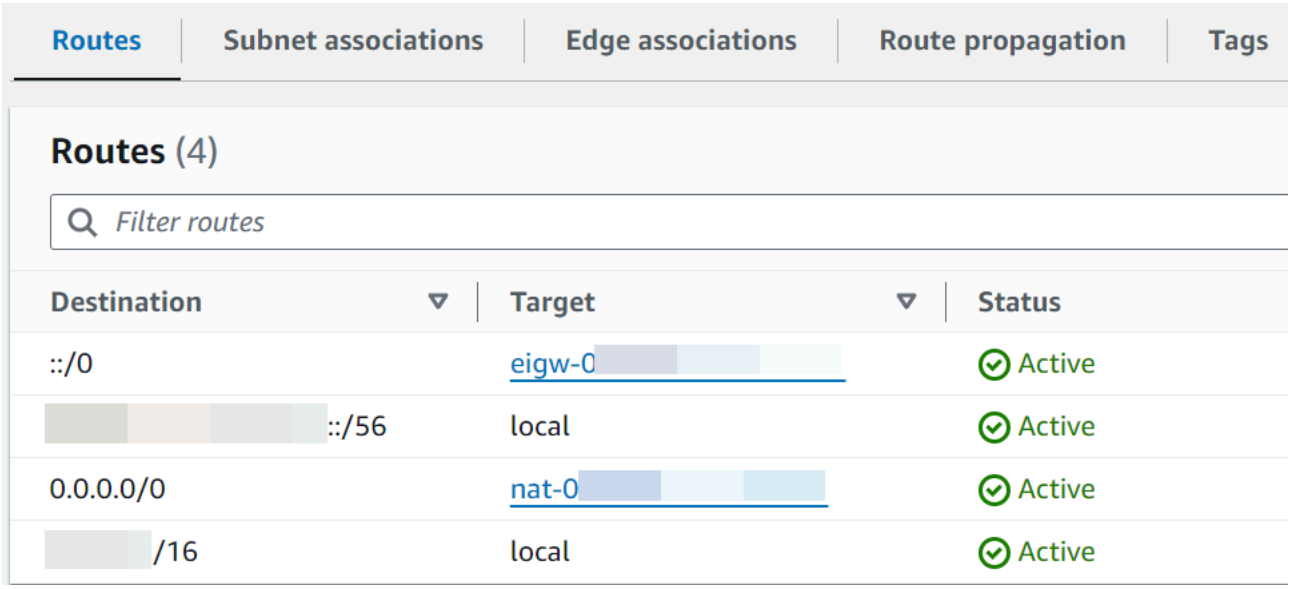
Example 公有子网路由表



- 将 IPv4 (`0.0.0.0/0`) 面向互联网的流量路由到与公有子网关联的 NAT 网关 (`nat-xxxxxxxxxx`)。这表示子网属于可以通过 NAT 网关访问互联网的私有子网。

> **Note**
>
> 如果子网具有 IPv6 CIDR 块，则路由表还必须将面向互联网的 IPv6 流量 (`::/0`) 路由到“仅出口互联网网关”(`eigw-xxxxxxxxxx`)。如果子网没有 IPv6 CIDR 块，则只能看到 IPv4 的路由 (`0.0.0.0/0`)。

Example 私有子网路由表



5. 重复上一步操作，直到查看了与 VPC 中子网关联的每个路由表，并确认存在带有互联网网关的路由表和带有 NAT 网关的路由表。

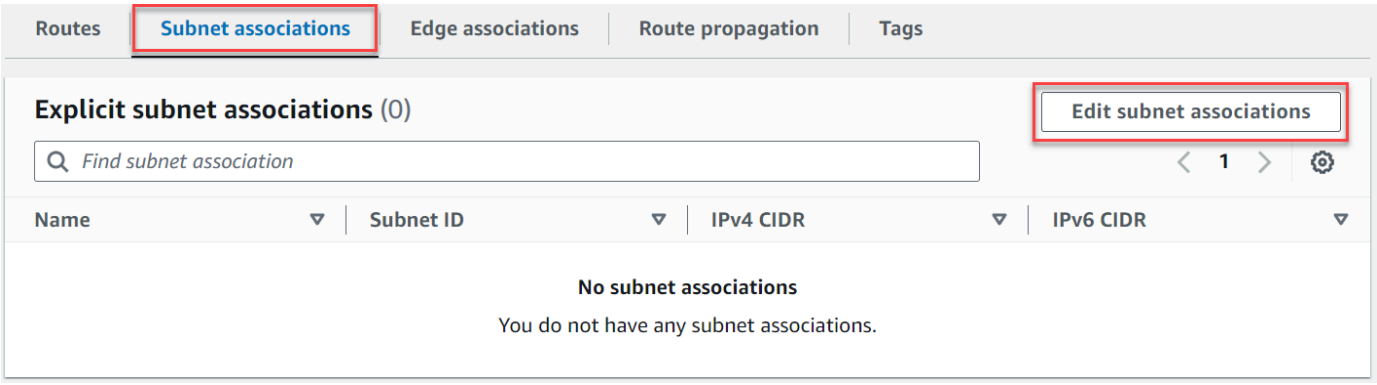
   如果没有两个路由表（一个带有指向互联网网关的路由，另一个带有指向 NAT 网关的路由），则按照以下步骤创建缺失的资源和路由表条目。

创建路由表

请按照以下步骤创建路由表，再将之与子网相关联。

使用 Amazon VPC 控制台创建自定义路由表

1. 通过以下网址打开 Amazon VPC 控制台：https://console.aws.amazon.com/vpc/。
2. 在导航窗格中，选择 Route tables（路由表）。
3. 选择创建路由表。
4. （可选）对于 Name（名称），为您的路由表输入名称。
5. 对于 VPC，选择您的 VPC。
6. （可选）若要添加标签，请选择 Add new tag（添加新标签），然后输入标签键和标签值。
7. 选择创建路由表。
8. 在 Subnet associations (子网关联) 选项卡上，选择 Edit subnet associations (编辑子网关联)。



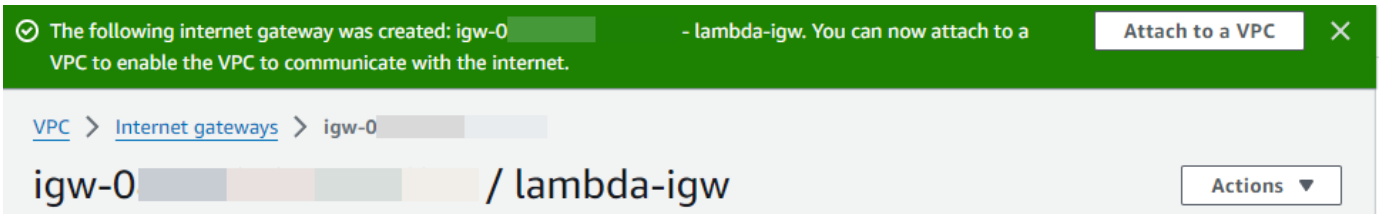
9. 选中要与路由表关联的子网的复选框。
10. 选择 Save associations (保存关联)。

创建 Internet 网关

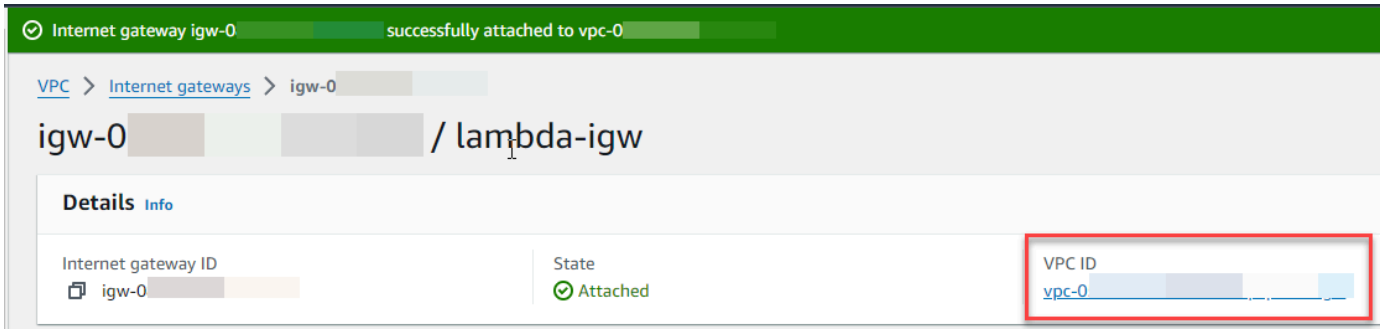
按照以下步骤创建互联网网关，再将其附加到 VPC，然后将之添加到公有子网的路由表中。

创建互联网网关

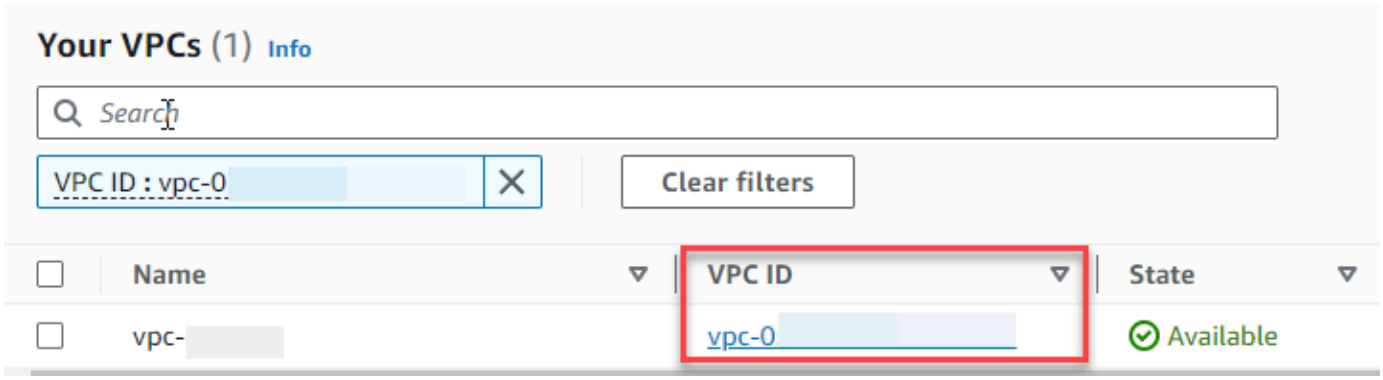
1. 通过以下网址打开 Amazon VPC 控制台：https://console.aws.amazon.com/vpc/。
2. 在导航窗格中，选择 Internet gateways（互联网网关）。
3. 选择创建互联网网关。
4. （可选）输入互联网网关的名称。
5. （可选）若要添加标签，请选择 Add new tag（添加新标签），然后输入该标签的键和值。
6. 选择创建互联网网关。
7. 从屏幕顶部的横幅中选择附加到 VPC，再选择可用的 VPC，然后选择附加互联网网关。



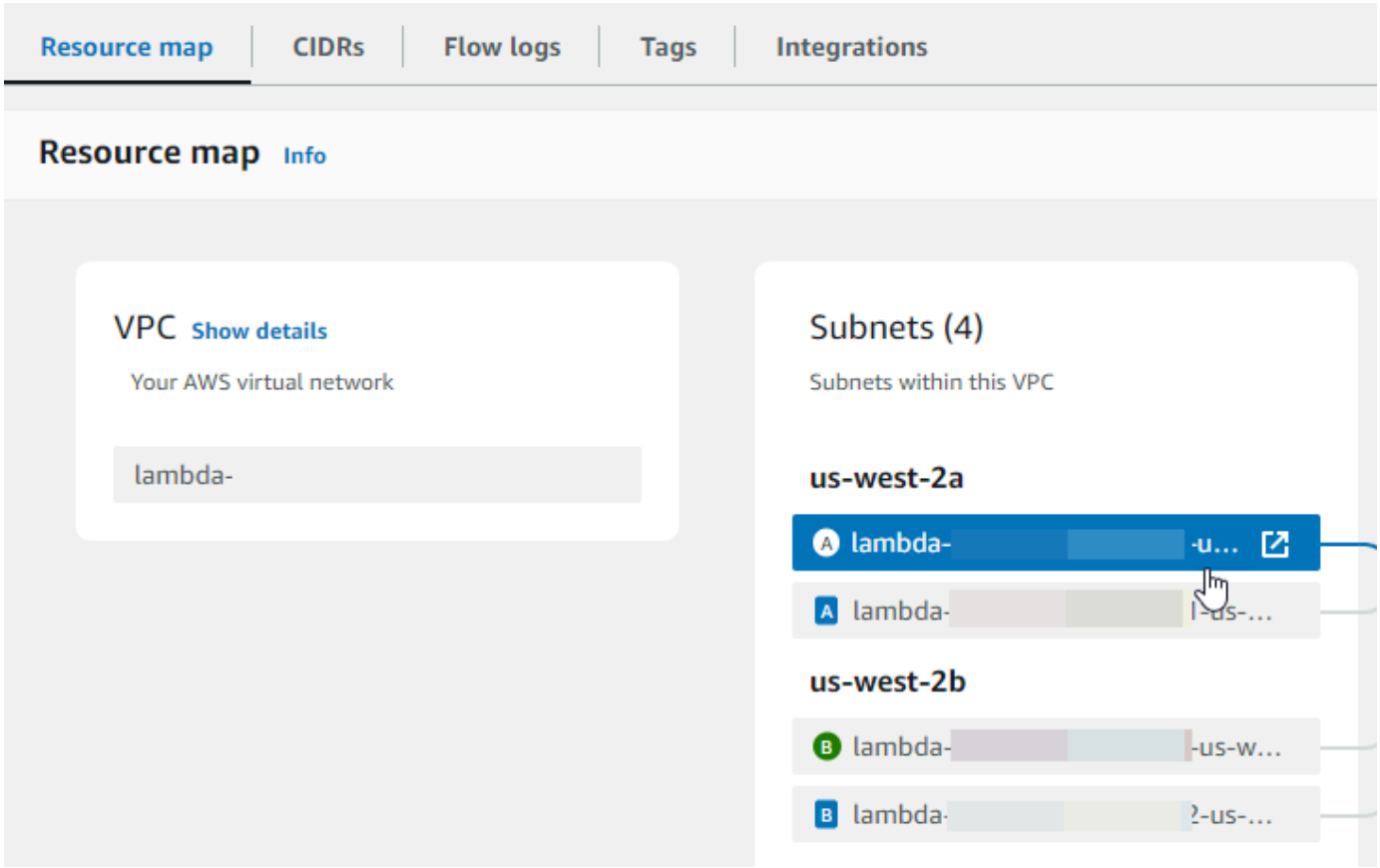
8. 选择 VPC ID。



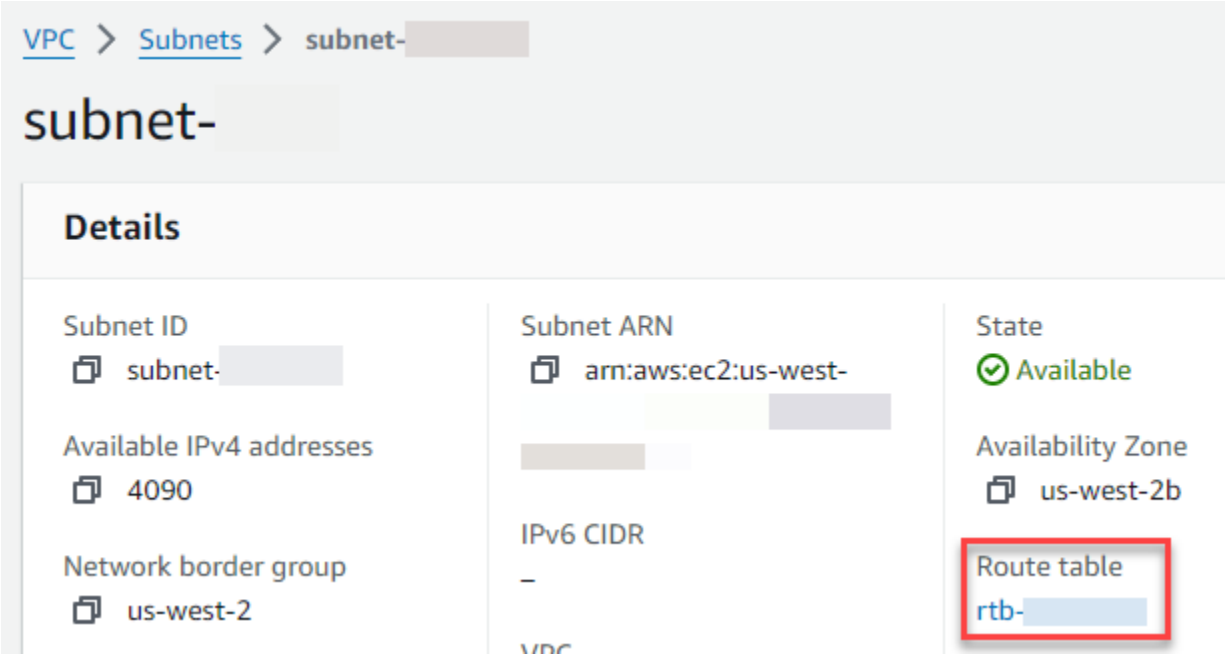
9. 再次选择 VPC ID 打开 VPC 详细信息页面。



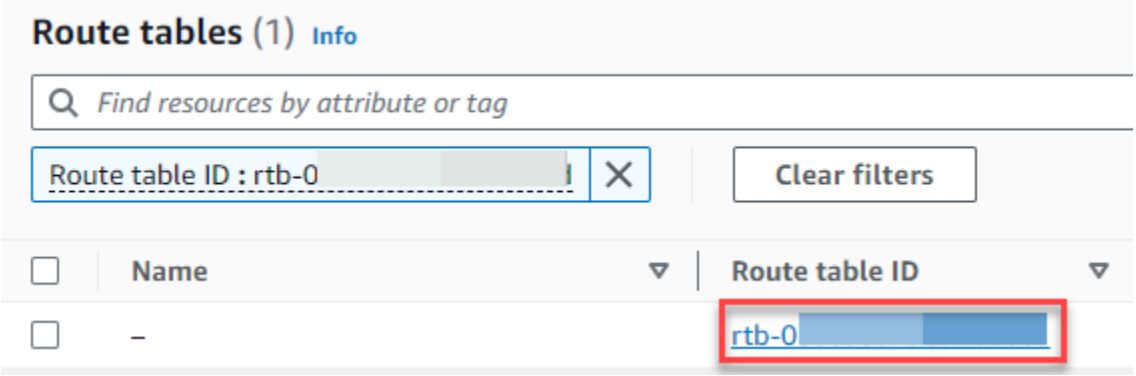
10. 向下滚动到资源映射部分，然后选择一个子网。子网详细信息显示在新选项卡中。



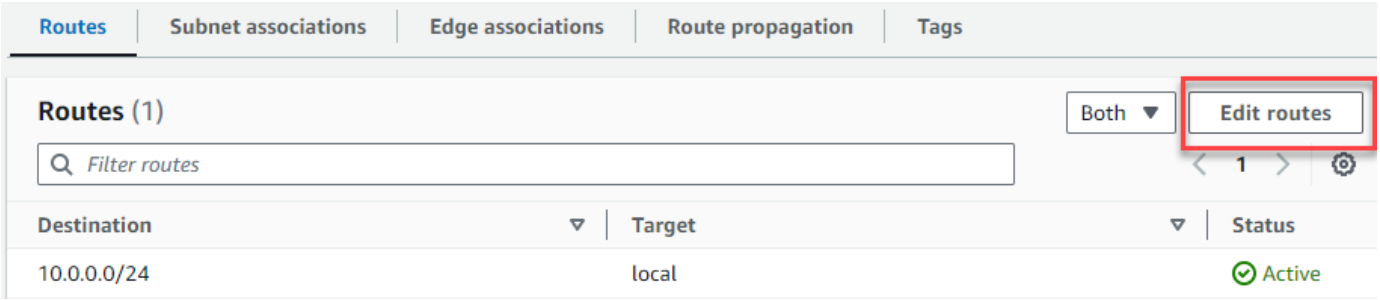
11. 选择路由表下的链接。



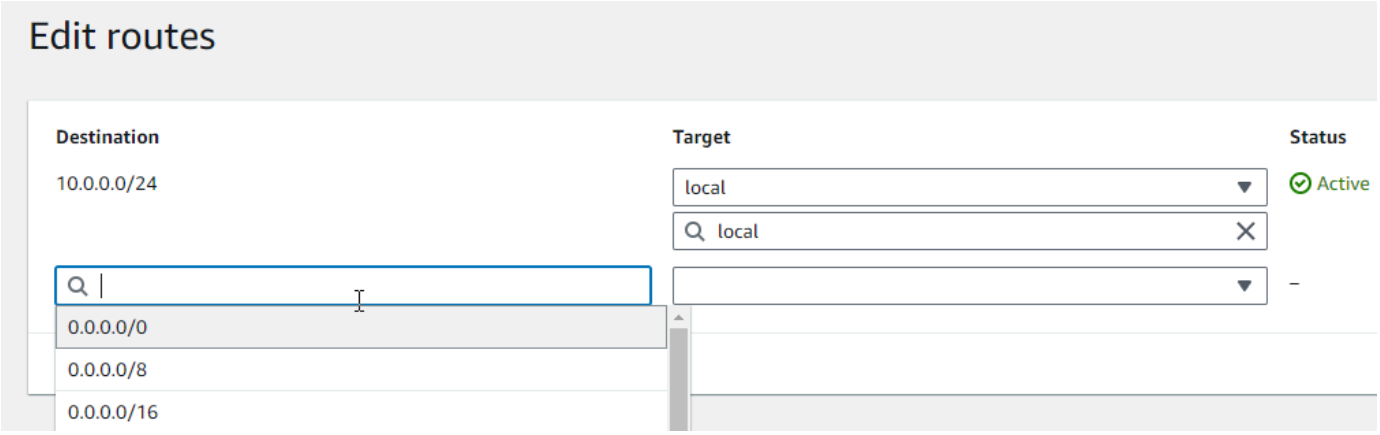
12. 选择路由表 ID 打开路由表的详细信息页面。



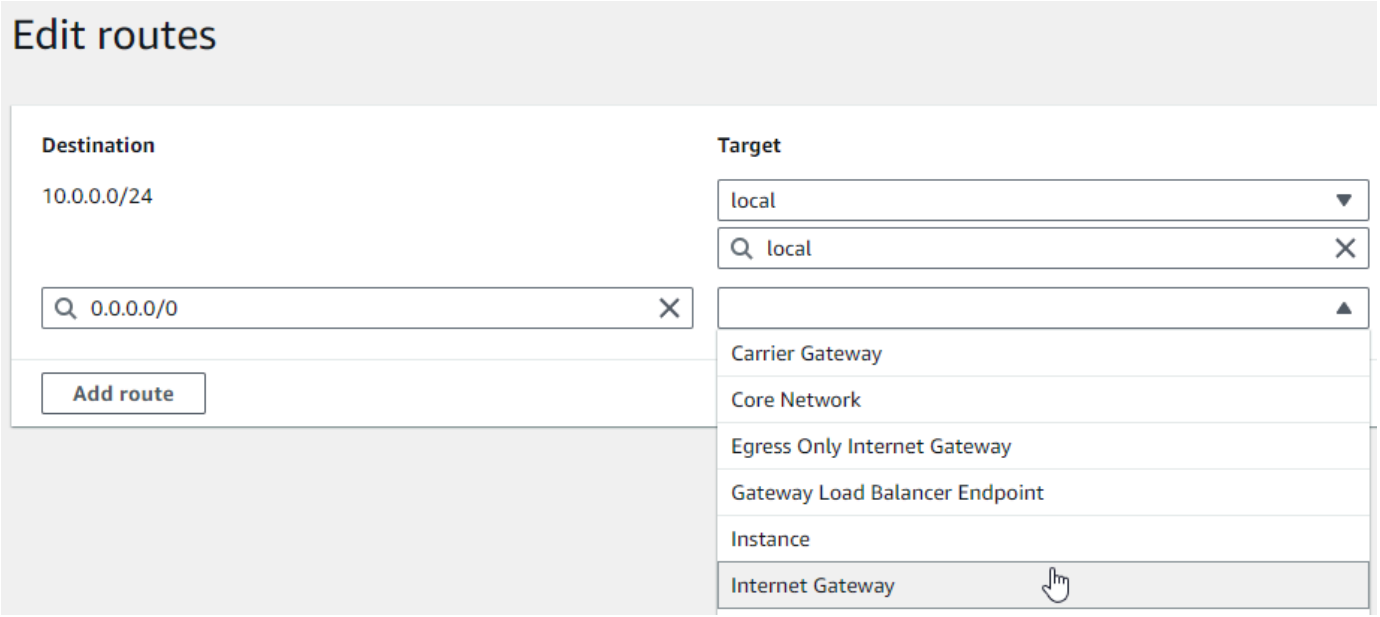
13. 在路由下，选择编辑路由。



14. 选择添加路由，然后在目的地框中输入 `0.0.0.0/0`。



15. 在目标中选择互联网网关，然后选择之前创建的互联网网关。如果子网具有 IPv6 CIDR 块，还须添加指向同一互联网网关的 `::/0` 路由。



16. 选择 Save changes（保存更改）。

创建 NAT 网关

按照以下步骤创建 NAT 网关，再将其与公有子网关联，然后将之添加到私有子网的路由表中。

创建 NAT 网关并将之与公有子网关联

1. 在导航窗格中，选择 NAT 网关。
2. 选择创建 NAT 网关。
3. （可选）输入 NAT 网关的名称。

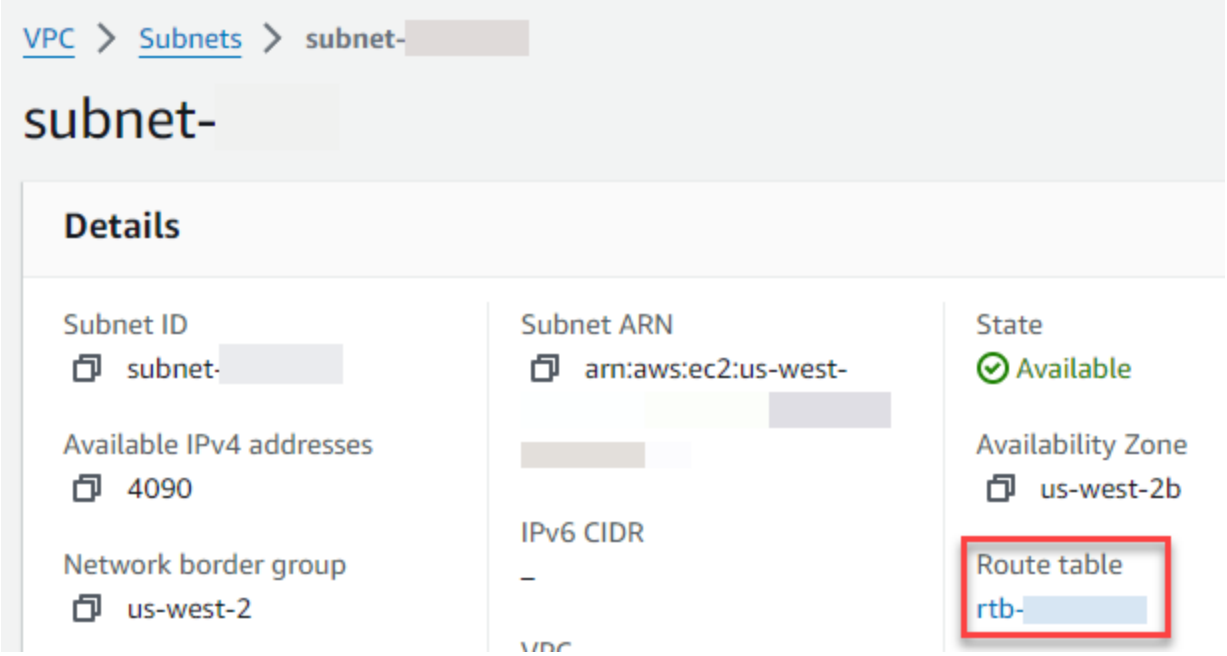
4. 对于子网，选择 VPC 中的公有子网。（公有子网是其路由表中具有指向互联网网关的直接路由的子网。）

> **Note**
>
> NAT 网关与公有子网关联，但路由表条目位于私有子网中。

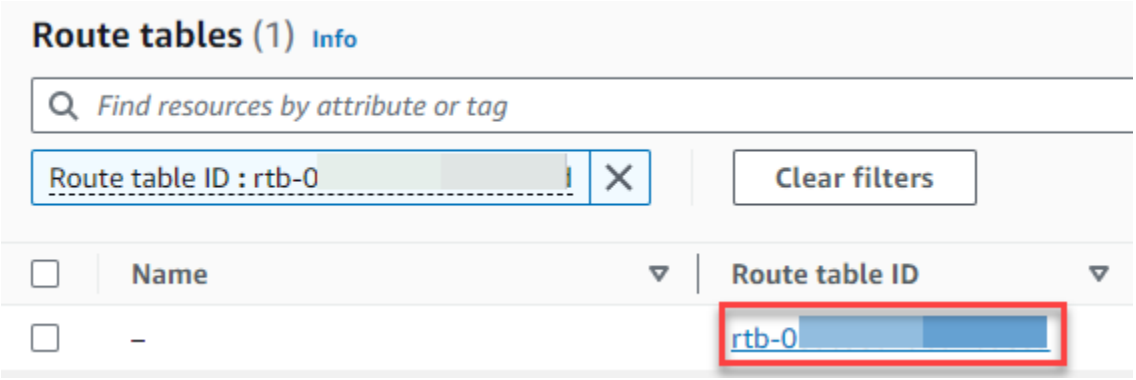
5. 对于弹性 IP 分配 ID，请选择弹性 IP 地址或选择分配弹性 IP。
6. 选择创建 NAT 网关。

在私有子网的路由表中向 NAT 网关添加路由

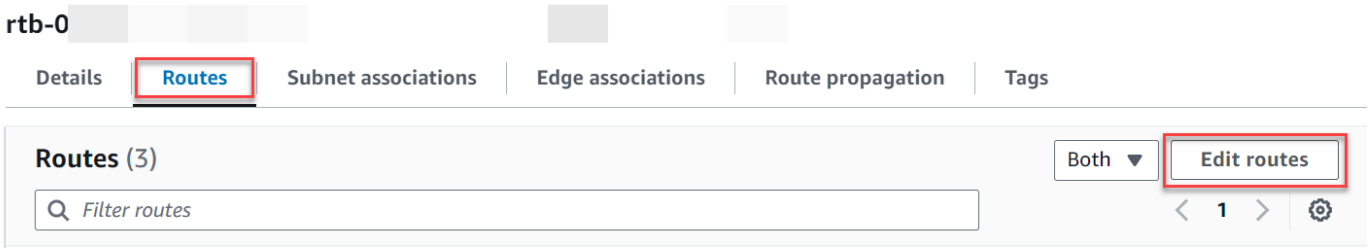
1. 在导航窗格中，选择 Subnets(子网)。
2. 选择 VPC 中的私有子网。（私有子网是其路由表中不具有指向互联网网关的路由的子网。）
3. 选择路由表下的链接。



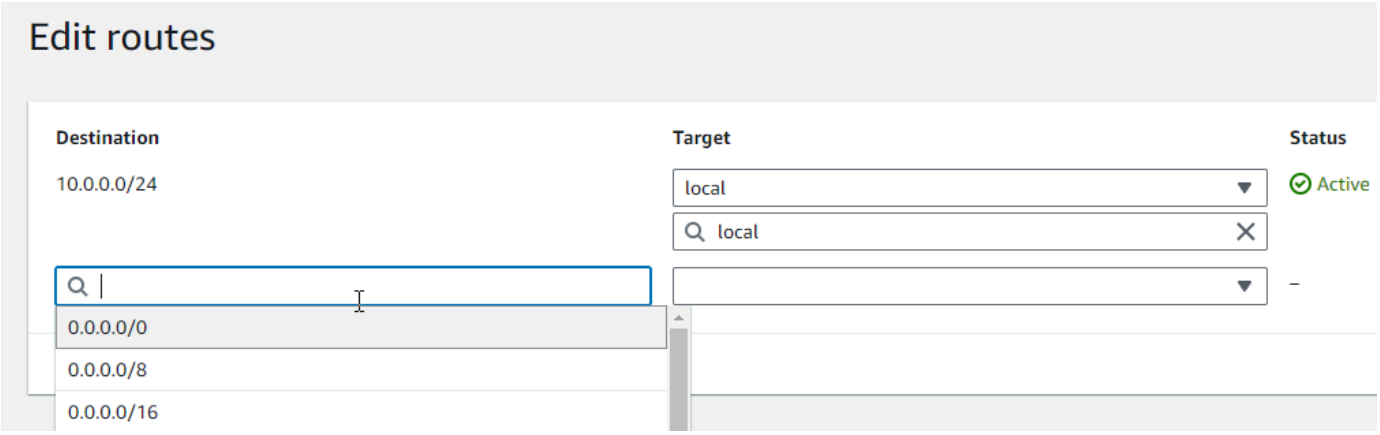
4. 选择路由表 ID 打开路由表的详细信息页面。



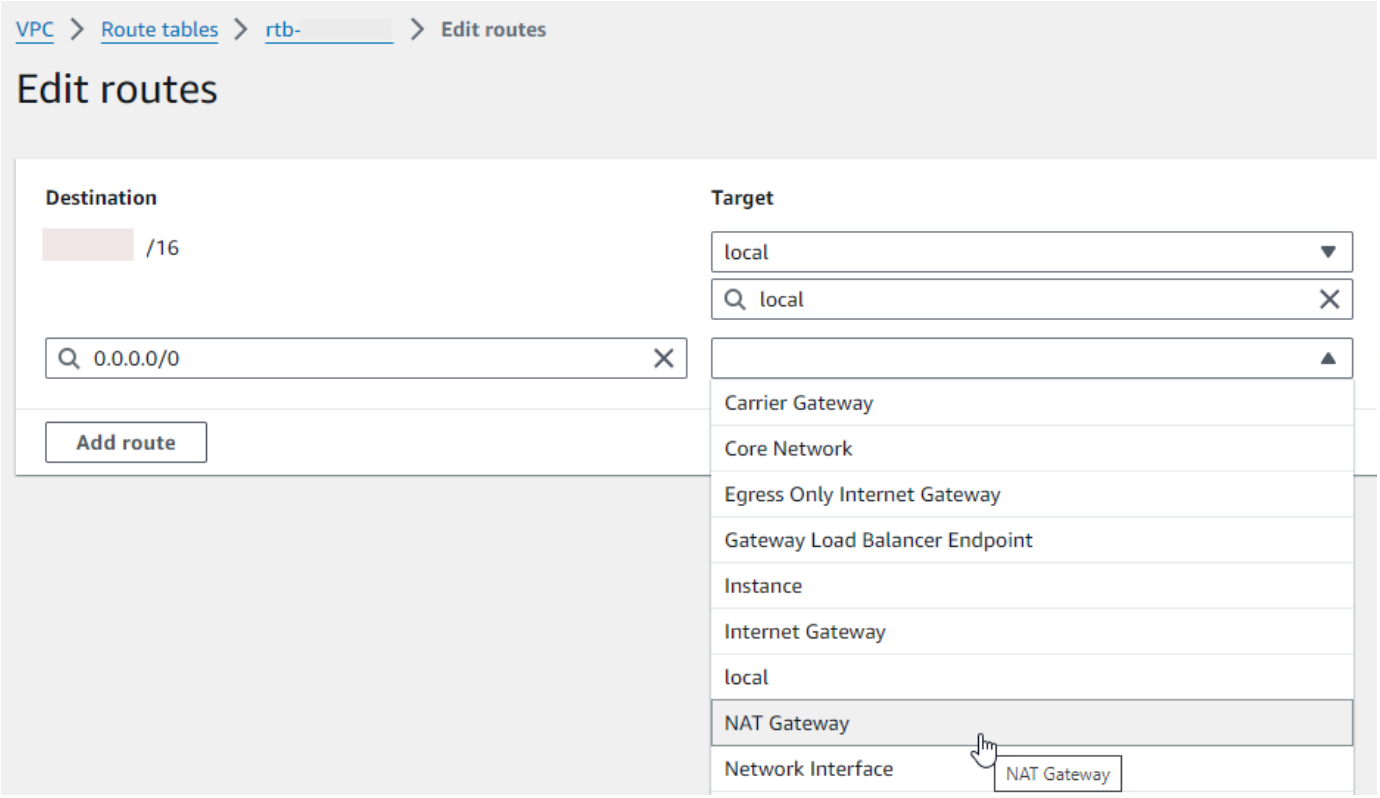
5. 向下滚动并选择路由选项卡，然后选择编辑路由。



6. 选择添加路由，然后在目的地框中输入 `0.0.0.0/0`。



7. 在目标中选择 NAT 网关，然后选择之前创建的 NAT 网关。



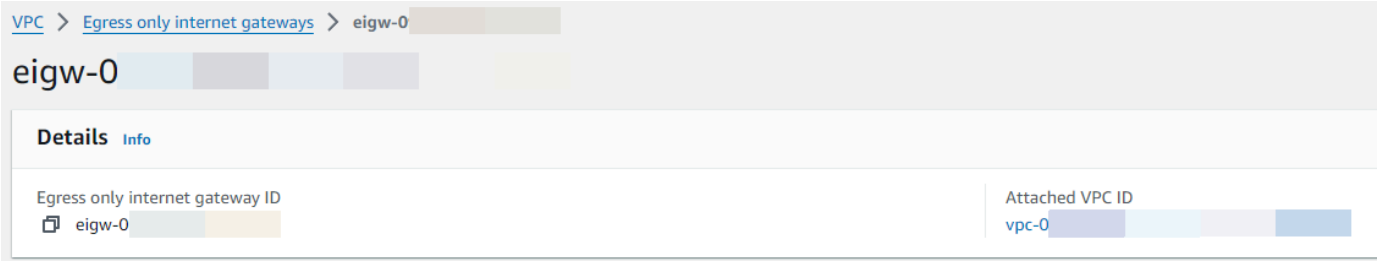
8. 选择 Save changes（保存更改）。

创建“仅出口互联网网关”（仅限 IPv6）

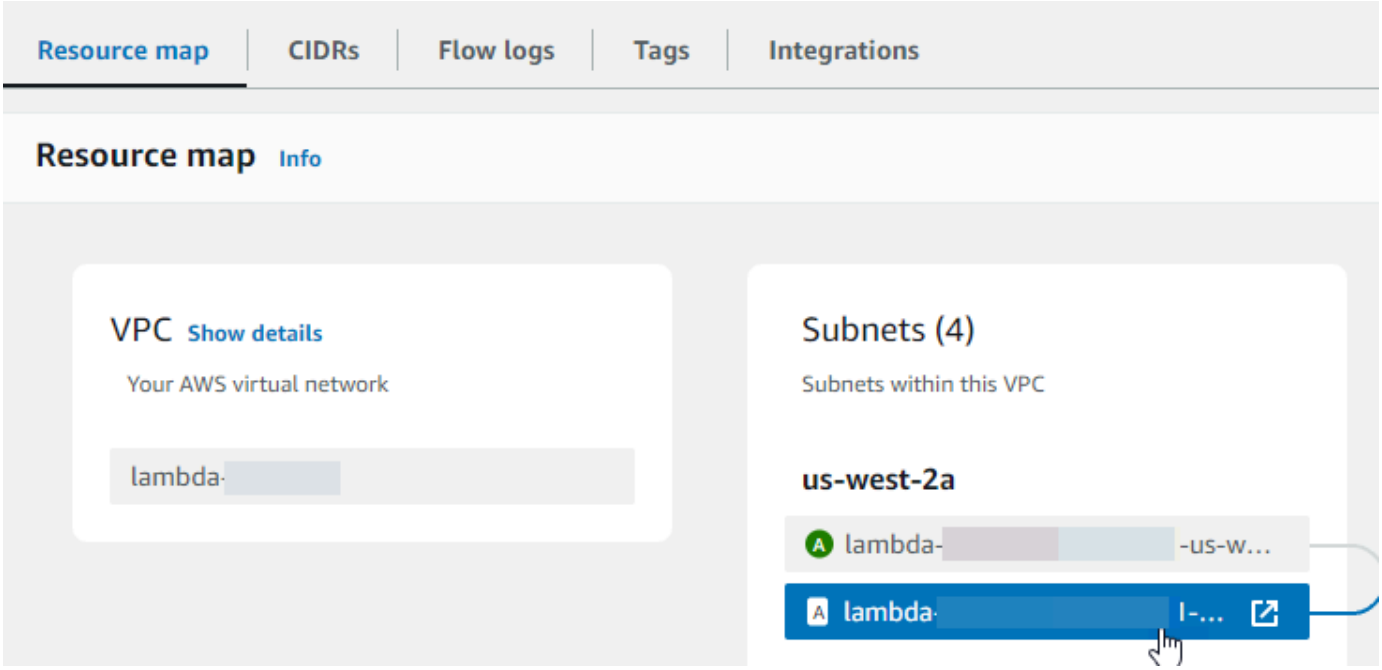
按照以下步骤创建“仅出口互联网网关”，在将之添加到私有子网的路由表中。

创建“仅出口互联网网关”

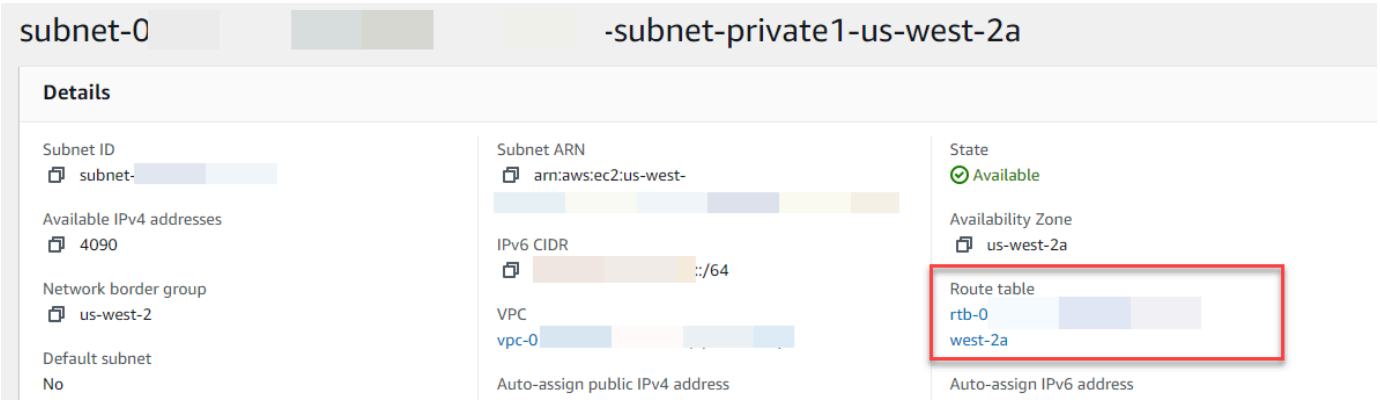
1. 在导航窗格中，选择仅出口互联网网关。
2. 选择创建仅出口互联网网关。
3. （可选）输入名称。
4. 选择要在其中创建“仅出口互联网网关”的 VPC。
5. 选择创建仅出口互联网网关。
6. 选择附加的 VPC ID下的链接。



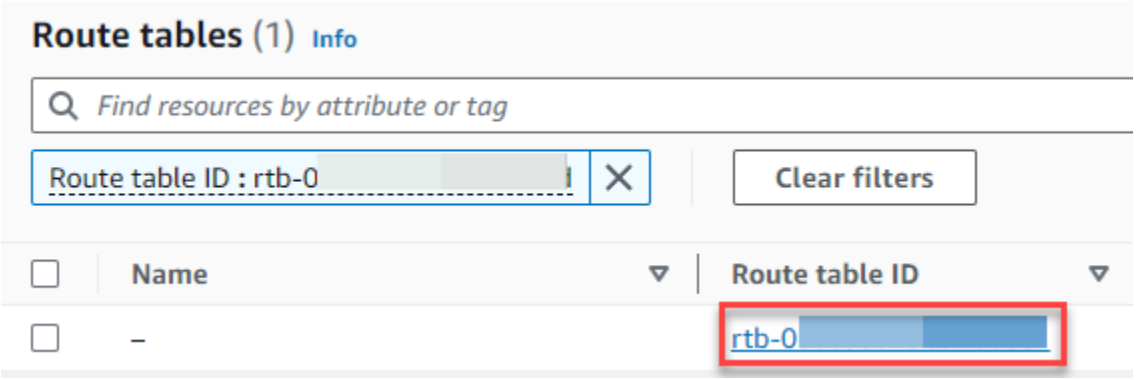
7. 选择 VPC ID 下的链接，打开 VPC 详细信息页面。
8. 向下滚动到资源映射部分，然后选择一个私有子网。（私有子网是其路由表中不具有指向互联网网关的路由的子网。）子网详细信息显示在新选项卡中。



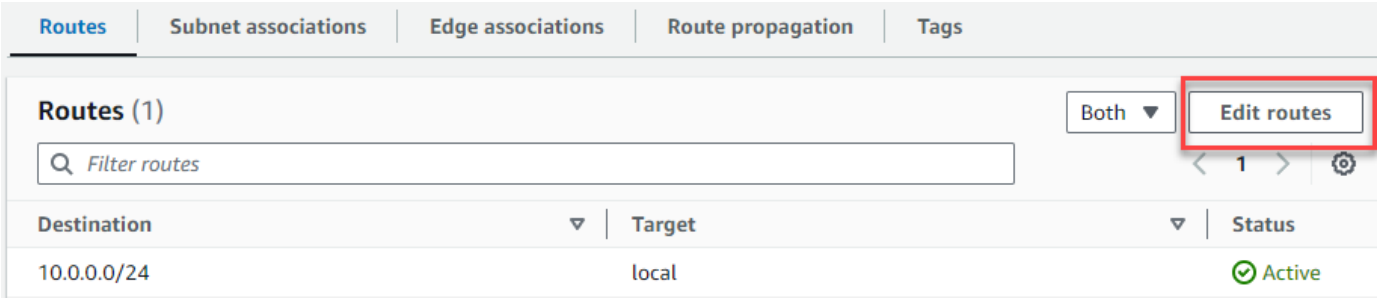
9. 选择路由表下的链接。



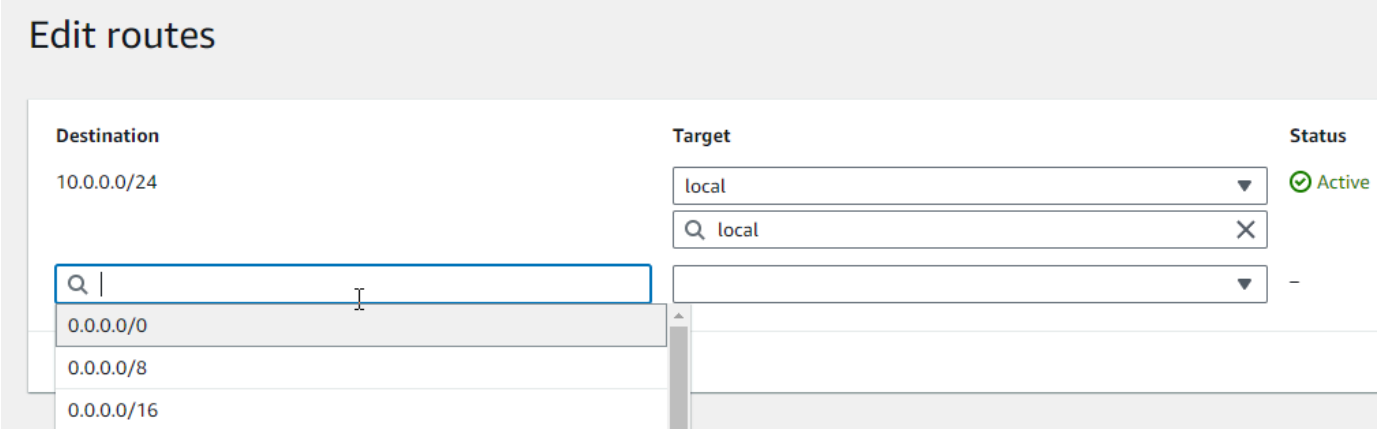
10. 选择路由表 ID 打开路由表的详细信息页面。



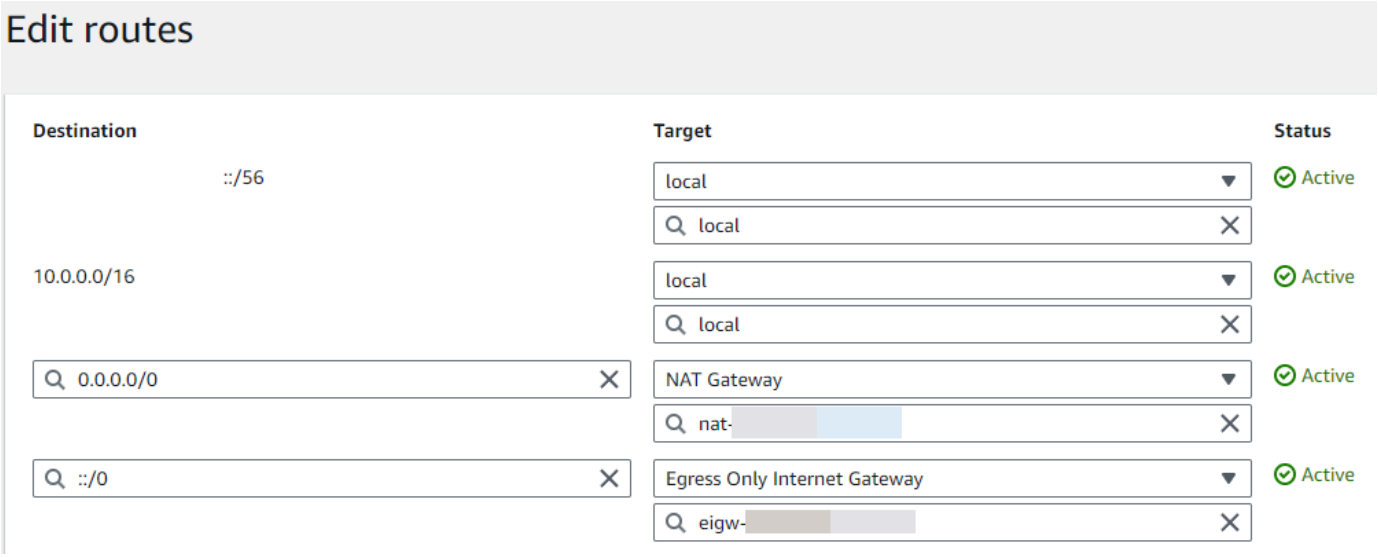
11. 在路由下，选择编辑路由。



12. 选择添加路由，然后在目的地框中输入 `::/0`。



13. 在目标中选择仅限出口互联网网关，然后选择之前创建的网关。



14. 选择 Save changes（保存更改）。

配置 Lambda 函数

在创建函数时配置 VPC

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择 Create function（创建函数）。
3. 在基本信息下的函数名称中输入函数的名称。
4. 展开 Advanced settings（高级设置）。
5. 选择启用 VPC，再选择一个 VPC。
6. （可选）要允许出站 IPv6 流量，请选择允许双堆栈子网的 IPv6 流量。
7. 对于子网，选择全部私有子网。私有子网可以通过 NAT 网关访问互联网。将函数连接到公有子网并不会授予其互联网访问权限。

> **Note**
>
> 如果您已选择允许双堆栈子网的 IPv6 流量，则所有选定的子网都必须具有 IPv4 CIDR 块和 IPv6 CIDR 块。

8. 对于安全组，选择一个允许出站流量的安全组。
9. 选择 Create function (创建函数)。

Lambda 使用 AWSLambdaVPCAccessExecutionRole AWS 托管策略自动创建执行角色。只有在为 VPC 配置创建弹性网络接口时需要此策略中的权限，调用函数时并不需要。要应用最低权限，可以在创建函数和 VPC 配置之后，从执行角色中删除 AWSLambdaVPCAccessExecutionRole 策略。有关更多信息，请参阅 所需的 IAM 权限。

为现有函数配置 VPC

要将 VPC 配置添加到现有函数，该函数的执行角色必须具有创建和管理弹性网络接口的权限。AWSLambdaVPCAccessExecutionRole AWS 托管策略包含这些必要权限。要应用最低权限，可以在创建 VPC 配置之后，从执行角色中删除 AWSLambdaVPCAccessExecutionRole 策略。

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择配置选项卡，然后选择 VPC。
4. 在 VPC 下，选择 Edit（编辑）。
5. 选择 VPC。
6. （可选）要允许出站 IPv6 流量，请选择允许双堆栈子网的 IPv6 流量。
7. 对于子网，选择全部私有子网。私有子网可以通过 NAT 网关访问互联网。将函数连接到公有子网并不会授予其互联网访问权限。

> **Note**
>
> 如果您已选择允许双堆栈子网的 IPv6 流量，则所有选定的子网都必须具有 IPv4 CIDR 块和 IPv6 CIDR 块。

8. 对于安全组，选择一个允许出站流量的安全组。
9. 选择保存。

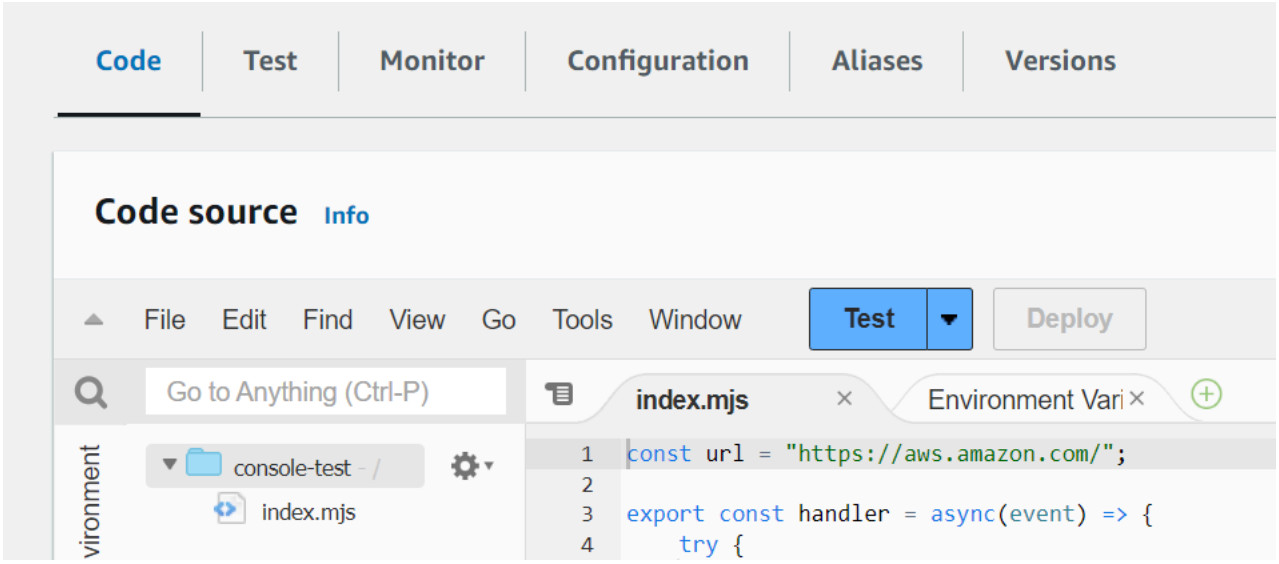
测试此函数

使用以下示例代码，确认连接到 VPC 的函数是否可以访问公有互联网。如果成功，代码将返回 `200` 状态代码。如果失败，函数将超时。

Node.js

此示例使用 `fetch`，它支持 `nodejs18.x` 和更高版本的运行时系统。

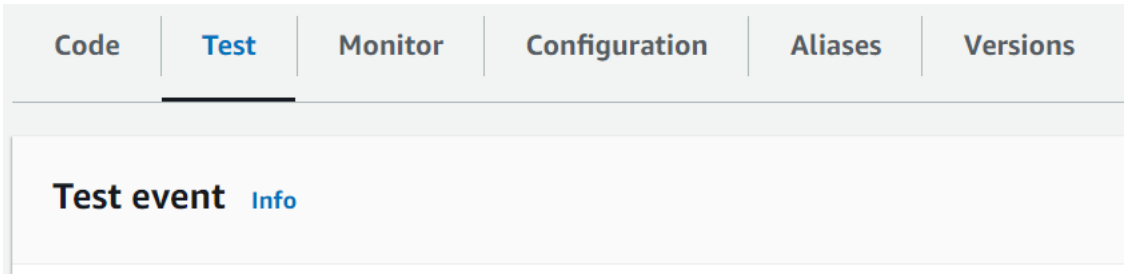
1. 在 Lambda 控制台的代码源窗格中，将以下代码粘贴到 index.mjs 文件中。该函数向公有端点发出 HTTP GET 请求，再返回 HTTP 响应代码来测试该函数是否可以访问公有互联网。



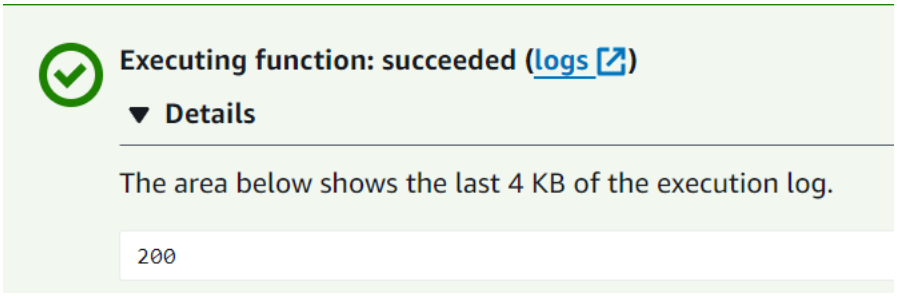
Example 示例：包含 async/await 的 HTTP 请求

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        // fetch is available with Node.js 18 and later runtimes
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

2. 选择部署。
3. 选择测试选项卡。



4. 选择测试。
5. 该函数返回 200 状态代码。此结果表示该函数具有出站互联网访问权限。

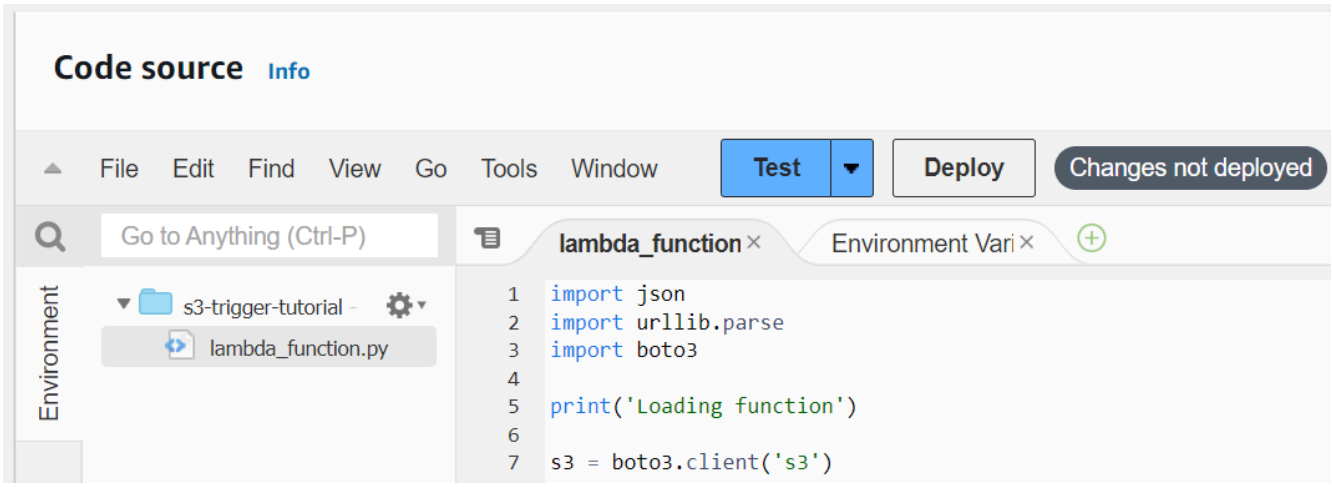


如果该函数无法访问公有互联网，则会收到如下错误消息：

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

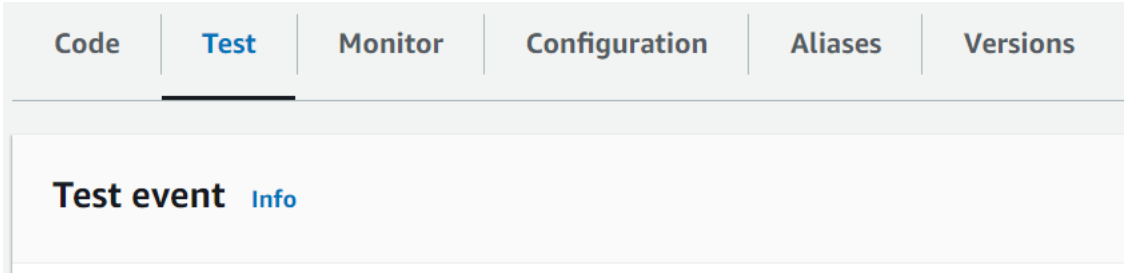
Python

1. 在 Lambda 控制台的代码源窗格中，将以下代码粘贴到 ambda_function.py 文件中。该函数向公有端点发出 HTTP GET 请求，再返回 HTTP 响应代码来测试该函数是否可以访问公有互联网。

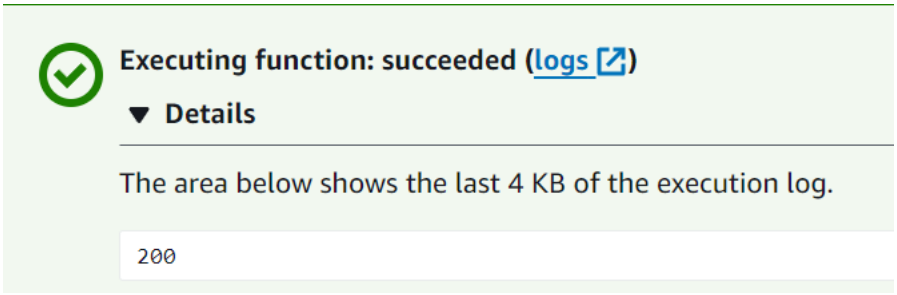



```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

2. 选择部署。
3. 选择测试选项卡。



4. 选择测试。
5. 该函数返回 200 状态代码。此结果表示该函数具有出站互联网访问权限。



如果该函数无法访问公有互联网，则会收到如下错误消息：

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

连接 Lambda 的入站接口 VPC 端点

如果您使用 Amazon Virtual Private Cloud (Amazon VPC) 托管AWS资源，则可以在您的 VPC 和 Lambda 之间建立连接。您可以使用此连接来调用您的 Lambda 函数，而无需跨越公有互联网。

您可以通过创建接口 VPC 终端节点在 VPC 和 Lambda 之间建立私有连接。接口端点由 AWS PrivateLink 提供支持，您可使用该技术通过私有连接访问 Lambda API，而无需采用互联网网关、NAT 设备、VPN 连接或 AWS Direct Connect 连接。VPC 中的实例即使没有公有 IP 地址也可与 Lambda API 进行通信。VPC 和 Lambda 之间的流量不会脱离 AWS 网络。

每个接口端点均由子网中的一个或多个弹性网络接口表示。网络接口提供一个私有 IP 地址，此地址可用作指向 Lambda 的流量的入口点。

小节目录

Lambda 接口端点的注意事项

请务必先查看 Amazon VPC 用户指南中的接口端点属性和限制，然后再为 Lambda 设置接口端点。

您可以从 VPC 调用任何 Lambda API 操作。例如，您可以通过在 VPC 内调用 `Invoke` API 来调用 Lambda 函数。有关 Lambda API 的完整列表，请参阅 Lambda API 参考中的操作。

`use1-az3` 是支持 Lambda VPC 函数的有限容量区域。您不应将此可用区中的子网与 Lambda 函数一起使用，因为这可能会在发生中断时导致可用区冗余减少。

对于持久性连接保持连接状态

Lambda 随时间推移逐渐清除空闲连接，因此您必须使用 keep-alive 指令来维护持久连接。在调用函数时尝试重用空闲连接会导致连接错误。要维护您的持久连接，请使用与运行时关联的 keep-alive 指令。有关示例，请参阅 AWS SDK for JavaScript 开发人员指南 中的在 Node.js 中重复使用具有保持连接功能的连接。

计费注意事项

通过接口端点访问 Lambda 函数不会产生额外费用。有关 Lambda 定价的更多信息，请参阅 AWS Lambda 定价。

AWS PrivateLink 的标准定价适用于 Lambda 的接口端点。您的AWS账户将按在每个可用区中预置的每小时以及通过接口端点处理的数据进行计费。有关接口端点定价的更多信息，请参阅 AWS PrivateLink 定价。

VPC 对等连接注意事项

您可以使用 VPC 对等连接通过接口端点将其他 VPC 连接到 VPC。VPC 对等连接是两个 VPC 之间的网络连接。您可以在自己的两个 VPC 之间建立 VPC 对等连接，或者在自己的 VPC 与其他 AWS 账户中的 VPC 之间建立此连接。VPC 也可以位于两个不同的AWS区域中。

对等 VPC 之间的流量保留在AWS网络上，不会穿越公有互联网。建立对等 VPC 连接后，两个 VPC 中的资源（如 Amazon Elastic Compute Cloud (Amazon EC2) 实例、Amazon Relational Database Service (Amazon RDS) 实例或启用 VPC 的 Lambda 函数等）可以通过在其中一个 VPC 中创建的接口端点访问 Lambda API。

为 Lambda 创建接口端点

您可以使用 Amazon VPC 控制台或 AWS Command Line Interface (AWS CLI) 为 Lambda 创建 VPC 端点。有关更多信息，请参阅 Amazon VPC 用户指南中的创建接口端点。

为 Lambda（控制台）创建接口端点

1. 打开 Amazon VPC 控制台的 Endpoints（端点）页面。
2. 选择 Create Endpoint（创建端点）。
3. 对于服务类别，请确保选择 AWS 服务。
4. 对于服务名称，选择 com.amazonaws.***region***.lambda。验证类型为接口。
5. 创建 VPC 和子网。
6. 要为接口端点启用私有 DNS，请选中 Enable DNS Name（启用 DNS 名称）复选框。我们建议您为 AWS 服务 的 VPC 端点启用私有 DNS 名称。这可以确保使用公有服务端点的请求（例如通过 AWS SDK 发出的请求）解析到您的 VPC 端点。
7. 对于安全组，选择一个或多个安全组。
8. 选择 Create Endpoint（创建端点）。

要使用私有 DNS 选项，您必须设置 VPC 的 `enableDnsHostnames` 和 `enableDnsSupportattributes`。有关更多信息，请参阅 Amazon VPC 用户指南中的查看和更新 VPC 的 DNS 支持。如果为接口端点启用私有 DNS，则可使用区域默认 DNS 名称向 Lambda 发出

API 请求，例如 `lambda.us-east-1.amazonaws.com`。有关更多服务端点，请参阅《AWS 一般参考》中的服务端点和配额。

有关更多信息，请参阅 Amazon VPC 用户指南中的通过接口端点访问服务。

有关使用 AWS CloudFormation 创建和配置端点的信息，请参阅 AWS CloudFormation 用户指南中的 AWS::EC2::VPCEndpoint 资源。

为 Lambda (AWS CLI) 创建接口端点

使用 create-vpc-endpoint 命令并指定 VPC ID、VPC 端点类型（接口）、服务名称、将使用端点的子网以及要与端点的网络接口关联的安全组。例如：

```
aws ec2 create-vpc-endpoint
  --vpc-id vpc-ec43eb89
  --vpc-endpoint-type Interface
  --service-name com.amazonaws.us-east-1.lambda
  --subnet-id subnet-abababab
  --security-group-id sg-1a2b3c4d
```

为 Lambda 创建接口端点策略

要控制哪些用户可以使用您的接口端点，以及用户可以访问哪些 Lambda 功能，您可以将端点策略附加到端点。该策略指定以下信息：

- 可执行操作的主体。
- 主体可以执行的操作。
- 委托人可以对其执行操作的资源。

有关更多信息，请参阅 Amazon VPC 用户指南 中的使用 VPC 终端节点控制对服务的访问。

示例：用于 Lambda 操作的接口端点策略

以下是用于 Lambda 的端点策略示例。连接到端点时，此策略允许 `MyUser` 用户调用函数 `my-function`。

> **Note**
>
> 资源中需要同时包含限定函数和非限定函数 ARN。

```
{
   "Statement":[
      {
         "Principal":
         {
             "AWS": "arn:aws:iam::111122223333:user/MyUser"
         },
         "Effect":"Allow",
         "Action":[
            "lambda:InvokeFunction"
         ],
         "Resource": [
               "arn:aws:lambda:us-east-2:123456789012:function:my-function",
               "arn:aws:lambda:us-east-2:123456789012:function:my-function:*"
            ]
      }
   ]
}
```

配置 Lambda 函数的文件系统访问

您可以配置函数以将 Amazon Elastic File System (Amazon EFS) 文件系统挂载到本地目录。借助 Amazon EFS，您的函数代码可以安全且高并发地访问和修改共享资源。

Sections

执行角色和用户权限

如果文件系统没有用户配置的 AWS Identity and Access Management（IAM）policy，EFS 将使用默认策略，该策略授予对可以使用文件系统挂载目标连接到文件系统的任何客户端的完全访问权限。如果文件系统具有用户配置的 IAM policy，则您的函数的执行角色必须具有正确的 `elasticfilesystem` 权限。

执行角色权限

- elasticfilesystem:ClientMount
- elasticfilesystem:ClientWrite（只读连接不需要）

这些权限包含在 AmazonElasticFileSystemClientReadWriteAccess 托管式策略中。此外，您的执行角色必须具有连接到文件系统的 VPC 所需的权限。

配置文件系统时，Lambda 使用您的权限来验证挂载目标。要配置函数以连接到文件系统，您的用户需要以下权限：

用户权限

- elasticfilesystem:DescribeMountTargets

配置文件系统和访问点

在 Amazon EFS 中创建一个文件系统，该文件系统在函数连接到的每个可用区都有一个挂载目标。为了获得性能和恢复能力，请至少使用两个可用区。例如，在简单配置中，您可能有一个 VPC 包含两个

私有子网，这两个子网位于不同的可用区。函数连接到两个子网，每个子网中都存在一个挂载目标。确保函数和挂载目标使用的安全组允许 NFS 流量（端口 2049）。

> **Note**
>
> 创建文件系统时，您可以选择以后无法更改的性能模式。General purpose (通用) 模式具有较低的延迟，Max I/O (最大 I/O) 模式支持较高的最大吞吐量和 IOPS。如需帮助选择，请参阅 Amazon Elastic File System 用户指南中的 Amazon EFS 性能。

访问点将函数的每个实例连接到该实例连接到的可用区的正确装挂目标。为了获得最佳性能，请使用非根路径创建访问点，并限制您在每个目录中创建的文件数。以下示例在文件系统上创建一个名为 `my-function` 的目录，然后将拥有者 ID 设置为 1001 并具有标准目录权限 (755)。

Example 访问点配置

- 名称 – `files`
- User ID（用户名 ID）– `1001`
- Group ID（组 ID）– `1001`
- 路径 – `/my-function`
- 权限 – `755`
- Owner user ID（拥有者用户 ID）– `1001`
- Group user ID（组用户 ID）– `1001`

当函数使用访问点时，它将被赋予用户 ID 1001 并具有对该目录的完全访问权限。

有关更多信息，请参阅 Amazon Elastic File System 用户指南中的以下主题：

- 为 Amazon EFS 创建资源
- 使用用户、组和权限

连接到文件系统（控制台）

函数通过 VPC 中的本地网络连接到文件系统。函数连接到的子网可以是包含文件系统挂载点的相同子网，也可以是位于同一可用区中的子网，这些子网可以将 NFS 流量（端口 2049）路由到文件系统。

> **Note**
>
> 如果函数尚未连接到 VPC，请参阅授予 Lambda 函数访问 Amazon VPC 中资源的权限。

配置文件系统访问

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration（配置），然后选择 File systems（文件系统）。
4. 在 File system (文件系统) 下，选择 Add file system (添加文件系统)。
5. 配置以下属性：
   - EFS file system（EFS 文件系统）– 同一 VPC 中的文件系统的访问点。
   - Local mount path（本地挂载路径）– 文件系统在 Lambda 函数上的挂载位置，以 `/mnt/` 开头。

> **定价**
>
> Amazon EFS 对存储和吞吐量收费，费率因存储类别而异。有关详细信息，请参阅 Amazon EFS 定价。
> Lambda 对 VPC 之间的数据传输收费。这仅在函数的 VPC 与另一个具有文件系统的 VPC 对等时才适用。费率与同一区域中 VPC 之间的 Amazon EC2 数据传输费率相同。有关详细信息，请参阅 Lambda 定价。

为 Lambda 函数创建别名

您可以为 Lambda 函数创建别名。Lambda 别名是您可以更新的指向函数版本的指针。函数的用户可以使用别名 Amazon 资源名称（ARN）访问函数版本。部署新版本时，您可以更新别名以使用新版本，或者在两个版本之间拆分流量。

Console

使用控制台创建别名

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Aliases（别名），然后选择 Create alias（创建别名）。
4. 在创建别名页面上，执行以下操作：
   a. 输入别名的名称。
   b. （可选）输入别名的描述。
   c. 对于版本，选择希望别名指向的函数版本。
   d. （可选）要在别名上配置路由，请展开加权别名。有关更多信息，请参阅为 Lambda 别名创建路由配置。
   e. 选择保存。

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建别名，请使用 create-alias 命令。

```
aws lambda create-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number \
  --description " "
```

要更改别名以便指向函数的新版本，请使用 update-alias 命令。

```
aws lambda update-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number
```

要删除别名，请使用 delete-alias 命令。

```
aws lambda delete-alias \
  --function-name my-function \
  --name alias-name
```

以上步骤中的 AWS CLI 命令对应于以下 Lambda API 操作：

- CreateAlias
- UpdateAlias
- DeleteAlias

在事件源和权限策略中使用 Lambda 别名

每个别名都有唯一的 ARN。别名只能指向函数版本，而不能指向其他别名。可以更新别名以便指向函数的新版本。

事件源（Amazon Simple Storage Service (Amazon S3) 等）会调用您的 Lambda 函数。这些事件源维护一个映射，该映射标识在发生事件时要调用的函数。如果在映射配置中指定 Lambda 函数别名，则当函数版本发生更改时，您不需要更新映射。有关更多信息，请参阅Lambda 如何处理来自基于流和队列的事件源的记录。

在资源策略中，您可以为事件源授予权限，以便使用您的 Lambda 函数。如果在策略中指定别名 ARN，则当函数版本发生更改时，您不需要更新策略。

资源策略

您可以使用基于资源的策略授予服务、资源或账户对您的函数的访问权限。该权限的范围取决于您要将其应用于别名、版本还是整个函数。例如，如果使用别名（如 `helloworld:PROD`），该权限将允许您使用别名 ARN (`helloworld`) 调用 `helloworld:PROD` 函数。

如果您尝试在不指定别名或具体版本的情况下调用该函数，则会出现权限错误。即使您尝试直接调用与别名关联的函数版本，也会发生此权限错误。

例如，以下 AWS CLI 命令会向 Amazon S3 授予在 Amazon S3 代表 `amzn-s3-demo-bucket` 执行操作时调用 `helloworld` 函数的 PROD 别名的权限。

```
aws lambda add-permission \
  --function-name helloworld \
  --qualifier PROD \
  --statement-id 1 \
  --principal s3.amazonaws.com \
  --action lambda:InvokeFunction \
  --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
  --source-account 123456789012
```

有关在策略中使用资源名称的更多信息，请参阅微调策略的“资源和条件”部分。

为 Lambda 别名创建路由配置

对别名使用路由配置，将一部分流量发送到第二个函数版本。例如，您可以通过配置别名以便将大部分流量发送到现有版本，并且仅将一小部分流量发送到新版本，来降低部署新版本的风险。

Lambda 使用简单的概率模型在两个函数版本之间分发流量。低流量级别情况下，您可能会看到每个版本上配置的流量百分比与实际流量百分比之间的差异很大。如果您的函数使用预配置并发，则可以避免溢出调用，方法是在别名路由处于活动状态期间配置更多的预配置并发实例。

您可以将别名最多指向两个 Lambda 函数版本。版本必须满足以下条件：

- 两个版本必须具有相同的执行角色。
- 两个版本必须具有相同的死信队列配置，或者都没有死信队列配置。
- 这两个版本必须都已发布。别名不能指向 $LATEST。

Console

使用控制台对别名配置路由

> **Note**
>
> 确保函数至少具有两个已发布版本。要创建其他版本，请按照 管理 Lambda 函数版本中的说明操作。

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Aliases（别名），然后选择 Create alias（创建别名）。
4. 在创建别名页面上，执行以下操作：
   a. 输入别名的名称。
   b. （可选）输入别名的描述。
   c. 对于版本，选择希望别名指向的第一个函数版本。
   d. 展开加权别名。
   e. 对于其他版本，请选择希望别名指向的第二个函数版本。
   f. 对于权重 (%)，输入函数的权重值。权重是在调用别名时分配给该版本的流量百分比。第一个版本接收剩余权重。例如，如果为其他版本指定 10%，则会自动为第一个版本分配 90%。
   g. 选择保存。

AWS CLI

使用 create-alias 和 update-alias AWS CLI 命令配置两个函数版本之间的流量权重。创建或更新别名时，您可以使用 `routing-config` 参数指定流量权重。

以下示例会创建一个名为 routing-alias 的 Lambda 函数别名，该别名指向函数的版本 1。函数的版本 2 接收 3% 的流量。其余 97% 的流量路由到版本 1。

```
aws lambda create-alias \
  --name routing-alias \
  --function-name my-function \
  --function-version 1  \
  --routing-config AdditionalVersionWeights={"2"=0.03}
```

使用 `update-alias` 命令可提高流入版本 2 的流量的百分比。在下面的示例中，将该流量提高到 5%。

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function \
  --routing-config AdditionalVersionWeights={"2"=0.05}
```

要将所有流量路由到版本 2，请使用 `update-alias` 命令更改 `function-version` 属性，以使别名指向版本 2。该命令还将重置路由配置。

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function  \
  --function-version 2 \
  --routing-config AdditionalVersionWeights={}
```

以上步骤中的 AWS CLI 命令对应于以下 Lambda API 操作：

- CreateAlias
- UpdateAlias

确定调用的版本

配置两个函数版本之间的流量权重时，可以通过两种方法确定已调用的 Lambda 函数版本：

- CloudWatch Logs – 对于每个函数调用，Lambda 会自动将包含调用的版本 ID 的 START 日志条目发出到 Amazon CloudWatch Logs。以下是示例：

  ```
  19:44:37 START RequestId: request id Version: $version
  ```

  对于别名调用，Lambda 会使用 Executed Version 维度按调用的版本筛选指标数据。有关更多信息，请参阅查看 Lambda 函数的指标。
- 响应负载 (同步调用) – 同步函数调用的响应包含 x-amz-executed-version 标头以指示已调用的函数版本。

管理 Lambda 函数版本

您可以使用版本来管理函数的部署。例如，您可以发布函数的新版本用于测试版测试，而不会影响稳定的生产版本的用户。您每次发布函数时，Lambda 都会为函数创建一个新版本。新版本是函数的未发布版本的副本。未发布版本的名为 $LATEST。

> **Note**
>
> 要创建函数的新版本，必须先更改未发布的版本（$LATEST）。这些更改可能会包括更新代码或修改配置设置。如果 $LATEST 与之前发布的版本相同，则在将更改部署到 $LATEST 之前，您无法创建新版本。

发布函数版本后，其代码、运行时、架构、内存、层和大多数其他配置设置都不可变。这意味着，如果不从 $LATEST 发布新版本，则无法更改这些设置。您可以为已发布函数版本配置以下项目：

- 触发
- 目标
- 预配置并发
- 异步调用
- 数据库连接和代理

> **Note**
>
> 在自动模式下使用运行时管理控件时，此函数版本使用的运行时版本会自动更新。使用 Function update（函数更新）或 Manual（手动）模式时，不会更新运行时版本。有关更多信息，请参阅 the section called “运行时版本更新”。

Sections

- 创建函数版本
- 使用版本
- 授予权限

创建函数版本

只能在函数的未发布版本上更改函数代码和设置。在您发布版本时，Lambda 会锁定代码和大多数设置，以便为该版本的用户维持一致的体验。

您可以使用 Lambda 控制台创建函数版本。

创建新函数版本

1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数，然后选择 Versions（版本）。
3. 在版本配置页面上，选择 Publish new version（发布新版本）。
4. （可选）输入版本说明。
5. 选择 发布。

或者，您可以使用 PublishVersion API 操作发布函数版本。

以下 AWS CLI 命令发布函数的新版本。响应返回有关新版本的配置信息，包括版本号和具有版本后缀的函数 ARN。

```
aws lambda publish-version --function-name my-function
```

您应看到以下输出：

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:1",
  "Version": "1",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "Runtime": "nodejs20.x",
  ...
}
```

> **Note**
>
> Lambda 会分配单调递增的序列号，以便进行版本控制。Lambda 不会重复使用版本号，即使是删除并重新创建函数之后也是如此。

使用版本

您可以使用限定的 ARN 或非限定的 ARN 来引用您的 Lambda 函数。

- 限定的 ARN – 具有版本后缀的函数 ARN。以下示例引用 `helloworld` 函数的版本 42。

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:42
```

- 非限定的 ARN – 不具有版本后缀的函数 ARN。

```
arn:aws:lambda:aws-region:acct-id:function:helloworld
```

您可以在所有相关 API 操作中使用限定或非非限定的 ARN。但是，不能使用非限定 ARN 来创建别名。

如果您决定不发布函数版本，可以使用事件源映射中的限定或非限定 ARN 来调用函数。当您使用非限定 ARN 调用函数时，Lambda 会隐式调用 $LATEST。

Lambda 仅在代码从未发布过或自上次发布版本后已发生更改时，才会发布新函数版本。如果没有任何更改，函数版本将保持为上次发布的版本。

每个 Lambda 函数版本的限定 ARN 都是唯一的。发布版本后，您无法更改 ARN 或函数代码。

授予权限

您可以使用基于资源的策略或基于身份的策略授予对函数的访问权限。权限的范围取决于您要将策略应用于整个函数还是函数的某个版本。有关策略中的函数资源名称的更多信息，请参阅微调策略的“资源和条件”部分。

您可以使用函数别名，简化事件源和 AWS Identity and Access Management (IAM) 策略的管理。有关更多信息，请参阅为 Lambda 函数创建别名。

在 Lambda 函数上使用标签

可以通过为函数添加标签来组织和管理资源。标签是与资源关联的自由格式键值对，在 AWS 服务中受支持。有关标签用例的更多信息，请参阅《Tagging AWS Resources and Tag Editor Guide》中的 Common tagging strategies。

标签应用在函数级别，而不是应用于版本或别名。标签不会包含在您发布版本时 AWS Lambda 创建快照的版本特定配置的快照中。可以使用 Lambda API 来查看和更新标签。在 Lambda 控制台中管理特定函数时，还可以查看和更新标签。

Sections

使用标签所需的权限

要允许 AWS Identity and Access Management（IAM）身份（用户、组或角色）读取资源或为其设置标签，请授予该身份相应的权限：

- lambda:ListTags – 当资源有标签时，将此权限授予需要在其上调用 `ListTags` 的任何人。对于带标签的函数，`GetFunction` 也需要此权限。
- lambda:TagResource – 将此权限授予需要调用 `TagResource` 或执行在创建时授予标记的操作的任何人。

有关更多信息，请参阅 Lambda 的基于身份的 IAM policy。

通过 Lambda 控制台使用标签

您可以使用 Lambda 控制台创建具有标签的函数、向现有函数添加标签以及按添加的标签筛选函数。

要在创建函数时添加标签

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择 Create function（创建函数）。
3. 选择 Author from scratch（从头开始编写）或 Container image（容器映像）。

4. 在基本信息下，设置您的函数。有关配置函数的更多信息，请参阅 配置函数。
5. 展开 Advanced settings（高级设置），然后选择 Enable tags（启用标签）。
6. 选择 Add new tag（添加新标签），然后输入 Key（键）和可选 Value（值）。要添加更多标签，请重复此步骤。
7. 选择 Create function（创建函数）。

要将标签添加到现有函数

1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数的名称。
3. 选择 Configuration（配置），然后选择 Tags（标签）。
4. 在标签下，选择管理标签。
5. 选择 Add new tag（添加新标签），然后输入 Key（键）和可选 Value（值）。要添加更多标签，请重复此步骤。
6. 选择保存。

使用标签过滤函数

1. 打开 Lamba 控制台的函数页面。
2. 选择搜索框以查看函数属性和标签键列表。
3. 选择一个标签键以查看当前 AWS 区域中正在使用的值的列表。
4. 选择使用：“tag-name”以查看所有使用此键标记的函数，或者选择一个运算符进一步按值筛选。
5. 选择标签值以按标签键和值的组合进行筛选。

搜索栏还支持搜索标签键。输入 `tag` 以仅查看标签键列表，或输入键的名称以在列表中查找它。

通过 AWS CLI 使用标签

可以使用 Lambda API 在现有 Lambda 资源（包括函数）上添加和删除标签。还可以在创建函数时添加标签，这样就可以在资源的整个生命周期中对其进行标记。

使用 Lambda 标签 API 更新标签

可以通过 TagResource 和 UntagResource API 操作，添加和删除受支持 Lambda 资源的标签。

可以使用 AWS CLI 调用这些操作。要向现有资源添加标签，请使用 `tag-resource` 命令。此示例添加了两个标签，一个带有键 *Department*，另一个带有键 *CostCenter*。

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

要删除标签，请使用 `untag-resource` 命令。此示例删除了键为 *Department* 的标签。

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

在创建函数时添加标签

要使用标签创建新的 Lambda 函数，请使用 CreateFunction API 操作。指定 `Tags` 参数。您可以使用 `create-function` CLI 命令和 `--tags` 选项调用此操作。在将标签参数与 `CreateFunction` 一起使用之前，请确保您的角色拥有标记资源的权限以及此操作所需的常规权限。有关标记权限的更多信息，请参阅 the section called “使用标签所需的权限”。此示例添加了两个标签，一个带有键 *Department*，另一个带有键 *CostCenter*。

```
aws lambda create-function --function-name my-function
--handler index.js --runtime nodejs20.x \
--role arn:aws:iam::123456789012:role/lambda-role \
--tags Department=Marketing,CostCenter=1234ABCD
```

查看函数上的标签

要查看应用于特定 Lambda 资源的标签，请使用 `ListTags` API 操作。有关更多信息，请参阅 ListTags。

可以提供 ARN（Amazon 资源名称），以使用 `list-tags` AWS CLI 命令调用此操作。

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier
```

可以通过 GetFunction API 操作来查看应用于特定资源的标签。类似的功能不适用于其他资源类型。

可以使用 `get-function` CLI 命令调用此操作：

```
aws lambda get-function --function-name my-function
```

按标签筛选资源

您可以使用 AWS Resource Groups Tagging API GetResources API 操作按标签筛选资源。GetResources 操作最多可接收 10 个筛选条件，每个筛选条件包含一个标签键和最多 10 个标签值。提供具有 ResourceType 的 GetResources，可按特定资源类型进行筛选。

可以使用 get-resources AWS CLI 命令调用此操作。有关使用 get-resources 的示例，请参阅《AWS CLI Command Reference》中的 get-resources。

Lambda 函数的响应流

配置 Lambda 函数 URL 以将响应负载流式传输回客户端。响应流式处理可通过提高首字节时间（TTFB）性能，使延迟敏感型应用程序受益。这是因为您可以在部分响应可用时将其发送回客户端。此外，您可以使用响应流式处理来构建返回较大负载的函数。响应流负载的软限制为 20MB，而缓冲响应的软限制为 6MB。流式处理响应还意味着您的函数不需要在内存中容纳整个响应。对于非常大的响应，这样可以减少您需要为函数配置的内存量。

Lambda 流式处理您的响应的速度取决于响应大小。函数响应的前 6MB 的流式处理速率无上限。对于大于 6MB 的响应，响应的其余部分受带宽上限的限制。有关流式处理带宽的详细信息，请参阅响应流式处理的带宽限制。

流式处理响应会产生费用。有关更多信息，请参阅AWS Lambda 定价。

Lambda 在 Node.js 托管式运行时系统上支持响应流式处理。对于其他语言，您也可以使用带有自定义运行时系统 API 集成的自定义运行时系统来流式处理响应或使用 Lambda Web Adapter。您可以通过 Lambda 函数 URL、AWS SDK 或使用 Lambda InvokeWithResponseStream API 来流式处理响应。

> **Note**
>
> 通过 Lambda 控制台测试函数时，将始终显示缓冲响应。

主题

响应流式处理的带宽限制

您的函数响应负载的前 6MB 的带宽无上限。在此初始突增之后，Lambda 会以最大 2MBps 的速率流式处理您的响应。如果您的函数响应从未超出 6MB，则此带宽限制永远不适用。

> **Note**
>
> 带宽限制仅适用于您的函数的响应负载，不适用于您的函数的网络访问。

无上限带宽的速率因多种因素而异，其中包括函数的处理速度。通常，函数响应的前 6MB 的速率高于 2MBps。如果您的函数将响应流式处理到 AWS 以外的目标，则流式处理速率还取决于外部互联网连接的速度。

编写支持响应流式处理的 Lambda 函数

为响应流式处理函数编写处理程序不同于典型的处理程序模式。编写流式处理函数时，请确保执行以下操作：

- 使用本机 Node.js 运行时系统提供的 `awslambda.streamifyResponse()` 装饰器包装函数。
- 正常结束流，以确保所有数据处理完成。

配置处理程序函数以流式处理响应

要向运行时系统指示 Lambda 应该流式处理函数的响应，您必须使用 `streamifyResponse()` 装饰器包装函数。从而指示运行时系统使用正确的响应流式处理逻辑路径，同时确保函数能够流式处理响应。

`streamifyResponse()` 装饰器接受可接受以下参数的函数：

- `event` – 提供有关函数 URL 的调用事件的信息，例如 HTTP 方法、查询参数和请求正文。
- `responseStream` – 提供可写流。
- `context` – 提供的方法和属性包含有关调用、函数和执行环境的信息。

`responseStream` 对象为 Node.js `writableStream`。与任何此类流一样，您应该使用 `pipeline()` 方法。

Example 支持响应流式处理的处理程序

```
const pipeline = require("util").promisify(require("stream").pipeline);
const { Readable } = require('stream');

exports.echo = awslambda.streamifyResponse(async (event, responseStream, _context) => {
  // As an example, convert event to a readable stream.
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));

  await pipeline(requestStream, responseStream);
});
```

尽管 responseStream 提供了写入流的 write() 方法，但建议您尽可能使用 pipeline()。使用 pipeline() 能够确保可写流不会被速度更快的可读流所淹没。

结束流

确保在处理程序返回之前正确结束流。pipeline() 方法会自动处理此问题。

对于其他使用案例，请调用 responseStream.end() 方法以正确结束流。此方法表示不应向流写入更多数据。如果您使用 pipeline() 或 pipe() 写入流，则不需要使用此方法。

Example 使用 pipeline() 结束流的示例

```
const pipeline = require("util").promisify(require("stream").pipeline);

exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  await pipeline(requestStream, responseStream);
});
```

Example 不使用 pipeline() 结束流的示例

```
exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  responseStream.write("Hello ");
  responseStream.write("world ");
  responseStream.write("from ");
  responseStream.write("Lambda!");
  responseStream.end();
});
```

使用 Lambda 函数 URL 调用支持响应流式处理的函数

> **Note**
>
> 您必须使用函数 URL 调用函数才能流式处理响应。

您可以通过更改函数 URL 的调用模式来调用支持响应流式处理的函数。调用模式决定 Lambda 使用哪个 API 操作来调用函数。可用的调用模式有：

- BUFFERED – 这是默认选项。Lambda 通过 Invoke API 操作调用函数。负载完成后，调用结果可用。最大负载大小为 6MB。
- RESPONSE_STREAM – 使函数能够在负载结果可用时对其进行流式处理。Lambda 通过 InvokeWithResponseStream API 操作调用函数。最大响应负载大小为 20MB。但是，您可以请求提高限额。

通过直接调用 Invoke API 操作，您仍然可以在不进行响应流式处理的情况下调用函数。但是，Lambda 会流式处理通过函数 URL 发出的调用的所有响应负载，直到您将调用模式更改为 BUFFERED。

Console

设置函数 URL 的调用模式（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择您要为其设置调用模式的函数的名称。
3. 选择 Configuration（配置）选项卡，然后选择 Function URL（函数 URL）。
4. 选择编辑，然后选择其他设置。
5. 在调用模式下，选择所需的调用模式。
6. 选择保存。

AWS CLI

设置函数 URL 的调用模式（AWS CLI）

```
aws lambda update-function-url-config \
  --function-name my-function \
  --invoke-mode RESPONSE_STREAM
```

AWS CloudFormation

设置函数 URL 的调用模式（AWS CloudFormation）

```
MyFunctionUrl:
  Type: AWS::Lambda::Url
  Properties:
    AuthType: AWS_IAM
    InvokeMode: RESPONSE_STREAM
```

有关配置函数 URL 的更多信息，请参阅 Lambda 函数 URL。

教程：使用函数 URL 创建响应流式处理 Lambda 函数

在本教程中，您将创建一个 Lambda 函数，该函数定义为 .zip 文件存档，其函数 URL 端点会返回响应流。有关配置函数 URL 的更多信息，请参阅 函数 URL。

先决条件

本教程假设您对 Lambda 基本操作和 Lambda 控制台有一定了解。如果您还没有了解，请按照 使用控制台创建 Lambda 函数 中的说明创建您的第一个 Lambda 函数。

要完成以下步骤，您需要 AWS CLI 版本 2。在单独的数据块中列出了命令和预期输出：

```
aws --version
```

您应看到以下输出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

对于长命令，使用转义字符 (\) 将命令拆分为多行。

在 Linux 和 macOS 中，可使用您首选的 shell 和程序包管理器。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常与 Lambda 一起使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。本指南中的示例 CLI 命令使用 Linux 格式。如果您使用的是 Windows CLI，则必须重新格式化包含内联 JSON 文档的命令。

创建执行角色

创建执行角色，向您的 Lambda 函数授予访问AWS资源的权限。

创建执行角色

1. 打开 AWS Identity and Access Management（IAM）控制台的 Roles（角色）页面。
2. 选择 Create role（创建角色）。

3. 创建具有以下属性的角色：

   - 可信实体类型 – AWS 服务
   - 使用场景 – Lambda
   - 权限 – AWSLambdaBasicExecutionRole
   - Role name（角色名称）– **response-streaming-role**

AWSLambdaBasicExecutionRole 策略具有函数将日志写入 Amazon CloudWatch Logs 所需的权限。创建角色后，记下其 Amazon 资源名称（ARN）。下一步中需要使用该值。

创建响应流式处理函数（AWS CLI）

使用 AWS Command Line Interface（AWS CLI）创建具有函数 URL 端点的响应流式处理 Lambda 函数。

创建能够流式处理响应的函数

1. 将以下代码示例复制到名为 `index.mjs` 的文件中。

```
import util from 'util';
import stream from 'stream';
const { Readable } = stream;
const pipeline = util.promisify(stream.pipeline);

/* global awslambda */
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));
  await pipeline(requestStream, responseStream);
});
```

2. 创建部署程序包。

```
zip function.zip index.mjs
```

3. 使用 `create-function` 命令创建 Lambda 函数。将 `--role` 的值替换为上一步中的角色 ARN。

```
aws lambda create-function \
  --function-name my-streaming-function \
```

```
  --runtime nodejs16.x \
  --zip-file fileb://function.zip \
  --handler index.handler \
  --role arn:aws:iam::123456789012:role/response-streaming-role
```

创建函数 URL

1. 向函数添加基于资源的策略，以允许对函数 URL 进行访问。将 `--principal` 的值替换为您的 AWS 账户 ID。

```
aws lambda add-permission \
  --function-name my-streaming-function \
  --action lambda:InvokeFunctionUrl \
  --statement-id 12345 \
  --principal 123456789012 \
  --function-url-auth-type AWS_IAM \
  --statement-id url
```

2. 使用 `create-function-url-config` 命令为函数创建 URL 端点。

```
aws lambda create-function-url-config \
  --function-name my-streaming-function \
  --auth-type AWS_IAM \
  --invoke-mode RESPONSE_STREAM
```

测试函数 URL 端点

通过调用函数来测试集成。您可以在浏览器中打开函数的 URL，也可以使用 curl。

```
curl --request GET "<function_url>" --user "<key:token>" --aws-sigv4 "aws:amz:us-east-1:lambda" --no-buffer
```

我们的函数 URL 使用 `IAM_AUTH` 身份验证类型。这意味着您必须使用 AWS 访问密钥和私有密钥才能签署请求。在上一条命令中，将 `<key:token>` 替换为 AWS 访问密钥 ID。请在出现提示时输入 AWS 私有密钥。如果没有 AWS 私有密钥，可以改为使用临时 AWS 凭证。

清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。
4. 在文本输入字段中输入角色名称，然后选择 Delete（删除）。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **`delete`**，然后选择 Delete（删除）。

了解 Lambda 函数调用方法

部署 Lambda 函数后，可以通过多种方式调用函数：

- Lambda 控制台：使用 Lambda 控制台快速创建测试事件来调用函数。
- AWS SDK：使用 AWS SDK 以编程方式调用函数。
- 调用 API – 使用 Lambda 调用 API 直接调用函数。
- AWS Command Line Interface（AWS CLI）：使用 `aws lambda invoke` AWS CLI 命令从命令行直接调用函数。
- 函数 URL HTTP(S) 端点：使用函数 URL 创建可用于调用函数的专用 HTTP(S) 端点。

这些全部都是直接调用函数的方法。在 Lambda 中，常见用例是根据应用程序中其他地方发生的事件调用函数。某些服务可以利用每个新事件调用 Lambda 函数。这称为触发器。对基于流和队列的服务，Lambda 使用批量记录调用该函数。这称为事件源映射。

调用函数时，您可以选择同步或异步调用。使用同步调用时，您将等待函数处理该事件并返回响应。使用异步调用时，Lambda 会将事件排队等待处理并立即返回响应。调用 API 中的 `InvocationType` 请求参数会决定 Lambda 如何调用函数。`RequestResponse` 的值表示同步调用，`Event` 的值则表示异步调用。

要通过 IPv6 调用您的函数，请使用 Lambda 的公共双堆栈端点。双堆栈端点同时支持 IPv4 和 IPv6。Lambda 双堆栈端点使用以下语法：

```
protocol://lambda.us-east-1.api.aws
```

您也可以使用 Lambda 函数 URL 通过 IPv6 调用函数。函数 URL 的端点具有以下格式：

```
https://url-id.lambda-url.us-east-1.on.aws
```

如果函数调用出现错误，对于同步调用，则查看响应中的错误消息并手动重试调用。对于异步调用，Lambda 会自动处理重试并将调用记录发送到目标。

同步调用 Lambda 函数

当您同步调用某个函数时，Lambda 会运行该函数并等待响应。当函数完成时，Lambda 返回来自函数代码的响应以及其他数据，例如已调用函数的版本。要使用 AWS CLI 同步调用函数，请使用 `invoke` 命令。

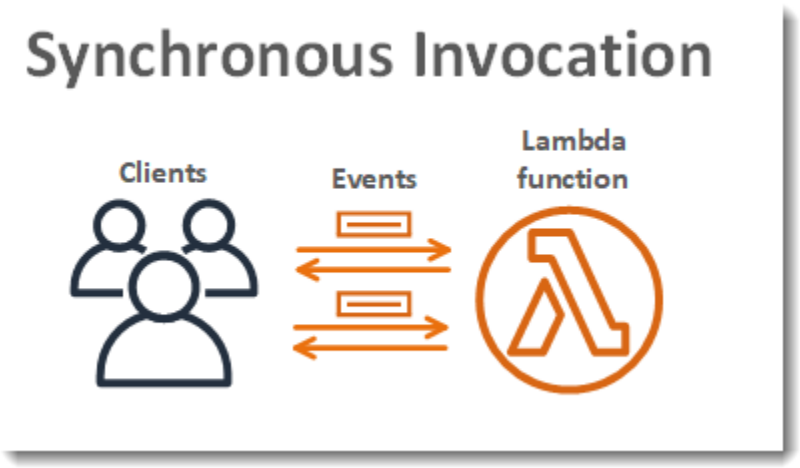
```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

您应看到以下输出：

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

下图显示了同步调用 Lambda 函数的客户端。Lambda 将事件直接发送给函数，并将函数的响应发送回调用方。



`payload` 是一个包含 JSON 格式事件的字符串。AWS CLI 用来写入函数响应的文件的名称为 `response.json`。如果函数返回对象或错误，则响应正文是 JSON 格式的对象或错误。如果函数退出时没有错误，则响应正文为 `null`。

> **Note**
>
> Lambda 在发送响应之前不会等待外部扩展完成。Lambda 扩展作为独立进程在执行环境中运行，并可以在完成函数调用后继续运行。有关更多信息，请参阅 使用 Lambda 扩展增强 Lambda 函数。

该命令的输出（显示在终端中）包含来自 Lambda 响应中的标头的信息。这包括处理事件的版本（在使用别名时非常有用），以及 Lambda 返回的状态代码。如果 Lambda 能够运行该函数，则状态代码为 200，即使该函数返回错误也是如此。

> **Note**
>
> 对于超时很长的函数，在等待响应的同步调用期间，客户端可能会断开连接。配置您的 HTTP 客户端、软件开发工具包、防火墙、代理或操作系统，以允许针对超时或保持活动设置保持长时间的连接。

如果 Lambda 无法运行该函数，则将在输出中显示错误。

```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload value response.json
```

您应看到以下输出：

```
An error occurred (InvalidRequestContentException) when calling the Invoke operation:
 Could not parse request body into json: Unrecognized token 'value': was expecting
 ('true', 'false' or 'null')
 at [Source: (byte[])"value"; line: 1, column: 11]
```

AWS CLI 是一种开源工具，让您能够在命令行 Shell 中使用命令与 AWS 服务进行交互。要完成本节中的步骤，您必须拥有 AWS CLI 版本 2。

您可以通过 AWS CLI，使用 `--log-type` 命令选项检索调用的日志。响应包含一个 `LogResult` 字段，其中包含多达 4KB 来自调用的 base64 编码日志。

Example 检索日志 ID

以下示例说明如何从 `LogResult` 字段中检索名为 `my-function` 的函数的日志 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解码日志

在同一命令提示符下，使用 base64 实用程序解码日志。以下示例说明如何为 my-function 检索 base64 编码的日志。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

您应看到以下输出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64 实用程序在 Linux、macOS 和 Ubuntu on Windows 上可用。macOS 用户可能需要使用 base64 -D。

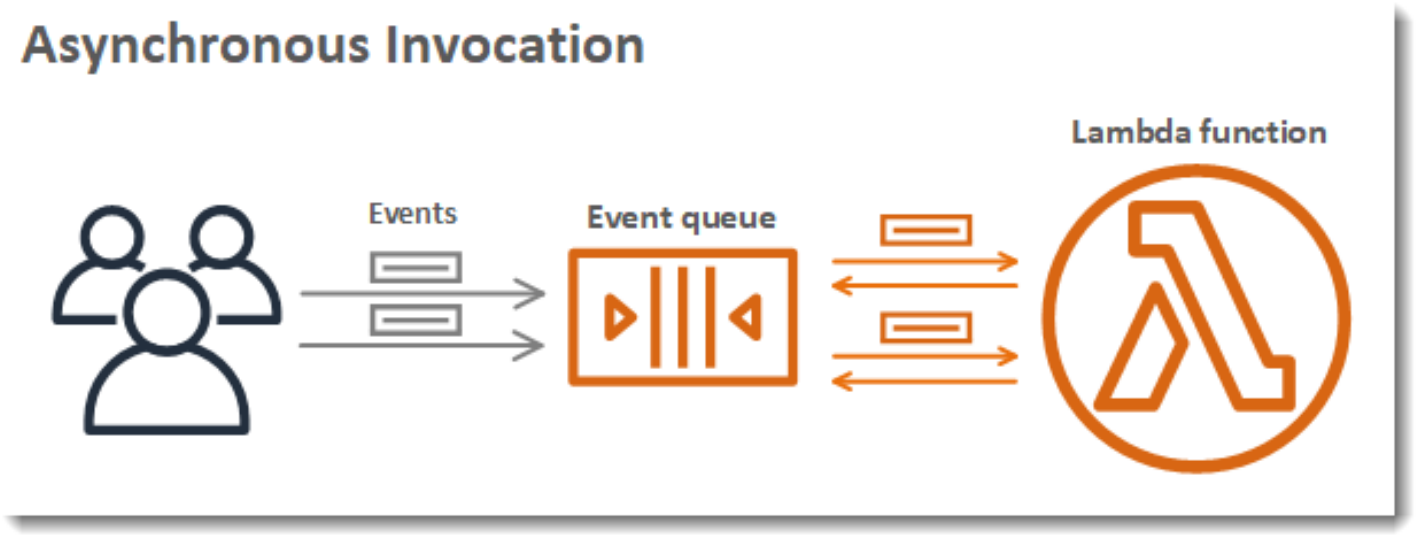
有关 Invoke API 的更多信息（包括参数、标头和错误的完整列表），请参阅调用。

当您直接调用函数时，可以检查错误的响应并重试。在出现客户端超时、节流和服务错误时，AWS CLI 和AWS开发工具包也会自动重试。有关更多信息，请参阅了解 Lambda 中的重试行为。

异步调用 Lambda 函数

多个 AWS 服务（如 Amazon Simple Storage Service（Amazon S3）和 Amazon Simple Notification Service（Amazon SNS）将异步调用函数以处理事件。您也可以使用 AWS Command Line Interface（AWS CLI）或一种 AWS SDK 来异步调用 Lambda 函数。在异步调用函数时，您不会等待函数代码的响应。您将事件交给 Lambda，Lambda 处理其余部分。您可以配置 Lambda 处理错误的方式，并将调用记录发送到下游资源，例如 Amazon Simple Queue Service（Amazon SQS）或 Amazon EventBridge（EventBridge），以将应用程序的组件链接在一起。

下图显示了异步调用 Lambda 函数的客户端。Lambda 在将事件发送到函数之前对事件排队。



对于异步调用，Lambda 将事件置于队列中并返回成功响应，而不返回其他信息。一个单独的进程会从队列中读取事件并将其发送到函数。

要使用 AWS Command Line Interface（AWS CLI）或一种 AWS SDK 来异步调用 Lambda 函数，请将 InvocationType 参数设置为 `Event`。以下示例显示了调用函数的 AWS CLI 命令。

```
aws lambda invoke \
  --function-name my-function  \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{ "key": "value" }' response.json
```

您应看到以下输出：

```
{
```

```
    "StatusCode": 202
}
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

输出文件 (`response.json`) 不包含任何信息，但运行此命令时仍会创建该文件。如果 Lambda 无法将事件添加到队列，则命令输出中将显示错误消息。

Lambda 如何处理异步调用的错误和重试

Lambda 管理函数的异步事件队列以及在出错时重试的尝试次数。如果函数返回错误，Lambda 默认会尝试再运行两次，前两次尝试之间等待一分钟，第二次与第三次尝试之间等待两分钟。函数错误包括函数代码返回的错误，以及函数运行时返回的错误，例如超时。

如果该函数没有足够的并发性可用于处理所有事件，则其他请求将受到限制。对于节流错误（429）和系统错误（500 系列），Lambda 会将事件返回到队列并尝试再次运行该函数，默认最长运行 6 小时。在第一次尝试后，重试间隔从 1 秒以指数级增加到最多 5 分钟。如果队列包含多个条目，Lambda 将增加重试间隔并降低从队列中读取事件的速率。

即使您的函数没有返回错误，它也可能多次从 Lambda 接收相同的事件，因为队列本身具有最终一致性。如果函数无法跟上传入事件，则也可能从队列中删除事件而不将其发送到函数。确保您的函数代码正常处理重复事件，并且您有足够的并发可用于处理所有调用。

当队列很长时，新事件可能会在 Lambda 有机会将它们发送到您的函数之前过期。当事件过期或所有处理尝试都失败时，Lambda 将放弃该事件。您可为函数配置错误处理以减少 Lambda 执行的重试次数，或者更快地放弃未处理的事件。

您还可以配置 Lambda 以将调用记录发送至其他服务。要了解更多信息，请参阅获取 Lambda 异步调用记录。

配置 Lambda 异步调用的错误处理设置

使用以下设置来配置 Lambda 如何处理异步函数调用的错误和重试：

- MaximumEventAgeInSeconds：Lambda 在异步事件队列中保留事件的最长时间（以秒为单位），此后将丢弃事件。
- MaximumRetryAttempts：函数返回错误时 Lambda 重试事件的最大次数。

使用 Lambda 控制台或 AWS CLI 配置有关函数、版本或别名的错误处理设置。

Console

配置错误处理

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration（配置），然后选择 Asynchronous invocation（异步调用）。
4. 在 Asynchronous invocation (异步调用) 下，选择 Edit (编辑)。
5. 配置以下设置。
   - Maximum age of event（事件的最长期限）– Lambda 在异步事件队列中保留事件的最长时间（最长 6 小时）。
   - Retry attempts（重试次数）– 函数返回错误时 Lambda 重试的次数（0 到 2 之间）。
6. 选择保存。

AWS CLI

要通过 AWS CLI 配置异步调用，请使用 put-function-event-invoke-config 命令。以下示例配置一个最长事件期限为 1 小时且无重试的函数。

```
aws lambda put-function-event-invoke-config \
  --function-name error \
  --maximum-event-age-in-seconds 3600 \
  --maximum-retry-attempts 0
```

`put-function-event-invoke-config` 命令覆盖函数、版本或别名上的任何现有配置。要配置某个选项而不重置其他选项，请使用 update-function-event-invoke-config。以下示例配置 Lambda，以便在无法处理事件时将记录发送到名为 `destination` 的标准 SQS 队列。

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

您应看到以下输出：

```
{
    "LastModified": 1573686021.479,
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {}
    }
}
```

当调用事件超过最长期限或所有重试均失败时，Lambda 会丢弃该事件。要保留已丢弃事件的副本，请配置失败事件目标。

获取 Lambda 异步调用记录

Lambda 可以将异步调用记录发送给以下某个 AWS 服务。

- Amazon SQS – 标准 SQS 队列。
- Amazon SNS – 标准 SNS 主题。
- AWS Lambda – Lambda 函数。
- Amazon EventBridge – EventBridge 事件总线。

调用记录包含有关 JSON 格式的请求和响应的详细信息。您可为成功处理的事件以及处理尝试失败的事件配置单独的目标。或者，您可以将标准 Amazon SQS 队列或标准 Amazon SNS 主题配置为丢弃事件的死信队列。对于死信队列，Lambda 只发送事件的内容，不包含有关响应的详细信息。

如果 Lambda 无法向您配置的目的地发送记录，它会向 Amazon CloudWatch 发送 `DestinationDeliveryFailures` 指标。如果您的配置包含不支持的目标类型，例如 Amazon SQS FIFO 队列或 Amazon SNS FIFO 主题，则可能会发生这种情况。权限误配和大小限制可能会导致发生传输错误。有关 Lambda 调用指标的更多信息，请参阅 调用指标。

> **Note**
>
> 要防止触发函数，可以将函数的预留并发设置为零。当您将异步调用函数的预留并发设置为零时，Lambda 会开始将新事件发送到已配置的死信队列或失败时的事件目标，且不会做出任何

重试。要处理在预留并发设置为零时发送的事件，您必须使用死信队列或失败时事件目标中的事件。

添加目标

要保留异步调用的记录，请向函数添加目标。您可以选择将成功或失败的调用发送到目标。每个函数可以有多个目标，因此您可以为成功和失败的事件配置不同的目标。发送到目标的每条记录都是一个 JSON 文档，其中包含有关调用的详细信息。与错误处理设置一样，您可以在函数版本或别名上配置目标。

> **Note**
>
> 您还可以保留以下事件源映射类型的失败调用记录：Amazon Kinesis、Amazon DynamoDB、自托管式 Apache Kafka 和 Amazon MSK。

下表列出了 Lambda 支持的异步调用记录目标。要让 Lambda 成功将记录发送到您选择的目标，请确保函数的执行角色也包含相关权限。该表还描述了每种目标类型如何接收 JSON 调用记录。

目标类型	所需的权限	特定于目标的 JSON 格式
Amazon SQS 队列	sqs:SendMessage	Lambda 将调用记录作为 `Message` 传递到目标。
Amazon SNS 主题	sns:Publish	Lambda 将调用记录作为 `Message` 传递到目标。
Lambda 函数	InvokeFunction	Lambda 将调用记录作为有效负载传递给函数。
EventBridge	events:PutEvents	• Lambda 将调用记录作为 `detail` 在 PutEvents 调用中传递。 • `source` 事件字段的值为 `lambda`。 • `detail-type` 事件字段的值为“Lambda

目标类型	所需的权限	特定于目标的 JSON 格式
		Function Invocation Result – Success”（Lambda 函数调用结果 – 成功）或“Lambda Function Invocation Result – Failure”（Lambda 函数调用结果 – 失败）。 • `resource` 事件字段包含函数和目标的 Amazon 资源名称（ARN）。 • 对于其他事件字段，请参阅 Amazon EventBridge 事件。

以下步骤介绍如何使用 Lambda 控制台和 AWS CLI 配置函数的目标。

Console

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 在 Function overview (函数概览) 下，选择 Add destination (添加目标)。
4. 对于 Source (源)，选择 Asynchronous invocation (异步调用)。
5. 对于 Condition (条件)，请从以下选项中选择：
   - On failure（失败时）– 当事件的所有处理尝试均失败或超过最长期限时发送记录。
   - On success（成功时）– 函数成功处理异步调用时发送记录。
6. 对于 Destination type (目标类型)，请选择接收调用记录的资源类型。
7. 对于 Destination (目标)，请选择一个资源。
8. 选择保存。

AWS CLI

要使用 AWS CLI 配置目标，请运行 update-function-event-invoke-config 命令。以下示例配置 Lambda，以便在无法处理事件时将记录发送到名为 `destination` 的标准 SQS 队列。

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

当调用与条件匹配时，Lambda 会向目标发送包含调用详细信息的 JSON 文档。以下示例显示了由于函数错误而导致三次处理尝试失败的事件的调用记录。

Example 调用记录

```
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_IDS": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a31ed15"
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18cdf61200e Process exited
 before completing request"
    }
}
```

调用记录包含有关事件、响应和记录发送原因的详细信息。

追踪发往目的地的请求

在每个请求排队、由 Lambda 函数处理并传递到目标服务时，您可以使用 AWS X-Ray 查看每个请求的连接视图。当您为调用函数的函数或服务激活 X-Ray 跟踪时，Lambda 会向请求添加 X-Ray 标头并将标头传递给目标服务。来自上游服务的跟踪会自动链接到下游 Lambda 函数的跟踪，从而创建整个应用程序的端到端视图。有关跟踪的更多信息，请参阅 使用 AWS X-Ray 可视化 Lambda 函数调用。

添加死信队列

作为失败时的目标的替代，您可以使用死信队列配置函数，以保存丢弃的事件供进一步处理。死信队列的作用与失败时的目标相同，在某个事件的所有处理尝试都失败或者已过期而未处理时使用。但是，您只能在函数级别添加或删除死信队列。函数版本使用与未发布的版本（$LATEST）相同的死信队列设置。失败时的目标还支持其他目标，并在调用记录中包含有关函数响应的详细信息。

要重新处理死信队列中的事件，您可以将其设置为 Lambda 函数的事件源。或者，您也可以手动检索事件。

您可以为死信队列选择 Amazon SQS 标准队列或 Amazon SNS 标准主题。不支持 FIFO 队列和 Amazon SNS FIFO 主题。

- Amazon SQS 队列 – 队列会保存失败的事件，直到检索这些事件为止。如果您希望单个实体（例如 Lambda 函数或 CloudWatch 告警）来处理失败事件，请选择 Amazon SQS 标准队列。有关更多信息，请参阅 将 Lambda 与 Amazon SQS 结合使用。
- Amazon SNS 主题 – 主题将失败的事件中继到一个或多个目标。如果您希望多个实体对失败事件采取行动，请选择 Amazon SNS 标准主题。例如，您可以配置主题以将事件发送到电子邮件地址、Lambda 函数或 HTTP 端点。有关更多信息，请参阅 使用 Amazon SNS 通知调用 Lambda 函数。

要将事件发送到队列或主题，您的函数需要其他权限。添加具有函数执行角色所需权限的策略。

如果已使用客户管理的密钥加密目标队列或主题，则执行角色也必须是密钥的基于资源的策略中的用户。

创建目标并更新函数的执行角色后，将死信队列添加到函数中。您可以配置多个函数，以便将事件发送到同一目标。

Console

1. 打开 Lamba 控制台的 Functions（函数）页面。

2. 选择函数。
3. 选择 Configuration（配置），然后选择 Asynchronous invocation（异步调用）。
4. 在 Asynchronous invocation (异步调用) 下，选择 Edit (编辑)。
5. 将死信队列服务设置为 Amazon SQS 或 Amazon SNS。
6. 选择目标队列或主题。
7. 选择保存。

AWS CLI

要通过 AWS CLI 配置死信队列，请使用 update-function-configuration 命令。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --dead-letter-config TargetArn=arn:aws:sns:us-east-1:123456789012:my-topic
```

Lambda 按原样将事件发送到死信队列，并在属性中包含其他信息。您可以使用此信息来标识函数返回的错误，或者将事件与日志或 AWS X-Ray 跟踪相关联。

死信队列消息属性

- RequestID（字符串）– 调用请求的 ID。请求 ID 显示在函数日志中。您还可以使用 X-Ray 开发工具包，在跟踪中的属性上记录请求 ID。然后，可以在 X-Ray 控制台中按请求 ID 搜索跟踪。
- ErrorCode（数字）– HTTP 状态代码。
- ErrorMessage（字符串）– 错误消息的第一个 1 KB 文本块。

如果 Lambda 无法向死信队列发送消息，则会删除该事件并发出 DeadLetterErrors 指标。之所以发生这种情况，可能是由于缺少权限，或者消息的总大小超过目标队列或主题的限制。例如，假设正文大小接近 256 KB 的 Amazon SNS 通知触发了一个导致错误的函数。在这种情况下，Amazon SNS 添加的事件数据加上 Lambda 添加的属性，可能会导致消息超过死信队列中允许的最大大小。

如果您正在使用 Amazon SQS 作为事件源，请在 Amazon SQS 队列本身而不是 Lambda 函数上配置死信队列。有关更多信息，请参阅 将 Lambda 与 Amazon SQS 结合使用。

Lambda 如何处理来自基于流和队列的事件源的记录

事件源映射是一种 Lambda 资源，它从流或基于队列的服务中读取项目并调用包含批次记录的函数。以下服务使用事件源映射调用 Lambda 函数：

- Amazon DocumentDB（与 MongoDB 兼容）（Amazon DocumentDB）
- Amazon DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- 自行管理的 Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)

> **Warning**
>
> Lambda 事件源映射至少处理每个事件一次，有可能出现重复处理记录的情况。为避免与重复事件相关的潜在问题，我们强烈建议您将函数代码设为幂等性。要了解更多信息，请参阅 AWS 知识中心的如何使我的 Lambda 函数具有幂等性。

事件源映射与直接触发器的区别

某些 AWS 服务可以使用触发器直接调用 Lambda 函数。这些服务将事件推送到 Lambda，并在指定事件发生时立即调用该函数。触发器适用于离散事件和实时处理。当您使用 Lambda 控制台创建触发器时，控制台会与相应的 AWS 服务交互以配置该服务的事件通知。触发器实际上由生成事件的服务而不是 Lambda 存储和管理。以下是一些使用触发器调用 Lambda 函数的服务示例：

- Amazon Simple Storage Service（Amazon S3）：当在存储桶中创建、删除或修改对象时调用函数。有关更多信息，请参阅 教程：使用 Amazon S3 触发器调用 Lambda 函数。
- Amazon Simple Notification Service（Amazon SNS）：将消息发布到 SNS 主题时调用函数。有关更多信息，请参阅 教程：将 AWS Lambda 与 Amazon Simple Notification Service 结合使用。
- Amazon API Gateway：在向特定端点发出 API 请求时调用函数。有关更多信息，请参阅 使用 Amazon API Gateway 端点调用 Lambda 函数。

事件源映射是在 Lambda 服务中创建和管理的 Lambda 资源。事件源映射旨在处理来自队列的大量流数据或消息。分批处理来自流或队列的记录比单独处理记录更高效。

批处理行为

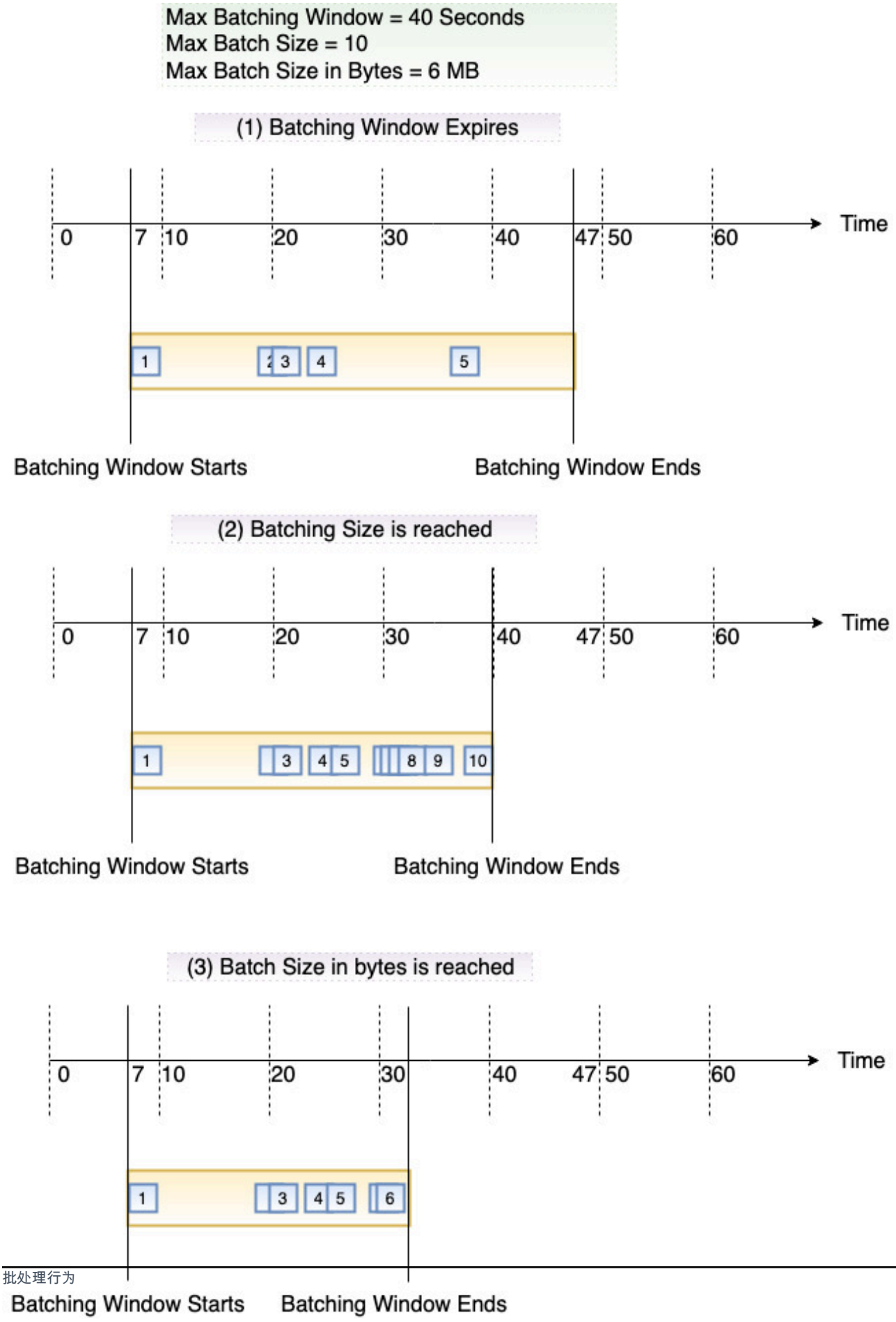
预设情况下，事件源映射会将记录合并为单个有效负载进行批处理，并由 Lambda 将其发送到您的函数。要微调批处理行为，您可以配置批处理时段（MaximumBatchingWindowInSeconds）和批处理大小（BatchSize）。批处理时段是将记录收集到单个有效负载中的最长时间。批处理大小是单个批处理中的最大记录数。满足以下三个条件中的任意一个时，Lambda 会调用您的函数：

- 批处理时段达到其最大值。默认的批处理时段行为因特定的事件源而异。
  - 对于 Kinesis、DynamoDB 和 Amazon SQS 事件源：原定设置的批处理时段是 0 秒。这意味着，一旦记录可用，Lambda 就会调用您的函数。要设置批处理时段，请配置 `MaximumBatchingWindowInSeconds`。您可以将此参数设置为介于 0 秒到 300 秒之间的任意值，以 1 秒为增量。如果您配置了批处理时段，则下一个时段将在上一个函数调用完成后立即开始计算。
  - 对于 Amazon MSK、自托管式 Apache Kafka、Amazon MQ 和 Amazon DocumentDB 事件源：默认批处理时段为 500 毫秒。您可以将 `MaximumBatchingWindowInSeconds` 配置为介于 0 秒到 300 秒之间的任意值，以秒的整数倍调整。第一条记录到达后，批处理时段将立即开始计算。

    > **Note**
    >
    > 由于您只能以秒的整数倍调整 `MaximumBatchingWindowInSeconds`，因此无法在更改该值后恢复到 500 毫秒的默认批处理时段。要恢复原定设置的批处理时段，必须创建新的事件源映射。

- 达到批处理大小。最小批处理大小为 1。原定设置和最大批处理大小取决于事件源。有关这些值的详细信息，请参阅 `CreateEventSourceMapping` API 操作的 BatchSize 规范。
- 有效负载大小达到 6MB。您不能修改此限制。

下图演示了这三个条件。假设批处理时段从 `t = 7` 秒开始。在第一种场景中，批处理时段累积 5 条记录后在 `t = 47` 秒达到 40 秒的最大值。在第二种场景中，批处理大小在批处理时段到期之前达到 10，因此批处理时段会提前结束。在第三种场景中，在批处理时段到期之前达到最大有效负载大小，因此批处理时段会提前结束。


Max Batching Window = 40 Seconds
Max Batch Size = 10
Max Batch Size in Bytes = 6 MB
(1) Batching Window Expires
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
Batching Window Starts
Batching Window Ends
(2) Batching Size is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
8
9
10
Batching Window Starts
Batching Window Ends
(3) Batch Size in bytes is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
6
Batching Window Starts
Batching Window Ends

建议您测试不同批处理和记录的大小，这样每个事件源的轮询频率都会根据函数完成任务的速度进行调整。CreateEventSourceMapping BatchSize 参数控制每次调用可向您的函数发送记录的最大数量。批处理大小如果较大，通常可以更有效地吸收大量记录的调用开销，从而增加吞吐量。

Lambda 在发送下次批处理之前不会等待任何配置的扩展完成。换句话说，扩展可能会在 Lambda 处理下一批记录时继续运行。如果您违反了账户的任何并发设置或限制，可能会导致节流问题。要检测这是否是潜在问题，请监控函数并检查所显示的并发指标是否高于事件源映射的预期。由于调用间隔时间较短，Lambda 可能会短暂报告高于分片数量的并发使用量。即使对于没有扩展名的 Lambda 函数也是如此。

预设情况下，如果函数返回错误，事件源映射会重新处理整个批处理，直到函数成功，或直到批处理中的项目到期。为确保按顺序处理，在错误得到解决之前，事件源映射会暂停处理受影响的分片。对于流源（DynamoDB 和 Kinesis），可以配置 Lambda 在函数返回错误时重试的最大次数。批次未到达您的函数时出现的服务错误或节流，不计入重试次数。您还可以配置事件源映射，以便在丢弃事件批处理时将调用记录发送到目标。

事件源映射 API

要使用 AWS Command Line Interface（AWS CLI） 或 AWS SDK 来管理事件源，可以使用以下 API 操作：

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

在事件源映射上使用标签

对事件源映射进行标记可以组织和管理资源。标签是与资源关联的自由格式键值对，在 AWS 服务中受支持。有关标签用例的更多信息，请参阅《Tagging AWS Resources and Tag Editor Guide》中的 Common tagging strategies。

事件源映射与函数相关联，而函数也可以有自己的标签。事件源映射并不会自动继承函数的标签。可以使用 AWS Lambda API 来查看和更新标签。在 Lambda 控制台中管理特定事件源映射时，还可以查看和更新标签。

使用标签所需的权限

要允许 AWS Identity and Access Management（IAM）身份（用户、组或角色）读取资源或为其设置标签，请授予该身份相应的权限：

- lambda:ListTags – 当资源有标签时，将此权限授予需要在其上调用 `ListTags` 的任何人。对于带标签的函数，`GetFunction` 也需要此权限。
- lambda:TagResource – 将此权限授予需要调用 `TagResource` 或执行在创建时授予标记的操作的任何人。

有关更多信息，请参阅 Lambda 的基于身份的 IAM policy。

通过 Lambda 控制台使用标签

可以使用 Lambda 控制台创建带标签的事件源映射、向现有事件源映射添加标签以及按标签筛选事件源映射。

使用 Lambda 控制台为支持的流服务和基于队列的服务添加触发器时，Lambda 会自动创建事件源映射。有关这些事件源的更多信息，请参阅 the section called “事件源映射”。若要在控制台中创建事件源映射，必须满足以下先决条件：

- 函数
- 来自受影响服务的事件源

可在用于创建或更新触发器的同一用户界面中添加标签。

在创建事件源映射时添加标签

1. 打开 Lamba 控制台的函数页面。
2. 选择函数的名称。
3. 在 Function overview（函数概览）下，选择 Add trigger（添加触发器）。
4. 在触发器配置下的下拉列表中，选择事件源所属服务的名称。
5. 为事件源提供核心配置。有关配置事件源的更多信息，请参阅 与其他服务集成中相关服务的部分。
6. 在事件源映射配置下选择其他设置。
7. 在标签下选择添加新标签。

8. 在键字段中输入标签键。有关标记限制的信息，请参阅《Tagging AWS Resources and Tag Editor Guide》中的 Tag naming limits and requirements。
9. 选择添加。

向现有事件源映射添加标签

1. 在 Lambda 控制台中打开事件源映射。
2. 在资源列表中，选择与函数和事件源 ARN 对应的事件源映射的 UUID。
3. 从常规配置窗格下方的选项卡列表中选择标签。
4. 选择管理标签。
5. 选择 Add new tag（添加新标签）。
6. 在键字段中输入标签键。有关标记限制的信息，请参阅《Tagging AWS Resources and Tag Editor Guide》中的 Tag naming limits and requirements。
7. 选择保存。

按标签筛选事件源映射

1. 在 Lambda 控制台中打开事件源映射。
2. 选择搜索框。
3. 在下拉列表中，从标签副标题下方选择标签键。
4. 选择使用：“tag-name”查看所有使用此键标记的事件源映射，或者选择一个运算符进一步按值筛选。
5. 选择标签值以按标签键和值的组合进行筛选。

搜索框还支持搜索标签键。输入键名称，即可在列表中查找该键。

通过 AWS CLI 使用标签

可以使用 Lambda API 在现有 Lambda 资源（包括事件源映射）上添加和删除标签。还可以在创建事件源映射时添加标签，这样就可以在资源的整个生命周期中对其进行标记。

使用 Lambda 标签 API 更新标签

可以通过 TagResource 和 UntagResource API 操作，添加和删除受支持 Lambda 资源的标签。

可以使用 AWS CLI 调用这些操作。要向现有资源添加标签，请使用 `tag-resource` 命令。此示例添加了两个标签，一个带有键 *Department*，另一个带有键 *CostCenter*。

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

要删除标签，请使用 `untag-resource` 命令。此示例删除了键为 *Department* 的标签。

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier \
--tag-keys Department
```

在创建事件源映射时添加标签

若要使用标签创建 Lambda 事件源映射，请使用 CreateEventSourceMapping API 操作。指定 `Tags` 参数。可以使用 `create-event-source-mapping` AWS CLI 命令和 `--tags` 选项调用此操作。有关使用此 CLI 命令的更多信息，请参阅《AWS CLI Command Reference》中的 create-event-source-mapping。

在将 `Tags` 参数与 `CreateEventSourceMapping` 一起使用之前，请确保角色拥有标记资源的权限以及此操作所需的常规权限。有关标记权限的更多信息，请参阅 the section called “使用标签所需的权限”。

使用 Lambda 标签 API 查看标签

要查看应用于特定 Lambda 资源的标签，请使用 `ListTags` API 操作。有关更多信息，请参阅 ListTags。

可以提供 ARN（Amazon 资源名称），以使用 `list-tags` AWS CLI 命令调用此操作。

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

按标签筛选资源

您可以使用 AWS Resource Groups Tagging API GetResources API 操作按标签筛选资源。`GetResources` 操作最多可接收 10 个筛选条件，每个筛选条件包含一个标签键和最多 10 个标签值。提供具有 `ResourceType` 的 `GetResources`，可按特定资源类型进行筛选。

可以使用 `get-resources` AWS CLI 命令调用此操作。有关使用 `get-resources` 的示例，请参阅《AWS CLI Command Reference》中的 get-resources。

控制 Lambda 向您的函数发送的事件

您可以使用事件筛选，控制 Lambda 将流或队列中的哪些记录发送给函数。例如，您可以添加筛选条件，以便函数仅处理包含特定数据参数的 Amazon SQS 消息。事件筛选仅适用于某些事件源映射。您可以将筛选条件添加到以下 AWS 服务的事件源映射中：

- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- 自行管理的 Apache Kafka
- Amazon Simple Queue Service(Amazon SQS)

有关在特定事件源中筛选的具体信息，请参阅 the section called “使用具有不同 AWS 服务 的筛选条件”。Lambda 不支持 Amazon DocumentDB 的事件筛选。

默认情况下，您可以为单个事件源映射定义最多五个不同的筛选条件。筛选条件是使用逻辑运算符 OR 组合起来的。如果来自事件源的记录符合一个或多个筛选条件，则 Lambda 会将该记录包含在发送到函数的下一个事件中。如果不符合任何筛选条件，Lambda 会丢弃该记录。

> **Note**
>
> 如果您需要为一个事件源定义 5 个以上的筛选条件，可请求将每个事件源的配额提高到 10 个筛选条件。如果您尝试添加的筛选条件数量超过当前配额允许的数量，则在创建事件源时，Lambda 将会返回错误。

主题

- 使用具有不同 AWS 服务 的筛选条件

了解事件筛选基础知识

筛选条件 (FilterCriteria) 对象是一个由筛选条件 (Filters) 列表组成的结构。每个筛选条件都是一个用于定义事件筛选模式 (Pattern) 的结构。模式是 JSON 筛选条件规则的字符串表示。FilterCriteria 对象的结构如下所示。

```
{
   "Filters": [
        {
            "Pattern": "{ \"Metadata1\": [ rule1 ], \"data\": { \"Data1\":
 [ rule2 ] }}"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 Pattern 的值。

```
{
    "Metadata1": [ rule1 ],
    "data": {
        "Data1": [ rule2 ]
    }
}
```

筛选条件模式可以包括元数据属性和/或数据属性。可用元数据参数和数据参数的格式根据充当事件源的 AWS 服务 而有所不同。例如，假设事件源映射从 Amazon SQS 队列接收到以下记录：

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\n "City": "Seattle",\n "State": "WA",\n "Temperature": "46"\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
```

```
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
    "awsRegion": "us-east-2"
}
```

- 元数据属性是包含有关创建记录的事件的信息的字段。在示例 Amazon SQS 记录中，元数据属性包括 `messageID`、`eventSourceArn` 和 `awsRegion` 等字段。
- 数据属性是包含流或队列中数据的记录的字段。在 Amazon SQS 事件示例中，数据字段的键是 `body`，数据属性是字段 `City`、`State` 和 `Temperature`。

不同类型的事件源为其数据字段使用不同的键值。要对数据属性进行筛选，请确保在筛选条件的模式中使用正确的键。如需数据筛选键列表，并查看每个支持的 AWS 服务 的筛选模式示例，请参阅 使用具有不同 AWS 服务 的筛选条件。

事件筛选可处理多层 JSON 筛选。例如，考虑以下来自 DynamoDB 流的记录片段：

```
"dynamodb": {
    "Keys": {
        "ID": {
            "S": "ABCD"
        }
        "Number": {
            "N": "1234"
    },
    ...
}
```

假设您只想处理排序键值 `Number` 为 4567 的记录。在这种情况下，您的 `FilterCriteria` 对象与以下类似：

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Number\": { \"N\":
 [ "4567" ] } } } }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 `Pattern` 的值。

```
{
    "dynamodb": {
        "Keys": {
            "Number": {
                "N": [ "4567" ]
                }
            }
        }
}
```

处理不符合筛选条件的记录

Lambda 如何处理不符合筛选条件的记录取决于事件源。

- 对于 Amazon SQS，如果消息不符合筛选条件，Lambda 会自动从队列中删除该消息。您无需在 Amazon SQS 中手动删除这些消息。
- 对于 Kinesis 和 DynamoDB，在筛选条件评估记录后，流迭代器会跳过此记录。如果记录不符合筛选条件，您无需从事件源手动删除记录。保留期结束后，Kinesis 和 DynamoDB 会自动删除这些旧记录。如果您希望提前删除记录，请参阅更改数据保留期。
- 对于 Amazon MSK、自行管理的 Apache Kafka 和 Amazon MQ 消息，Lambda 会丢弃与筛选条件中包含的所有字段不匹配的消息。对于 Amazon MSK 和自行管理的 Apache Kafka，Lambda 在成功调用函数后，会为匹配和不匹配的消息提交偏移。对于 Amazon MQ，Lambda 在成功调用函数后确认匹配的消息，并在筛选不匹配的消息时确认此类消息。

筛选条件规则语法

对于筛选条件规则，Lambda 支持 Amazon EventBridge 规则，并使用与 EventBridge 相同的语法。有关更多信息，请参阅 Amazon EventBridge 用户指南中的 Amazon EventBridge 事件模式。

以下是可用于 Lambda 事件筛选的所有比较运算符的汇总。

比较运算符	示例	Rule syntax（规则语法）
Null	用户 ID 为空	"UserID": [null]
空	姓氏为空	"LastName": [""]
等于	名字为“Alice”	"Name": ["Alice"]

比较运算符	示例	Rule syntax（规则语法）
等于（忽略大小写）	名字为“Alice”	“Name”：[{ "equals-ignore-case": "alice" }]
并且	位置为“纽约”，日期为“星期一”	"Location": ["New York"], "Day": ["Monday"]
Or	付款类型为“信用卡”或“借记卡”	"PaymentType": ["Credit", "Debit"]
或（多个字段）	位置为“纽约”，或日期为“星期一”。	“$or”：[{ "Location": ["New York"] }, { "Day": ["Monday"] }]
非	天气是除“下雨”以外的任何天气	"Weather": [{ "anything-but": ["Raining"] }]
数值（等于）	价格为 100	"Price": [{ "numeric": ["=", 100] }]
数值（范围）	价格大于 10，且小于等于 20	"Price": [{ "numeric": [">", 10, "<=", 20] }]
存在	产品名存在	"ProductName": [{ "exists": true }]
不存在	产品名不存在	"ProductName": [{ "exists": false }]
始于	地区位于美国	"Region": [{"prefix": "us-" }]
结束于	文件名以 .png 扩展名结尾。	“FileName”：[{ "suffix": ".png" }]

> **Note**
>
> 与 EventBridge 一样，对于字符串，Lambda 使用精确的逐个字符匹配，而不进行小写化或任何其他字符串标准化。此外，对于数值，Lambda 使用字符串表示。例如，300、300.0 和 3.0e2 不相等。

请注意，Exists 运算符仅适用于事件源 JSON 中的叶节点，与中间节点不匹配。例如，使用以下 JSON，筛选条件模式 `{ "person": { "address": [ { "exists": true } ] } }` 找不到匹配项，因为 `"address"` 是中间节点。

```
{
  "person": {
    "name": "John Doe",
    "age": 30,
    "address": {
      "street": "123 Main St",
      "city": "Anytown",
      "country": "USA"
    }
  }
}
```

将筛选条件附加到事件源映射（控制台）

按照以下步骤使用 Lambda 控制台创建包含筛选条件的新事件源映射。

使用筛选条件创建新事件源映射（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择要为其创建事件源映射的函数名称。
3. 在 Function overview（函数概览）下，选择 Add trigger（添加触发器）。
4. 对于 Trigger configuration（触发器配置），请选择支持事件筛选的触发器类型。有关受支持的服务列表，请参阅本页开头的列表。
5. 展开其他设置。
6. 在 Filter criteria（筛选条件）下，选择 Add（添加），然后定义并输入筛选条件。例如，您可以输入以下筛选条件。

```
{ "Metadata" : [ 1, 2 ] }
```

这指示 Lambda 只处理字段 Metadata 等于 1 或 2 的记录。您可以继续选择添加，以添加更多筛选条件，但不得超过最大允许数量。

7. 添加完筛选条件后，选择保存。

使用控制台输入筛选条件时，只需输入筛选条件模式，无需提供 Pattern 键或转义引号。在上述说明的第 6 步中，{ "Metadata" : [1, 2] } 对应于下面的 FilterCriteria。

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

在控制台中创建事件源映射后，您可以在触发器详细信息中看到格式化后的 FilterCriteria。有关使用控制台创建事件筛选条件的更多示例，请参阅 使用具有不同 AWS 服务 的筛选条件。

将筛选条件附加到事件源映射（AWS CLI）

假设您希望事件源映射具有以下 FilterCriteria：

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

此 create-event-source-mapping 命令将为包含指定 FilterCriteria 的函数 my-function 创建新的 Amazon SQS 事件源映射。

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

请注意，要更新事件源映射，您需要其 UUID。您可以通过 list-event-source-mappings 调用获取 UUID。Lambda 还会在 create-event-source-mapping CLI 响应中返回 UUID。

要从事件源中删除筛选条件，您可以运行以下包含空 FilterCriteria 对象的 update-event-source-mapping 命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria "{}"
```

有关使用 AWS CLI 创建事件筛选条件的更多示例，请参阅 使用具有不同 AWS 服务 的筛选条件。

将筛选条件附加到事件源映射（AWS SAM）

假设您要在 AWS SAM 中配置事件源以使用以下筛选条件：

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

要将这些筛选条件添加到事件源映射中，请将以下代码段插入事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{"Metadata": [1, 2]}'
```

有关为事件源映射创建和配置 AWS SAM 模板的更多信息，请参阅《AWS SAM 开发人员指南》中的 EventSource 部分。有关使用 AWS SAM 模板创建事件筛选条件的更多示例，请参阅 使用具有不同 AWS 服务 的筛选条件。

筛选条件的加密

默认情况下，Lambda 不会加密您的筛选条件对象。对于可能在筛选条件对象中包含敏感信息的应用场景，您可以使用自己的 KMS 密钥对其进行加密。

加密筛选条件对象后，您可以使用 GetEventSourceMapping API 调用查看其纯文本版本。您必须拥有 `kms:Decrypt` 权限才能成功查看纯文本筛选条件。

> **Note**
>
> 如果您的筛选条件对象已加密，Lambda 会在 ListEventSourceMappings 调用的响应中编辑 `FilterCriteria` 字段的值。相反，此字段显示为 `null`。要查看 `FilterCriteria` 的真实值，请使用 GetEventSourceMapping API。
> 要在控制台中查看 `FilterCriteria` 解密后的值，请确保您的 IAM 角色包含使用 GetEventSourceMapping 的权限。

您可以通过控制台、API/CLI 或 AWS CloudFormation 指定自己的 KMS 密钥。

使用客户自有 KMS 密钥加密筛选条件（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择添加触发器。如果您已有触发器，请选择配置选项卡，然后选择触发器。选择现有触发器，然后选择编辑。
3. 选中使用客户自主管理型 KMS 密钥加密旁边的复选框。
4. 在选择客户自主管理型 KMS 加密密钥中，选择已启用的现有密钥或创建新密钥。根据操作的不同，您需要以下部分或全部权限：`kms:DescribeKey`、`kms:GenerateDataKey` 和 `kms:Decrypt`。使用 KMS 密钥政策授予这些权限。

如果您使用自己的 KMS 密钥，密钥政策中必须允许以下 API 操作：

- `kms:Decrypt` – 必须授予区域 Lambda 服务主体 (`lambda.`*`AWS_region`*`.amazonaws.com`)。允许 Lambda 使用此 KMS 密钥解密数据。

- 为防止跨服务混淆代理问题，密钥政策使用 aws:SourceArn 全局条件密钥。aws:SourceArn 密钥的正确值是您的事件源映射资源的 ARN，因此只有在知道其 ARN 之后，才能将其添加到政策中。在向 KMS 发出解密请求时，Lambda 还会在加密上下文中转发 aws:lambda:FunctionArn 和 aws:lambda:EventSourceArn 密钥及其各自的值。这些值必须与密钥政策中的指定条件相匹配，解密请求才能成功。您无需为自行管理的 Kafka 事件源添加 EventSourceArn，因为它们没有 EventSourceArn。

- kms:Decrypt – 还必须授予打算使用密钥在 GetEventSourceMapping 或 DeleteEventSourceMapping API 调用中查看纯文本筛选条件的主体。
- kms:DescribeKey – 提供客户自主管理型密钥详细信息以允许主体使用密钥。
- kms:GenerateDataKey – 为 Lambda 提供代表指定主体生成数据密钥以加密筛选条件的权限（信封加密）。

您可以使用 AWS CloudTrail 来跟踪 Lambda 代表您发出的 AWS KMS 请求。有关 CloudTrail 事件示例，请参阅???。

我们还建议使用 kms:ViaService 条件密钥将 KMS 密钥的使用限制为仅限来自 Lambda 的请求。此密钥的值是区域 Lambda 服务主体 (lambda.*AWS_region*.amazonaws.com)。以下是授予所有相关权限的密钥政策示例：

Example AWS KMS 密钥政策

```
{
    "Version": "2012-10-17",
    "Id": "example-key-policy-1",
    "Statement": [
        {
            "Sid": "Allow Lambda to decrypt using the key",
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.us-east-1.amazonaws.com"
            },
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "*",
            "Condition": {
                "ArnEquals" : {
                    "aws:SourceArn": [
```

```
                        "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:<esm_uuid>"
                    ]
                },
                "StringEquals": {
                    "kms:EncryptionContext:aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:test-function",
                    "kms:EncryptionContext:aws:lambda:EventSourceArn": "arn:aws:sqs:us-
east-1:123456789012:test-queue"
                }
            }
        },
        {
            "Sid": "Allow actions by an AWS account on the key",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:root"
            },
            "Action": "kms:*",
            "Resource": "*"
        },
        {
            "Sid": "Allow use of the key to specific roles",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:role/ExampleRole"
            },
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals" : {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com"
                }
            }
        }
    ]
}
```

要使用自己的 KMS 密钥对筛选条件进行加密，也可以使用以下 CreateEventSourceMapping AWS CLI 命令。使用 `--kms-key-arn` 标志指定 KMS 密钥 ARN。

```
aws lambda create-event-source-mapping --function-name my-function \
    --maximum-batching-window-in-seconds 60 \
    --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
    --filter-criteria "{\"filters\": [{\"pattern\": \"{\"a\": [\"1\", \"2\"]}\" }]}" \
    --kms-key-arn arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

如果您已有事件源映射，请改用 UpdateEventSourceMapping AWS CLI 命令。使用 `--kms-key-arn` 标志指定 KMS 密钥 ARN。

```
aws lambda update-event-source-mapping --function-name my-function \
    --maximum-batching-window-in-seconds 60 \
    --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
    --filter-criteria "{\"filters\": [{\"pattern\": \"{\"a\": [\"1\", \"2\"]}\" }]}" \
    --kms-key-arn arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

此操作将覆盖先前指定的任何 KMS 密钥。如果您指定 `--kms-key-arn` 标志和空实参，Lambda 将停止使用您的 KMS 密钥来加密筛选条件。相反，Lambda 默认使用 Amazon 拥有的密钥。

要在 AWS CloudFormation 模板中指定您自己的 KMS 密钥，请使用 `AWS::Lambda::EventSourceMapping` 资源类型的 `KMSKeyArn` 属性。例如，您可以将以下代码段插入事件源的 YAML 模板中。

```
MyEventSourceMapping:
  Type: AWS::Lambda::EventSourceMapping
  Properties:
    ...
    FilterCriteria:
      Filters:
        - Pattern: '{"a": [1, 2]}'
    KMSKeyArn: "arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599"
    ...
```

要能够在 GetEventSourceMapping 或 DeleteEventSourceMapping API 调用中以纯文本形式查看加密的筛选条件，您必须拥有 `kms:Decrypt` 权限。

从 2024 年 8 月 6 日起，如果您的函数不使用事件筛选，FilterCriteria 字段将不再显示在 CreateEventSourceMapping、UpdateEventSourceMapping 和 DeleteEventSourceMapping API 调用的AWS CloudTrail 日志中。如果您的函数确实使用了事件筛选，则 FilterCriteria 字段将显示为空 ({})。如果拥有正确 KMS 密钥的 kms:Decrypt 权限，则仍然可以在响应 GetEventSourceMapping API 调用时以纯文本形式查看筛选条件。

Create/Update/DeleteEventSourceMapping 调用的 CloudTrail 日志条目示例

在以下 CreateEventSourceMapping 调用的 AWS CloudTrail 示例日志条目中，由于该函数使用事件筛选，FilterCriteria 显示为空 ({})。即使 FilterCriteria 对象包含函数正在使用的有效筛选条件，也是如此。如果该函数不使用事件筛选，则 CloudTrail 根本不会在日志条目中显示 FilterCriteria 字段。


```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:userid1",
        "arn": "arn:aws:sts::123456789012:assumed-role/Example/example-role",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA987654321EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/User1",
                "accountId": "123456789012",
                "userName": "User1"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2024-05-09T20:35:01Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "AWS Internal"
    },
    "eventTime": "2024-05-09T21:05:41Z",
    "eventSource": "lambda.amazonaws.com",
    "eventName": "CreateEventSourceMapping20150331",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "AWS Internal",
```

```
    "userAgent": "AWS Internal",
    "requestParameters": {
        "eventSourceArn": "arn:aws:sqs:us-east-2:123456789012:example-queue",
        "functionName": "example-function",
        "enabled": true,
        "batchSize": 10,
        "filterCriteria": {},
        "kMSKeyArn": "arn:aws:kms:us-east-2:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "scalingConfig": {},
        "maximumBatchingWindowInSeconds": 0,
        "sourceAccessConfigurations": []
    },
    "responseElements": {
        "uUID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
        "batchSize": 10,
        "maximumBatchingWindowInSeconds": 0,
        "eventSourceArn": "arn:aws:sqs:us-east-2:123456789012:example-queue",
        "filterCriteria": {},
        "kMSKeyArn": "arn:aws:kms:us-east-2:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:example-
function",
        "lastModified": "May 9, 2024, 9:05:41 PM",
        "state": "Creating",
        "stateTransitionReason": "USER_INITIATED",
        "functionResponseTypes": [],
        "eventSourceMappingArn": "arn:aws:lambda:us-east-2:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb"
    },
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE33333",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "sessionCredentialFromConsole": "true"
}
```

使用具有不同 AWS 服务 的筛选条件

不同类型的事件源为其数据字段使用不同的键值。要对数据属性进行筛选，请确保在筛选条件的模式中使用正确的键。下表给出了每个受支持 AWS 服务 的筛选键。

AWS 服务	筛选键
DynamoDB	`dynamodb`
Kinesis	`data`
Amazon MQ	`data`
Amazon MSK	`value`
自行管理的 Apache Kafka	`value`
Amazon SQS	`body`

以下各节介绍了不同类型事件源的筛选条件模式示例。还为每个受支持的服务提供支持的传入数据格式和筛选条件模式正文格式的定义。

- the section called “事件筛选”
- the section called “事件筛选”
- the section called “事件筛选”
- the section called “事件筛选”
- the section called “事件筛选”
- the section called “事件筛选”

在控制台中测试 Lambda 函数

您可以在控制台中使用测试事件调用函数，从而测试 Lambda 函数。测试事件是函数的一个 JSON 输入。如果函数不需要输入，则事件可以是空文档（{}）。

在控制台中运行测试时，Lambda 会使用测试事件同步调用您的函数。函数运行时系统将事件 JSON 转换为一个对象，并将该对象传递给代码的处理程序方法以进行处理。

> **创建测试事件**
>
> 您需要先创建一个私有或可共享的测试事件，然后才能在控制台中进行测试。

使用测试事件调用函数

测试函数

1. 打开 Lamba 控制台的函数页面。
2. 选择要测试的函数的名称。
3. 选择测试选项卡。
4. 在测试事件下，选择创建新事件或编辑已保存的事件，然后选择要使用的已保存事件。
5. 可选：为事件 JSON 选择一个模板。
6. 选择测试。
7. 在 Execution result（执行结果）下，展开 Details（详细信息）以查看测试结果。

要在不保存测试事件的情况下调用函数，请在保存之前选择 Test（测试）。这将创建一个未保存的测试事件，Lambda 仅会在会话期间内保存该事件。

对于 Node.js、Python 和 Ruby 运行时，您还可以在代码选项卡上访问现有已保存和未保存的测试事件。在主侧栏中，展开测试事件部分以创建、编辑和运行测试。

创建私有测试事件

私人测试事件仅供事件创建者使用，并且不需要额外的权限即可使用。每个函数最多可以创建和保存 10 个测试事件。

创建私有测试事件

1. 打开 Lamba 控制台的函数页面。
2. 选择要测试的函数的名称。
3. 选择测试选项卡。
4. 在 Test event（测试事件）下执行以下操作：
   a. 选择一个 Template（模板）。
   b. 输入测试的 Name（名称）。
   c. 在文本输入框中，输入 JSON 测试事件。
   d. 在 Event sharing settings（事件共享设置）下，选择 Private（私有）。
5. 选择 Save changes（保存更改）。

对于 Node.js、Python 和 Ruby 运行时，您还可以在代码选项卡上创建测试事件。在主侧栏中，展开测试事件部分以创建、编辑和运行测试。

创建可共享测试事件

可共享测试事件是您可与同一 AWS 账户中的其他用户共享的测试事件。您可以编辑其他用户的可共享测试事件并使用这些测试事件调用您的函数。

Lambda 将可共享测试事件作为 Schema 保存在一个名为 `lambda-testevent-schemas` 的 Amazon EventBridge（CloudWatch Events）Schema 注册表中。由于 Lambda 利用此注册表来存储和调用您创建的可共享测试事件，因此我们建议您不要编辑此注册表或使用 `lambda-testevent-schemas` 名称创建注册表。

要查看、共享和编辑可共享测试事件，您必须拥有以下所有 EventBridge（CloudWatch Events）Schema 注册表 API 操作的权限：

- `schemas.CreateRegistry`
- `schemas.CreateSchema`
- `schemas.DeleteSchema`
- `schemas.DeleteSchemaVersion`
- `schemas.DescribeRegistry`
- `schemas.DescribeSchema`
- `schemas.GetDiscoveredSchema`

- schemas.ListSchemaVersions
- schemas.UpdateSchema

请注意，保存对可共享测试事件所做的编辑将覆盖该事件。

如果您无法创建、编辑或查看可共享测试事件，请检查您的账户是否具有这些操作所需的权限。如果您拥有所需的权限但仍然无法访问可共享测试事件，请检查任何基于资源的策略，因为这些策略可能会限制对 EventBridge（CloudWatch Events）注册表的访问权限。

创建可共享测试事件

1. 打开 Lamba 控制台的函数页面。
2. 选择要测试的函数的名称。
3. 选择测试选项卡。
4. 在 Test event（测试事件）下执行以下操作：
   a. 选择一个 Template（模板）。
   b. 输入测试的 Name（名称）。
   c. 在文本输入框中，输入 JSON 测试事件。
   d. 在 Event sharing settings（事件共享设置）下，选择 Shareable（可共享）。
5. 选择 Save changes（保存更改）。

> 通过 AWS Serverless Application Model 使用可共享的测试事件。
>
> 您可以使用 AWS SAM 调用可共享的测试事件。请参阅《AWS Serverless Application Model 开发人员指南》中的 `sam remote test-event`。

删除可共享测试事件 Schema

当您删除可共享测试事件时，Lambda 会将其从 `lambda-testevent-schemas` 注册表移除。如果您从注册表中移除了最后一个可共享测试事件，Lambda 将删除该注册表。

如果您删除了函数，Lambda 不会删除任何关联的可共享测试事件 Schema。您必须从 EventBridge（CloudWatch Events）控制台手动清除这些资源。

Lambda 函数状态

Lambda 在所有函数的函数配置中包含一个状态字段，以指示您的函数何时可以调用。State 提供了有关函数当前状态的信息，包括您是否可以成功调用该函数。函数状态不会改变函数调用的行为或函数运行代码的方式。函数状态包括：

- Pending – Lambda 创建函数后，它将状态设置为待处理。处于待处理状态时，Lambda 会尝试为函数创建或配置资源，例如 VPC 或 EFS 资源。Lambda 在待处理状态期间不调用函数。在函数上运行的任何调用或其他 API 操作都将失败。
- Active – Lambda 完成资源配置和预置后，函数将转换为激活状态。函数只能在激活时成功调用。
- Failed – 表示资源配置或预置遇到错误。
- Inactive – 当函数空闲时间足够长，以便 Lambda 回收为其配置的外部资源时，函数变为非激活状态。当您尝试调用非激活函数时，调用会失败，Lambda 将函数设置为待处理状态，直到重新创建函数资源。如果 Lambda 无法重新创建资源，则函数将返回到非激活状态。如果您的函数停留在非活动状态，则请参阅函数的 StatusCode 和 StatusCodeReason 属性以进行进一步的故障排除。您可能需要解决错误并重新部署函数以将其恢复到活动状态。

如果您使用基于 SDK 的自动化工作流程或直接调用 Lambda 的服务 API，请确保在调用之前检查函数的状态以验证函数是否处于活动状态。您可以使用 Lambda API 操作 GetFunction 来执行此任务，或者使用 AWS SDK for Java 2.0 来配置 Waiter。

```
aws lambda get-function --function-name my-function --query 'Configuration.[State,
 LastUpdateStatus]'
```

您应看到以下输出：

```
[
 "Active",
 "Successful"
]
```

在函数创建处于挂起状态时，以下操作会失败：

- Invoke
- UpdateFunctionCode
- UpdateFunctionConfiguration

- PublishVersion

更新时的函数状态

Lambda 为使用 LastUpdateStatus 属性进行更新的函数提供其他上下文，这些函数可能为以下其他状态：

- InProgress – 正在对现有函数进行更新。在进行函数更新时，调用将转到函数以前的代码和配置。
- Successful – 更新已完成。Lambda 完成更新后，该设置将保留，直到进一步更新为止。
- Failed – 函数更新失败。Lambda 中止更新，并且函数以前的代码和配置保持可用。

Example

以下是正在进行更新的函数上的 get-function-configuration 的结果。

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs20.x",
    "VpcConfig": {
        "SubnetIds": [
            "subnet-071f712345678e7c8",
            "subnet-07fd123456788a036",
            "subnet-0804f77612345cacf"
        ],
        "SecurityGroupIds": [
            "sg-085912345678492fb"
        ],
        "VpcId": "vpc-08e1234569e011e83"
    },
    "State": "Active",
    "LastUpdateStatus": "InProgress",
    ...
}
```

FunctionConfiguration 还有两个其他属性，LastUpdateStatusReason 和 LastUpdateStatusReasonCode，以帮助您解决与更新相关的问题。

正在进行异步更新时，以下操作会失败：

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion
- TagResource

了解 Lambda 中的重试行为

直接调用函数时，您需要确定处理与函数代码相关的错误的策略。Lambda 不会自动代表您重试这些类型的错误。要进行重试，您可以说动重新调用函数，将事件发送到队列以进行调试，或者忽略该错误。您的函数代码可能已完全、部分或根本不运行。如要重试，请确保您的函数代码可以多次处理相同的事件，而不会导致重复的事务或其他不必要的副作用。

间接调用函数时，您需要了解调用者的重试行为以及请求在此过程中遇到的任何服务。这包括以下场景。

- 异步调用 – Lambda 会针对函数错误重试两次。如果该函数没有足够的容量来处理所有传入请求，则事件可能会在队列中等待数小时才能发送到该函数。您可以在函数上配置死信队列以捕获未成功处理的事件。有关更多信息，请参阅异步调用 Lambda 函数。
- 事件源映射 – 从流中读取的事件源映射将重试整个批次的项。重复错误会阻止受影响的分片的处理，直到错误得到解决或项失效为止。要检测停顿的分片，您可以监控迭代器期限指标。

  对于从队列中读取的事件源映射，您可以通过在源队列上配置可见性超时和重新驱动策略来确定失败事件的重试次数以及目标之间的时间长度。有关更多信息，请参阅 Lambda 如何处理来自基于流和队列的事件源的记录 以及 使用来自其 AWS 他服务的事件调用 Lambda 下特定于服务的主题。
- AWS服务 – AWS服务可能同步或异步调用您的函数。对于同步调用，服务将决定是否重试。例如，如果 Lambda 函数返回 `TemporaryFailure` 响应代码，则 Amazon S3 批处理操作会重试该操作。代理来自上游用户或客户端的请求的服务可能具有重试策略，或者可能将错误响应中继回请求者。例如，API Gateway 始终将错误响应中继回请求者。

  对于异步调用，无论调用源如何，重试逻辑都是相同的。默认情况下，对于失败的异步调用，Lambda 最多会重试两次。有关更多信息，请参阅 Lambda 如何处理异步调用的错误和重试。
- 其他账户和客户端 – 当您向其他账户授予访问权限时，可以使用基于资源的策略来限制可配置为调用您的函数的服务或资源。为了保护您的函数不发生过载情况，请考虑使用 Amazon API Gateway 在您的函数前面放置一个 API 层。

为了帮助您处理 Lambda 应用程序中的错误，Lambda 集成了 Amazon CloudWatch 和 AWS X-Ray 等服务。您可以结合使用日志、指标、警报和跟踪来快速检测和识别函数代码，API 或支持您的应用程序的其他资源中的问题。有关更多信息，请参阅 监控 Lambda 函数并进行故障排除。

使用 Lambda 递归循环检测来防止无限循环

当您配置 Lambda 函数输出到调用该函数的同一服务或资源时，就可能会创建无限递归循环。例如，Lambda 函数可能会向 Amazon Simple Queue Service（Amazon SQS）队列写入一条消息，该队列随即调用同一函数。此调用导致该函数向队列写入另一条消息，而队列反过来再次调用该函数。

意外发生的递归循环可能会让您的 AWS 账户 产生意外费用。循环还可能导致 Lambda 扩展并使用您账户的所有可用并发。为了帮助减轻意外循环的影响，Lambda 可以在某些类型的递归循环发生后不久将其检测出。默认情况下，在检测到递归循环时，Lambda 会停止调用函数并向您发送通知。如果您的设计有意使用递归模式，则可以更改函数的默认配置，以允许其以递归方式调用。请参阅the section called “允许 Lambda 函数在递归循环中运行”了解更多信息。

Sections

- 了解递归循环检测
- 受支持的 AWS 服务 和开发工具包
- 递归循环通知
- 响应递归循环检测通知
- 允许 Lambda 函数在递归循环中运行
- 支持 Lambda 递归循环检测的区域

了解递归循环检测

Lambda 中的递归循环检测通过跟踪事件来工作。Lambda 是一种事件驱动型计算服务，可在某些事件发生时运行您的函数代码。例如，将项目添加到 Amazon SQS 队列或 Amazon Simple Notification Service（Amazon SNS）主题时，就会如此。Lambda 将事件作为 JSON 对象传递给您的函数，其中包含有关系统状态变化的信息。如果某一事件导致您的函数运行时，这就称为调用。

为了检测递归循环，Lambda 会使用 AWS X-Ray 跟踪标头。当支持递归循环检测的 AWS 服务 将事件发送到 Lambda 时，这些事件将自动使用元数据进行注释。当您的 Lambda 函数使用支持的 AWS 开发工具包版本将其中一个事件写入另一个支持的 AWS 服务 时，就会更新此元数据。更新后的元数据包括事件调用该函数的次数计数。

> **Note**
>
> 您无需启用 X-Ray 主动追踪，即可使用此功能。默认情况下，所有 AWS 客户都启用递归循环检测。使用该功能不会产生任何费用。

请求链是由同一触发事件引起的一系列 Lambda 调用。例如，假设 Amazon SQS 队列调用了您的 Lambda 函数。然后，Lambda 函数将处理过的事件发送回同一 Amazon SQS 队列，该队列将再次调用您的函数。在此示例中，函数的每次调用都属于同一请求链。

如果您的函数在同一请求链中被调用的次数超过 16 次，Lambda 会自动停止该请求链中的下一次函数调用并向您发送通知。如果您的函数配置了多个触发器，来自其他触发器的调用则不会受到影响。

> **Note**
>
> 即使在源队列的重新驱动策略的 `maxReceiveCount` 设置高于 16 时，Lambda 递归保护也不会阻止 Amazon SQS 在检测到递归循环并终止后重试消息。当 Lambda 检测到递归循环并丢弃后续调用时，它会向事件源映射返回 `RecursiveInvocationException`。这会增加消息的 `receiveCount` 值。Lambda 会继续重试该消息，并继续阻止函数调用，直到 Amazon SQS 确定已超出 `maxReceiveCount` 并将消息发送到配置的死信队列。

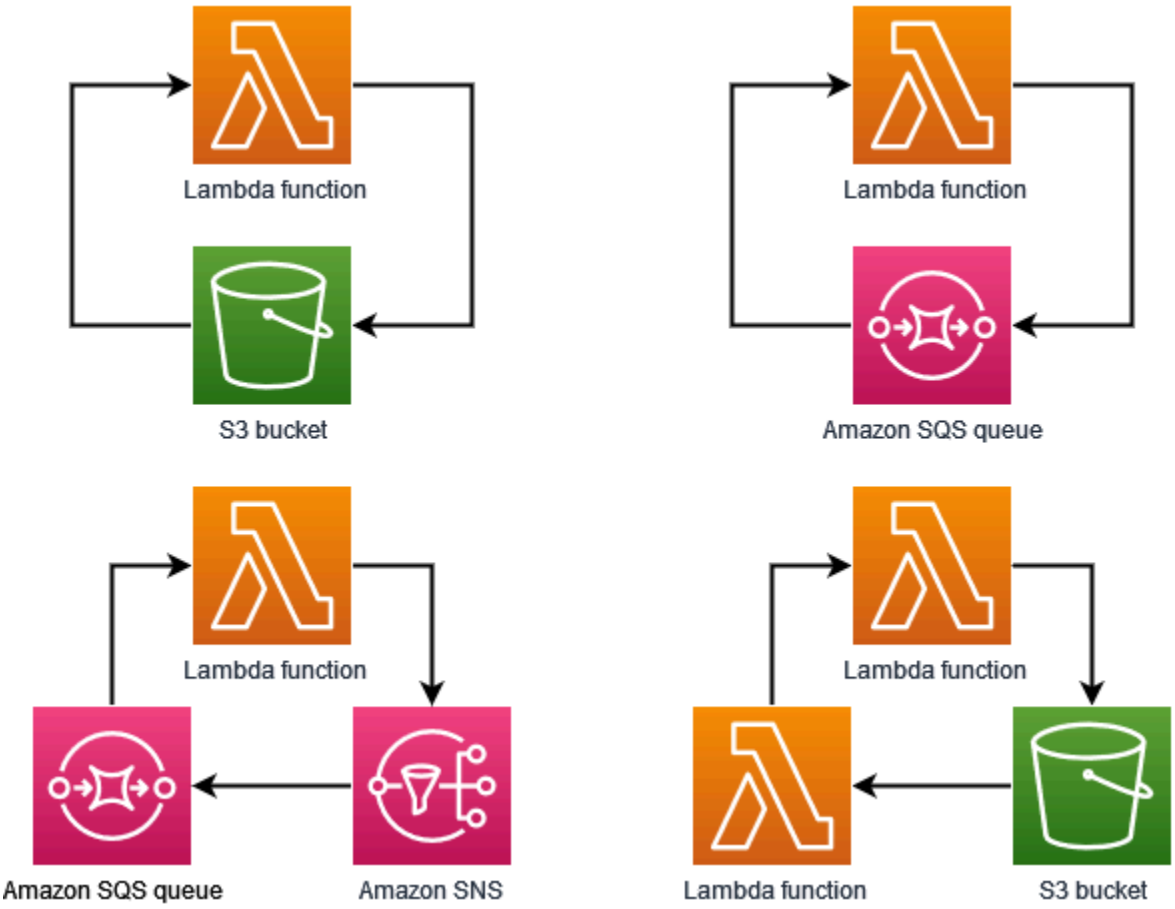
如果您为函数配置了失败时的目标或死信队列，则 Lambda 还会将已停止调用的事件发送到您的目标或死信队列。为函数配置目标或死信队列时，确保不要将函数也在使用的 Amazon SNS 主题或 Amazon SQS 队列，用作事件触发器或事件源映射。如果您将事件发送到调用函数的同一资源，则可以创建另一个递归循环。

受支持的 AWS 服务 和开发工具包

Lambda 只能检测包含某些受支持的 AWS 服务 的递归循环。为了检测到递归循环，您的函数还必须使用一种受支持的 AWS 开发工具包。

支持 AWS 服务

Lambda 目前可检测您的函数、Amazon SQS、Amazon S3 和 Amazon SNS 之间的递归循环。Lambda 还会检测仅由 Lambda 函数组成的循环，这些函数可以同步或异步地相互调用。下图显示了 Lambda 可以检测的一些循环示例：



如果 Amazon DynamoDB 等其他 AWS 服务构成循环的一部分，Lambda 目前无法对其进行检测和阻止。

由于 Lambda 目前仅检测涉及 Amazon SQS、Amazon S3 和 Amazon SNS 的递归循环，因此涉及其他 AWS 服务的循环仍有可能导致您的 Lambda 函数遭意外使用。

为防止 AWS 账户 产生意外费用，我们建议您配置 Amazon CloudWatch 警报，提醒自己注意异常使用模式。例如，您可以将 CloudWatch 配置为在 Lambda 函数并发或调用出现峰值时向自己发送通知。您还可以配置账单警报，以便在账户中的支出超过指定的阈值时通知您。您也可以使用 AWS Cost Anomaly Detection 来提醒自己注意异常的计费模式。

受支持的 AWS 开发工具包

要让 Lambda 检测递归循环，您的函数必须使用以下开发工具包版本中的一种版本或更高版本：

运行时系统	所需 AWS 开发工具包的最低版本
Node.js	2.1147.0（开发工具包版本 2）

运行时系统	所需 AWS 开发工具包的最低版本
	3.105.0（开发工具包版本 3）
Python	1.24.46（boto3）
	1.27.46（botocore）
Java 8 和 Java 11	2.17.135
Java 17	2.20.81
Java 21	2.21.24
.NET	3.7.293.0
Ruby	3.134.0
PHP	3.232.0
Go	SDK V2（使用最新版本）

某些 Lambda 运行时系统（例如 Python 和 Node.js）包含 AWS 开发工具包的一个版本。如果函数运行时系统中包含的开发工具包版本低于所需的最低版本，您可以将受支持的开发工具包版本添加到函数的部署包中。您还可以使用 Lambda 层将受支持的开发工具包版本添加到自己的函数中。有关每个 Lambda 运行时系统包含的开发工具包的列表，请参阅 Lambda 运行时。

递归循环通知

当 Lambda 停止递归循环时，您会通过 AWS Health Dashboard 或电子邮件收到通知。您还可以使用 CloudWatch 指标来监控 Lambda 已停止的递归的调用次数。

AWS Health Dashboard 通知

当 Lambda 停止递归调用时，AWS Health Dashboard 会在账户运行状况页面的未决问题和近期问题下显示一条通知。请注意，在 Lambda 停止递归调用后，最多可能需要三个小时才能显示此通知。有关在 AWS Health Dashboard 中查看账户事件的更多信息，请参阅《AWS 运行状况用户指南》中的 Getting started with your AWS Health Dashboard – Your account health。

电子邮件警报

在首次停止函数的递归调用时，Lambda 会向您发送电子邮件警报。Lambda 每 24 小时最多为您 AWS 账户 中的每个函数发送一封电子邮件。在 Lambda 发送电子邮件通知后，即使 Lambda 停止对该函数的进一步递归调用，您也不会在接下来的 24 小时内再收到该函数的更多电子邮件。请注意，在 Lambda 停止递归调用后，最多可能需要三个小时您才会收到此通知。

Lambda 会向您 AWS 账户 的主要账户联系人和备用运营联系人发送递归循环电子邮件警报。有关查看或更新账户中电子邮件地址的信息，请参阅《AWS 一般参考》中的 Updating contact information。

Amazon CloudWatch 指标

CloudWatch 指标 `RecursiveInvocationsDropped` 会记录 Lambda 因您的函数在单个请求链中被调用次数超过约 16 次而停止的函数调用次数。Lambda 会在停止递归调用后立即发出此指标。要查看此指标，请按照在 CloudWatch 控制台上查看指标的说明进行操作，然后选择指标 `RecursiveInvocationsDropped`。

响应递归循环检测通知

当同一触发事件调用您的函数超过约 16 次时，Lambda 会停止该事件的下一次函数调用，以便中断递归循环。为防止 Lambda 中断的递归循环再次出现，请执行以下操作：

- 将函数的可用并发减少到零，即可限制未来发生的所有调用。
- 移除或禁用会调用函数的触发器或事件源映射。
- 识别并修复会将事件写回会调用函数的 AWS 资源的代码缺陷。在使用变量定义函数的事件源和目标时，就会出现常见的缺陷来源。请检查并确认您为两个变量使用的是不同值。

此外，如果您的 Lambda 函数的事件源是 Amazon SQS 队列，则可以考虑在源队列上配置死信队列。

> **Note**
>
> 确保在源队列上配置死信队列，而不是在 Lambda 函数上配置。您在函数上配置的死信队列用于函数的异步调用队列，而不是用于事件源队列。

如果事件源是 Amazon SNS 主题，请考虑为您的函数添加失败时的目标。

将函数的可用并发减少到零（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择函数的名称。
3. 选择限制。
4. 在限制函数对话框中，选择确认。

删除函数的触发器或事件源映射（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择函数的名称。
3. 选择配置选项卡，然后选择触发器。
4. 在触发器下，选择要删除的触发器或事件源映射，然后选择删除。
5. 在删除触发器对话框中，选择删除。

禁用函数的事件源映射（AWS CLI）

1. 要找到要禁用的事件源映射的 UUID，请运行 AWS Command Line Interface（AWS CLI）list-event-source-mappings 命令。

```
aws lambda list-event-source-mappings
```

2. 要禁用事件源映射，请运行以下 AWS CLI update-event-source-mapping 命令。

```
aws lambda update-event-source-mapping --function-name MyFunction \
--uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 --no-enabled
```

允许 Lambda 函数在递归循环中运行

如果您的设计有意使用递归循环，则可以配置 Lambda 函数以允许递归调用该函数。我们建议避免在设计中使用递归循环。实施错误可能导致递归调用使用您 AWS 账户 的所有可用并发量，并向您的账户收取意外费用。

> **Important**
>
> 如果您使用递归循环，请谨慎对待。实施最佳实践防护轨道，以最大限度地降低实施错误的风险。要了解使用递归模式的最佳实践的更多信息，请参阅 Serverless Land 中的 Recursive patterns that cause run-away Lambda functions。

您可以使用 Lambda 控制台、AWS Command Line Interface（AWS CLI）和 PutFunctionRecursionConfig API 将函数配置为允许递归循环。您也可以在 AWS SAM 和 AWS CloudFormation 中配置函数的递归循环检测设置。

默认情况下，Lambda 会检测并终止递归循环。除非您的设计有意使用递归循环，否则我们建议您不要更改函数的默认配置。

请注意，当您将函数配置为允许递归循环时，不会发出 CloudWatch 指标 `RecursiveInvocationsDropped`。

Console

允许函数在递归循环中运行（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择函数的名称，打开函数详细信息页面。
3. 选择配置选项卡，然后选择并发和递归检测。
4. 在递归循环检测旁边，选择编辑。
5. 选择允许递归循环。
6. 选择保存。

AWS CLI

您可以使用 PutFunctionRecursionConfig API 来允许在递归循环中调用您的函数。为递归循环参数指定 `Allow`。例如，您可以使用 `put-function-recursion-config` AWS CLI 命令调用此 API：

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Allow
```

您可以将函数的配置更改回默认设置，以便 Lambda 在检测到递归循环时将其终止。使用 Lambda 控制台或 AWS CLI 编辑函数的配置。

Console

配置函数以终止递归循环（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择函数的名称，打开函数详细信息页面。
3. 选择配置选项卡，然后选择并发和递归检测。
4. 在递归循环检测旁边，选择编辑。
5. 选择终止递归循环。
6. 选择保存。

AWS CLI

您可以使用 PutFunctionRecursionConfig API 来配置您的函数，这样 Lambda 在检测到递归循环时将其终止。为递归循环参数指定 `Terminate`。例如，您可以使用 `put-function-recursion-config` AWS CLI 命令调用此 API：

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Terminate
```

支持 Lambda 递归循环检测的区域

以下 AWS 区域 中支持 Lambda 递归循环检测。

- 美国东部（弗吉尼亚州北部）
- 美国东部（俄亥俄州）
- 美国西部（加利福尼亚北部）
- 美国西部（俄勒冈州）
- 非洲（开普敦）
- Asia Pacific (Hong Kong)
- 亚太地区（雅加达）
- 亚太地区（孟买）

- 亚太地区（大阪）
- 亚太地区（首尔）
- 亚太地区（新加坡）
- 亚太地区（悉尼）
- 亚太地区（东京）
- 加拿大（中部）
- 欧洲地区（法兰克福）
- 欧洲地区（爱尔兰）
- 欧洲地区（伦敦）
- 欧洲地区（米兰）
- 欧洲地区（巴黎）
- 欧洲地区（斯德哥尔摩）
- 中东（巴林）
- 南美洲（圣保罗）

创建和管理 Lambda 函数 URL

函数 URL 是 Lambda 函数的专用 HTTP(S) 端点。您可以通过 Lambda 控制台或 Lambda API 创建和配置函数 URL。创建函数 URL 时，Lambda 会自动为您生成唯一的 URL 端点。创建函数 URL 后，其 URL 端点永远不会改变。函数 URL 的端点具有以下格式：

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 以下 AWS 区域 不支持函数 URL：亚太地区（海得拉巴）(ap-south-2)、亚太地区（墨尔本）(ap-southeast-4)、亚太地区（马来西亚）(ap-southeast-5)、加拿大西部（卡尔加里）(ca-west-1)、欧洲（西班牙）(eu-south-2)、欧洲（苏黎世）(eu-central-2)、以色列（特拉维夫）(il-central-1) 和中东（阿联酋）(me-central-1)。

函数 URL 启用了双堆栈，支持 IPv4 和 IPv6。为函数配置函数 URL 后，可以通过 Web 浏览器、curl、Postman 或任何 HTTP 客户端通过其 HTTP（S）端点调用函数。

> **Note**
>
> 您只能通过公共 Internet 访问自己的函数 URL。虽然 Lambda 函数确实支持 AWS PrivateLink，但函数 URL 并不支持。

Lambda 函数 URL 使用基于资源的策略进行安全和访问控制。函数 URL 还支持跨源资源共享 (CORS) 配置选项。

可以将函数 URL 应用于任何函数别名或 $LATEST 未发布的函数版本。不能将函数 URL 添加到任何其他函数版本。

以下部分介绍如何使用 Lambda 控制台、AWS CLI 和 AWS CloudFormation 模板创建和管理函数 URL

主题

- 创建函数 URL（控制台）
- 创建函数 URL (AWS CLI)

- 向 CloudFormation 模板添加函数 URL
- 跨源资源共享 (CORS)
- 节流函数 URL
- 停用函数 URL
- 删除函数 URL
- 控制对 Lambda 函数 URL 的访问
- 调用 Lambda 函数 URL
- 监控 Lambda 函数 URL
- 教程：使用函数 URL 创建 Lambda 函数

创建函数 URL（控制台）

按照以下步骤，使用控制台创建函数 URL。

为现有函数创建函数 URL（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择您要为其创建函数 URL 的函数的名称。
3. 选择 Configuration（配置）选项卡，然后选择 Function URL（函数 URL）。
4. 选择 Create function URL（创建函数 URL）。
5. 对于 Auth type（身份验证类型），选择 AWS_IAM 或 NONE。有关函数 URL 身份验证的更多信息，请参阅 访问控制。
6. （可选）选择 Configure cross-origin resource sharing (CORS)（配置跨源资源共享），然后为函数 URL 配置 CORS 设置。有关 CORS 的更多信息，请参阅 跨源资源共享 (CORS)。
7. 选择保存。

这将为函数的 $LATEST 未发布版本创建函数 URL。函数 URL 将显示在控制台的 Function overview（函数概览）部分。

为现有别名创建函数 URL（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择具有您要为其创建函数 URL 的别名的函数的名称。

3. 选择 Aliases（别名）选项卡，然后选择要为其创建函数 URL 的别名的名称。
4. 选择 Configuration（配置）选项卡，然后选择 Function URL（函数 URL）。
5. 选择 Create function URL（创建函数 URL）。
6. 对于 Auth type（身份验证类型），选择 AWS_IAM 或 NONE。有关函数 URL 身份验证的更多信息，请参阅 访问控制。
7. （可选）选择 Configure cross-origin resource sharing (CORS)（配置跨源资源共享），然后为函数 URL 配置 CORS 设置。有关 CORS 的更多信息，请参阅 跨源资源共享 (CORS)。
8. 选择保存。

这将为函数别名创建函数 URL。函数 URL 将显示在控制台中您的别名的 Function overview（函数概览）部分。

创建具有函数 URL 的新函数（控制台）

创建具有函数 URL 的新函数（控制台）

1. 打开 Lamba 控制台的 Functions page（函数页面）。
2. 选择 Create function（创建函数）。
3. 在 Basic information（基本信息）中，执行以下操作：
   a. 在 Function name（函数名称）中，输入您的函数的名称，例如 **`my-function`**。
   b. 对于 Runtime（运行时），请选择您首选的语言运行时，例如 Node.js 18.x。
   c. 对于 Architecture（架构），请选择 x86_64 或 arm64。
   d. 展开 Permissions（权限），然后选择是创建新的执行角色还是使用现有的执行角色。
4. 展开 Advanced settings（高级设置），然后选择 Function URL（函数 URL）。
5. 对于 Auth type（身份验证类型），选择 AWS_IAM 或 NONE（无）。有关函数 URL 身份验证的更多信息，请参阅 访问控制。
6. （可选）选择 Configure cross-origin resource sharing (CORS)（配置跨源资源共享）。如果在函数创建过程中选择此选项，则原定设置下，函数 URL 将允许来自所有来源的请求。创建函数后，可以编辑函数 URL 的 CORS 设置。有关 CORS 的更多信息，请参阅 跨源资源共享 (CORS)。
7. 选择 Create function (创建函数)。

这将为函数的 $LATEST 未发布版本创建一个具有函数 URL 的新函数。函数 URL 将显示在控制台的 Function overview（函数概览）部分。

创建函数 URL (AWS CLI)

要使用 AWS Command Line Interface (AWS CLI) 为现有 Lambda 函数创建函数 URL，请运行以下命令：

```
aws lambda create-function-url-config \
    --function-name my-function \
    --qualifier prod \ // optional
    --auth-type AWS_IAM
    --cors-config {AllowOrigins="https://example.com"} // optional
```

这会将函数 URL 添加到函数 **my-function** 的 **prod** 限定符中。有关这些配置参数的更多信息，请参阅 API 参考中的CreateFunctionUrlConfig。

> **Note**
>
> 要通过 AWS CLI 创建函数 URL，该函数必须已经存在。

向 CloudFormation 模板添加函数 URL

要向 AWS CloudFormation 模板中添加 `AWS::Lambda::Url` 资源，请使用以下语法：

JSON

```
{
  "Type" : "AWS::Lambda::Url",
  "Properties" : {
      "AuthType" : String,
      "Cors" : Cors,
      "Qualifier" : String,
      "TargetFunctionArn" : String
    }
}
```

YAML

```
Type: AWS::Lambda::Url
Properties:
  AuthType: String
```

```
  Cors:
    Cors
  Qualifier: String
  TargetFunctionArn: String
```

参数

- （必需）AuthType – 定义函数 URL 的身份验证类型。可能的值为 AWS_IAM 或 NONE。若要将访问权限限制为仅经过身份验证的用户，请设置为 AWS_IAM。要绕过 IAM 身份验证并允许任何用户对您的函数发出请求，请设置为 NONE。
- （可选）Cors – 定义函数 URL 的 CORS settings（CORS 设置）。要在 CloudFormation 中向 AWS::Lambda::Url 资源添加 Cors，请使用以下语法。

  Example AWS::Lambda::Url.Cors (JSON)

  ```
  {
    "AllowCredentials" : Boolean,
    "AllowHeaders" : [ String, ... ],
    "AllowMethods" : [ String, ... ],
    "AllowOrigins" : [ String, ... ],
    "ExposeHeaders" : [ String, ... ],
    "MaxAge" : Integer
  }
  ```

  Example AWS::Lambda::Url.Cors (YAML)

  ```
    AllowCredentials: Boolean
    AllowHeaders:
      - String
    AllowMethods:
      - String
    AllowOrigins:
      - String
    ExposeHeaders:
      - String
    MaxAge: Integer
  ```

- （可选）Qualifier – 别名。
- （必需）TargetFunctionArn – Lambda 函数的名称或 Amazon 资源名称（ARN）。有效的名称格式包括：

- 函数名称 – `my-function`
- 函数 ARN – `arn:aws:lambda:us-west-2:123456789012:function:my-function`
- 部分 ARN – `123456789012:function:my-function`

跨源资源共享 (CORS)

要定义不同来源如何访问函数 URL，请使用 跨源资源共享 (CORS)。如果要从其他域调用函数 URL，建议配置 CORS。Lambda 支持函数 URL 的以下 CORS 标头。

CORS 标头	CORS 配置属性	示例值

Access-Control-Allow-Origin	`AllowOrigins`	`*`（允许所有源）
		`https://www.example.com`
		`http://localhost:60905`

Access-Control-Allow-Methods	`AllowMethods`	`GET, POST, DELETE, *`
Access-Control-Allow-Headers	`AllowHeaders`	`Date, Keep-Alive ,X-Custom-Header`
Access-Control-Expose-Headers	`ExposeHeaders`	`Date, Keep-Alive ,X-Custom-Header`
Access-Control-Allow-Credentials	`AllowCredentials`	`TRUE`
Access-Control-Max-Age	`MaxAge`	`5 (默认值)300`

使用 Lambda 控制台或 AWS CLI 为函数 URL 配置 CORS 时，Lambda 会通过函数 URL 自动将 CORS 标头添加到所有响应中。或者，您也可以手动将 CORS 标头添加到函数响应中。如果存在冲突的标头，则预期行为取决于请求的类型：

- 对于 OPTIONS 请求之类的预检请求，以函数 URL 上配置的 CORS 标头为准。Lambda 在响应中仅返回这些 CORS 标头。

- 对于 GET 或 POST 请求之类的非预检请求，Lambda 会同时返回函数 URL 上配置的 CORS 标头以及函数返回的 CORS 标头。这可能会导致响应中出现重复的 CORS 标头。您将看到类似以下的错误：`The 'Access-Control-Allow-Origin' header contains multiple values '*, *', but only one is allowed`。

通常，我们建议在函数 URL 上配置所有 CORS 设置，而不是在函数响应中手动发送 CORS 标头。

节流函数 URL

节流限制了函数处理请求的速率。这在很多情况下都很有用，比如防止函数过载下游资源，或者处理请求突然激增的情况。

通过配置预留并发，可以节流 Lambda 函数通过函数 URL 处理请求的速率。预留并发限制了函数的最大并发调用次数。函数的最大每秒请求速率 (RPS) 相当于配置的预留并发的 10 倍。例如，如果将函数配置为预留并发 100，则最大 RPS 为 1,000。

当函数并发超过预留并发时，函数 URL 将返回 HTTP 429 状态码。如果您的函数收到的请求超过了基于您配置的预留并发的最大 10 倍 RPS，那么您也会收到一个 HTTP 429 错误。有关预留并发的更多信息，请参阅 为函数配置预留并发。

停用函数 URL

在紧急情况下，您可能希望拒绝函数 URL 的所有流量。要停用函数 URL，请将预留并发设置为零。这会节流对函数 URL 的所有请求，从而产生 HTTP 429 状态响应。要重新激活函数 URL，请删除预留的并发配置，或将配置设置为大于零的数量。

删除函数 URL

删除函数 URL 后，将无法恢复。创建一个新函数 URL 将导致不同的 URL 地址。

> **Note**
>
> 如果您删除身份验证类型为 `NONE` 的函数 URL，Lambda 不会自动删除关联的基于资源的策略。如果要删除此策略，您必须手动执行该操作。

1. 打开 Lamba 控制台的函数页面。
2. 选择此函数的名称。
3. 选择 Configuration（配置）选项卡，然后选择 Function URL（函数 URL）。

4. 选择删除。
5. 在此字段中输入 delete（删除）短语以确认删除。
6. 选择删除。

> **Note**
>
> 删除具有函数 URL 的函数后，Lambda 会异步删除函数 URL。如果您立即在同一个账户中创建具有相同名称的新函数，则原始函数 URL 可能会被映射到新函数，而不是被删除。

控制对 Lambda 函数 URL 的访问

您可以使用 `AuthType` 参数和附上特定函数的基于资源的策略来控制对 Lambda 函数 URL 的访问。这两个组件的配置决定了谁可以对函数 URL 调用或执行其他管理操作。

`AuthType` 参数确定了 Lambda 如何对函数 URL 的请求进行身份验证或授权。配置函数 URL 时，必须指定以下 `AuthType` 选项之一：

- `AWS_IAM` – Lambda 根据 IAM 主体的身份策略和函数基于资源的策略使用 AWS Identity and Access Management (IAM) 对请求进行身份验证和授权。如果只希望经过身份验证的 用户和角色通过函数 URL 调用函数，请选择此选项。

- `NONE` – Lambda 在调用函数之前不会执行任何身份验证。但是，函数基于资源的策略始终有效，并且必须在函数 URL 接收请求之前授予公有访问权限。选择此选项可允许对函数 URL 进行未经身份验证的公有访问。

除 `AuthType` 之外，您还可以使用基于资源的策略向其他 AWS 账户 授予调用您的函数的权限。有关更多信息，请参阅 在 Lambda 中使用基于资源的 IAM 策略。

要了解更多关于安全性的洞察，可以使用 AWS Identity and Access Management Access Analyzer 获取对函数 URL 的外部访问的全面分析。IAM Access Analyzer 还可以监控 Lambda 函数的新权限或更新权限，以帮助您识别授予公有和跨账户访问的权限。IAM Access Analyzer 可供任何 AWS 客户免费使用。要开始使用 IAM Access Analyzer，请参阅使用 AWS IAM Access Analyzer。

该页面包含两种身份验证类型的基于资源的策略示例，以及如何使用 AddPermission API 操作或 Lambda 控制台创建这些策略。有关设置权限后如何调用函数 URL 的信息，请参阅 调用 Lambda 函数 URL。

主题

使用 **AWS_IAM** 身份验证类型

如果选择 AWS_IAM 身份验证类型，则需要调用 Lambda 函数 URL 的用户必须具有 `lambda:InvokeFunctionUrl` 权限。根据发出调用请求的人员，您可能需要使用基于资源的策略授予此权限。

如果发出请求的主体与函数 URL 的 AWS 账户 相同，则主体必须或者在其基于身份的策略中拥有 `lambda:InvokeFunctionUrl` 权限，或者在函数基于资源的策略中获授权限。换句话说，如果用户已经在其基于身份的策略中拥有 `lambda:InvokeFunctionUrl` 权限，则基于资源的策略为可选。策略评估遵循确定是允许还是拒绝账户内的请求中概述的规则。

如果发出请求的主体位于不同的账户中，则主体必须同时具有基于身份的策略（该策略为其提供 `lambda:InvokeFunctionUrl` 权限）和基于资源的策略（基于其尝试调用的函数）中授予的权限。在这些跨账户情况下，策略评估遵循确定是否允许跨账户请求中概述的规则。

对于跨账户交互示例，以下基于资源的策略允许 AWS 账户 444455556666 中的 example 角色调用与函数 my-function 关联的函数 URL：

Example 函数 URL 跨账户调用策略

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:InvokeFunctionUrl",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
```

```
        }
    ]
}
```

您可以按照以下步骤通过控制台创建此策略语句：

将 URL 调用权限授予另一个账户（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择要为其授予 URL 调用权限的函数的名称。
3. 选择 Configuration（配置）选项卡，然后选择 Permissions（权限）。
4. 在 Resource-based policy（基于资源的策略）下，选择 Add permissions（添加权限）。
5. 选择 Function URL（函数 URL）。
6. 对于 Auth type（身份验证类型），选择 AWS_IAM。
7. （可选）在 Statement ID（语句 ID）中，为策略语句输入语句 ID。
8. 对于主体，请输入要向其授予权限的用户或角色的 Amazon 资源名称（ARN）。例如：**444455556666**。
9. 选择保存。

或者，也可以使用以下 add-permission AWS Command Line Interface (AWS CLI) 命令创建此策略语句：

```
aws lambda add-permission --function-name my-function \
    --statement-id example0-cross-account-statement \
    --action lambda:InvokeFunctionUrl \
    --principal 444455556666 \
    --function-url-auth-type AWS_IAM
```

在上一个示例中，`lambda:FunctionUrlAuthType` 条件键值为 `AWS_IAM`。此策略仅在函数 URL 的身份验证类型也为 `AWS_IAM` 时允许访问。

使用 **NONE** 身份验证类型

> **Important**
>
> 当您的函数 URL 身份验证类型为 `NONE` 且您有基于资源的策略授予公有访问权限时，任何使用您函数 URL 的未经身份验证的用户都可以调用您的函数。

在某些情况下，您可能希望函数 URL 为公有。例如，您可能希望处理直接从 Web 浏览器发出的请求。要允许函数 URL 公有访问权限，请选择 NONE 身份验证类型。

如果选择 NONE 身份验证类型，Lambda 不会使用 IAM 对函数 URL 的请求进行身份验证。但是，用户仍必须拥有 `lambda:InvokeFunctionUrl` 权限才能成功调用函数 URL。您可以使用以下基于资源的策略授予 `lambda:InvokeFunctionUrl` 权限：

Example 适用于所有未经身份验证的主体的函数 URL 调用策略

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunctionUrl",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "NONE"
                }
            }
        }
    ]
}
```

> **Note**
>
> 通过控制台或 AWS Serverless Application Model (AWS SAM) 创建具有身份验证类型 NONE 的函数 URL 时，Lambda 会自动创建前面的基于资源的策略语句。如果策略已经存在，或者创建应用程序的用户或角色没有相应的权限，Lambda 将不会为您创建该策略。如果直接使用 AWS CLI、AWS CloudFormation 或 Lambda API，则必须自己添加 `lambda:InvokeFunctionUrl` 权限。这会让您的函数变为公有。
>
> 此外，如果您删除身份验证类型为 NONE 的函数 URL，Lambda 不会自动删除关联的基于资源的策略。如果要删除此策略，您必须手动执行该操作。

在此语句中，`lambda:FunctionUrlAuthType` 条件键值为 NONE。此策略仅在函数 URL 的身份验证类型也为 NONE 时允许访问。

如果函数基于资源的策略未授予 `lambda:invokeFunctionUrl` 权限，则用户在尝试调用函数 URL 时，将收到 403 禁止的错误代码，即使函数 URL 使用 `NONE` 身份验证类型也是如此。

治理和访问控制

除了函数 URL 调用权限外，还可以控制对用于配置函数 URL 的操作的访问。Lambda 支持以下针对函数 URL 的 IAM policy 操作：

- `lambda:InvokeFunctionUrl` – 使用函数 URL 调用 Lambda 函数。
- `lambda:CreateFunctionUrlConfig` – 创建函数 URL 并设置其 `AuthType`。
- `lambda:UpdateFunctionUrlConfig` – 更新函数 URL 配置及其 `AuthType`。
- `lambda:GetFunctionUrlConfig` – 查看函数 URL 的详细信息。
- `lambda:ListFunctionUrlConfigs` – 列出函数 URL 配置。
- `lambda:DeleteFunctionUrlConfig` – 删除函数 URL。

> **Note**
>
> Lambda 控制台仅支持为 `lambda:InvokeFunctionUrl` 添加权限。对于所有其他操作，必须使用 Lambda API 或 AWS CLI 添加权限。

要允许或拒绝对其他 AWS 实体的函数 URL 访问，请在 IAM policy 中包含这些操作。例如，以下策略授予 AWS 账户 444455556666 中的 `example` 角色在账户 123456789012 中更新函数 **my-function** 的函数 URL 的权限。

Example 跨账户函数 URL 策略

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:UpdateFunctionUrlConfig",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function"
        }
    ]
```

```
}
```

条件键

要对函数 URL 进行精细访问控制，请使用条件键。Lambda 支持函数 URL 的一个附加条件键：FunctionUrlAuthType。FunctionUrlAuthType 键定义了一个枚举值，描述函数 URL 使用的身份验证类型。该值可以是 AWS_IAM 或 NONE。

可以在与函数关联的策略中使用此条件键。例如，您可能希望限制谁可以对函数 URL 进行配置更改。要拒绝对 URL 身份验证类型 NONE 的任何函数的所有 UpdateFunctionUrlConfig 请求，可以定义以下策略：

Example 带有显式拒绝的函数 URL 策略

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action":[
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "NONE"
                }
            }
        }
    ]
}
```

要授予 AWS 账户 444455556666 中的 example 角色对 URL 身份验证类型 AWS_IAM 的函数进行 CreateFunctionUrlConfig 和 UpdateFunctionUrlConfig 请求的权限，可以定义以下策略：

Example 带有显式允许的函数 URL 策略

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

您还可以在服务控制策略（SCP）中使用此条件键。使用 SCP 在 AWS Organizations 中管理整个企业的权限。例如，要拒绝用户创建或更新使用除 AWS_IAM 身份验证类型以外的任何身份验证类型的函数 URL，请使用以下服务控制策略：

Example 带有显式拒绝的函数 URL SCP

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:*:123456789012:function:*",
            "Condition": {
                "StringNotEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

调用 Lambda 函数 URL

函数 URL 是 Lambda 函数的专用 HTTP(S) 端点。您可以通过 Lambda 控制台或 Lambda API 创建和配置函数 URL。创建函数 URL 时，Lambda 会自动为您生成唯一的 URL 端点。创建函数 URL 后，其 URL 端点永远不会改变。函数 URL 的端点具有以下格式：

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 以下 AWS 区域 不支持函数 URL：亚太地区（海得拉巴）(ap-south-2)、亚太地区（墨尔本）(ap-southeast-4)、亚太地区（马来西亚）(ap-southeast-5)、加拿大西部（卡尔加里）(ca-west-1)、欧洲（西班牙）(eu-south-2)、欧洲（苏黎世）(eu-central-2)、以色列（特拉维夫）(il-central-1) 和中东（阿联酋）(me-central-1)。

函数 URL 启用了双堆栈，支持 IPv4 和 IPv6。配置函数 URL 后，可以通过 Web 浏览器、curl、Postman 或任何 HTTP 客户端通过其 HTTP(S) 端点调用函数。要调用函数 URL，您必须具有 `lambda:InvokeFunctionUrl` 权限。有关更多信息，请参阅 访问控制。

主题

- 函数 URL 调用基础
- 请求和响应有效负载

函数 URL 调用基础

如果函数 URL 使用 `AWS_IAM` 身份验证类型，则必须使用 AWS 签名版本 4（SigV4）对每个 HTTP 请求进行签名。awscurl、Postman 和 AWS SigV4 代理等工具提供了内置的方法使用 SigV4 对请求进行签名。

如果不使用工具向函数 URL 对 HTTP 请求进行签名，则必须使用 SigV4 手动对每个请求进行签名。当函数 URL 收到请求时，Lambda 还会计算 SigV4 签名。Lambda 仅在签名匹配时处理请求。有关如何使用 SigV4 手动对请求进行签名的说明，请参阅《Amazon Web Services 一般参考 指南》中的利用签名版本 4 对 AWS 请求进行签名。

如果函数 URL 使用 `NONE` 身份验证类型，则不必使用 SigV4 对请求进行签名。您可以使用 Web 浏览器、curl、Postman 或任何 HTTP 客户端来调用函数。

要测试对函数的简单 GET 请求，请使用 Web 浏览器。例如，如果您的函数 URL 为 `https://abcdefg.lambda-url.us-east-1.on.aws`，并且包含一个字符串参数 `message`，那么您的请求 URL 可能如下所示：

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

要测试其他 HTTP 请求，例如 POST 请求，可以使用 curl 之类的工具。例如，如果希望在对函数 URL 的 POST 请求中包含一些 JSON 数据，可以使用以下 curl 命令：

```
curl -v 'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
-H 'content-type: application/json' \
-d '{ "example": "test" }'
```

请求和响应有效负载

当客户端调用函数 URL 时，Lambda 会将请求映射到事件对象，然后再将其传递给函数。然后，函数的响应将映射到一个 HTTP 响应，Lambda 会通过函数 URL 将该响应发送回客户端。

请求和响应事件格式遵循与 Amazon API Gateway 有效负载格式版本 2.0 相同的模式。

请求有效负载格式

请求有效负载具有以下结构：

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": [
    "cookie1",
    "cookie2"
  ],
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
```

```
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "<urlid>",
    "authentication": null,
    "authorizer": {
        "iam": {
                "accessKey": "AKIA...",
                "accountId": "111122223333",
                "callerId": "AIDA...",
                "cognitoIdentity": null,
                "principalOrgId": null,
                "userArn": "arn:aws:iam::111122223333:user/example-user",
                "userId": "AIDA..."
        }
    },
    "domainName": "<url-id>.lambda-url.us-west-2.on.aws",
    "domainPrefix": "<url-id>",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "123.123.123.123",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "body": "Hello from client!",
  "pathParameters": null,
  "isBase64Encoded": false,
  "stageVariables": null
}
```

参数	描述	示例
version	此事件的有效负载格式版本。Lambda 函数 URL 目前支持有效负载格式版本 2.0。	2.0

参数	描述	示例
routeKey	函数 URL 不使用此参数。Lambda 将其设置为 $default，作为占位符。	$default
rawPath	请求路径。例如，如果请求 URL 为 https://{url-id}.lambda-url.{region}.on.aws/example/test/demo ，则原始路径值为 /example/test/demo 。	/example/test/demo
rawQueryString	包含请求的查询字符串参数的原始字符串。支持的字符包括 a-z、A-Z、0-9、.、_、-、%、&、= 和 +。	"?parameter1=value1&parameter2=value2"
cookies	数组包含发送的部分请求的所有 Cookie。	["Cookie_1=Value_1", "Cookie_2=Value_2"]
headers	请求标头的列表，以键值对的形式显示。	{"header1": "value1", "header2": "value2"}
queryStringParameters	请求的查询参数。例如，如果请求 URL 为 https://{url-id}.lambda-url.{region}.on.aws/example?name=Jane ，则 queryStringParameters 值是一个 JSON 对象，其键为 name，值为 Jane。	{"name": "Jane"}

参数	描述	示例
requestContext	一个包含有关请求的附加信息的对象，例如 requestId 、请求的时间以及通过 AWS Identity and Access Management (IAM) 授权的调用者身份。	
requestContext.accountId	函数拥有者的 AWS 账户 ID。	"123456789012"
requestContext.apiId	函数 URL 的 ID。	"33anwqw8fj"
requestContext.authentication	函数 URL 不使用此参数。Lambda 会将其设置为 null。	null
requestContext.authorizer	对象包含有关调用者身份的信息（如果函数 URL 使用 AWS_IAM 身份验证类型）。否则，Lambda 会将其设置为 null。	
requestContext.authorizer.iam.accessKey	调用者身份的访问密钥。	"AKIAIOSFODNN7EXAMPLE"
requestContext.authorizer.iam.accountId	调用者身份的 AWS 账户 ID。	"111122223333"
requestContext.authorizer.iam.callerId	调用者的 ID（用户 ID）。	"AIDACKCEVSQ6C2EXAMPLE"
requestContext.authorizer.iam.cognitoIdentity	函数 URL 不使用此参数。Lambda 会将其设置为 null，或将其从 JSON 中排除。	null

参数	描述	示例
requestContext.authorizer.iam.principalOrgId	与调用者身份关联的主体企业 ID。	"AIDACKCEVSQORGEXAMPLE"
requestContext.authorizer.iam.userArn	调用者身份的用户 Amazon 资源名称（ARN）。	"arn:aws:iam::111122223333:user/example-user"
requestContext.authorizer.iam.userId	调用者身份的用户 ID。	"AIDACOSFODNN7EXAMPLE2"
requestContext.domainName	函数 URL 的域名。	"<url-id>.lambda-url.us-west-2.on.aws"
requestContext.domainPrefix	函数 URL 的域前缀。	"<url-id>"
requestContext.http	包含有关 HTTP 请求的详细信息的对象。	
requestContext.http.method	此请求中使用的 HTTP 方法。有效值包括 GET、POST、PUT、HEAD、OPT 和 DELETE。	GET
requestContext.http.path	请求路径。例如，如果请求 URL 为 https://{url-id}.lambda-url.{region}.on.aws/example/test/demo ，则路径值为 /example/test/demo 。	/example/test/demo
requestContext.http.protocol	请求的协议。	HTTP/1.1

参数	描述	示例
requestContext.http.sourceIp	发出请求的即时 TCP 连接的源 IP 地址。	123.123.123.123
requestContext.http.userAgent	用户代理请求标头值。	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/20100101 Firefox/42.0
requestContext.requestId	调用请求的 ID。可以使用此 ID 跟踪与函数相关的调用日志。	e1506fd5-9e7b-434f-bd42-4f8fa224b599
requestContext.routeKey	函数 URL 不使用此参数。Lambda 将其设置为 $default，作为占位符。	$default
requestContext.stage	函数 URL 不使用此参数。Lambda 将其设置为 $default，作为占位符。	$default
requestContext.time	请求的时间戳。	"07/Sep/2021:22:50:22 +0000"
requestContext.timeEpoch	请求的时间戳，用 Unix 纪元时间表示。	"1631055022677"
body	请求的正文。如果请求的内容类型为二进制，则正文为 base64 编码。	{"key1": "value1", "key2": "value2"}
pathParameters	函数 URL 不使用此参数。Lambda 会将其设置为 null，或将其从 JSON 中排除。	null
isBase64Encoded	如果正文为二进制有效负载，并且为 base64 编码，则为 TRUE。否则为 FALSE。	FALSE

参数	描述	示例
stageVariables	函数 URL 不使用此参数。Lambda 会将其设置为 null，或将其从 JSON 中排除。	null

响应有效负载格式

当函数返回响应时，Lambda 会解析响应并将其转换为 HTTP 响应。函数响应有效负载的格式如下：

```
{
   "statusCode": 201,
    "headers": {
        "Content-Type": "application/json",
        "My-Custom-Header": "Custom Value"
    },
    "body": "{ \"message\": \"Hello, world!\" }",
    "cookies": [
        "Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=78000"
    ],
    "isBase64Encoded": false
}
```

Lambda 会为您推断响应格式。如果您的函数返回有效的 JSON 并且没有返回 statusCode，Lambda 会做出以下假设：

- statusCode 为 200。
- content-type 为 application/json。
- body 是函数响应。
- isBase64Encoded 为 false。

以下示例显示了 Lambda 函数的输出如何映射到响应有效负载，以及响应有效负载如何映射到最终 HTTP 响应。当客户端调用函数 URL 时，就会看到 HTTP 响应。

字符串响应的输出示例

Lambda 函数输出	解释响应输出	HTTP 响应（客户端看到的内容）

`"Hello, world!"`	`{ "statusCode": 200, "body": "Hello, world!", "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200`
		`date: Wed, 08 Sep 2021 18:02:24 GMT`
		`content-type: application/json`
		`content-length: 15`
		
		`"Hello, world!"`

JSON 响应的输出示例

Lambda 函数输出	解释响应输出	HTTP 响应（客户端看到的内容）

`{ "message": "Hello, world!" }`	`{ "statusCode": 200, "body": { "message": "Hello, world!" }, "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200`
		`date: Wed, 08 Sep 2021 18:02:24 GMT`
		`content-type: application/json`
		`content-length: 34`
		
		`{ "message": "Hello, world!" }`

自定义响应的输出示例

Lambda 函数输出	解释响应输出	HTTP 响应（客户端看到的内容）

<pre>{
  "statusCode": 201,
  "headers": {
  "Content-Type": "application/json",
  "My-Custom-Header": "Custom Value"
  },
  "body": JSON.stringify({
  "message": "Hello, world!"
  }),
  "isBase64Encoded": false
}</pre>
	<pre>{
	  "statusCode": 201,
	  "headers": {
	  "Content-Type": "application/json",
	  "My-Custom-Header": "Custom Value"
	  },
	  "body": JSON.stringify({
	  "message": "Hello, world!"
	  }),
	  "isBase64Encoded": false
	}</pre>
		<pre>HTTP/2 201
		date: Wed, 08 Sep 2021 18:02:24 GMT
		content-type: application/json
		content-length: 27
		my-custom-header: Custom Value
		
		{
		  "message": "Hello, world!"
		}</pre>

Cookie

要从函数返回 Cookie，请不要手动添加 `set-cookie` 标头。相反，请在响应有效负载对象中包含 Cookie。Lambda 会自动进行解释，并将其作为 `set-cookie` 标头添加到 HTTP 响应中，如下例所示。

Lambda 函数输出	HTTP 响应（客户端看到的内容）

<pre>{
  "statusCode": 201,
  "headers": {
  "Content-Type": "application/json",
  "My-Custom-Header": "Custom Value"
  },
  "body": JSON.stringify({
  "message": "Hello, world!"
  }),</pre>
	<pre>HTTP/2 201
	date: Wed, 08 Sep 2021 18:02:24 GMT
	content-type: application/json
	content-length: 27
	my-custom-header: Custom Value
	set-cookie: Cookie_1=Value2; Expires=21 Oct 2021 07:48 GMT
	set-cookie: Cookie_2=Value2; Max-Age=78000
	
	{</pre>

Lambda 函数输出	HTTP 响应（客户端看到的内容）

<pre>"cookies": [
"Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT",
"Cookie_2=Value2; Max-Age=78000"
],
"isBase64Encoded": false
}</pre>
	<pre>"message": "Hello, world!"
	}</pre>

监控 Lambda 函数 URL

您可以使用 AWS CloudTrail 和 Amazon CloudWatch 来监控您的函数 URL。

主题

使用 CloudTrail 监控函数 URL

对于函数 URL，Lambda 自动支持将以下 API 操作记录为 CloudTrail 日志文件中的事件：

- CreateFunctionUrlConfig
- UpdateFunctionUrlConfig
- DeleteFunctionUrlConfig
- GetFunctionUrlConfig
- ListFunctionUrlConfigs

每个日志条目都包含有关调用者身份、发出请求的时间以及其他详细信息的信息。通过查看 CloudTrail Event history（事件历史记录），可以看到过去 90 天内的所有事件。要保留过去 90 天的记录，可以创建跟踪记录。

原定设置下，CloudTrail 不会录入 `InvokeFunctionUrl` 请求，这些请求将被视为数据事件。但是，您可以在 CloudTrail 中打开数据事件日志记录。有关更多信息，请参阅 AWS CloudTrail 用户指南中的记录数据事件以便跟踪。

函数 URL 的 CloudWatch 指标

Lambda 会向 CloudWatch 发送关于函数 URL 请求的聚合指标。借助这些指标，您可以在 CloudWatch 控制台中监控函数 URL、构建控制面板和配置告警。

函数 URL 支持以下调用指标。我们建议使用 `Sum` 统计数据查看这些指标。

- `UrlRequestCount` – 向该函数 URL 发出的请求数。
- `Url4xxCount` – 返回 4XX HTTP 状态码的请求数。4XX 系列代码表示客户端错误，例如错误请求。

- Url5xxCount – 返回 5XX HTTP 状态码的请求数。5XX 系列代码表示服务器端错误，例如函数错误和超时。

函数 URL 还支持以下性能指标。我们建议使用 Average 或 Max 统计数据查看此指标。

- UrlRequestLatency – 函数 URL 收到请求和函数 URL 返回响应之间的时间。

每个调用和性能指标都支持以下维度：

- FunctionName – 查看分配给函数 $LATEST 未发布版本或任何函数别名的函数 URL 的聚合指标。例如，hello-world-function。
- Resource – 查看特定函数 URL 的指标。其由函数名称、函数的 $LATEST 未发布版本或函数的别名之一定义。例如，hello-world-function:$LATEST。
- ExecutedVersion – 根据执行的版本查看特定函数 URL 的指标。您可以主要使用此维度跟踪分配给 $LATEST 未发布版本的函数 URL。

教程：使用函数 URL 创建 Lambda 函数

在本教程中，您将创建一个 Lambda 函数，该函数定义为 .zip 文件存档，其公共函数 URL 端点返回两个数字的乘积。有关配置函数 URL 的更多信息，请参阅 函数 URL。

先决条件

本教程假设您对 Lambda 基本操作和 Lambda 控制台有一定了解。如果您还没有了解，请按照 使用控制台创建 Lambda 函数 中的说明创建您的第一个 Lambda 函数。

要完成以下步骤，您需要 AWS CLI 版本 2。在单独的数据块中列出了命令和预期输出：

```
aws --version
```

您应看到以下输出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

对于长命令，使用转义字符 (\) 将命令拆分为多行。

在 Linux 和 macOS 中，可使用您首选的 shell 和程序包管理器。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常与 Lambda 一起使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。本指南中的示例 CLI 命令使用 Linux 格式。如果您使用的是 Windows CLI，则必须重新格式化包含内联 JSON 文档的命令。

创建执行角色

创建执行角色，向您的 Lambda 函数授予访问AWS资源的权限。

创建执行角色

1. 打开 AWS Identity and Access Management（IAM）控制台的 Roles（角色）页面。
2. 选择 Create role（创建角色）。
3. 对于可信实体类型，选择 AWS 服务，然后对于应用场景，选择 Lambda。

4. 选择下一步。
5. 在权限策略窗格中，在搜索框中输入 **AWSLambdaBasicExecutionRole**。
6. 勾选 AWSLambdaBasicExecutionRole AWS 托管策略旁边的复选框，然后选择下一步。
7. 对于角色名称，输入 **lambda-url-role**，然后选择创建角色。

AWSLambdaBasicExecutionRole 策略具有函数将日志写入 Amazon CloudWatch Logs 所需的权限。在本教程的后面部分，您需要角色的 Amazon 资源名称（ARN）创建您的 Lambda 函数。

查找执行角色的 ARN

1. 打开 AWS Identity and Access Management（IAM）控制台的 Roles（角色）页面。
2. 选择您刚刚创建的角色（lambda-url-role）。
3. 在摘要窗格中，复制 ARN。

使用函数 URL（.zip 文件归档）创建 Lambda 函数

使用 .zip 文件归档创建一个具有函数 URL 端点的 Lambda 函数

创建函数

1. 将以下代码示例复制到名为 index.js 的文件中。

   Example index.js

   ```
   exports.handler = async (event) => {
       let body = JSON.parse(event.body);
       const product = body.num1 * body.num2;
       const response = {
           statusCode: 200,
           body: "The product of " + body.num1 + " and " + body.num2 + " is " +
    product,
       };
       return response;
   };
   ```

2. 创建部署程序包。

   ```
   zip function.zip index.js
   ```

3. 使用 `create-function` 命令创建 Lambda 函数。请务必将角色 ARN 替换为您在本教程前面部分中复制的您自己的执行角色的 ARN。

```
aws lambda create-function \
    --function-name my-url-function \
    --runtime nodejs18.x \
    --zip-file fileb://function.zip \
    --handler index.handler \
    --role arn:aws:iam::123456789012:role/lambda-url-role
```

4. 向函数添加基于资源的策略，授予允许对您的函数 URL 进行公有访问的权限。

```
aws lambda add-permission \
    --function-name my-url-function \
    --action lambda:InvokeFunctionUrl \
    --principal "*" \
    --function-url-auth-type "NONE" \
    --statement-id url
```

5. 使用 `create-function-url-config` 命令为函数创建 URL 端点。

```
aws lambda create-function-url-config \
    --function-name my-url-function \
    --auth-type NONE
```

测试函数 URL 端点

通过使用诸如 curl 或 Postman 之类的 HTTP 客户端调用函数 URL 端点来调用 Lambda 函数。

```
curl 'https://abcdefg.lambda-url.us-east-1.on.aws/' \
-H 'Content-Type: application/json' \
-d '{"num1": "10", "num2": "10"}'
```

您应看到以下输出：

```
The product of 10 and 10 is 100
```

使用函数 URL 创建 Lambda 函数 (CloudFormation)

还可以使用 AWS CloudFormation 类型 `AWS::Lambda::Url` 创建带有函数 URL 端点的 Lambda 函数。

```
Resources:
  MyUrlFunction:
    Type: AWS::Lambda::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs18.x
      Role: arn:aws:iam::123456789012:role/lambda-url-role
      Code:
        ZipFile: |
          exports.handler = async (event) => {
              let body = JSON.parse(event.body);
              const product = body.num1 * body.num2;
              const response = {
                  statusCode: 200,
                  body: "The product of " + body.num1 + " and " + body.num2 + " is " +
 product,
              };
              return response;
          };
      Description: Create a function with a URL.
  MyUrlFunctionPermissions:
    Type: AWS::Lambda::Permission
    Properties:
      FunctionName: !Ref MyUrlFunction
      Action: lambda:InvokeFunctionUrl
      Principal: "*"
      FunctionUrlAuthType: NONE
  MyFunctionUrl:
    Type: AWS::Lambda::Url
    Properties:
      TargetFunctionArn: !Ref MyUrlFunction
      AuthType: NONE
```

使用函数 URL 创建 Lambda 函数 (AWS SAM)

还可以使用 AWS Serverless Application Model (AWS SAM) 创建配置有函数 URL 的 Lambda 函数。

```
ProductFunction:
```

```
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: function/.
    Handler: index.handler
    Runtime: nodejs18.x
    AutoPublishAlias: live
    FunctionUrlConfig:
      AuthType: NONE
```

清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。
4. 在文本输入字段中输入角色名称，然后选择 Delete（删除）。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **`delete`**，然后选择 Delete（删除）。

了解 Lambda 函数扩展

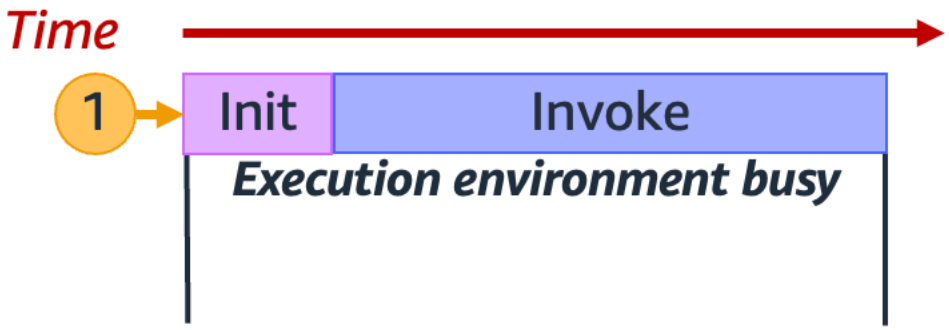
并发是您的 AWS Lambda 函数同时处理的正在进行的请求数。对于每个并发请求，Lambda 会预置单独的执行环境实例。当您的函数收到更多请求时，Lambda 会自动处理执行环境数量的扩展，直到您达到账户的并发限制。默认情况下，Lambda 为您的账户提供的一个 AWS 区域 中所有函数总并发上限为 1000 个并发执行。为了支持您的特定账户需求，您可以申请增加限额，并配置函数级并发控制，这样您的关键函数就不会节流。

本主题介绍了 Lambda 中的并发概念和函数横向缩减。在本主题结束时，您将能够了解如何计算并发、如何可视化两个主要的并发控制选项（预留类和预置类）、估计适当的并发控制设置，以及查看用于进一步优化的指标。

Sections

了解和可视化并发

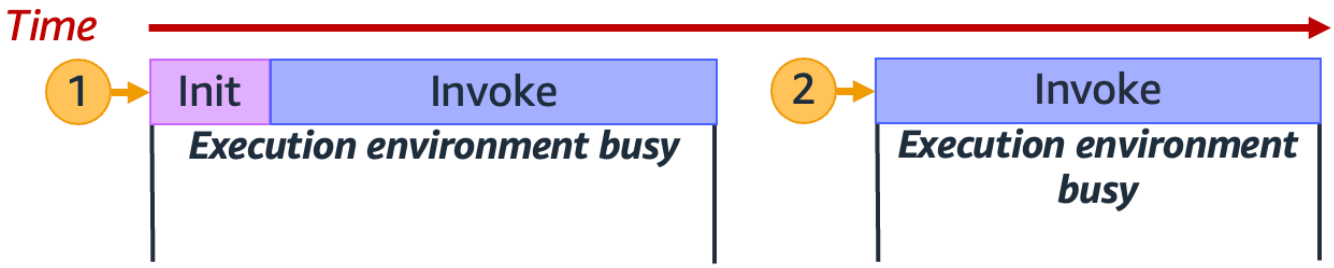
Lambda 调用一个安全和隔离的执行环境中的函数。要处理请求，Lambda 必须先初始化执行环境（Init 阶段），然后再使用它来调用您的函数（Invoke 阶段）：



> **Note**
>
> 实际的 Init 和 Invoke 持续时间可能因多种因素而异，例如您选择的运行时和 Lambda 函数代码。前面的图的目的并不是表示 Init 和 Invoke 阶段持续时间的确切比例。

上图使用矩形表示单个执行环境。当函数收到其第一个请求（由带标签 1 的黄色圆圈表示）时，Lambda 会创建一个新的执行环境并在初始化阶段在主处理程序之外运行代码。然后，Lambda 在 Invoke 阶段运行函数的主处理程序代码。在整个过程中，此执行环境繁忙，无法处理其他请求。

当 Lambda 处理完第一个请求后，此执行环境就可以处理针对同一函数的其他请求。对于后续请求，Lambda 无需重新初始化环境。

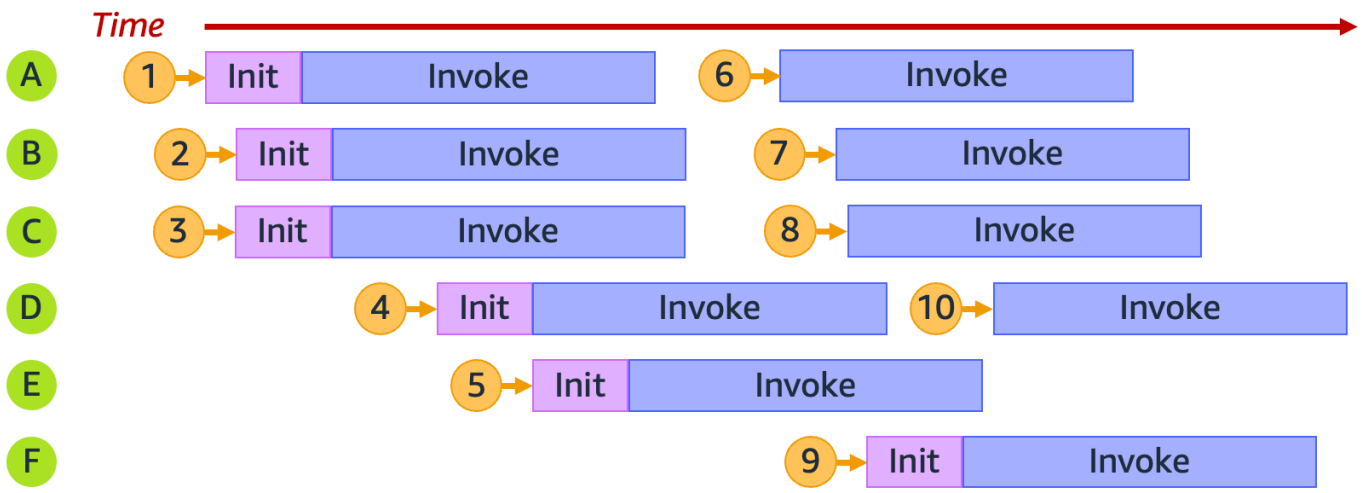


在上图中，Lambda 重复使用执行环境来处理第二个请求（由带标签 2 的黄色圆圈表示）。

到目前为止，我们只关注您的执行环境的单个实例（即并发为 1）。实际上，Lambda 可能需要并行预置多个执行环境实例来处理所有传入请求。当您的函数收到新请求时，可能会发生以下两种情况之一：

- 如果预初始化的执行环境实例可用，Lambda 会使用它来处理请求。
- 否则，Lambda 会创建一个新的执行环境实例来处理请求。

例如，让我们来看看当您的函数收到 10 个请求时会发生什么：

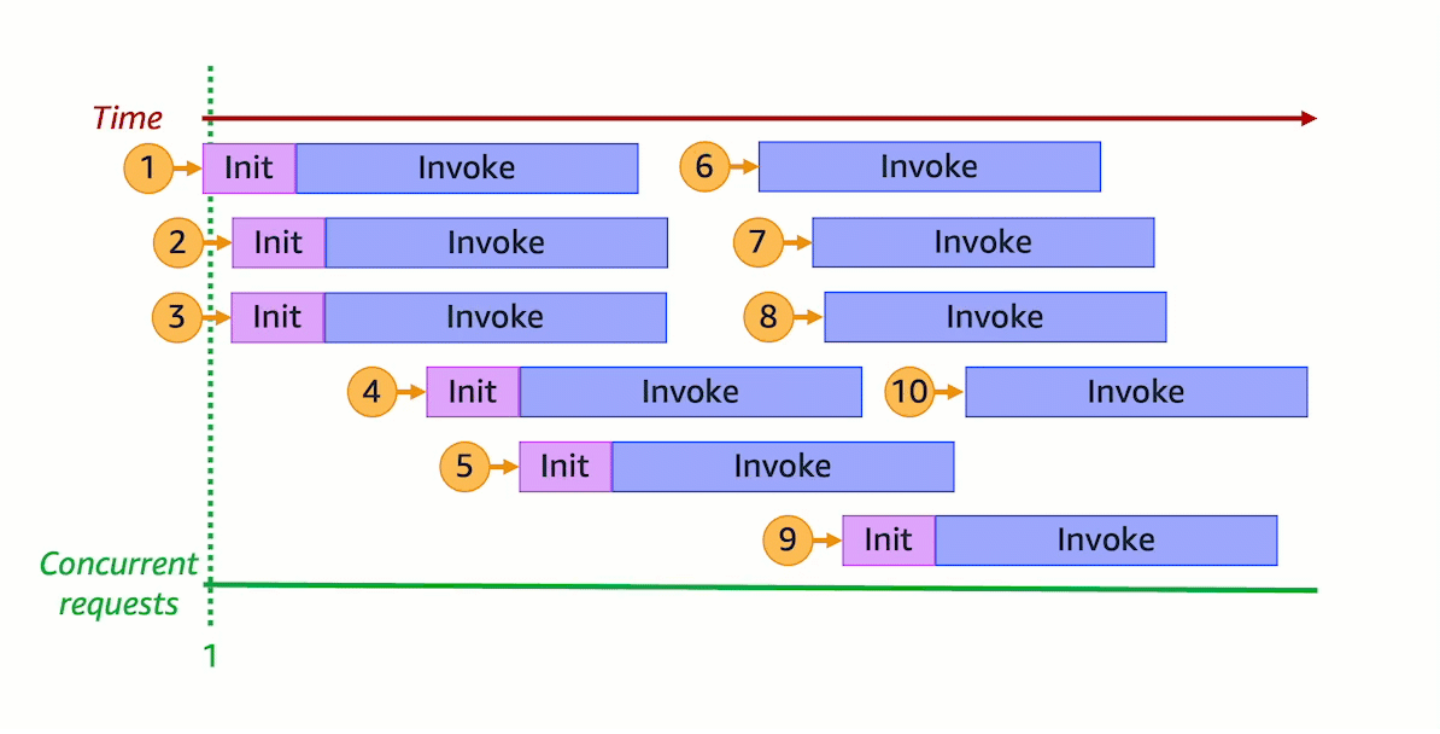


在上图中，每个水平平面代表一个执行环境实例（标记为从 A 到 F）。以下是 Lambda 处理每个请求的方式：

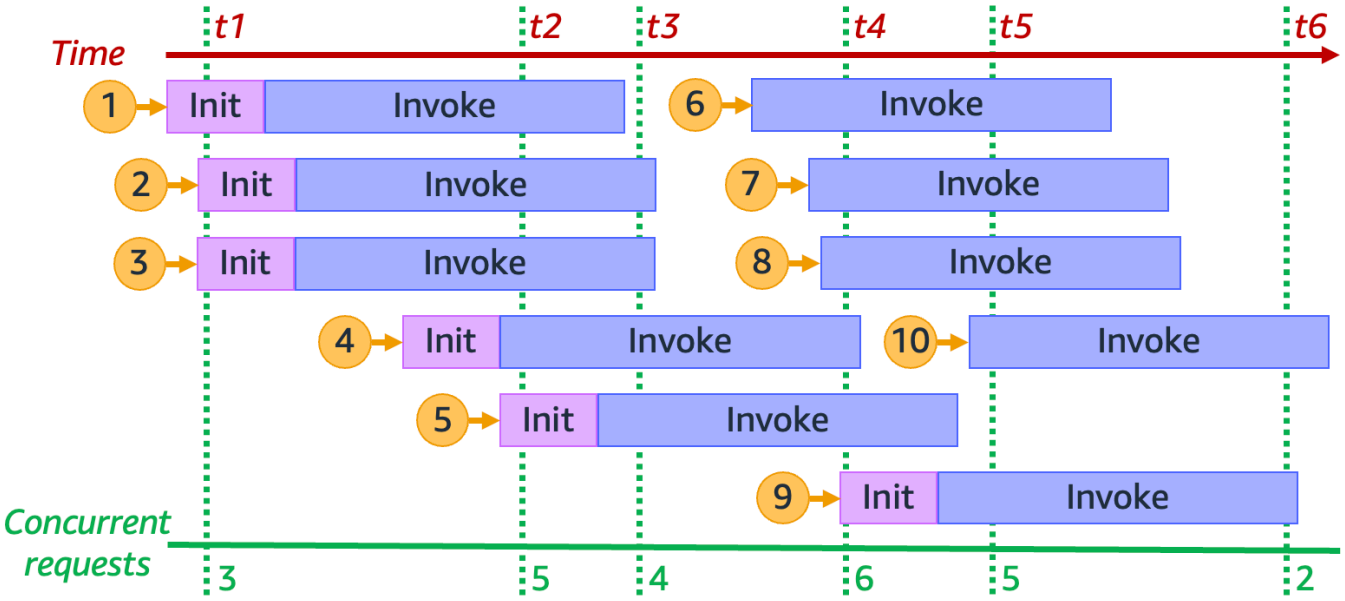
请求	Lambda 行为	Reasoning
1	预置新环境 A	这是第一个请求；没有可用的执行环境实例。
2	预置新环境 B	现有执行环境实例 A 繁忙。
3	预置新环境 C	现有执行环境实例 A 和 B 都繁忙。
4	预置新环境 D	现有执行环境实例 A、B 和 C 都繁忙。
5	预置新环境 E	现有执行环境实例 A、B、C 和 D 都繁忙。
6	重用环境 A	执行环境实例 A 已处理完请求 1，现已可用。
7	重用环境 B	执行环境实例 B 已处理完请求 2，现已可用。

请求	Lambda 行为	Reasoning
8	重用环境 C	执行环境实例 C 已处理完请求 3，现已可用。
9	预置新环境 F	现有执行环境实例 A、B、C、D 和 E 都繁忙。
10	重用环境 D	执行环境实例 D 已处理完请求 4，现已可用。

随着您的函数收到更多的并发请求，Lambda 会纵向扩展响应中的执行环境实例的数量。以下动画跟踪一段时间内的并发请求数：



通过将之前的动画冻结在六个不同的时间点，我们得到下图：



在上图中，我们可以在任何时间点绘制一条垂直线，并计算与该直线相交的环境数量。这为我们提供了该时间点的并发请求数。例如，时间 `t1` 处有三个处于活动状态的环境在处理三个并发请求。此模拟中的最大并发请求数发生在时间 `t4`，此时有六个处于活动状态的环境处理六个并发请求。

总而言之，函数并发是它同时处理并发请求的数目。为了应对函数并发的增加，Lambda 预置了更多的执行环境实例以满足请求需求。

计算函数的并发

通常，系统的并发是指同时处理多个任务的能力。在 Lambda 中，并发是您的函数同时处理的正在进行的请求数。一种衡量 Lambda 函数并发的快速而实用的方法是使用以下公式：

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

并发不同于每秒的请求数。例如，假设您的函数平均每秒接收 100 个请求。如果平均请求持续时间为一秒，那么并发确实也是 100：

```
Concurrency = (100 requests/second) * (1 second/request) = 100
```

但是，如果平均请求持续时间为 500 毫秒，则并发为 50：

```
Concurrency = (100 requests/second) * (0.5 second/request) = 50
```

实际上，并发为 50 意味着什么？ 如果平均请求持续时间为 500 毫秒，则可以将函数的实例视为每秒能够处理两个请求。然后，您的函数需要 50 个实例才能处理每秒 100 个请求的负载。并发为 50 意味着 Lambda 必须预置 50 个执行环境实例才能在没有任何节流的情况下高效处理此工作负载。以下是用方程式表示这种情况的方法：

```
Concurrency = (100 requests/second) / (2 requests/second) = 50
```

如果您的函数收到的请求数是原来的两倍（每秒 200 个请求），但只需要一半的时间来处理每个请求（250 毫秒），则并发仍为 50：

```
Concurrency = (200 requests/second) * (0.25 second/request) = 50
```

测试您对并发的理解

假设您有一个平均运行时间为 200 毫秒的函数。在峰值负载期间，可每秒观察 5000 个请求。在峰值负载期间，您的函数的并发是多少？

回答

函数的平均持续时间为 200 毫秒或 0.2 秒。使用并发公式，您可以插入数字以获取 1,000 的并发：

```
Concurrency = (5,000 requests/second) * (0.2 seconds/request) = 1,000
```

或者，函数平均持续时间为 200 毫秒意味着您的函数每秒可处理 5 个请求。要处理每秒 5000 个请求的工作负载，您需要 1000 个执行环境实例。因此，并发为 1000：

```
Concurrency = (5,000 requests/second) / (5 requests/second) = 1,000
```

了解预留并发和预置并发

默认情况下，您的账户有某个区域内的所有函数的并发上限，该上限为 1000 个并行执行。您的函数按需共享这个拥有 1000 个并发的并发池。如果您用尽了可用的并发，您的函数将节流（即开始丢弃请求）。

您的某些函数可能比其他函数更重要。因此，您可能需要配置并发设置，以确保关键函数获得所需的并发。有两种并发控制：预留并发和预置并发。

- 使用预留并发可为函数预留账户并发的某部分。如果您不想让其他函数占用所有可用的非预留并发，这非常有用。

- 使用预置并发为一个函数预先初始化多个环境实例。这对于减少冷启动延迟很有用。

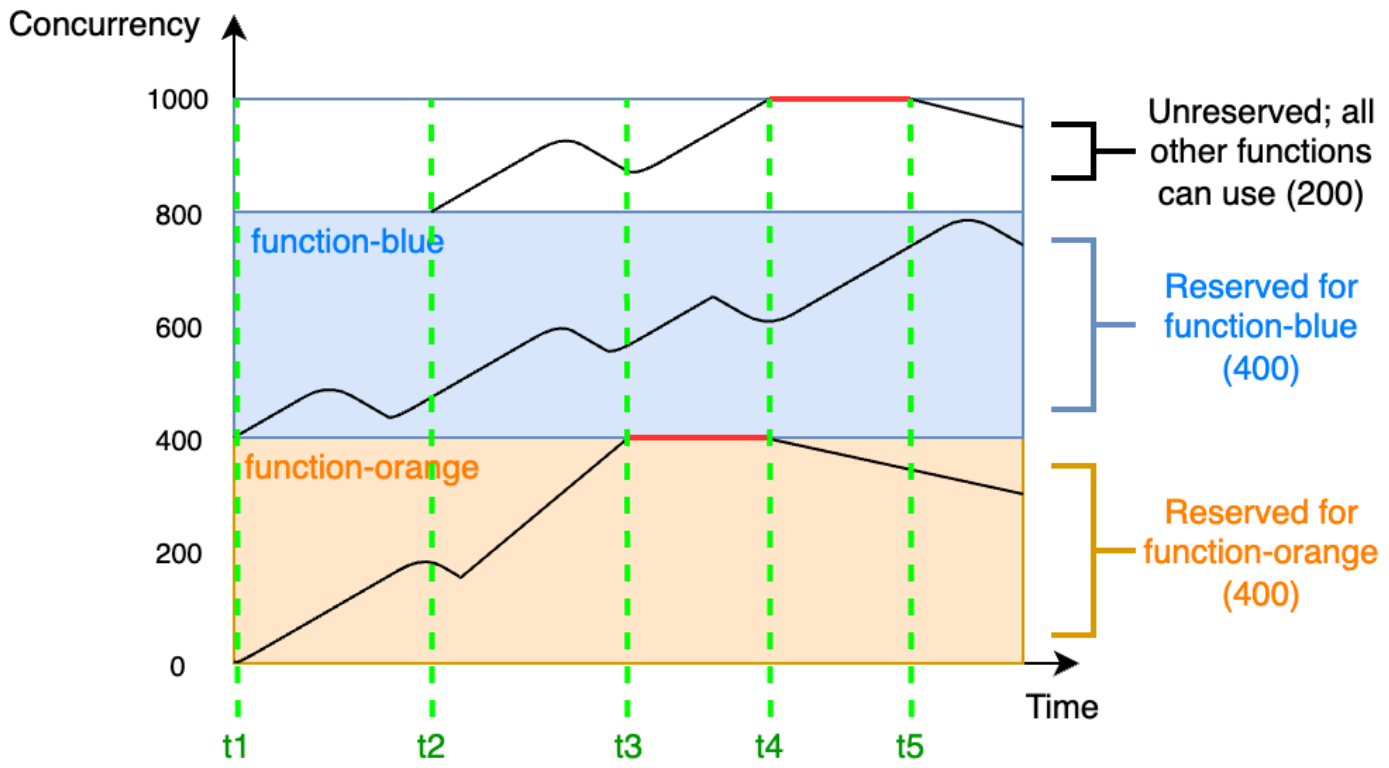
预留并发

若想保证您的函数在任何时候都有一定数量的并发可用，请使用预留并发。

预留并发是您要分配给函数的最大并发实例数。当将预留并发提供给函数时，任何其他函数都不可以使用该并发。换言之，设置预留并发会影响可用于其他函数的并发池。没有预留并发的函数共享剩余的非预留并发池。

配置预留并发将计入您的账户总并发上限。为函数配置预留并发不收取任何费用。

为了更好地理解预留并发，请细看下图：



在此图中，此区域中的所有函数的账户并发限制为默认限制 1000。假设您有两个关键函数 `function-blue` 和 `function-orange`，它们通常会预估获得很高的调用量。您决定将 400 个单位的预留并发分配给 `function-blue`，将 400 个单位的预留并发分配给 `function-orange`。在此示例中，您账户中的所有其他函数必须共享剩余的 200 个单位的非预留并发。

该图有五个兴趣点：

- 在 t1，function-orange 和 function-blue 都开始接收请求。每个函数开始用完其预留并发单位的分配部分。
- 在 t2，function-orange 和 function-blue 可稳步接收更多请求。同时，您部署了某些其他 Lambda 函数，这些函数开始接收请求。您不将预留并发分配给这些其他函数。它们开始使用剩余的 200 个单位的非预留并发。
- 在 t3，function-orange 达到最大并发 400。尽管您的账户中的其他地方有未使用的并发，但 function-orange 无法访问它。红线表示 function-orange 处于节流状态，Lambda 可能会丢弃请求。
- 在 t4，function-orange 开始接收更少的请求并且不再节流。但是，您的其他函数会遇到流量峰值并开始节流。尽管您的账户中的其他地方有未使用的并发，但其他函数无法访问它。红线表示您的其他函数处于节流状态。
- 在 t5，其他函数开始接收更少的请求并且不再节流。

在此示例中，请注意，预留并发具有以下效果：

- 您的函数可以独立于账户中的其他函数进行扩缩。在没有预留并发的同一区域中，您所有账户的函数共享非预留并发池。如果没有预留并发，其他函数可能会耗尽所有您的可用的并发。从而导致关键函数无法根据需要进行纵向扩展。
- 您的函数不能无节制地扩缩。预留并发对函数的最大并发设置了上限。这意味着您的函数不能使用为其他函数预留的并发，也不能使用非预留池中的并发。您可以预留并发以防止您的函数使用您账户中的所有可用并发，或者防止下游资源过载。
- 您可能无法使用账户的所有可用并发。预留并发计入您的账户并发上限，但这也意味着其他函数无法使用这一大部分预留并发。如果您的函数没有用完您为它预留的所有并发，那么您实际上是在浪费这个并发。除非您账户中的其他函数可以从浪费的并发中受益，否则这不是问题。

要了解如何管理函数的预留并发设置，请参阅 为函数配置预留并发。

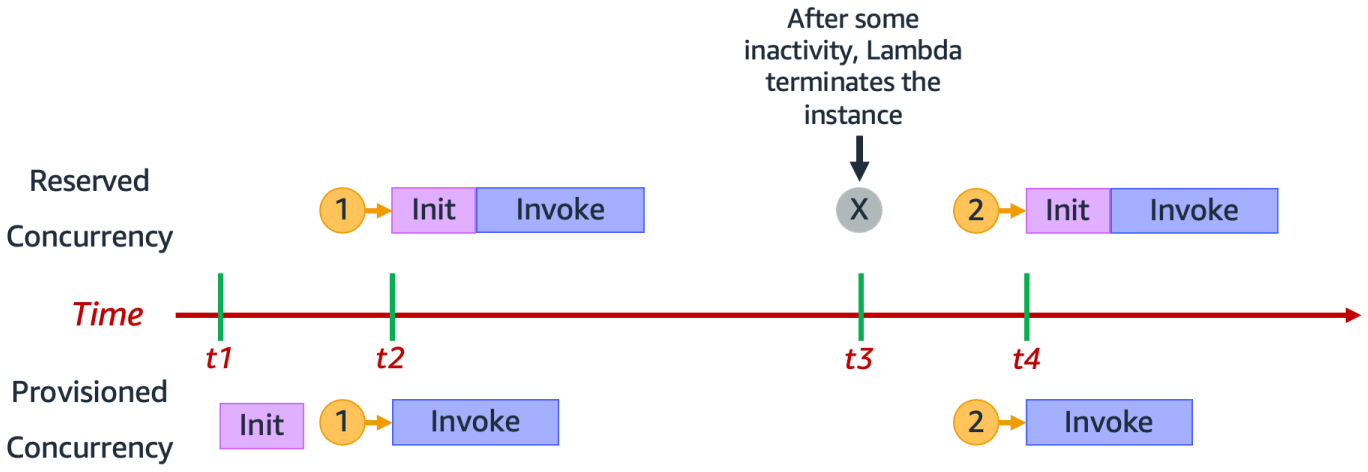
预配置并发

您可以使用预留并发来定义为 Lambda 函数预留的最大执行环境数。但是，这些环境都不会进行预先初始化。因此，您的函数调用可能需要更长的时间，因为 Lambda 必须先初始化新环境，然后才能使用它来调用您的函数。当 Lambda 必须初始化新环境才能执行调用时，这称为冷启动。为了减少冷启动，您可以使用预置并发。

预置并发是您要分配给函数的预初始化执行环境的数量。如果您在函数上设置预置并发，Lambda 会初始化该执行环境数量，以便它们准备好立即响应函数请求。

> **Note**
>
> 使用预置并发会让您的账户产生费用。如果您使用的是 Java 11 或 Java 17 运行时系统，则也可以使用 Lambda SnapStart 来缓解冷启动问题，而无需支付额外费用。SnapStart 使用执行环境的缓存快照来显著提高启动性能。您不能对同一个函数版本同时使用 SnapStart 和预置并发。有关 SnapStart 的功能、限制和支持的区域的更多信息，请参阅 使用 Lambda SnapStart 提高启动性能。

当使用预置并发时，Lambda 仍会在后台回收执行环境。但是，在任何给定时间，Lambda 始终确保预初始化环境的数量等于您的函数预置并发设置的值。这种行为与预留并发不同，在预留并发中，Lambda 可能会在不活动时段后完全终止环境。当您使用预留并发而不是预置并发来配置函数时，下图通过比较单个执行环境的生命周期来说明这一点。

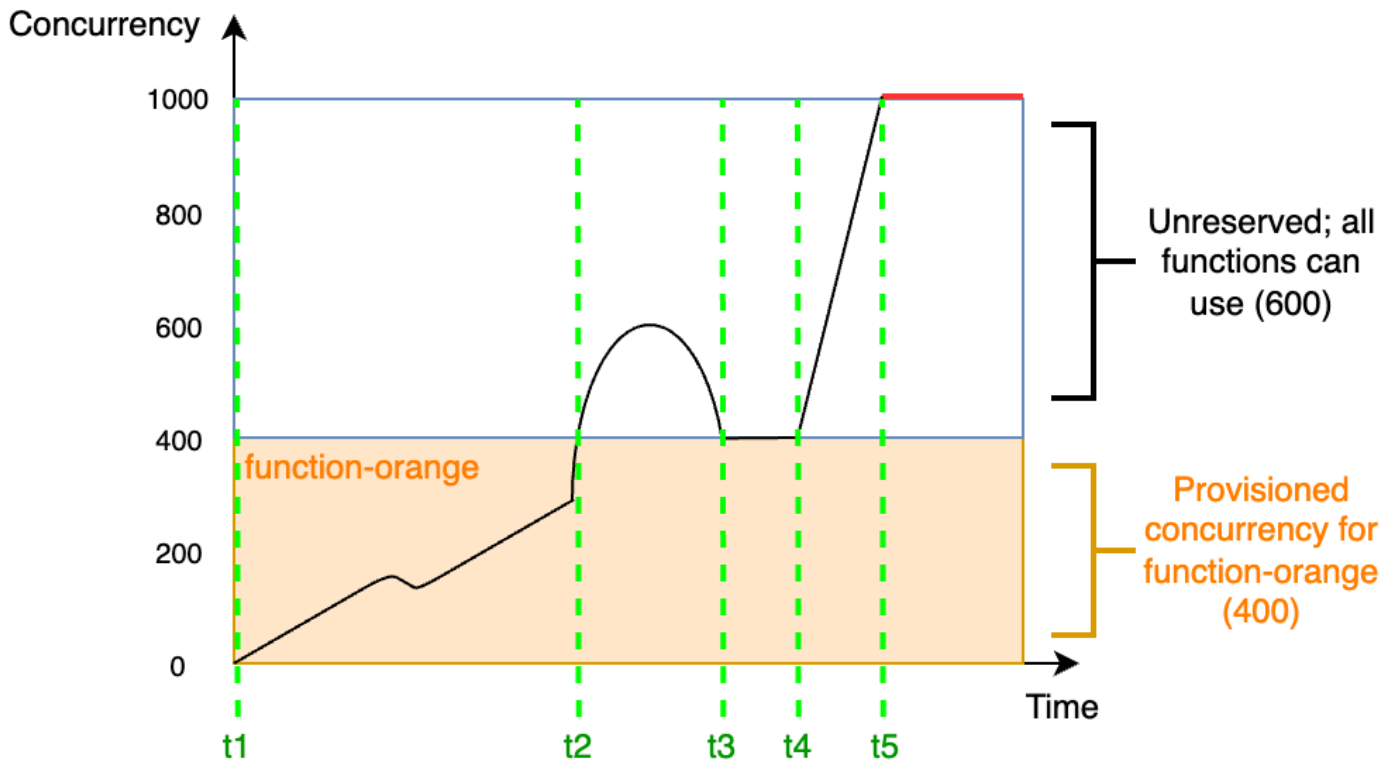


该图有四个兴趣点：

时间	预留并发	预配置并发
t1	什么都未发生。	Lambda 预初始化一个执行环境实例。
t2	请求 1 传入。Lambda 必须初始化一个新的执行环境实例。	请求 1 传入。Lambda 使用预初始化的环境实例。

时间	预留并发	预配置并发
t3	在经过一段时间的不活动状态后，Lambda 会终止处于活动状态的环境实例。	什么都未发生。
t4	请求 2 传入。Lambda 必须初始化一个新的执行环境实例。	请求 2 传入。Lambda 使用预初始化的环境实例。

为了更好地了解预置并发，请细看下图：



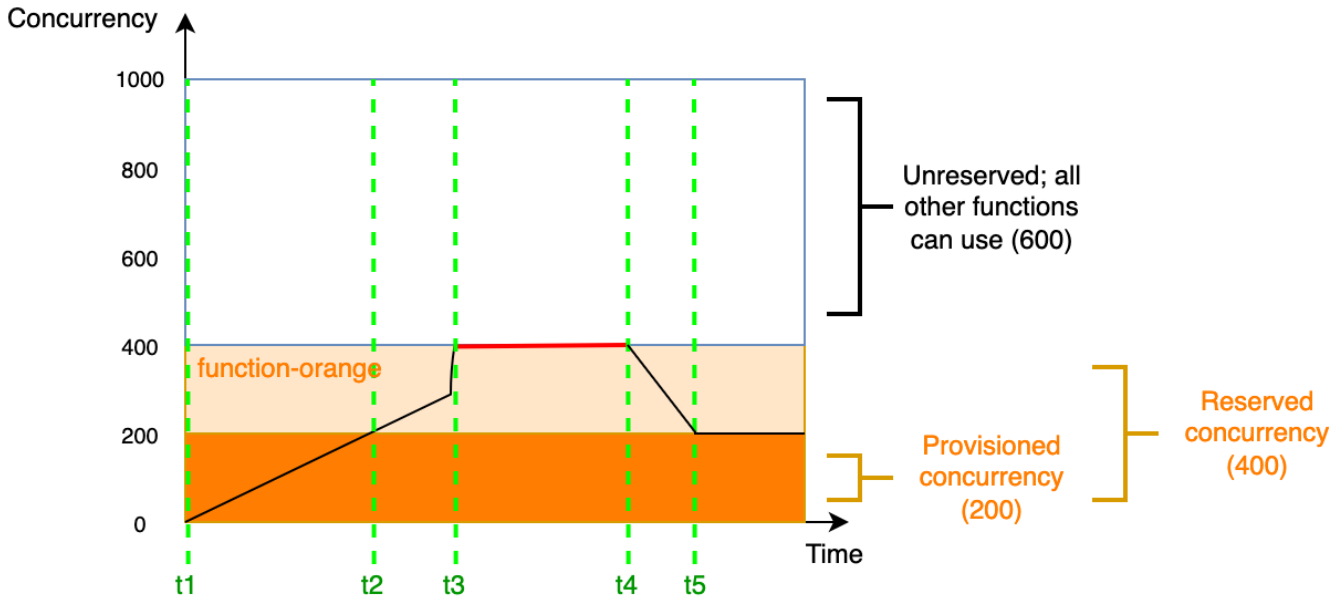
在此图中，您的账户并发限制为 1,000。您决定将 400 个单位的预置并发量分配给 `function-orange`。您账户中的所有函数，包括 `function-orange`，都可以使用剩余的 600 个单位的非预留并发。

该图有五个兴趣点：

- 在 `t1`，`function-orange` 开始接收请求。由于 Lambda 已经预先初始化 400 个执行环境实例，`function-orange` 可以立即调用。

- 在 t2，function-orange 达到 400 个并发请求。因此，function-orange 已用尽预置并发。但是，由于仍有非预留并发可用，Lambda 可以使用它来处理对 function-orange 的额外请求（没有节流）。Lambda 必须创建新实例来处理这些请求，并且您的函数可能会遇到冷启动延迟。
- 在 t3，function-orange 在短暂的流量峰值后返回 400 个并发请求。Lambda 能够再次在没有冷启动延迟的情况下处理所有请求。
- 在 t4，您账户中的函数会遇到流量突增的情况。这种突增可能产生于 function-orange 或您账户中的任何其他函数。Lambda 使用非预留并发来处理这些请求。
- 在 t5，您的账户中的函数达到 1,000 的最大并发上限，并且会节流。

前面的示例仅考虑了预置并发。实际上，您可以设置函数的预置并发和预留并发。如果您有一个函数可以处理工作日的恒定调用负载，但是在周末经常出现流量峰值，那么您可以这样做。在这种情况下，您可以使用预置并发来设置基准环境数量以在工作日处理请求，并使用预留并发来处理周末的流量峰值。请细看以下图：



在此图中，假设您为 function-orange 配置了 200 个单位的预置并发和 400 个单位的预留并发。由于您配置了预留并发，因此 function-orange 无法使用 600 个单位的非预留并发中的任何一个。

该图有五个兴趣点：

- 在 t1，function-orange 开始接收请求。由于 Lambda 已经预先初始化 200 个执行环境实例，function-orange 可以立即调用。

- 在 t2，function-orange 已用尽其所有的预置并发。function-orange 可以继续使用预留并发来处理请求，但这些请求可能会遇到冷启动延迟。
- 在 t3，function-orange 达到 400 个并发请求。因此，function-orange 已用尽其所有预留并发。由于 function-orange 无法使用非预留并发，因此请求开始节流。
- 在 t4，function-orange 开始接收更少的请求并且不再节流。
- 在 t5，function-orange 降至 200 个并发请求，因此所有请求都能够再次使用预置并发（即没有冷启动延迟）。

预留并发和预置并发均计入您的账户并发限制和区域限额。换言之，分配预留和预置并发会影响可用于其他函数的并发池。配置预置并发会让您的 AWS 账户 产生费用。

> **Note**
>
> 如果函数版本与别名功能上的预配置并发数加起来达到函数的预留并发，则所有调用都在预配置并发上运行。此配置还具有限制函数 ($LATEST) 未发布版本的效果，从而阻止其执行。为函数分配的预配置并发数不能超过预留并发数。

要管理函数的预置并发设置，请参阅 为函数配置预置并发。要根据计划或应用程序利用率自动执行预置并发扩展，请参阅 使用 Application Auto Scaling 自动执行预置并发管理。

Lambda 如何分配预置并发

配置后，预置并发并不会立即生效。Lambda 会在一两分钟的准备时间后开始分配预配置并发。无论所处的 AWS 区域 如何，Lambda 每分钟最多可为每个函数预置 6000 个执行环境。这与函数的并发扩展速率完全相同。

当您提交分配预置并发的请求时，在 Lambda 完全完成分配之前，您无法访问任何这些环境。例如，若请求 5000 个预置并发，则需等待 Lambda 完全完成对 5000 个执行环境的分配后，请求才能使用预置的并发。

对比预留并发和预置并发。

下表总结并对比了预留并发和预置并发。

主题	预留并发	预配置并发
定义	您的函数的最大执行环境实例数。	设置您的函数的预置执行环境实例数。
预置行为	Lambda 按需预置新实例。	Lambda 预置实例（即在您的函数开始接收请求之前）。
冷启动行为	由于 Lambda 必须按需创建新实例，因此可能出现冷启动延迟。	由于 Lambda 不必按需创建实例，因此不可能发生冷启动延迟。
节流行为	当达到预留并发限制时，函数会被节流。	如果未设置预留并发：当达到预置并发限制时，函数将使用非预留并发。 如果设置了预留并发：当达到预留并发限制时，函数会被节流。
如果未设置，则为默认行为	函数使用您的账户中可用的非预留并发。	Lambda 不预置任何实例。相反，如果没有设置预留并发：函数使用您的账户中可用的非预留并发。 如果设置了预留并发：函数使用预留并发。
定价	无额外费用。	会产生额外费用。

了解并发和每秒请求数

如上一节中所述，并发不同于每秒的请求数。在处理平均请求持续时间小于 100 毫秒的函数时，这是一项特别重要的区别。

对于账户中的所有函数，Lambda 强制执行每秒请求数限制，此限制相当于您的账户并发数的 10 倍。例如，由于默认账户并发限制为 1000，因此您账户中的函数每秒最多可以处理 10000 个请求。

例如，考虑一个平均请求持续时间为 50 毫秒的函数。在每秒 20000 个请求时，以下是此函数的并发：

```
Concurrency = (20,000 requests/second) * (0.05 second/request) = 1,000
```

基于此结果，您可能会认为 1000 的账户并发限制足以处理此负载。但是，由于每秒 10000 个请求的限制，您的函数每秒只能处理 20000 个请求总数中的 10000 个请求。此函数会受到节流。

教训是，在为函数配置并发设置时，必须同时考虑并发和每秒请求数。在这种情况下，您需要请求将账户并发限制增加到 2000，因为这将使每秒请求总数限制增加到 20000。

> **Note**
>
> 基于此每秒请求数限制，说每个 Lambda 执行环境每秒最多只能处理 10 个请求是不正确的。在计算配额时，Lambda 不会观察任何单个执行环境中的负载，而是仅考虑总体并发数和每秒总请求数。

测试您对并发的理解（低于 100 毫秒的函数）

假设您有一个平均运行时间为 20 毫秒的函数。在峰值负载期间，可每秒观察 30000 个请求。在峰值负载期间，您的函数的并发是多少？

回答

函数的平均持续时间为 20 毫秒或 0.02 秒。使用并发公式，您可以插入数字以获取 600 的并发数：

```
Concurrency = (30,000 requests/second) * (0.02 seconds/request) = 600
```

默认情况下，账户并发限制 1000 似乎足以处理此负载。但是，每秒 10000 个请求的限制不足以处理每秒传入 30000 个请求。要完全满足 30000 个请求，您需要请求将账户并发限制增加到 3000 或以上。

每秒请求数限制适用于 Lambda 中涉及并发的所有配额。换句话说，它适用于同步按需函数、使用预置并发的函数和并发扩展行为。例如，在以下几种情况下，您必须仔细考虑并发数和每秒请求数限制：

- 使用按需并发的函数可能会每 10 秒突增 500 个并发，或者每 10 秒突增 5000 个请求（以先发生者为准）。
- 假设您有一个函数，其预置并发分配为 10。此函数在每秒 10 个并发或 100 个请求后溢出到按需并发（以先发生者为准）。

并发限额

对于可跨区域中所有函数使用的并发的总量，Lambda 可设置限额。这些限额分为两个级别：

- 在账户级别，默认情况下，您的函数最多可以有 1000 个单位的并发。要提高此限制，请参阅 Service Quotas User Guide（《服务限额用户指南》）中的 Requesting a quota increase（请求增加限额）。
- 在函数级别，默认情况下，您可以为所有函数保留最多 900 个单位的并发。无论您的账户总并发限制如何设置，Lambda 始终为未明确保留并发的函数预留 100 个并发单位。例如，如果您将账户并发限制提高到 2000，则可以在函数级别预留最多 1900 个单位的并发。
- 在账户级别和函数级别，Lambda 还强制执行每秒请求数限制，此限制相当于相应并发配额的 10 倍。例如，这适用于账户级并发、使用按需并发的函数、使用预置并发的函数和并发扩展行为。有关更多信息，请参阅 the section called “了解并发和每秒请求数”。

要检查您的当前账户级别并发限额，请使用 AWS Command Line Interface（AWS CLI）运行以下命令：

```
aws lambda get-account-settings
```

您应该会看到类似如下输出：

```
{
    "AccountLimit": {
        "TotalCodeSize": 80530636800,
        "CodeSizeUnzipped": 262144000,
        "CodeSizeZipped": 52428800,
        "ConcurrentExecutions": 1000,
        "UnreservedConcurrentExecutions": 900
    },
    "AccountUsage": {
        "TotalCodeSize": 410759889,
        "FunctionCount": 8
    }
}
```

`ConcurrentExecutions` 是您的账户级别的总并发限额。`UnreservedConcurrentExecutions` 是您仍然可以分配给函数的预留并发。

当您的函数收到更多请求时，Lambda 会自动纵向扩展执行环境的数量来处理这些请求，直到账户达到其并发限额。但是，为了防止因突然的流量爆发而出现过度扩展，Lambda 限制了函数的扩展速度。此并发扩展速率是您账户中的函数在应对增加的请求时可以扩展的最大速率。（也就是 Lambda 创建新执行环境的速度。）并发扩展速率不同于账户级别的并发限制，后者是函数可用的并发总量。

在每个 AWS 区域 中，对于每个函数，您的并发扩展速率为每 10 秒 1000 个执行环境实例（或每 10 秒 10000 个请求）。换句话说，每 10 秒 Lambda 最多可以为每个函数分配 1000 个额外的执行环境实例，或者每秒满足 10000 个额外的请求。

通常，您无需关注此限制。Lambda 的扩展速率足以满足大多数用例的需求。

重要的是，并发扩展速率是函数级别的限制。这意味着您账户中的每个函数可以独立于其他函数进行扩展。

有关扩展行为的更多信息，请参阅 Lambda 扩展行为。

为函数配置预留并发

在 Lambda 中，并发指您的函数当前正在进行的请求数。有两种类型的并发控件可用：

- 预留并发 – 指分配给函数的最大并发实例数。当一个函数有预留并发时，任何其他函数都不可以使用该并发。对于确保最关键的函数始终具有足够的并发性来处理传入请求，预留并发非常有用。为函数配置预留并发不产生任何额外费用。
- 预置并发 – 指分配给函数的预初始化执行环境的数量。这些执行环境已准备就绪，可以立即响应传入的函数请求。预置并发对于缩短函数冷启动延迟很有用。配置预置并发会让您的 AWS 账户 产生额外费用。

本主题详细介绍了如何管理和配置预留并发。有关这两种并发控制的概念概述，请参阅预留并发和预置并发。有关配置预置并发的信息，请参阅 the section called “配置预配置并发”。

> **Note**
>
> 关联到 Amazon MQ 事件源映射的 Lambda 函数具有默认的最大并发数。对于 Apache Active MQ，最大并发实例数为 5。对于 Rabbit MQ，最大并发实例数为 1。为函数设置预留或预调配的并发不会更改这些限制。要在使用 Amazon MQ 时请求增加默认的最大并发数，请联系 AWS Support。

Sections

配置预留并发

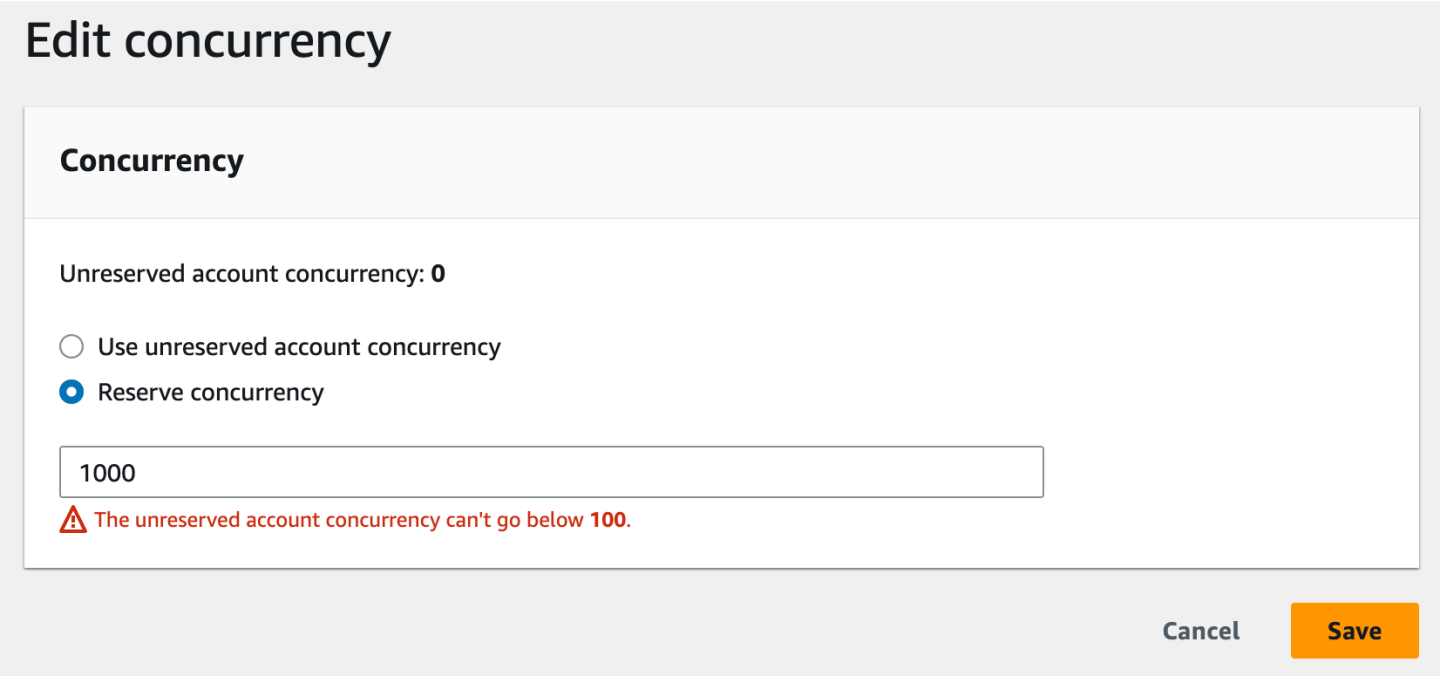
您可以使用 Lambda 控制台或 Lambda API 为函数配置预留并发设置。

为函数预留并发（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择要为其预留并发的函数。
3. 选择 Configuration（配置），然后选择 Concurrency（并发）。
4. 在并发下，选择编辑。

5. 选择预留并发。输入要为该函数预留的并发数量。
6. 选择 Save（保存）。

您最多可以预留的单位数量为非预留账户并发值减去 100。剩余 100 个单位的并发可用于不会使用预留并发的函数。例如，如果您的账户的并发上限为 1000，则不能将所有 1000 个单位的并发预留给单个函数。



为一个函数预留并发会影响可用于其他函数的并发池。例如，如果您为 `function-a` 预留 100 个单位的并发，则即使 `function-a` 不使用所有 100 个单位的预留并发，您账户中的其他函数也必须共享剩余的 900 个单位的并发。

要有意限制函数，请将其预留并发设置为 0。这将停止函数处理任何事件，直到您删除限制。

要使用 Lambda API 配置预留并发，请使用以下 API 操作。

- PutFunctionConcurrency
- GetFunctionConcurrency
- DeleteFunctionConcurrency

例如，要使用 AWS Command Line Interface（CLI）配置预留并发，请使用 `put-function-concurrency` 命令。以下命令为名为 `my-function` 的函数预留 100 个单位的并发：

```
aws lambda put-function-concurrency --function-name my-function \
```

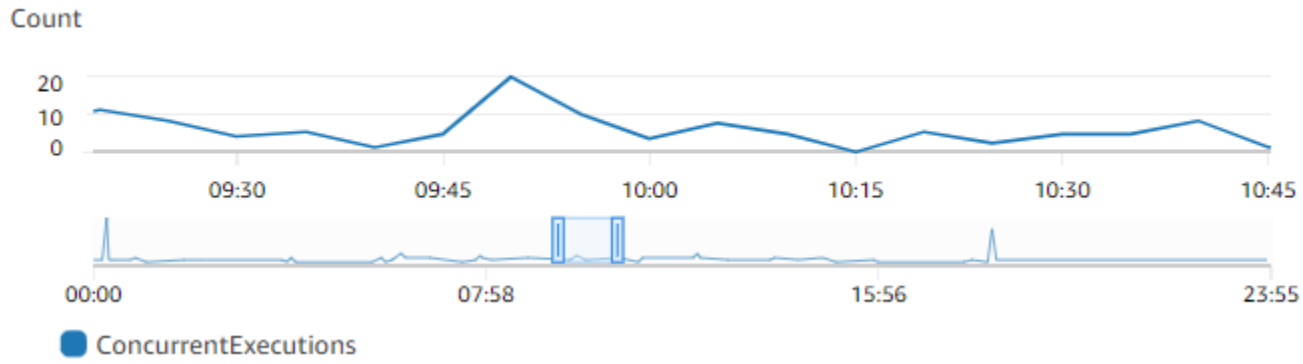
```
    --reserved-concurrent-executions 100
```

您应该会看到类似如下输出：

```
{
    "ReservedConcurrentExecutions": 100
}
```

准确估计函数所需的预留并发

如果您的函数当前正在提供流量，则可以使用 CloudWatch 指标轻松查看其并发指标。具体而言，ConcurrentExecutions 指标显示了您账户中每个函数的并发调用数。



前面的图表显示，在给定的任何时间，此函数平均处理 5 到 10 个并发请求，峰值通常为 20 个请求。假设您的账户中还有许多其他函数。如果此函数对您的应用程序至关重要，并且您不想丢弃任何请求，请使用大于或等于 20 的数字作为预留的并发设置。

请注意，您也可以使用以下公式计算并发：

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

将每秒的平均请求数与平均请求持续时间（以秒为单位）相乘可以粗略估计您需要预留多少并发。您可以使用 Invocation 指标估算每秒的平均请求数，并使用 Duration 指标估计平均请求持续时间（以秒为单位）。有关更多信息，请参阅 查看 Lambda 函数的指标。

您还应该熟悉上游和下游吞吐量限制。虽然 Lambda 函数可随负载无缝扩展，但上游和下游依赖项可能不具有相同的吞吐量。如果您需要限制函数可以扩展的幅度，可以为函数配置预留并发。

为函数配置预置并发

在 Lambda 中，并发指您的函数当前正在进行的请求数。有两种类型的并发控件可用：

- 预留并发 – 指分配给函数的最大并发实例数。当一个函数有预留并发时，任何其他函数都不可以使用该并发。对于确保最关键的函数始终具有足够的并发性来处理传入请求，预留并发非常有用。为函数配置预留并发不产生任何额外费用。
- 预置并发 – 指分配给函数的预初始化执行环境的数量。这些执行环境已准备就绪，可以立即响应传入的函数请求。预置并发对于缩短函数冷启动延迟很有用。配置预置并发会让您的 AWS 账户 产生额外费用。

本主题详细介绍了如何管理和配置预置并发。有关这两种并发控制的概念概述，请参阅预留并发和预置并发。有关配置预留并发的更多信息，请参阅the section called “配置预留并发”。

> **Note**
>
> 关联到 Amazon MQ 事件源映射的 Lambda 函数具有默认的最大并发数。对于 Apache Active MQ，最大并发实例数为 5。对于 Rabbit MQ，最大并发实例数为 1。为函数设置预留或预调配的并发不会更改这些限制。要在使用 Amazon MQ 时请求增加默认的最大并发数，请联系 AWS Support。

Sections

配置预配置并发

您可以使用 Lambda 控制台或 Lambda API 为函数配置预置并发设置。

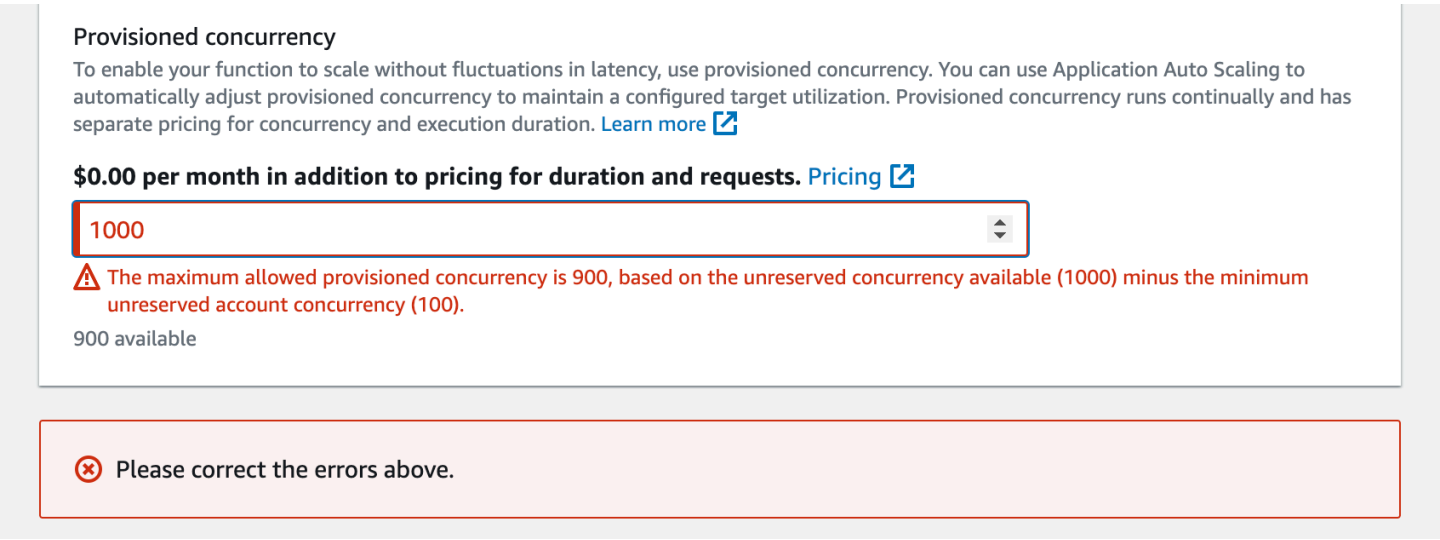
要为函数分配预置并发（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择要为其分配预置并发的函数。
3. 选择 Configuration（配置），然后选择 Concurrency（并发）。
4. 在 Provisioned concurrency configurations（预配置并发配置）下，选择 Add configuration（添加配置）。
5. 选择限定符类型以及别名或版本。

> **Note**
>
> 您不能将预置并发与任何函数的 $LATEST 版本一起使用。
> 如果函数具有事件源，请确保该事件源指向正确的函数别名或版本。否则，您的函数将不会使用预置并发环境。

6. 在预置并发下输入一个数字。Lambda 会提供每月成本的估算值。
7. 选择保存。

您最多可以在账户中配置的单位数量为非预留账户并发减去 100。剩余 100 个单位的并发可用于不会使用预留并发的函数。例如，如果您的账户的并发限制为 1000，并且您没有为任何其他函数分配任何预留或预置并发，则可以为单个函数配置最多 900 个单位的预置并发。



为函数配置预置并发会影响可用于其他函数的并发池。例如，如果您为 `function-a` 配置 100 个单位的预置并发，则您账户中的其他函数必须共享剩余的 900 个单位的并发。即使 `function-a` 不使用所有 100 个单位的预置并发也是如此。

可以为同一函数同时分配预留并发和预置并发。在这种情况下，预置并发数不能超过预留并发数。

此限制也适用于函数版本。您可以分配给特定函数版本的最大预置并发数等于该函数的预留并发减去其他函数版本上配置的预置并发。

要使用 Lambda API 配置预置并发，请使用以下 API 操作：

- PutProvisionedConcurrencyConfig
- GetProvisionedConcurrencyConfig
- ListProvisionedConcurrencyConfigs
- DeleteProvisionedConcurrencyConfig

例如，要使用 AWS Command Line Interface（CLI）配置预置并发，请使用 `put-provisioned-concurrency-config` 命令。以下命令将为名为 `my-function` 的 BLUE 别名分配 100 个单位的预置并发：

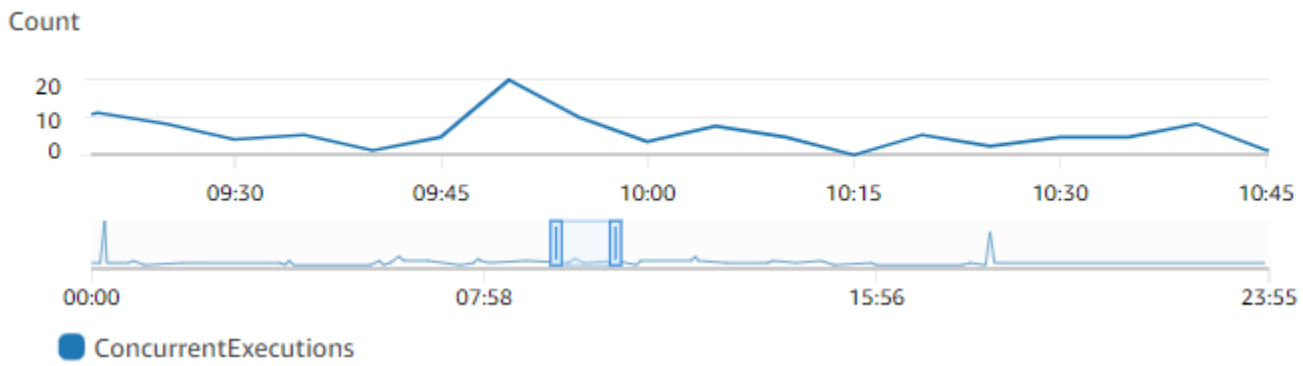
```
aws lambda put-provisioned-concurrency-config --function-name my-function \
  --qualifier BLUE \
  --provisioned-concurrent-executions 100
```

您应该会看到类似如下输出：

```
{
  "Requested ProvisionedConcurrentExecutions": 100,
  "Allocated ProvisionedConcurrentExecutions": 0,
  "Status": "IN_PROGRESS",
  "LastModified": "2023-01-21T11:30:00+0000"
}
```

准确估计函数所需的预置并发

您可以使用 CloudWatch 指标查看任何活动函数的并发指标。具体而言，`ConcurrentExecutions` 指标显示了您账户中函数的并发调用数。



前面的图表显示，在给定的任何时间，此函数平均处理 5 到 10 个并发请求，峰值为 20 个请求。假设您的账户中还有许多其他函数。如果此函数对您的应用程序至关重要，并且您的每次调用都需要低延迟响应，请配置至少 20 个单位的预置并发。

请注意，您也可以使用以下公式计算并发数：

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

要估计您需要的并发数，请将每秒的平均请求数与平均请求持续时间（以秒为单位）相乘。您可以使用 `Invocation` 指标估算每秒的平均请求数，并使用 `Duration` 指标估计平均请求持续时间（以秒为单位）。

对于配置预置并发，Lambda 建议在函数通常需要的并发的基础上添加 10% 的浮动。例如，如果函数通常在出现 200 个并发请求时达到峰值，请将预置并发设置为 220（200 个并发请求 + 10% = 220 个预置并发）。

使用预置并发时优化函数代码

如果您使用预置并发，则请考虑重构函数代码以优化实现低延迟。对于使用预置并发的函数，Lambda 会在分配期间运行任何初始化代码（例如加载库和实例化客户端）。因此，建议将尽可能多的初始化放在主函数处理程序之外，以避免影响实际函数调用期间的延迟。相比之下，如果您在主处理程序代码中初始化库或实例化客户端，则意味着您的函数必须在每次调用时运行该代码（无论您是否使用预置并发，都会发生这种情况）。

对于按需调用，每次函数出现冷启动时，Lambda 可能需要重新运行您的初始化代码。对于此类函数，您可以选择将特定功能的初始化推迟到函数需要该功能的时候。例如，请为 Lambda 处理程序考虑以下控制流：

```
def handler(event, context):
    ...
```

```
    if ( some_condition ):
        // Initialize CLIENT_A to perform a task
    else:
        // Do nothing
```

在前面的示例中，开发人员没有在主处理程序之外初始化 CLIENT_A，而是在 if 语句中对其进行初始化。这样一来，Lambda 只有在 some_condition 得到满足时才会运行此代码。如果您在主处理程序之外初始化 CLIENT_A，Lambda 会在每次冷启动时运行该代码。这可能会增加总体延迟。

使用环境变量查看和控制预置并发行为

函数可能会用尽其所有预置并发。Lambda 使用按需实例来处理任何过多流量。为了确定 Lambda 将哪种类型的初始化用于特定环境，请检查 AWS_LAMBDA_INITIALIZATION_TYPE 环境变量的值。此变量可能有两个值：provisioned-concurrency 和 on-demand。AWS_LAMBDA_INITIALIZATION_TYPE 的值是不可变的，并且在环境的整个生命周期中保持不变。要查看函数代码中环境变量的值，请参阅 ???。

如果使用的是 .NET 6 或 .NET 7 运行时系统，则可以配置 AWS_LAMBDA_DOTNET_PREJIT 环境变量以改善函数的延迟，即使函数不使用预置并发。.NET 运行时系统会延时编译和初始化代码首次调用的每个库。因此，首次调用 Lambda 函数的时间可能比后续调用的时间更长。要缓解此问题，可以为 AWS_LAMBDA_DOTNET_PREJIT 从以下三个值中任选一个：

- ProvisionedConcurrency：Lambda 会使用预置并发为所有环境执行提前 JIT 编译。这是默认值。
- Always：Lambda 会为每个环境执行提前 JIT 编译，即使函数不使用预置并发也是如此。
- Never：Lambda 会为所有环境禁用提前 JIT 编译。

了解使用预置并发的日志记录和计费行为

对于预置并发环境，函数的初始化代码在分配期间定期运行一次，因为 Lambda 会回收环境实例。在环境实例处理请求后，您可以在日志和跟踪中查看初始化时间。需要注意的是，即使环境实例从不处理请求，Lambda 也会对初始化进行计费。预配置并发连续运行，并且与初始化和调用成本分开计费。有关更多详细信息，请参阅 AWS Lambda 定价。

此外，当您配置具有预调配并发的 Lambda 函数时，Lambda 会预先初始化该执行环境，使其在函数调用请求之前可用。但是，只有在实际调用函数时，您的函数才会将调用日志发布到 CloudWatch。因此，初始化持续时间字段会出现在第一次函数调用的 REPORT 日志行中，即使初始化是提前进行的。这并不意味着该函数经历了冷启动。

使用 Application Auto Scaling 自动执行预置并发管理

您可以使用 Application Auto Scaling 根据计划或基于利用率管理预置并发。如果函数的流量模式是可预测的，请使用计划扩展。如果您希望函数保持特定利用率，请使用目标跟踪扩展策略。

计划扩展

借助 Application Auto Scaling，您可以按照可预测的负载变化来设置自己的扩展计划。有关更多信息和示例，请参阅《Application Auto Scaling 用户指南》中的 Application Auto Scaling 的计划扩展，以及在 AWS 计算博客上的为经常出现的峰值使用情况计划 AWS Lambda 预置并发。

目标跟踪

借助目标跟踪，Application Auto Scaling 可根据您定义扩展策略的方式创建和管理一组 CloudWatch 警报。当这些警报激活时，Application Auto Scaling 会使用预置并发自动调整分配的环境数量。针对流量模式不可预测的应用程序，使用目标跟踪。

要使用目标跟踪扩展预置并发，请使用 `RegisterScalableTarget` 和 `PutScalingPolicy` Application Auto Scaling API 操作。例如，如果您使用的是 AWS Command Line Interface（CLI），请按照以下步骤操作：

1. 将函数的别名注册为扩展目标。以下示例注册名为 `my-function` 的函数的 BLUE 别名：

```
aws application-autoscaling register-scalable-target --service-namespace lambda \
    --resource-id function:my-function:BLUE --min-capacity 1 --max-capacity 100 \
    --scalable-dimension lambda:function:ProvisionedConcurrency
```

2. 将扩展策略应用于目标。以下示例配置 Application Auto Scaling，用于调节别名的预置并发配置，让利用率接近 70%，但也可以应用 10% 到 90% 之间的任意值。

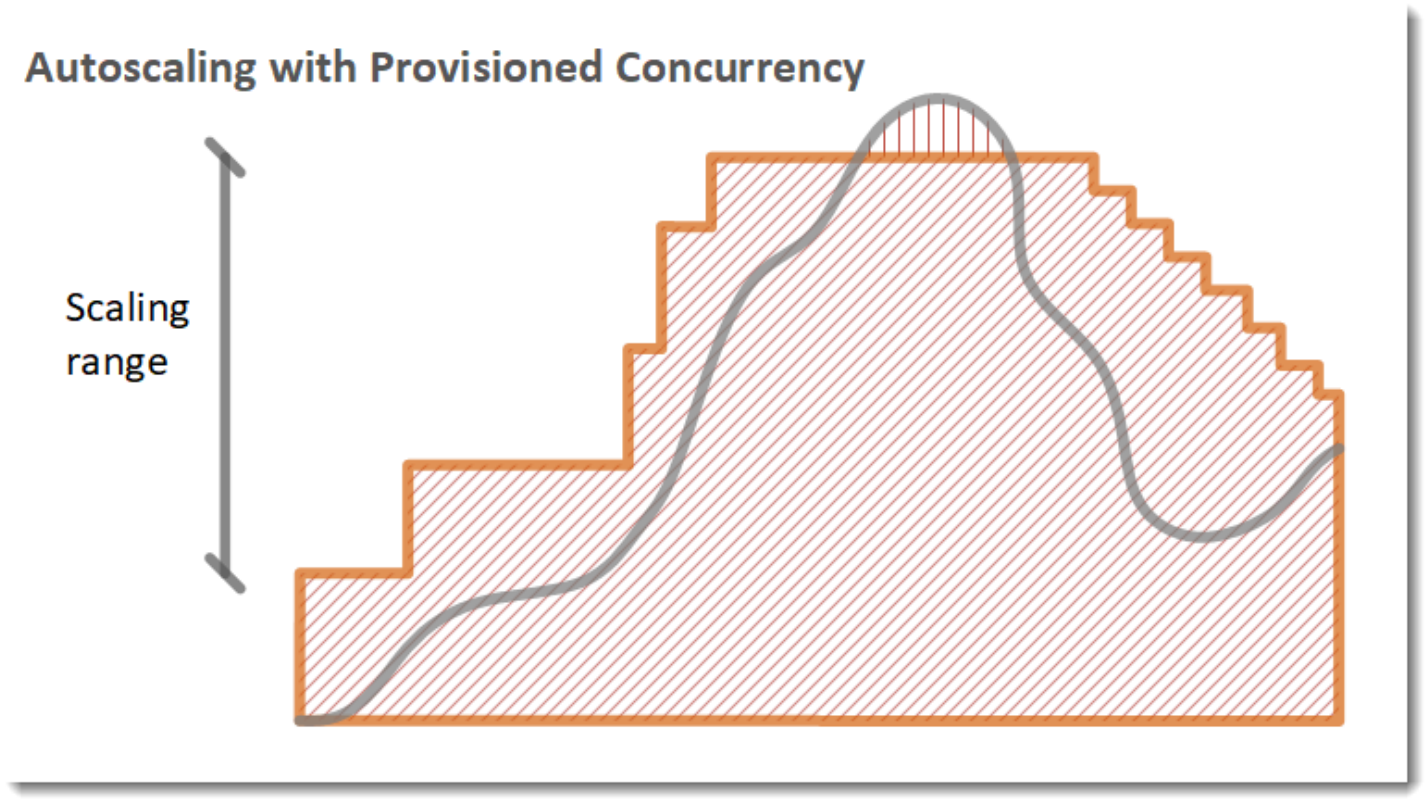
```
aws application-autoscaling put-scaling-policy \
    --service-namespace lambda \
    --scalable-dimension lambda:function:ProvisionedConcurrency \
    --resource-id function:my-function:BLUE \
    --policy-name my-policy \
    --policy-type TargetTrackingScaling \
    --target-tracking-scaling-policy-configuration '{ "TargetValue":
 0.7, "PredefinedMetricSpecification": { "PredefinedMetricType":
 "LambdaProvisionedConcurrencyUtilization" }}'
```

应看到类似如下内容的输出：

```
{
    "PolicyARN": "arn:aws:autoscaling:us-
east-2:123456789012:scalingPolicy:12266dbb-1524-xmpl-a64e-9a0a34b996fa:resource/lambda/
function:my-function:BLUE:policyName/my-policy",
    "Alarms": [
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7"
        },
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66"
        }
    ]
}
```

Application Auto Scaling 会在 CloudWatch 中创建两个警报。当预置并发的利用率持续超过 70% 时，第一个警报会触发。发生这种情况时，Application Auto Scaling 会分配更多的预配置并发以降低利用率。当利用率持续低于 63%（70% 目标的 90%）时，第二个警报会触发。发生这种情况时，Application Auto Scaling 会减少别名的预配置并发。

在以下示例中，函数会根据利用率在最小和最大预置并发量之间进行扩缩。



图例

- 函数实例
- 打开请求
- 预配置并发
- 标准并发

打开的请求数量增加时，Application Auto Scaling 会大幅度增加预置并发，直到达到配置的最大值。达到最大值后，如果账户尚未达到账户并发数限制，则函数可能会在标准的非预留并发上继续扩展。利用率下降并保持较低时，Application Auto Scaling 会以较短周期逐步减少预置并发。

默认情况下，Application Auto Scaling 的两个警报都使用平均统计数据。具有快速突发流量的函数可能不会触发这些警报。例如，假设 Lambda 函数执行速度很快（如 20-100 毫秒），并且您的流量出现快速突发。在这种情况下，请求数超过了在突发期间分配的预置并发数。但是，突发负载需要维持

至少 3 分钟，Application Auto Scaling 才能预置更多环境。此外，两个 CloudWatch 警报都需要获得 3 个达到目标平均水平的数据点，然后才能激活自动扩缩策略。如果函数遭遇快速的流量爆发，使用 Maximum 统计数据而非 Average 统计数据可以更有效地扩展预置并发数，从而最大限度地减少冷启动。

有关目标跟踪扩展策略的更多信息，请参阅适用于 Application Auto Scaling 的目标跟踪扩展策略。

Lambda 扩展行为

当您的函数收到更多请求时，Lambda 会自动纵向扩展执行环境的数量来处理这些请求，直到账户达到其并发限额。但是，为了防止因突然的流量爆发而出现过度扩展，Lambda 限制了函数的扩展速度。此并发扩展速率是您账户中的函数在应对增加的请求时可以扩展的最大速率。（也就是 Lambda 创建新执行环境的速度。）并发扩展速率不同于账户级别的并发限制，后者是函数可用的并发总量。

并发扩展速率

在每个 AWS 区域 中，对于每个函数，您的并发扩展速率为每 10 秒 1000 个执行环境实例（或每 10 秒 10000 个请求）。换句话说，每 10 秒 Lambda 最多可以为每个函数分配 1000 个额外的执行环境实例，或者每秒满足 10000 个额外的请求。

通常，您无需关注此限制。Lambda 的扩展速率足以满足大多数用例的需求。

重要的是，并发扩展速率是函数级别的限制。这意味着您账户中的每个函数可以独立于其他函数进行扩展。

> **Note**
>
> 实际上，Lambda 会尽最大努力随着时间的推移持续重新填充您的并发扩展速率，而不是每隔 10 秒一次填充 1,000 个单位。

Lambda 不会累积并发扩展速率中未使用的部分。这意味着，无论在任何时候，扩展速率的最大值始终为 1,000 个并发单位。例如，如果您在 10 秒间隔内没有使用可用的 1,000 个并发单元中的任何一个，则在下一个 10 秒间隔内，不会累加 1,000 个额外的单位。在下一个 10 秒间隔内，您的并发扩展速率仍为 1,000。

只要您的函数继续收到越来越多的请求，Lambda 就会以您可用的最快速率进行扩展，直至达到账户的并发限制。您可以通过配置预留并发来限制单个函数可以使用的并发数量。如果请求进入的速度超过函数可扩展的速度，或者如果函数处于最大并发，则其他请求会因节流错误而失败（状态代码为 429）。

监控并发

Lambda 会发送 Amazon CloudWatch 指标，以帮助监控函数的并发。本主题解释了这些指标以及如何解读指标。

Sections

一般并发指标

使用以下指标监控 Lambda 函数并发。各指标的粒度为 1 分钟。

- `ConcurrentExecutions` – 给定时间点处于活动状态的并发调用数。Lambda 会针对所有函数、别名和版本发送该指标。对于 Lambda 控制台中的任何函数，Lambda 会在指标下的监控选项卡中原生显示 `ConcurrentExecutions` 的图表。使用 MAX 查看该指标。
- `UnreservedConcurrentExecutions` – 使用非预留并发的处于活动状态的并发调用数。Lambda 会向区域中所有函数发出该指标。使用 MAX 查看该指标。
- `ClaimedAccountConcurrency` – 不可用于按需调用的并发数。`ClaimedAccountConcurrency` 等于 `UnreservedConcurrentExecutions` 加上分配的并发数（即总预留并发数加上总预置并发数）。如果 `ClaimedAccountConcurrency` 超出账户并发限制，则可以申请更高的账户并发数限制。使用 MAX 查看该指标。有关更多信息，请参阅 使用 `ClaimedAccountConcurrency` 指标。

预配置并发指标

使用预置并发时，使用以下指标监控 Lambda 函数。各指标的粒度为 1 分钟。

- `ProvisionedConcurrentExecutions` – 正在积极处理预置并发调用的执行环境实例的数目。Lambda 会根据配置的预置并发，为每个函数版本和别名发送该指标。使用 MAX 查看该指标。

`ProvisionedConcurrentExecutions` 与您分配的预置并发总数不同。例如，假设您为某个函数版本分配了 100 个单位的预置并发。在任何给定的一分钟内，如果这 100 个执行环境中最多有 50 个会同时处理调用，则 MAX (`ProvisionedConcurrentExecutions`) 的值为 50。

- ProvisionedConcurrencyInvocations – Lambda 使用预置并发调用函数代码的次数。Lambda 会根据配置的预置并发，为每个函数版本和别名发送该指标。使用 SUM 查看该指标。

ProvisionedConcurrencyInvocations 与 ProvisionedConcurrentExecutions 的不同之处在于，ProvisionedConcurrencyInvocations 会计算调用总次数，而 ProvisionedConcurrentExecutions 会计算处于活动状态的环境的数量。要理解这种区别，请考虑以下情形：

在本例中，假设您每分钟收到 1 次调用，并且每次调用需要 2 分钟才能完成。每个橙色水平条形代表一个请求。假设您为该函数分配 10 个单位的预置并发，此时每个请求都会在预置并发上运行。

在 0 到 1 分钟之间，Request 1 会传入。在第 1 分钟，MAX (ProvisionedConcurrentExecutions) 的值为 1，因为在过去一分钟内，至多 1 个执行环境处于活动状态。SUM (ProvisionedConcurrencyInvocations) 的值也为 1，因为在过去一分钟内传入了 1 个新请求。

在第 1 分钟到第 2 分钟之间，Request 2 会传入，并且 Request 1 会继续运行。在第 2 分钟，MAX (ProvisionedConcurrentExecutions) 的值为 2，因为在过去一分钟内至多 2 个执行环境处于活动状态。但是，SUM (ProvisionedConcurrencyInvocations) 的值为 1，因为在过去一分钟内只传入了 1 个新请求。这种指标行为一直持续到示例结束。

- ProvisionedConcurrencySpilloverInvocations – 当所有预置并发均处于使用状态时，Lambda 根据标准（预留或非预留）并发调用函数的次数。Lambda 会根据配置的预置并发，为每个函数版本和别名发送该指标。使用 SUM 查看该指标。ProvisionedConcurrencyInvocations +

ProvisionedConcurrencySpilloverInvocations 的值应等于函数调用的总次数（即 Invocations 指标）。

ProvisionedConcurrencyUtilization – 正在使用的预置并发的百分比（即 ProvisionedConcurrentExecutions 除以分配的预置并发总量的值）。Lambda 会根据配置的预置并发，为每个函数版本和别名发送该指标。使用 MAX 查看该指标。

例如，假设您为某个函数版本预置了 100 个单位的预置并发。在任何给定的一分钟内，如果这 100 个执行环境中最多有 60 个同时处理调用，则 MAX (ProvisionedConcurrentExecutions) 的值为 60，MAX (ProvisionedConcurrencyUtilization) 的值为 0.6。

ProvisionedConcurrencySpilloverInvocations 的值较高可能表示您需要为函数分配限额外的预置并发。或者，您可以配置 Application Auto Scaling 以根据预定义的阈值处理预置并发的自动扩缩。

相反，ProvisionedConcurrencyUtilization 的值持续较低可能表明您为函数分配了过多的预置并发。

使用 ClaimedAccountConcurrency 指标

Lambda 使用 ClaimedAccountConcurrency 指标来确定您的账户可用于按需调用的并发数。Lambda 将使用以下公式计算 ClaimedAccountConcurrency：

```
ClaimedAccountConcurrency = UnreservedConcurrentExecutions + (allocated concurrency)
```

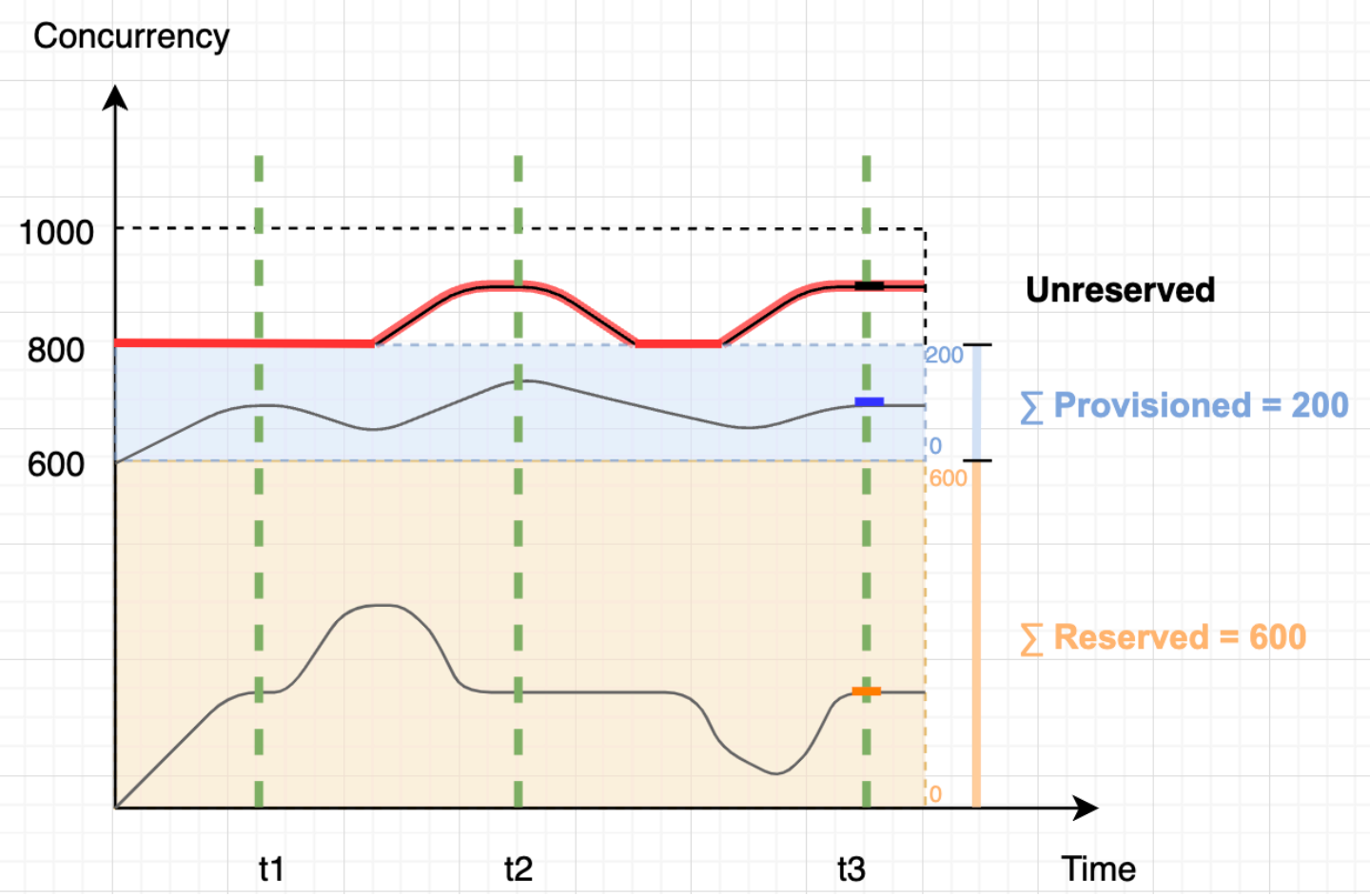
UnreservedConcurrentExecutions 指使用非预留并发的处于活动状态的并发调用数。分配的并发是以下两部分的总和（将 RC 替换为“预留并发”，PC 替换为“预置并发”）：

- 区域中所有函数的总 RC。
- 一个区域中所有使用 PC 的函数的总 PC，不包括使用 RC 的函数。

> **Note**
>
> 为一个函数分配的 PC 不能超过 RC。因此，函数的 RC 总是大于或等于其 PC。为了计算同时使用 PC 和 RC 的此类函数对分配并发的贡献，Lambda 只考虑两者中最大的 RC。

Lambda 使用 ClaimedAccountConcurrency 指标，而不是 ConcurrentExecutions 指标，来确定可用于按需调用的并发数。虽然 ConcurrentExecutions 指标非常适用于跟踪活动并发调用数，但其并不总是能反映真实的并发可用性。这是因为 Lambda 还会考虑预留并发和预置并发来确定可用性。

要说明 ClaimedAccountConcurrency，假设在一个场景中，您在基本上未使用的函数中配置了大量预留并发和预置并发。在以下示例中，假设您的账户并发数限制为 1000，并且账户中有两个主要函数：function-orange 和 function-blue。您为 function-orange 分配了 600 个单位的预留并发。您为 function-blue 分配了 200 个单位的预置并发。假设随着时间的推移，您会部署其他函数并观测到以下流量模式：



在上图中，黑线表示一段时间内的实际并发使用情况，红线表示一段时间内的 ClaimedAccountConcurrency 值。尽管各函数的实际并发利用率较低，但在整个场景中，ClaimedAccountConcurrency 至少为 800。这是因为您总共为 function-orange 和 function-blue 分配了 800 个单位的并发。从 Lambda 的角度来看，您已经“申请”了此并发以供使用，因此其他函数实际上只剩下 200 个单位的并发。

该场景中，ClaimedAccountConcurrency 公式中分配的并发数为 800。然后我们可以推导出图中不同点处的 ClaimedAccountConcurrency 值：

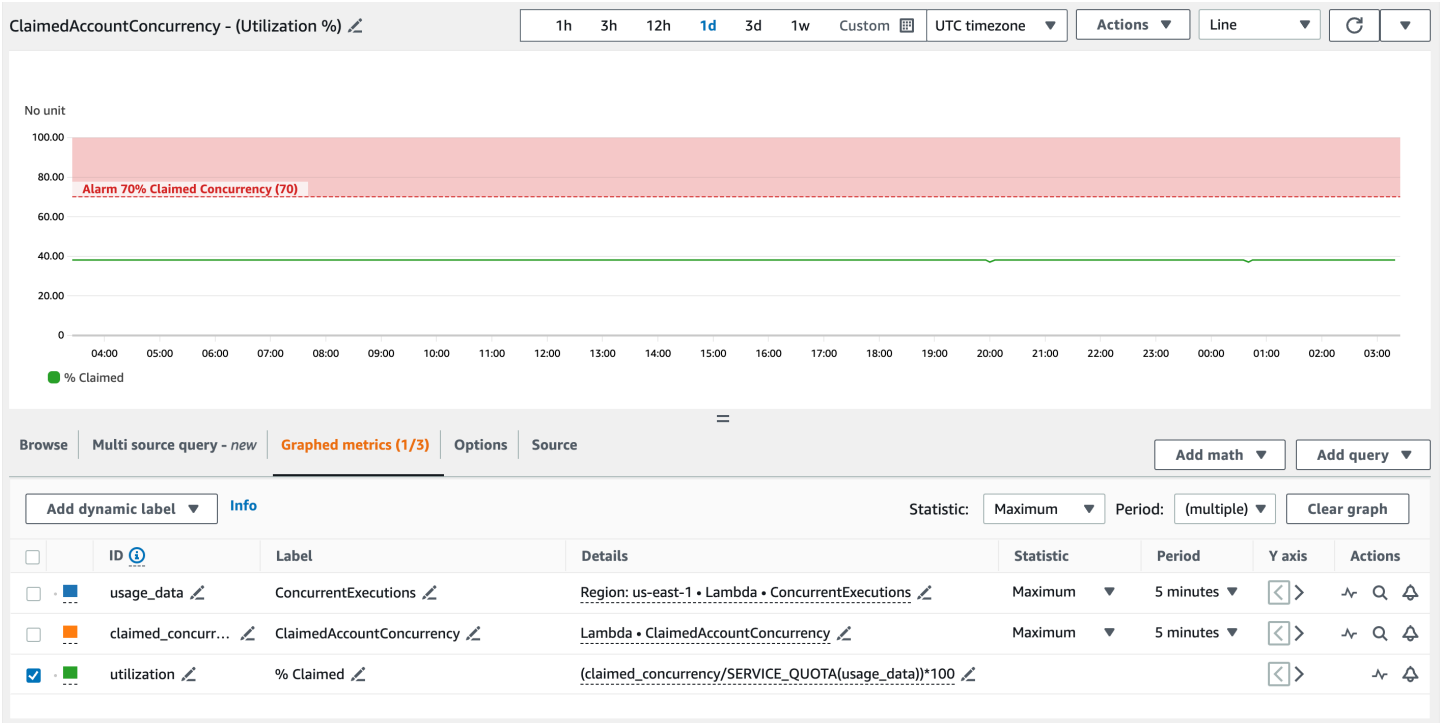
- t1 点处，ClaimedAccountConcurrency 等于 800 (800 + 0 UnreservedConcurrentExecutions)。
- t2 点处，ClaimedAccountConcurrency 等于 900 (800 + 100 UnreservedConcurrentExecutions)。
- t3 点处，ClaimedAccountConcurrency 又等于 900 (800 + 100 UnreservedConcurrentExecutions)。

在 CloudWatch 中设置 **ClaimedAccountConcurrency** 指标

Lambda 在 CloudWatch 中发出 ClaimedAccountConcurrency 指标。使用此指标以及 SERVICE_QUOTA(ConcurrentExecutions) 值可以获得账户中并发利用率的百分比，如以下公式所示：

```
Utilization = (ClaimedAccountConcurrency/SERVICE_QUOTA(ConcurrentExecutions)) * 100%
```

以下屏幕截图说明了如何在 CloudWatch 中绘制此公式的图表。绿 claim_utilization 线表示此账户中的并发利用率，约为 40%：



上一个屏幕截图还包括 CloudWatch 警报，当并发利用率超过 70% 时，该警报会进入 ALARM 状态。您可以使用 ClaimedAccountConcurrency 指标以及类似警报来主动确定何时可能需要请求更高的账户并发数限制。

使用 Node.js 构建 Lambda 函数

您可以使用 Node.js 在 AWS Lambda 中运行 JavaScript 代码。Lambda 可为 Node.js 提供运行时，用于运行代码来处理事件。您的代码在包含 AWS SDK for JavaScript 的环境中运行，其中包含来自您管理的 AWS Identity and Access Management (IAM) 角色的凭证。要了解有关 Node.js 运行时随附的 SDK 版本的更多信息，请参阅 the section called “包含运行时的 SDK 版本”。

Lambda 支持以下 Node.js 运行时。

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
Node.js 20	`nodejs20.x`	Amazon Linux 2023	未计划	未计划	未计划
Node.js 18	`nodejs18.x`	Amazon Linux 2	2025 年 7 月 31 日	2025 年 9 月 1 日	2025 年 10 月 1 日

> **Note**
>
> Node.js 18 和更高版本运行时系统使用适用于 JavaScript v3 的 AWS SDK。要从较早的运行时系统迁移函数，请关注 GitHub 上的 migration workshop（迁移研讨会）。有关适用于 JavaScript 版本 3 的 AWS SDK的更多信息，请参阅适用于 JavaScript 的模块化 AWS SDK 现已正式发布博客文章。

创建 Node.js 函数

1. 打开 Lambda 控制台。
2. 选择 Create function（创建函数）。
3. 配置以下设置：
   - 函数名称：输入函数名称。
   - 运行时系统：选择 Node.js 20.x。
4. 选择 Create function（创建函数）。
5. 要配置测试事件，请选择测试。

6. 对于事件名称，输入 **test**。
7. 选择 Save changes（保存更改）。
8. 要调用该函数，请选择 Test（测试）。

控制台使用名为 `index.js` 或 `index.mjs` 的单个源文件创建一个 Lambda 函数。您可以在内置代码编辑器中编辑此文件并添加更多文件。要保存您的更改，请选择 Save（保存）。然后，要运行代码，请选择 Test（测试）。

该 `index.js` 或 `index.mjs` 文件会导出一个名为 `handler` 的函数，此函数将接受事件对象和上下文对象。这是 Lambda 在调用函数时调用的处理函数。Node.js 函数运行时从 Lambda 获取调用事件并将其传递到处理程序。在函数配置中，处理程序值为 `index.handler`。

保存函数代码时，Lambda 控制台会创建一个 .zip 文件归档部署包。在控制台外部开发函数代码时（使用 IDE），您需要创建部署程序包将代码上载到 Lambda 函数。

除了调用事件之外，函数运行时还将上下文对象传递给处理程序。上下文对象包含有关调用、函数和执行环境的其他信息。环境变量中提供了更多信息。

您的 Lambda 函数附带了 CloudWatch Logs 日志组。函数运行时会将每次调用的详细信息发送到 CloudWatch Logs。该运行时会中继调用期间函数输出的任何日志。如果您的函数返回错误，则 Lambda 将为错误设置格式，并将其返回给调用方。

主题

- Node.js 初始化
- 包含运行时的 SDK 版本
- 对 TCP 连接使用“保持连接”
- 正在加载 CA 证书
- 定义采用 Node.js 的 Lambda 函数处理程序
- 使用 .zip 文件归档部署 Node.js Lambda 函数
- 使用容器映像部署 Node.js Lambda 函数
- 使用 Node.js Lambda 函数的层
- 使用 Lambda 上下文对象检索 Node.js 函数信息
- Node.js Lambda 函数日志记录和监控
- 在 AWS Lambda 中检测 Node.js 代码

Node.js 初始化

Node.js 使用独有的事件循环模型，导致其初始化行为与其他运行时不同。具体来说，Node.js 使用支持异步操作的非阻止式 I/O 模型。此模型允许 Node.js 高效地执行大多数工作负载。例如，假设 Node.js 函数进行网络调用，则该请求可能会被指定为异步操作并放入回调队列中。函数可能会继续处理主调用堆栈内的其他操作，而不会因等待网络调用返回而被阻止。完成网络调用后，将会执行对它的回调，然后从回调队列中删除。

某些初始化任务可能会异步运行。这些异步任务不能保证在调用之前完成执行。例如，在 Lambda 执行处理函数时，进行网络调用以从 AWS 参数存储中获取参数的代码可能尚未完成。因此在调用过程中，变量可能为空。为避免这种情况，请首先确保变量和其他异步代码已完全初始化，然后再继续使用函数的其余核心业务逻辑。

或者，您可以将函数代码指定为 ES 模块，以便在函数处理程序的范围之外，在文件的最顶层使用 `await`。当您 `await` 每个 `Promise` 时，异步初始化代码会在处理程序调用之前完成，从而最大限度地提高预置并发在减少冷启动延迟方面的有效性。有关更多信息和示例，请参阅在 AWS Lambda 中使用 Node.js ES 模块和顶级等待。

将函数处理程序指定为 ES 模块

默认情况下，Lambda 将带有 `.js` 后缀的文件视为 CommonJS 模块。或者，您可以将您的代码指定为 ES 模块。您可以通过两种方式执行此操作：在函数的 `package.json` 文件中将 `type` 指定为 `module`，或者使用 `.mjs` 文件扩展名。在第一种方式中，函数代码将所有 `.js` 文件视为 ES 模块，而在第二种场景中，只有使用 `.mjs` 指定的文件才是 ES 模块。您可以通过将它们分别命名为 `.mjs` 和 `.cjs` 来混合 ES 模块和 CommonJS 模块，因为 `.mjs` 文件始终是 ES 模块，`.cjs` 文件始终是 CommonJS 模块。

Lambda 会在加载 ES 模块时在 `NODE_PATH` 环境变量中搜索文件夹。您可以使用 ES 模块 `import` 语句加载运行时中包含的 AWS SDK。您也可以从层加载 ES 模块。

包含运行时的 SDK 版本

Node.js 运行时中包含的 AWS SDK 版本取决于运行时版本和您的 AWS 区域。要查找您正在使用的运行时中包含的 SDK 的版本，请使用以下代码创建 Lambda 函数。

> **Note**
>
> 下面显示的 Node.js 版本 18 及以上版本的示例代码使用 CommonJS 格式。如果您在 Lambda 控制台中创建函数，则请务必将包含代码的文件重命名为 `index.js`。

Example Node.js 18 及更高版本

```
const { version } = require("@aws-sdk/client-s3/package.json");

exports.handler = async () => ({ version });
```

这会返回以下格式的响应：

```
{
  "version": "3.462.0"
}
```

对 TCP 连接使用“保持连接”

默认的 Node.js HTTP/HTTPS 代理会为每个新请求创建一个新的 TCP 连接。为了避免建立新连接的成本，您可以使用 `keepAlive: true` 来重用函数使用适用于 JavaScript 的 AWS SDK 创建的连接。“保持连接”可以减少使用 SDK 进行多次 API 调用的 Lambda 函数的请求时间。

在适用于 JavaScript 的 AWS SDK 版本 3.x（包含在 `nodejs18.x` 和更高版本的 Lambda 运行时系统中）中，默认启用“保持连接”。要禁用“保持连接”，请参阅《AWS SDK for JavaScript 3.x Developer Guide》中的 Reusing connections with keep-alive in Node.js。有关使用“保持连接”的更多信息，请参阅 AWS 开发工具博客上的 HTTP keep-alive is on by default in modular AWS SDK for JavaScript。

正在加载 CA 证书

对于 Node.js 18 之前的 Node.js 运行时系统版本，Lambda 会自动加载特定于 Amazon 的 CA（证书颁发机构）证书，让您更轻松地创建与其他 AWS 服务交互的函数。例如，Lambda 包括验证安装在 Amazon RDS 数据库上的服务器身份证书所必需的 Amazon RDS 证书。这种行为可能会对冷启动期间的性能产生影响。

从 Node.js 20 开始，Lambda 默认不再加载其他 CA 证书。Node.js 20 运行时系统包含一个证书文件，其中所有 Amazon CA 证书都位于 `/var/runtime/ca-cert.pem`。要从 Node.js 18 及更早版本

的运行时系统恢复相同的行为，请将 NODE_EXTRA_CA_CERTS 环境变量设置为 /var/runtime/ca-cert.pem。

为了获得最佳性能，我们建议仅将所需的证书与部署包捆绑在一起，并通过 NODE_EXTRA_CA_CERTS 环境变量进行加载。证书文件应包含一个或多个 PEM 格式的可信根证书或中间 CA 证书。例如，对于 RDS，将所需的证书与代码一起包含在 certificates/rds.pem。然后，通过将 NODE_EXTRA_CA_CERTS 设置为 /var/task/certificates/rds.pem 来加载证书。

定义采用 Node.js 的 Lambda 函数处理程序

Lambda 函数处理程序是函数代码中处理事件的方法。当调用函数时，Lambda 运行处理程序方法。您的函数会一直运行，直到处理程序返回响应、退出或超时。

主题

Node.js 处理程序基础知识

以下示例函数记录事件对象的内容并返回日志的位置。

> **Note**
>
> 本页显示了 CommonJS 和 ES 模块处理程序的示例。要了解这两种处理程序之间的差异，请参阅 将函数处理程序指定为 ES 模块。

ES module handler

Example

```
export const handler = async (event, context) => {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2));
  return context.logStreamName;
};
```

CommonJS module handler

Example

```
exports.handler = async function (event, context) {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2));
  return context.logStreamName;
```

```
};
```

配置函数时，处理程序设置的值是文件的名称，也是导出的处理程序方法的名称（由点分隔）。控制台中的默认值为 `index.handler`，这也是本指南所用示例的值。这表示从 `handler` 文件中导出的 `index.js` 方法。

运行时会将参数传递到处理程序方法。第一个参数是 `event` 对象，它包含来自调用方的信息。调用程序在调用 Invoke 时将该信息作为 JSON 格式字符串传递，运行时将它转换为对象。当 AWS 服务调用您的函数时，事件结构因服务而异。

第二个参数是 context 对象，它包含有关调用、函数和执行环境的信息。在前面的示例中，函数将从 context 对象获取日志流的名称，然后将其返回给调用方。

您也可以使用回调参数，这是一种可在非异步处理程序中进行调用以发送响应的函数。我们建议您使用 Async/await（异步/等待），而不使用 Callback（回调）。Async/await 具有更好的易读性、错误处理能力和效率。有关 Async/await 与 Callback 之间差异的更多信息，请参阅 使用回调。

命名

配置函数时，处理程序设置的值是文件的名称，也是导出的处理程序方法的名称（由点分隔）。控制台中创建的函数的默认值为 `index.handler`，这也是本指南所用示例的值。这表示从 `index.js` 或 `index.mjs` 文件中导出的 `handler` 方法。

如果您在控制台中使用不同的文件名或函数处理程序名称创建函数，则必须编辑默认处理程序名称。

更改函数处理程序名称（控制台）

1. 打开 Lambda 控制台的函数页面，然后选择一个函数。
2. 选择节点选项卡。
3. 向下滚动到运行时设置窗格并选择编辑。
4. 在处理程序中，输入函数处理程序的新名称。
5. 选择保存。

使用异步/等待

如果您的代码执行异步任务，则使用 Async/await 模式来确保处理程序会完成运行。Async/await 是一种简洁、易读的 Node.js 异步代码编写方式，无需嵌套回调或链式承诺。使用 Async/await 时，您编写的代码看起来与同步代码类似，同时仍然是异步和非阻止式的。

async 关键字会将函数标记为异步，await 关键字会暂停函数的执行，直到 Promise 完成解析为止。

> **Note**
>
> 确保等待异步事件完成。如果函数在异步事件完成之前返回，则该函数可能会失败或导致应用程序出现意外行为。当 forEach 循环包含异步事件时可能会发生这种情况。forEach 循环需要同步调用。有关更多信息，请参阅 Mozilla 文档中的 Array.prototype.forEach()。

ES module handler

Example – 包含 async/await 的 HTTP 请求

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        // fetch is available in Node.js 18 and later runtimes
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

CommonJS module handler

Example – 包含 async/await 的 HTTP 请求

```
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = async function (event) {
  let statusCode;
  await new Promise(function (resolve, reject) {
    https.get(url, (res) => {
```

```
        statusCode = res.statusCode;
        resolve(statusCode);
      }).on("error", (e) => {
        reject(Error(e));
      });
  });
  console.log(statusCode);
  return statusCode;
};
```

下一个示例使用 Async/await 来列出您的 Amazon Simple Storage Service 存储桶。

> **Note**
>
> 在使用此示例之前，请确保您的函数的执行角色具有 Amazon S3 读取权限。

ES module handler

Example – 使用 Async/await 的 AWS SDK v3

此示例使用 AWS SDK for JavaScript v3，它支持 `nodejs18.x` 和更高版本的运行时系统。

```
import {S3Client, ListBucketsCommand} from '@aws-sdk/client-s3';
const s3 = new S3Client({region: 'us-east-1'});

export const handler = async(event) => {
  const data = await s3.send(new ListBucketsCommand({}));
  return data.Buckets;

};
```

CommonJS module handler

Example – 使用 Async/await 的 AWS SDK v3

此示例使用 AWS SDK for JavaScript v3，它支持 `nodejs18.x` 和更高版本的运行时系统。

```
const { S3Client, ListBucketsCommand } = require('@aws-sdk/client-s3');
const s3 = new S3Client({ region: 'us-east-1' });
```

```
exports.handler = async (event) => {
  const data = await s3.send(new ListBucketsCommand({}));
  return data.Buckets;
};
```

使用回调

我们建议您使用 Async/await 来声明函数处理程序，而不是使用回调。Async/await 是更好的选择，原因有以下几点：

- 易读性：Async/await 代码比回调代码更易阅读和理解，回调代码很快就会变得难以理解，从而导致回调失败。
- 调试和错误处理：基于回调的代码可能非常难以调试。调用堆栈可能变得难以理解，并且很容易出现错误。使用 Async/await 时，您可以使用 try/catch 块来处理错误。
- 效率：回调通常需要在代码的不同部分之间切换。Async/await 可以减少上下文切换的数量，从而提高代码效率。

当您在处理程序中使用回调时，函数会一直执行，直到 事件循环 为空或函数超时为止。在完成所有事件循环任务之前，不会将响应发送给调用方。如果函数超时，则会返回 error。可以通过将 context.callbackWaitsForEmptyEventLoop 设置为 false，从而将运行时配置为立即发送响应。

callback 函数有两个参数：一个是 `Error`，一个是响应。响应对象必须与 `JSON.stringify` 兼容。

以下示例函数检查 URL 并向调用方返回状态代码。

ES module handler

Example – 包含 callback 的 HTTP 请求

```
import https from "https";
let url = "https://aws.amazon.com/";

export function handler(event, context, callback) {
  https.get(url, (res) => {
      callback(null, res.statusCode);
    }).on("error", (e) => {
      callback(Error(e));
    });
```

```
}
```

CommonJS module handler

Example – 包含 callback 的 HTTP 请求

```
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = function (event, context, callback) {
  https.get(url, (res) => {
      callback(null, res.statusCode);
    }).on("error", (e) => {
      callback(Error(e));
    });
};
```

在下一示例中，来自 Amazon S3 的响应将在可用时立即返回给调用方。针对事件循环运行的超时被冻结，并在下次调用该函数时继续运行。

> **Note**
>
> 在使用此示例之前，请确保您的函数的执行角色具有 Amazon S3 读取权限。

ES module handler

Example – 包含 callbackWaitsForEmptyEventLoop 的 AWS SDK v3

此示例使用 AWS SDK for JavaScript v3，它支持 `nodejs18.x` 和更高版本的运行时系统。

```
import AWS from "@aws-sdk/client-s3";
const s3 = new AWS.S3({});

export const handler = function (event, context, callback) {
  context.callbackWaitsForEmptyEventLoop = false;
  s3.listBuckets({}, callback);
  setTimeout(function () {
    console.log("Timeout complete.");
  }, 5000);
};
```

CommonJS module handler

Example – 包含 callbackWaitsForEmptyEventLoop 的 AWS SDK v3

此示例使用 AWS SDK for JavaScript v3，它支持 nodejs18.x 和更高版本的运行时系统。

```
const AWS = require("@aws-sdk/client-s3");
const s3 = new AWS.S3({});

exports.handler = function (event, context, callback) {
  context.callbackWaitsForEmptyEventLoop = false;
  s3.listBuckets({}, callback);
  setTimeout(function () {
    console.log("Timeout complete.");
  }, 5000);
};
```

Node.js Lambda 函数的代码最佳实践

在构建 Lambda 函数时，请遵循以下列表中的指南，采用最佳编码实践：

- 从核心逻辑中分离 Lambda 处理程序。这样您可以创建更容易进行单元测试的函数。在 Node.js 中可能如下所示：

```
exports.myHandler = function(event, context, callback) {
 var foo = event.foo;
 var bar = event.bar;
 var result = MyLambdaFunction (foo, bar);

 callback(null, result);
}

function MyLambdaFunction (foo, bar) {
 // MyLambdaFunction logic here
}
```

- 控制函数部署程序包中的依赖关系。AWS Lambda 执行环境包含许多库。对于 Node.js 和 Python 运行时，其中包括 AWS SDK。Lambda 会定期更新这些库，以支持最新的功能组合和安全更新。这些更新可能会使 Lambda 函数的行为发生细微变化。要完全控制您的函数所用的依赖项，请使用部署程序包来打包所有依赖项。
- 将依赖关系的复杂性降至最低。首选在执行环境启动时可以快速加载的更简单的框架。

- 将部署程序包大小精简为只包含运行时必要的部分。这样会减少调用前下载和解压缩部署程序包所需的时间。

- 利用执行环境重用来提高函数性能。连接软件开发工具包 (SDK) 客户端和函数处理程序之外的数据库，并在 `/tmp` 目录中本地缓存静态资产。由函数的同一实例处理的后续调用可重用这些资源。这样就可以通过缩短函数运行时间来节省成本。

  为了避免调用之间潜在的数据泄露，请不要使用执行环境来存储用户数据、事件或其他具有安全影响的信息。如果您的函数依赖于无法存储在处理程序的内存中的可变状态，请考虑为每个用户创建单独的函数或单独的函数版本。
- 使用 keep-alive 指令来维护持久连接。Lambda 会随着时间的推移清除空闲连接。在调用函数时尝试重用空闲连接会导致连接错误。要维护您的持久连接，请使用与运行时关联的 keep-alive 指令。有关示例，请参阅在 Node.js 中通过 Keep-Alive 重用连接。
- 使用环境变量将操作参数传递给函数。例如，您在写入 Amazon S3 存储桶时，不应对要写入的存储桶名称进行硬编码，而应将存储桶名称配置为环境变量。
- 避免在 Lambda 函数中使用递归调用，在这种情况下，函数会调用自己或启动可能再次调用该函数的进程。这可能会导致意想不到的函数调用量和升级成本。如果您看到意外的调用量，请立即将函数保留并发设置为 `0` 来限制对函数的所有调用，同时更新代码。
- Lambda 函数代码中不要使用非正式的非公有 API。对于 AWS Lambda 托管式运行时，Lambda 会定期为 Lambda 的内部 API 应用安全性和功能更新。这些内部 API 更新可能不能向后兼容，会导致意外后果，例如，假设您的函数依赖于这些非公有 API，则调用会失败。请参阅 API 参考以查看公开发布的 API 列表。
- 编写幂等代码。为您的函数编写幂等代码可确保以相同的方式处理重复事件。您的代码应该正确验证事件并优雅地处理重复事件。有关更多信息，请参阅如何使我的 Lambda 函数具有幂等性？。

使用 .zip 文件归档部署 Node.js Lambda 函数

AWS Lambda 函数的代码包含一个 .js 或 .mjs 文件，其中包含函数的处理程序代码，以及代码所依赖的任何其他包和模块。要将此函数部署到 Lambda，您可以使用部署包。此包可以是 .zip 文件归档或容器映像。有关在 Node.js 中使用容器映像的更多信息，请参阅使用容器映像部署 Node.js Lambda 函数。

要创建 .zip 文件归档格式的部署包，可以使用命令行工具内置的 .zip 文件归档实用工具或任何其他 .zip 文件实用工具（例如 7zip）。以下各部分中显示的示例假设您在 Linux 或 macOS 环境中使用命令行 `zip` 工具。要在 Windows 中使用相同命令，您可以安装 Windows Subsystem for Linux，以获取 Windows 集成版本的 Ubuntu 和 Bash。

请注意，Lambda 使用 POSIX 文件权限，因此在创建 .zip 文件归档之前，您可能需要为部署包文件夹设置权限。

主题

Node.js 中的运行时系统依赖项

对于使用 Node.js 运行时系统的 Lambda 函数，依赖项可以是任何 Node.js 模块。Node.js 运行时系统包括许多常用库以及一个 AWS SDK for JavaScript 版本。`nodejs16.x` Lambda 运行时系统包括 SDK 的版本 2.x。运行时系统版本 `nodejs18.x` 及更高版本包括 SDK 的版本 3。要将 SDK 版本 2 与运行时系统版本 `nodejs18.x` 及更高版本结合使用，请将该 SDK 添加到您的 .zip 文件部署包中。如果所选运行时系统包含您正在使用的开发工具包版本，则无需在.zip 文件中包含开发工具包库。要查找正在使用的运行时系统中包含哪个版本的 SDK，请参阅 the section called “包含运行时的 SDK 版本”。

Lambda 会定期更新 Node.js 运行时系统中的开发工具包库，以包含最新更新和安全补丁。Lambda 还会对运行时系统中包含的其他库应用安全补丁和更新。要完全控制包中的依赖项，您可以将任何包含运行时系统依赖项的首选版本添加到部署包中。例如，如果您想使用适用于 JavaScript 的开发工具包的特定版本，则可以将其作为依赖项包含在.zip 文件中。有关向.zip 文件添加包含运行时系统的依赖项的更多信息，请参阅 依赖项搜索路径和包含运行时系统的库。

在 AWS 责任共担模式下，您负责管理函数部署包中的所有依赖项。这包括应用更新和安全补丁。要更新函数部署包中的依赖项，请先创建一个新的 .zip 文件，然后将其上传到 Lambda 中。有关更多信息，请参阅 创建含依赖项的 .zip 部署包 和 使用 .zip 文件创建和更新 Node.js Lambda 函数。

创建不含依赖项的 .zip 部署包

如果除了包含在 Lamba 运行时系统中的库外，您的函数代码没有其他依赖项，则 .zip 文件仅包含带有函数处理程序代码的 `index.js` 或 `index.mjs` 文件。使用您的首选 zip 实用工具创建一个 .zip 文件，并将 `index.js` 或 `index.mjs` 文件置于根目录中。如果包含您处理程序代码的文件不在 .zip 文件的根目录中，Lambda 将无法运行代码。

要了解如何部署 .zip 文件以创建新的 Lambda 函数或更新现有函数，请参阅 使用 .zip 文件创建和更新 Node.js Lambda 函数。

创建含依赖项的 .zip 部署包

如果函数代码依赖未包含在 Lamba Node.js 运行时系统中的程序包或模块，您可以使用函数代码将这些依赖项添加到 .zip 文件中，也可以使用 Lambda 层。本部分中的说明向您展示了如何将依赖项包含在 .zip 部署包中。有关如何将依赖项包含在层中的说明，请参阅 the section called “为依赖项创建 Node.js 层”。

以下示例 CLI 命令将创建名为 `my_deployment_package.zip` 的 .zip 文件，其中包含 `index.js` 或 `index.mjs` 文件，以及您的函数处理程序代码及其依赖项。在示例中，您要使用 npm 程序包管理器来安装依赖项。

创建部署包

1. 导航到包含 `index.js` 或 `index.mjs` 源代码文件的项目目录。在此示例中，该目录名为 `my_function`。

   ```
   cd my_function
   ```

2. 使用 `npm install` 命令在 `node_modules` 目录中安装函数所需的库。在此示例中，您要安装 AWS X-Ray SDK for Node.js。

   ```
   npm install aws-xray-sdk
   ```

   这将创建一个类似于下面的文件夹结构：

   ```
   ~/my_function
   ```

```
### index.mjs
### node_modules
    ### async
    ### async-listener
    ### atomic-batcher
    ### aws-sdk
    ### aws-xray-sdk
    ### aws-xray-sdk-core
```

您还可以将自己创建的自定义模块添加到部署包中。在 node_modules 下创建一个以模块命名的目录，然后将自定义编写程序包保存在此处。

3. 在根目录下创建一个包含您的项目文件夹内容的 .zip 文件。使用 r （递归）选项来确保 zip 压缩子文件夹。

```
zip -r my_deployment_package.zip .
```

为依赖项创建 Node.js 层

本部分中的说明旨在向您展示如何将依赖项包含在层中。有关如何将依赖项包含在部署包中的说明，请参阅 the section called “创建含依赖项的 .zip 部署包”。

当您向函数添加层时，Lambda 会将层内容加载到该执行环境的 /opt 目录中。对于每个 Lambda 运行时系统，PATH 变量都包括 /opt 目录中的特定文件夹路径。为确保 PATH 变量能够获取层内容，层 .zip 文件应在以下文件夹路径中具有依赖项：

- nodejs/node_modules
- nodejs/node16/node_modules (NODE_PATH)
- nodejs/node18/node_modules (NODE_PATH)
- nodejs/node20/node_modules (NODE_PATH)

例如，层.zip 文件结构可能如下所示：

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

此外，Lambda 会自动检测 /opt/lib 目录中的任何库，以及 /opt/bin 目录中的任何二进制文件。为确保 Lambda 正确获取层内容，还可以创建包含以下结构的层：

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

打包层后，请参阅 the section called “创建和删除层” 和 the section called “添加层” 以完成层设置。

依赖项搜索路径和包含运行时系统的库

Node.js 运行时系统包括许多常用库以及一个 AWS SDK for JavaScript 版本。如果您想使用包含运行时系统的库的不同版本，可以通过将其与自己的函数捆绑或将其作为依赖项添加到部署包中来实现。例如，您可以通过将开发工具包添加到 .zip 部署包中来使用其他版本的开发工具包。您也可以将其包含在函数的 Lambda 层中。

在代码中使用 `import` 或 `require` 语句时，Node.js 运行时系统会搜索 `NODE_PATH` 路径中的目录，直到找到相应模块。默认情况下，运行时系统搜索的第一个位置是解压缩并安装 .zip 部署包的目录 (`/var/task`)。如果在部署包中包含运行时系统的库的某个版本，则此版本将优先于运行时系统中包含的版本。部署包中的依赖项也优先于层中的依赖项。

向层添加依赖项时，Lambda 会将其提取到 `/opt/nodejs/nodexx/node_modules` 中，其中 `nodexx` 表示正在使用的运行时系统版本。在搜索路径中，此目录优先于包含运行时系统的库的目录 (`/var/lang/lib/node_modules`)。因此，函数层中的库优先于运行时系统中包含的版本。

通过添加以下代码行，您可以查看 Lambda 函数的完整搜索路径。

```
console.log(process.env.NODE_PATH)
```

您还可以在 .zip 程序包内的单独文件夹中添加依赖项。例如，您可以将自定义模块添加到 .zip 程序包中名为 `common` 的文件夹中。解压缩并安装 .zip 程序包后，此文件夹将放置在 `/var/task` 目录中。要在代码中使用 .zip 部署包中某个文件夹的依赖项，请使用 `import { } from` 或 `const { } = require()` 语句，具体取决于您使用的是 CJS 还是 ESM 模块解析。例如：

```
import { myModule } from './common'
```

如果您将代码与 `esbuild`、`rollup` 或类似内容捆绑在一起，则函数使用的依赖项将捆绑在一个或多个文件中。我们建议尽量使用此方法来提供依赖项。与向部署包添加依赖项相比，由于减少了 I/O 操作，捆绑代码可以提高性能。

使用 .zip 文件创建和更新 Node.js Lambda 函数

创建 .zip 部署包后，您可以用其创建新的 Lambda 函数或更新现有的 Lambda 函数。您可以使用 Lambda 控制台、AWS Command Line Interface 和 Lambda API 部署 .zip 程序包。您也可以使用 AWS Serverless Application Model（AWS SAM）和 AWS CloudFormation 创建和更新 Lambda 函数。

Lambda 的 .zip 部署包的最大大小为 250MB（已解压缩）。请注意，此限制适用于您上传的所有文件（包括任何 Lambda 层）的组合大小。

Lambda 运行时需要权限才能读取部署包中的文件。在 Linux 权限八进制表示法中，Lambda 对于不可执行文件（rw-r--r--）需要 644 个权限，对于目录和可执行文件需要 755 个权限（rwxr-xr-x）。

在 Linux 和 MacOS 中，使用 chmod 命令更改部署包中文件和目录的文件权限。例如，要为可执行文件提供正确的权限，请运行以下命令。

```
chmod 755 <filepath>
```

要在 Windows 中更改文件权限，请参阅 Microsoft Windows 文档中的 Set, View, Change, or Remove Permissions on an Object。

使用控制台通过 .zip 文件创建和更新函数

要创建新函数，必须先在控制台中创建该函数，然后上传您的 .zip 归档。要更新现有函数，请打开函数页面，然后按照相同的步骤添加更新的 .zip 文件。

如果您的 .zip 文件小于 50MB，则可以通过直接从本地计算机上传该文件来创建或更新函数。对于大于 50MB 的 .zip 文件，必须首先将您的程序包上传到 Amazon S3 存储桶。有关如何使用 AWS Management Console 将文件上传到 Amazon S3 存储桶的说明，请参阅 Amazon S3 入门。要使用 AWS CLI 上传文件，请参阅《AWS CLI 用户指南》中的移动对象。

> **Note**
>
> 您无法更改现有函数的部署包类型（.zip 或容器映像）。例如，您无法将容器映像函数转换为使用 .zip 文件归档。您必须创建新函数。

创建新函数（控制台）

1. 打开 Lambda 控制台的“函数”页面，然后选择创建函数。

2. 选择从头开始创作。
3. 在基本信息中，执行以下操作：
   a. 对于函数名称，输入函数的名称。
   b. 对于运行时系统，选择要使用的运行时系统。
   c. （可选）对于架构，选择要用于函数的指令集架构。默认架构为 x86_64。确保您的函数的 .zip 部署包与您选择的指令集架构兼容。
4. （可选）在 Permissions（权限）下，展开 Change default execution role（更改默认执行角色）。您可以创建新的执行角色，也可以使用现有角色。
5. 选择 Create function (创建函数)。Lambda 使用您选择的运行时系统创建基本“Hello world”函数。

从本地计算机上传 .zip 归档（控制台）

1. 在 Lambda 控制台的“函数”页面中，选择要为其上传 .zip 文件的函数。
2. 选择代码选项卡。
3. 在代码源窗格中，选择上传自。
4. 选择 .zip 文件。
5. 要上传 .zip 文件，请执行以下操作：
   a. 选择上传，然后在文件选择器中选择您的 .zip 文件。
   b. 选择打开。
   c. 选择保存。

从 Amazon S3 存储桶上传 .zip 归档（控制台）

1. 在 Lambda 控制台的“函数”页面中，选择要为其上传新 .zip 文件的函数。
2. 选择代码选项卡。
3. 在代码源窗格中，选择上传自。
4. 选择 Amazon S3 位置。
5. 粘贴 .zip 文件的 Amazon S3 链接 URL，然后选择保存。

使用控制台代码编辑器更新 .zip 文件函数

对于某些带有 .zip 部署包的函数，您可以使用 Lambda 控制台的内置代码编辑器直接更新函数代码。要使用此功能，函数必须满足以下条件：

- 函数必须使用一种解释性语言运行时系统（Python、Node.js 或 Ruby）
- 函数的部署包必须小于 50 MB（未压缩状态）。

带有容器映像部署包的函数的代码不能直接在控制台中编辑。

要使用控制台代码编辑器更新函数代码。

1. 打开 Lambda 控制台的“函数”页面，然后选择函数。
2. 选择代码选项卡。
3. 在代码源窗格中，选择源代码文件并在集成的代码编辑器中对其进行编辑。
4. 编辑完代码后，展开主侧栏中的部署部分，然后选择部署。

使用 AWS CLI 通过 .zip 文件创建和更新函数

您可以使用 AWS CLI 创建新函数或使用 .zip 文件更新现有函数。使用 create-function 和 update-function-code 命令部署 .zip 程序包。如果您的 .zip 文件小于 50MB，则可以从本地生成计算机上的文件位置上传 .zip 程序包。对于较大的文件，必须从 Amazon S3 存储桶上传 .zip 程序包。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

> **Note**
>
> 如果您使用 AWS CLI 从 Amazon S3 存储桶上传 .zip 文件，则该存储桶必须与您的函数位于同一个 AWS 区域 中。

要通过 AWS CLI 使用 .zip 文件创建新函数，则必须指定以下内容：

- 函数的名称 (`--function-name`)
- 函数的运行时系统 (`--runtime`)
- 函数的执行角色 (`--role`) 的 Amazon 资源名称（ARN）
- 函数代码 (`--handler`) 中处理程序方法的名称

还必须指定 .zip 文件的位置。如果 .zip 文件位于本地生成计算机上的文件夹中，请使用 `--zip-file` 选项指定文件路径，如以下示例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime nodejs20.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

要指定 .zip 文件在 Amazon S3 存储桶中的位置，请使用 `--code` 选项，如以下示例命令所示。您只需对版本控制对象使用 `S3ObjectVersion` 参数。

```
aws lambda create-function --function-name myFunction \
--runtime nodejs20.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

要使用 CLI 更新现有函数，请使用 `--function-name` 参数指定函数的名称。您还必须指定要用于更新函数代码的 .zip 文件的位置。如果 .zip 文件位于本地生成计算机上的文件夹中，请使用 `--zip-file` 选项指定文件路径，如以下示例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

要指定 .zip 文件在 Amazon S3 存储桶中的位置，请使用 `--s3-bucket` 和 `--s3-key` 选项，如以下示例命令所示。您只需对版本控制对象使用 `--s3-object-version` 参数。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

使用 Lambda API 通过 .zip 文件创建和更新函数

要使用 .zip 文件归档创建和更新函数，请使用以下 API 操作：

- CreateFunction
- UpdateFunctionCode

使用 AWS SAM 通过 .zip 文件创建和更新函数

AWS Serverless Application Model（AWS SAM）是一个工具包，可帮助简化在 AWS 上构建和运行无服务器应用程序的过程。您可以在 YAML 或 JSON 模板中为应用程序定义资源，并使用 AWS SAM 命令行界面（AWS SAM CLI）构建、打包和部署应用程序。当您通过 AWS SAM 模板构建 Lambda 函数时，AWS SAM 会使用您的函数代码和您指定的任何依赖项自动创建 .zip 部署包或容器映像。要了解有关使用 AWS SAM 构建和部署 Lambda 函数的更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 AWS SAM 入门。

您可以使用 AWS SAM 创建使用现有 .zip 文件归档的 Lambda 函数。要使用 AWS SAM 创建 Lambda 函数，您可以将 .zip 文件保存在 Amazon S3 存储桶或生成计算机上的本地文件夹中。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

在 AWS SAM 模板中，`AWS::Serverless::Function` 资源将指定 Lambda 函数。在此资源中，设置以下属性以创建使用 .zip 文件归档的函数：

- `PackageType` – 设置为 `Zip`
- `CodeUri` – 设置为函数代码的 Amazon S3 URI、本地文件夹的路径或 FunctionCode 对象
- `Runtime` – 设置为您选择的运行时系统

使用 AWS SAM，如果 .zip 文件大于 50MB，则不需要先将其上传到 Amazon S3 存储桶。AWS SAM 可以从本地生成计算机上的某个位置上传最大允许大小为 250MB（已解压缩）的 .zip 程序包。

要了解有关在 AWS SAM 中使用 .zip 文件部署函数的更多信息，请参阅《AWS SAM 开发人员指南》中的 AWS::Serverless::Function。

使用 AWS CloudFormation 通过 .zip 文件创建和更新函数

您可以使用 AWS CloudFormation 创建使用 .zip 文件归档的 Lambda 函数。要从 .zip 文件创建 Lambda 函数，必须先将您的文件上传到 Amazon S3 存储桶。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

在 AWS CloudFormation 模板中，`AWS::Lambda::Function` 资源将指定 Lambda 函数。在此资源中，设置以下属性以创建使用 .zip 文件归档的函数：

- `PackageType` – 设置为 `Zip`
- `Code` – 在 `S3Bucket` 和 `S3Key` 字段中输入 Amazon S3 存储桶名称和 .zip 文件名。
- `Runtime` – 设置为您选择的运行时系统

AWS CloudFormation 生成的 .zip 文件不能超过 4MB。要了解有关在 AWS CloudFormation 中使用 .zip 文件部署函数的更多信息，请参阅《AWS CloudFormation 用户指南》中的 AWS::Lambda::Function。

使用容器映像部署 Node.js Lambda 函数

有三种方法可以为 Node.js Lambda 函数构建容器映像：

- 使用 Node.js 的 AWS 基本映像

  AWS 基本映像会预加载一个语言运行时系统、一个用于管理 Lambda 和函数代码之间交互的运行时系统接口客户端，以及一个用于本地测试的运行时系统接口仿真器。

- 使用 AWS 仅限操作系统的基础镜像

  AWS 仅限操作系统的运行时系统包含 Amazon Linux 发行版和运行时系统接口模拟器。这些镜像通常用于为编译语言（例如 Go 和 Rust）以及 Lambda 未提供基础映像的语言或语言版本（例如 Node.js 19）创建容器镜像。您也可以使用仅限操作系统的基础映像来实施自定义运行时系统。要使映像与 Lambda 兼容，您必须在映像中包含 Node.js 的运行时系统接口客户端。

- 使用非 AWS 基本映像

  您还可以使用其他容器注册表的备用基本映像，例如 Alpine Linux 或 Debian。您还可以使用您的组织创建的自定义映像。要使映像与 Lambda 兼容，您必须在映像中包含 Node.js 的运行时系统接口客户端。

> **Tip**
>
> 要缩短 Lambda 容器函数激活所需的时间，请参阅 Docker 文档中的使用多阶段构建。要构建高效的容器映像，请遵循编写 Dockerfiles 的最佳实践。

此页面介绍了如何为 Lambda 构建、测试和部署容器映像。

主题

Node.js 的 AWS 基本映像

AWS 为 Node.js 提供了以下基本映像：

标签	运行时	操作系统	Dockerfile	淘汰
20	Node.js 20	Amazon Linux 2023	GitHub 上适用于 Node.js 20 的 Dockerfile	未计划
18	Node.js 18	Amazon Linux 2	GitHub 上适用于 Node.js 18 的 Dockerfile	2025 年 7 月 31 日

Amazon ECR 存储库：gallery.ecr.aws/lambda/nodejs

Node.js 20 及更高版本的基础映像基于 Amazon Linux 2023 最小容器映像。早期的基础映像使用 Amazon Linux 2。与 Amazon Linux 2 相比，AL2023 具有多项优势，包括较小的部署占用空间以及 `glibc` 等更新版本的库。

基于 AL2023 的映像使用 `microdnf`（符号链接为 `dnf`）作为软件包管理器，而不是 Amazon Linux 2 中的默认软件包管理器 `yum`。`microdnf` 是 `dnf` 的独立实现。有关基于 AL2023 的映像中已包含软件包的列表，请参阅 Comparing packages installed on Amazon Linux 2023 Container Images 中的 Minimal Container 列。有关 AL2023 和 Amazon Linux 2 之间区别的更多信息，请参阅 AWS 计算博客上的 Introducing the Amazon Linux 2023 runtime for AWS Lambda。

> **Note**
>
> 要在本地运行基于 AL2023 的映像，包括使用 AWS Serverless Application Model（AWS SAM），您必须使用 Docker 版本 20.10.10 或更高版本。

使用 Node.js 的 AWS 基本映像

先决条件

要完成本节中的步骤，您必须满足以下条件：

- AWS CLI 版本 2
- Docker（Node.js 20 及更高版本基础映像的最低版本为 20.10.10）
- Node.js

从基本映像创建映像

从 Node.js 的 AWS 的基本映像创建容器映像

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir example
cd example
```

2. 使用 npm 创建新的 Node.js 项目。要在交互式体验中接受提供的默认选项，请按 Enter。

```
npm init
```

3. 创建名为 index.js 的新文件。您可以将以下示例函数代码添加到文件中进行测试，也可以使用您自己的函数代码。

   Example CommonJS 处理程序

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. 如果函数依赖于 AWS SDK for JavaScript 之外的库，请使用 npm 将其添加到程序包。
5. 使用以下配置创建一个新的 Dockerfile。

   - 将 FROM 属性设置为基本映像的 URI。
   - 使用 COPY 命令将函数代码和运行时系统依赖项复制到 {LAMBDA_TASK_ROOT}，此为 Lambda 定义的环境变量。
   - 将 CMD 参数设置为 Lambda 函数处理程序。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 root 用户。

Example Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:20

# Copy function code
COPY index.js ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "index.handler" ]
```

6. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 docker-image 并为其提供 test 标签。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 该命令指定了 --platform linux/amd64 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 --platform linux/arm64 选项。

（可选）在本地测试映像

1. 使用 docker run 命令启动 Docker 映像。在此示例中，docker-image 是映像名称，test 是标签。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令会将映像作为容器运行，并在 localhost:9000/2015-03-31/functions/function/invocations 创建本地端点。

> **Note**
>
> 如果为 ARM64 指令集架构创建 Docker 映像，请务必使用 --platform linux/*arm64* 选项，而不是 --platform linux/*amd64* 选项。

2. 在新的终端窗口中，将事件发布到本地端点。

   Linux/macOS

   在 Linux 和 macOS 中，运行以下 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   在 PowerShell 中，运行以下 `Invoke-WebRequest` 命令：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

3. 获取容器 ID。

   ```
   docker ps
   ```

4. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

   ```
   docker kill 3766c4ab331c
   ```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 `111122223333` 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
```

```
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。
   - 将 `<ECRrepositoryUri>` 替换为复制的 `repositoryUri`。确保 URI 末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 `ImageUri`，指定之前的存储库 URI。确保 URI 末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，`--publish` 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

将备用基本映像与运行时系统接口客户端配合使用

如果使用仅限操作系统的基础映像或者备用基础映像，则必须在映像中包括运行时系统接口客户端。运行时系统接口客户端可扩展 将 Lambda 运行时 API 用于自定义运行时，用于管理 Lambda 和函数代码之间的交互。

使用 npm 包管理器安装 Node.js 运行时系统接口客户端：

```
npm install aws-lambda-ric
```

您也可以从 GitHub 下载 Node.js 运行时接口客户端。运行时系统接口客户端支持以下 NodeJS 版本：

- 14.x

- 16.x
- 18.x
- 20.x

以下示例演示了如何使用非 AWS 基本映像为 Node.js 构建容器映像。示例 Dockerfile 使用 `buster` 基本映像。Docker 包含运行时系统接口客户端。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- AWS CLI 版本 2
- Docker
- Node.js

从备用基本映像创建映像

要从非 AWS 基本映像创建容器映像

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir example
cd example
```

2. 使用 npm 创建新的 Node.js 项目。要在交互式体验中接受提供的默认选项，请按 `Enter`。

```
npm init
```

3. 创建名为 `index.js` 的新文件。您可以将以下示例函数代码添加到文件中进行测试，也可以使用您自己的函数代码。

   Example CommonJS 处理程序

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
```

```
};
```

4. 创建新 Dockerfile。以下 Dockerfile 使用 `buster` 基本映像而不是 AWS 基本映像。Dockerfile 包含运行时系统接口客户端，该客户端可使映像与 Lambda 兼容。Dockerfile 使用多阶段构建。第一阶段将创建一个构建映像，作为安装函数依赖项的标准 Node.js 环境。第二阶段创建一个更精简的映像，其中包含函数代码及其依赖项。此机制可缩减最终映像大小。

   - 将 FROM 属性设置为基本映像标识符。
   - 使用 COPY 命令将复制函数代码和运行时系统依赖项。
   - 将 ENTRYPOINT 设置为您希望 Docker 容器在启动时运行的模块。在本例中，模块为运行时系统接口客户端。
   - 将 CMD 参数设置为 Lambda 函数处理程序。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 `root` 用户。

   Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM node:20-buster as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Install build dependencies
RUN apt-get update && \
    apt-get install -y \
    g++ \
    make \
    cmake \
    unzip \
    libcurl4-openssl-dev

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}
```

```
WORKDIR ${FUNCTION_DIR}

# Install Node.js dependencies
RUN npm install

# Install the runtime interface client
RUN npm install aws-lambda-ric

# Grab a fresh slim copy of the image to reduce the final size
FROM node:20-buster-slim

# Required for Node runtimes which use npm@8.6.0+ because
# by default npm writes logs under /home/.npm and Lambda fs is read-only
ENV NPM_CONFIG_CACHE=/tmp/.npm

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT ["/usr/local/bin/npx", "aws-lambda-ric"]
# Pass the name of the function handler as an argument to the runtime
CMD ["index.handler"]
```

5. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 `docker-image` 并为其提供 `test` 标签。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 该命令指定了 `--platform linux/amd64` 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 `--platform linux/arm64` 选项。

（可选）在本地测试映像

使用运行时系统接口仿真器在本地测试映像。您可以将仿真器构建到映像中，也可以使用以下程序将其安装在本地计算机上。

在本地计算机上安装并运行运行时系统接口仿真器

1. 从项目目录中，运行以下命令以从 GitHub 下载运行时系统接口仿真器（x86-64 架构）并将其安装在本地计算机上。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   要安装 arm64 仿真器，请将上一条命令中的 GitHub 存储库 URL 替换为以下内容：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   要安装 arm64 模拟器，请将 `$downloadLink` 替换为以下内容：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

2. 使用 docker run 命令启动 Docker 映像。请注意以下几点：

- `docker-image` 是映像名称，`test` 是标签。
- `/usr/local/bin/npx aws-lambda-ric index.handler` 是 `ENTRYPOINT`，后跟您 Dockerfile 中的 `CMD`。

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/npx aws-lambda-ric index.handler
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/local/bin/npx aws-lambda-ric index.handler
```

此命令会将映像作为容器运行，并在 `localhost:9000/2015-03-31/functions/function/invocations` 创建本地端点。

> **Note**
>
> 如果为 ARM64 指令集架构创建 Docker 映像，请务必使用 `--platform linux/arm64` 选项，而不是 `--platform linux/amd64` 选项。

3. 将事件发布到本地端点。

Linux/macOS

在 Linux 和 macOS 中，运行以下 `curl` 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，运行以下 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 获取容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 111122223333 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。

- 将 <ECRrepositoryUri> 替换为复制的 repositoryUri。确保 URI 末尾包含 :latest。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 :latest。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 ImageUri，指定之前的存储库 URI。确保 URI 末尾包含 :latest。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
```

```
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，`--publish` 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

使用 Node.js Lambda 函数的层

Lambda 层是包含补充代码或数据的 .zip 文件存档。层通常包含库依赖项、自定义运行时系统或配置文件。创建层涉及三个常见步骤：

1. 打包层内容。此步骤需要创建 .zip 文件存档，其中包含要在函数中使用的依赖项。
2. 在 Lambda 中创建层。
3. 将层添加到函数。

本主题包含有关如何正确打包并创建具有外部库依赖项的 Node.js Lambda 层的步骤和指南。

主题

先决条件

要完成本部分中的步骤，您必须满足以下条件：

- Node.js 20 和 npm 软件包管理器。有关安装 Node.js 的更多信息，请参阅 Node.js 文档中的 Installing Node.js via package manager。
- AWS CLI 版本 2

在本主题中，我们引用了 aws-lambda-developer-guide GitHub 存储库中的 layer-nodejs 示例应用程序。该应用程序包含用于下载依赖项并将其打包到层中的脚本。该应用程序还包含相应的函数，函数使用来自层的依赖项。创建层后，即可部署并调用相应的函数来验证一切是否正常运行。该示例应用程序使用 Node.js 20 运行时。如果您在层中添加其他依赖项，则它们必须与 Node.js 20 兼容。

`layer-nodejs` 示例应用程序会将 lodash 库打包到 Lambda 层中。`layer` 目录包含用于生成层的脚本。`function` 目录包含示例函数，用于帮助测试该层是否正常工作。本文档将演示如何创建、打包、部署并测试该层。

Node.js 层与 Lambda 运行时环境的兼容性

在 Node.js 层中打包代码时，需要指定与该代码兼容的 Lambda 运行时环境。要评测代码与运行时的兼容性，请考虑代码是为哪些版本的 Node.js、操作系统和指令集架构设计的。

Lambda Node.js 运行时指定其 Node.js 版本和操作系统。在本文档中，您将使用基于 AL2023 的 Node.js 20 运行时。有关运行时版本的更多信息，请参阅the section called “支持的运行时”。在创建 Lambda 函数时，您可以指定指令集架构。在本文档中，您将使用 `arm64` 架构。有关 Lambda 中的架构的更多信息，请参阅the section called “指令集（ARM/x86）”。

如果您使用软件包中提供的代码，每个软件包维护者都会独立定义其兼容性。大多数 Node.js 开发都是为了独立于操作系统和指令集架构而设计。此外，打破与新 Node.js 版本的不兼容性的情况并不常见。与评测软件包与 Node.js 版本、操作系统或指令集架构的兼容性相比，应该花更多的时间来评测软件包之间的兼容性。

有时，Node.js 包包含编译后的代码，这需要您考虑操作系统和指令集架构的兼容性。如果您确实需要评测软件包的代码兼容性，则需要检查软件包及其文档。NPM 中的软件包可以通过 `package.json` 清单文件的 `engines`、`os` 和 `cpu` 字段来指定其兼容性。有关 `package.json` 文件的更多信息，请参阅 NPM 文档中的 package.json。

Node.js 运行时的层路径

当您向函数添加层时，Lambda 会将层内容加载到执行环境中。如果您的层将依赖项打包到特定的文件夹路径中，Node.js 执行环境将识别这些模块，并且您可以从函数代码中引用这些模块。

为确保您的模块被拾取，请使用以下文件夹路径之一，将它们打包到层 .zip 文件中。这些文件存储在 `/opt` 中，并将文件夹路径加载到 `PATH` 环境变量中。

- `nodejs/node_modules`
- `nodejs/node`*`X`*`/node_modules`

例如，您在本教程中创建生成的层.zip 文件具有以下目录结构：

```
layer_content.zip
# nodejs
    # node20
        # node_modules
            # lodash
            # <other potential dependencies>
```

```
            # ...
```

您将把 lodash 库放在 nodejs/node20/node_modules 目录中。这可确保 Lambda 在函数调用期间可以找到该库。

打包层内容

在本示例中，您要将 lodash 库打包在一个层 .zip 文件中。完成以下步骤，安装并打包层内容。

安装并打包层内容

1. 克隆 GitHub 存储库中的 aws-lambda-developer-guide，其中包含 sample-apps/layer-nodejs 目录中需要的示例代码。

   ```
   git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
   ```

2. 导航到 layer-nodejs 示例应用程序的 layer 目录。此目录包含用于正确创建并打包层的脚本。

   ```
   cd aws-lambda-developer-guide/sample-apps/layer-nodejs/layer
   ```

3. 确保 package.json 文件列出 lodash。此文件定义了要包含在层中的依赖项。您可以更新此文件，纳入要包含在层中的任何依赖项。

   > **Note**
   >
   > 在您的依赖项上传到 Lambda 层后，此步骤中使用的 package.json 不会存储起来，也不会与依赖项一起使用。它仅在层打包过程中使用，不像 Node.js 应用程序或已发布包中的文件那样指定运行命令和兼容性。

4. 确保您拥有运行 layer 目录中脚本的 shell 权限。

   ```
   chmod 744 1-install.sh && chmod 744 2-package.sh
   ```

5. 使用以下命令运行 1-install.sh 脚本：

   ```
   ./1-install.sh
   ```

   此脚本将运行 npm install，它会读取您的 package.json 并下载其中定义的依赖项。

Example 1-install.sh

```
npm install .
```

6. 使用以下命令运行 2-package.sh 脚本：

```
./2-package.sh
```

此脚本将内容从 node_modules 目录复制到名为 nodejs/node20 的新目录中，随后将 nodejs 目录的内容压缩到一个名为 layer_content.zip 的文件中。这便是层的 .zip 文件。您可以解压缩文件，验证是否包含正确的文件结构，如 the section called “Node.js 运行时的层路径”部分所示。

Example 2-package.sh

```
mkdir -p nodejs/node20
cp -r node_modules nodejs/node20/
zip -r layer_content.zip nodejs
```

创建层

获取在上一部分中生成的 layer_content.zip 文件，将其作为 Lambda 层上传。您可以使用 AWS Management Console 上传层，也可以通过 AWS Command Line Interface（AWS CLI）使用 Lambda API 上传层。上传层 .zip 文件时，在以下 PublishLayerVersion AWS CLI 命令中，将 nodejs20.x 指定为兼容的运行时系统，并将 arm64 指定为兼容的架构。

```
aws lambda publish-layer-version --layer-name node-lodash-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes nodejs20.x \
    --compatible-architectures "arm64"
```

注意响应中的 LayerVersionArn，与 arn:aws:lambda:us-east-1:*123456789012*:layer:node-lodash-layer:1 类似。在本教程的下一步中，在将层添加到函数时，您要用到此 Amazon 资源名称（ARN）。

将层添加到函数

部署在函数代码中使用 `lodash` 库的示例 Lambda 函数，然后附加创建的层。要部署该函数，您需要一个执行角色。有关更多信息，请参阅 the section called “执行角色（函数访问其他资源的权限）”。如果目前没有执行角色，则按照可折叠部分中的步骤操作。若有，则跳至下一部分来部署函数。

（可选）创建执行角色

创建执行角色

1. 在 IAM 控制台中，打开 Roles（角色）页面。
2. 选择创建角色。
3. 创建具有以下属性的角色。
   - Trusted entity（可信任的实体）– Lambda。
   - Permissions（权限）– AWSLambdaBasicExecutionRole。
   - Role name（角色名称）– **`lambda-role`**。

   AWSLambdaBasicExecutionRole 策略具有函数将日志写入 CloudWatch Logs 所需的权限。

示例函数代码使用 lodash `_.findLastIndex` 方法来读取对象数组。它将对象与标准进行比较，以找到匹配项的索引。然后，它会返回 Lambda 响应中对象的索引和值。

```
import _ from "lodash"

export const handler = async (event) => {

  var users = [
  { 'user': 'Carlos',  'active': true },
  { 'user': 'Gil-dong',    'active': false },
  { 'user': 'Pat', 'active': false }
  ];

  let out = _.findLastIndex(users, function(o) { return o.user == 'Pat'; });
  const response = {
    statusCode: 200,
    body: JSON.stringify(out + ", " + users[out].user),
  };
  return response;
```

```
};
```

部署 Lambda 函数

1. 导航到 function/ 目录。如果当前在 layer/ 目录中，请运行以下命令：

```
cd ../function-js
```

2. 使用以下命令创建 .zip 文件部署包：

```
zip my_deployment_package.zip index.mjs
```

3. 部署函数。在以下 AWS CLI 命令中，将 --role 参数替换为执行角色 ARN：

```
aws lambda create-function --function-name nodejs_function_with_layer \
    --runtime nodejs20.x \
    --architectures "arm64" \
    --handler index.handler \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://my_deployment_package.zip
```

4. 将层附加到函数。在以下 AWS CLI 命令中，将 --layers 参数替换为之前记下的层版本 ARN：

```
aws lambda update-function-configuration --function-name nodejs_function_with_layer
 \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:nodejs-lodash-layer:1"
```

5. 调用函数，使用以下 AWS CLI 命令验证它是否可以正常工作：

```
aws lambda invoke --function-name nodejs_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{}' response.json
```

应看到类似如下内容的输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

输出 `response.json` 文件包含有关响应的详细信息。

（可选）清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除 Lambda 层

1. 打开 Lambda 控制台的 Layers page（层页面）。
2. 选择您创建的层。
3. 选择删除，然后再次选择删除。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **`delete`**，然后选择 Delete（删除）。

使用 Lambda 上下文对象检索 Node.js 函数信息

当 Lambda 运行您的函数时，它会将上下文对象传递到处理程序。此对象提供的方法和属性包含有关调用、函数和执行环境的信息。

上下文方法

- `getRemainingTimeInMillis()` – 返回执行超时前剩余的毫秒数。

上下文属性

- `functionName` – Lambda 函数的名称。
- `functionVersion` – 函数的版本
- `invokedFunctionArn` – 用于调用函数的 Amazon Resource Name (ARN)。表明调用者是否指定了版本号或别名。
- `memoryLimitInMB` – 为函数分配的内存量。
- `awsRequestId` – 调用请求的标识符。
- `logGroupName` – 函数的日志组。
- `logStreamName` – 函数实例的日志流。
- `identity` – （移动应用程序）有关授权请求的 Amazon Cognito 身份的信息。
  - `cognitoIdentityId` – 经过身份验证的 Amazon Cognito 身份。
  - `cognitoIdentityPoolId` – 授权调用的 Amazon Cognito 身份池。
- `clientContext` – （移动应用程序）客户端应用程序提供给 Lambda 的客户端上下文。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`
  - `env.make`
  - `env.model`
  - `env.locale`

  - `Custom` – 由客户应用程序设置的自定义值。
- `callbackWaitsForEmptyEventLoop` – 设置为 false 可在回调运行时立即发送响应，而不是等待 Node.js 事件循环成空。如果为 false，则任何未完成的事件将在下次调用期间继续运行。

以下示例函数记录了上下文信息并返回了日志的位置。

Example index.js 文件

```
exports.handler = async function(event, context) {
  console.log('Remaining time: ', context.getRemainingTimeInMillis())
  console.log('Function name: ', context.functionName)
  return context.logStreamName
}
```

Node.js Lambda 函数日志记录和监控

AWS Lambda 将代表您自动监控 Lambda 函数并将日志记录发送至 Amazon CloudWatch。您的 Lambda 函数带有一个 CloudWatch Logs 日志组以及函数的每个实例的日志流。Lambda 运行时环境会将每个调用的详细信息发送到日志流，然后中继函数代码的日志和其他输出。有关更多信息，请参阅将 CloudWatch Logs 日志与 Lambda 结合使用。

本页旨在介绍如何从 Lambda 函数的代码生成日志输出，并使用 AWS Command Line Interface、Lambda 控制台或 CloudWatch 控制台访问日志。

Sections

创建返回日志的函数

要从函数代码输出日志，您可以使用控制台对象的方法或使用写入到 `stdout` 或 `stderr` 的任何日志记录库。以下示例记录环境变量和事件对象的值。

> **Note**
>
> 建议在记录输入时使用输入验证和输出编码等技术。如果直接记录输入数据，攻击者可能会利用您的代码，让篡改难以检测、伪造日志条目或绕过日志监视器。有关更多信息，请参阅《Common Weakness Enumeration》中的 Improper Output Neutralization for Logs。

Example index.js 文件 – 日志记录

```
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.info("EVENT\n" + JSON.stringify(event, null, 2))
  console.warn("Event not processed.")
```

```
  return context.logStreamName
}
```

Example 日志格式

```
START RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Version: $LATEST
2019-06-07T19:11:20.562Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO ENVIRONMENT
 VARIABLES
{
  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/my-function",
  "AWS_LAMBDA_LOG_STREAM_NAME": "2019/06/07/[$LATEST]e6f4a0c4241adcd70c262d34c0bbc85c",
  "AWS_EXECUTION_ENV": "AWS_Lambda_nodejs12.x",
  "AWS_LAMBDA_FUNCTION_NAME": "my-function",
  "PATH": "/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin",
  "NODE_PATH": "/opt/nodejs/node10/node_modules:/opt/nodejs/node_modules:/var/runtime/
node_modules",
  ...
}
2019-06-07T19:11:20.563Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO EVENT
{
  "key": "value"
}
2019-06-07T19:11:20.564Z c793869b-ee49-115b-a5b6-4fd21e8dedac WARN Event not processed.
END RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac
REPORT RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Duration: 128.83 ms Billed
 Duration: 200 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 166.62 ms
 XRAY TraceId: 1-5d9d007f-0a8c7fd02xmpl480aed55ef0 SegmentId: 3d752xmpl1bbe37e Sampled:
 true
```

Node.js 运行时记录每次调用的 START、END 和 REPORT 行。它向函数记录的每个条目添加时间戳、请求 ID 和日志级别。报告行提供了以下详细信息。

REPORT 行数据字段

- RequestId – 调用的唯一请求 ID。
- Duration（持续时间）– 函数的处理程序方法处理事件所花费的时间。
- Billed Duration（计费持续时间）– 针对调用计费的时间量。
- Memory Size（内存大小）– 分配给函数的内存量。
- Max Memory Used（最大内存使用量）– 函数使用的内存量。如果调用共享执行环境，Lambda 会报告所有调用使用的最大内存。此行为可能会导致报告值高于预期。

- Init Duration（初始持续时间）– 对于提供的第一个请求，为运行时在处理程序方法外部加载函数和运行代码所花费的时间。
- XRAY TraceId – 对于追踪的请求，为 AWS X-Ray 追踪 ID。
- SegmentId – 对于追踪的请求，为 X-Ray 分段 ID。
- Sampled（采样）– 对于追踪的请求，为采样结果。

您可以在 Lambda 控制台中、在 CloudWatch Logs 控制台中或从命令行查看日志。

在 Node.js 中使用 Lambda 高级日志记录控件

为了让您更好地控制如何捕获、处理和使用函数日志，您可以为支持的 Node.js 运行时系统配置以下日志记录选项：

- 日志格式 - 为函数日志选择纯文本或结构化的 JSON 格式
- 日志级别 - 对于 JSON 格式的日志，选择 Lambda 发送到 Amazon CloudWatch 的日志的详细信息级别，例如 ERROR、DEBUG 或 INFO
- 日志组 - 选择您的函数发送日志的目标 CloudWatch 日志组

有关这些日志记录选项的更多信息以及如何通过配置来使用函数的说明，请参阅 the section called “配置函数日志”。

要在 Node.js Lambda 函数中使用日志格式和日志级别选项，请参阅以下各节中的指南。

在 Node.js 中使用结构化的 JSON 日志

如果您为函数的日志格式选择 JSON，Lambda 将使用 `console.trace`、`console.debug`、`console.log`、`console.info`、`console.error` 和 `console.warn` 的控制台方法将日志输出作为结构化 JSON 发送到 CloudWatch。每个 JSON 日志对象包含至少四个键值对和以下键：

- `"timestamp"` - 生成日志消息的时间
- `"level"` - 分配给消息的日志级别
- `"message"` - 日志消息的内容
- `"requestId"` - 函数调用的唯一请求 ID

根据函数使用的日志记录方法，此 JSON 对象还可能包含其他密钥对。例如，如果函数通过 `console` 方法使用多个参数记录错误对象，则 JSON 对象将包含带有键 `errorMessage`、`errorType`、和 `stackTrace` 的额外键值对。

如果您的代码已经使用其他日志记录库（例如 Powertools for AWS Lambda）来生成 JSON 结构化日志，则无需进行任何更改。Lambda 不会对任何已采用 JSON 编码的日志进行双重编码，因此仍将像以前一样捕获函数的应用程序日志。

有关使用 Powertools for AWS Lambda 日志记录包在 Node.js 运行时系统中创建 JSON 结构化日志的更多信息，请参阅 the section called “日志记录”。

JSON 格式的日志输出示例

以下示例显示了当您将函数的日志格式设置为 JSON 时，如何在 CloudWatch Logs 中捕获使用具有单个和多个参数的 `console` 方法生成的各种日志输出。

第一个示例使用 `console.error` 方法输出一个简单的字符串。

Example Node.js 日志记录代码

```
export const handler = async (event) => {
  console.error("This is a warning message");
  ...
}
```

Example JSON 日志记录

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"ERROR",
    "message":"This is a warning message",
    "requestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

您还可以使用 `console` 方法中的单个或多个参数输出更复杂的结构化日志消息。在下一个示例中，您可以使用 `console.log` 的单个参数输出两个键值对。请注意，Lambda 发送到 CloudWatch Logs 的 JSON 对象中的 `"message"` 字段未进行字符串化。

Example Node.js 日志记录代码

```
export const handler = async (event) => {
```

```
  console.log({data: 12.3, flag: false});
  ...
}
```

Example JSON 日志记录

```
{
    "timestamp": "2023-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "data": 12.3,
        "flag": false
    }
}
```

在下一个示例中，您可以再次使用 `console.log` 方法创建日志输出。这次，该方法采用两个参数，即一个包含两个键值对的映射和一个标识字符串。请注意，在本例中，由于您提供了两个参数，因此 Lambda 会对 `"message"` 字段进行字符串化。

Example Node.js 日志记录代码

```
export const handler = async (event) => {
  console.log('Some object - ', {data: 12.3, flag: false});
  ...
}
```

Example JSON 日志记录

```
{
    "timestamp": "2023-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "Some object -  { data: 12.3, flag: false }"
}
```

Lambda 为使用 `console.log` 生成的输出分配日志级别 INFO。

最后一个示例说明了如何使用 `console` 方法将错误对象输出到 CloudWatch Logs。请注意，当您使用多个参数记录错误对象时，Lambda 会将字段 `errorMessage`、`errorType` 和 `stackTrace` 添加到日志输出中。

Example Node.js 日志记录代码

```
export const handler = async (event) => {
  let e1 = new ReferenceError("some reference error");
  let e2 = new SyntaxError("some syntax error");
  console.log(e1);
  console.log("errors logged - ", e1, e2);
};
```

Example JSON 日志记录

```
{
    "timestamp": "2023-12-08T23:21:04.632Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "errorType": "ReferenceError",
        "errorMessage": "some reference error",
        "stackTrace": [
            "ReferenceError: some reference error",
            "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
            "    at Runtime.handleOnceNonStreaming (file:///var/runtime/
index.mjs:1173:29)"
        ]
    }
}

{
    "timestamp": "2023-12-08T23:21:04.646Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "errors logged -  ReferenceError: some reference error
\n    at Runtime.handler (file:///var/task/index.mjs:3:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29) SyntaxError: some syntax
 error\n    at Runtime.handler (file:///var/task/index.mjs:4:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29)",
    "errorType": "ReferenceError",
    "errorMessage": "some reference error",
    "stackTrace": [
        "ReferenceError: some reference error",
        "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
```

```
        "    at Runtime.handleOnceNonStreaming (file:///var/runtime/index.mjs:1173:29)"
    ]
}
```

记录多种错误类型时，会从提供给 `console` 方法的第一个错误类型中提取额外的 `errorMessage`、`errorType` 和 `stackTrace` 字段。

使用带有结构化的 JSON 日志的嵌入式指标格式（EMF）客户端库

AWS 提供了用于 Node.js 的开源客户端库，可供您用来创建嵌入式指标格式（EMF）日志。如果您有使用这些库的现有函数，并且将函数的日志格式更改为 JSON，则 CloudWatch 可能无法再识别您的代码发出的指标。

如果您的代码当前直接使用 `console.log` 或使用 Powertools for AWS Lambda（TypeScript）发出 EMF 日志，若将函数的日志格式更改为 JSON，CloudWatch 也将无法解析这些日志。

> **Important**
>
> 为确保 CloudWatch 继续正确解析函数的 EMF 日志，请将库的 EMF 和 Powertools for AWS Lambda 更新到最新版本。如果切换到 JSON 日志格式，我们还建议您进行测试以确保与函数的嵌入式指标兼容。如果您的代码直接使用 `console.log` 发出 EMF 日志，请更改您的代码以将这些指标直接输出到 `stdout`，如以下示例代码所示。

Example 向 **`stdout`** 发送嵌入式指标的代码

```
process.stdout.write(JSON.stringify(
    {
        "_aws": {
            "Timestamp": Date.now(),
            "CloudWatchMetrics": [{
                "Namespace": "lambda-function-metrics",
                "Dimensions": [["functionVersion"]],
                "Metrics": [{
                    "Name": "time",
                    "Unit": "Milliseconds",
                    "StorageResolution": 60
                }]
            }]
        },
        "functionVersion": "$LATEST",
```

```
        "time": 100,
        "requestId": context.awsRequestId
    }
) + "\n")
```

在 Node.js 中使用日志级别筛选

为了让 AWS Lambda 根据日志级别筛选应用程序日志，您的函数必须使用 JSON 格式的日志。您可以通过两种方式实现这一点：

- 使用标准控制台方法创建日志输出，并将您的函数配置为使用 JSON 日志格式。然后 AWS Lambda 使用 the section called “在 Node.js 中使用结构化的 JSON 日志” 中所述 JSON 对象中的“级别”键值对筛选日志输出。要了解如何配置函数的日志格式，请参阅 the section called “配置函数日志”。
- 使用其他日志记录库或方法在代码中创建 JSON 结构化日志，其中包含定义日志输出级别的“级别”键值对。例如，您可以使用 Powertools for AWS Lambda 通过代码生成 JSON 结构化日志输出。要了解有关在 Node.js 运行时系统中使用 Powertools 的更多信息，请参阅 the section called “日志记录”。

  要让 Lambda 筛选函数的日志，还必须在 JSON 日志输出中包含一个 `"timestamp"` 键值对。必须以有效的 RFC 3339 时间戳格式指定时间。如果您未提供有效的时间戳，Lambda 将为日志分配 INFO 级别并为您添加时间戳。

将函数配置为使用日志级别筛选时，您可以从以下选项中选择希望 AWS Lambda 发送到 CloudWatch Logs 的日志级别：

日志级别	标准使用情况
TRACE（最详细）	用于跟踪代码执行路径的最精细信息
调试	系统调试的详细信息
信息	记录函数正常运行情况的消息
警告	有关潜在错误的消息，如果不加以解决，这些错误可能会导致意外行为
错误	有关会阻碍代码按预期执行的问题的消息
FATAL（最简略）	有关导致应用程序停止运行的严重错误的消息

Lambda 仅将选定级别及更低级别的系统日志发送到 CloudWatch。例如，如果您将日志级别配置为 WARN，Lambda 将发送与 WARN、ERROR 和 FATAL 级别相对应的日志。

在 Lambda 控制台中查看日志

调用 Lambda 函数后，您可以使用 Lambda 控制台查看日志输出。

如果可以在嵌入式代码编辑器中测试代码，则可以在执行结果中找到日志。使用控制台测试功能调用函数时，可以在详细信息部分找到日志输出。

在 CloudWatch 控制台中查看日志

您可以使用 Amazon CloudWatch 控制台查看所有 Lambda 函数调用的日志。

使用 CloudWatch 控制台查看日志

1. 打开 CloudWatch 控制台的 Log groups（日志组页面）。
2. 选择您的函数 (/aws/lambda/***your-function-name***) 的日志组。
3. 创建日志流。

每个日志流对应一个函数实例。日志流会在您更新 Lambda 函数以及创建更多实例来处理多个并发调用时显示。要查找特定调用的日志，建议您使用 AWS X-Ray 检测函数。X-Ray 会在追踪中记录有关请求和日志流的详细信息。

使用 AWS Command Line Interface（AWS CLI）查看日志

AWS CLI 是一种开源工具，让您能够在命令行 Shell 中使用命令与 AWS 服务进行交互。要完成本节中的步骤，您必须拥有 AWS CLI 版本 2。

您可以通过 AWS CLI，使用 `--log-type` 命令选项检索调用的日志。响应包含一个 `LogResult` 字段，其中包含多达 4KB 来自调用的 base64 编码日志。

Example 检索日志 ID

以下示例说明如何从 `LogResult` 字段中检索名为 `my-function` 的函数的日志 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您应看到以下输出：

```
{
```

```
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解码日志

在同一命令提示符下，使用 base64 实用程序解码日志。以下示例说明如何为 my-function 检索 base64 编码的日志。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

您应看到以下输出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64 实用程序在 Linux、macOS 和 Ubuntu on Windows 上可用。macOS 用户可能需要使用 base64 -D。

Example get-logs.sh 脚本

在同一命令提示符下，使用以下脚本下载最后五个日志事件。此脚本使用 sed 从输出文件中删除引号，并休眠 15 秒以等待日志可用。输出包括来自 Lambda 的响应，以及来自 get-log-events 命令的输出。

复制以下代码示例的内容并将其作为 get-logs.sh 保存在 Lambda 项目目录中。

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux（仅限）

在同一命令提示符下，macOS 和 Linux 用户可能需要运行以下命令以确保脚本可执行。

```
chmod -R 755 get-logs.sh
```

Example 检索最后五个日志事件

在同一命令提示符下，运行以下脚本以获取最后五个日志事件。

```
./get-logs.sh
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
```

```
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

删除日志

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期之后，日志将自动删除。

在 AWS Lambda 中检测 Node.js 代码

Lambda 与 AWS X-Ray 集成，以帮助您跟踪、调试和优化 Lambda 应用程序。您可以在某个请求遍历应用程序中的资源（其中可能包括 Lambda 函数和其他 AWS 服务）时，使用 X-Ray 跟踪该请求。

要将跟踪数据发送到 X-Ray，您可以使用以下两个软件开发工具包 (SDK) 库之一：

- 适用于 OpenTelemetry 的 AWS 发行版 (ADOT) – 一种安全、可供生产、支持 AWS 的 OpenTelemetry (OTel) SDK 的分发版本。
- AWS X-Ray SDK for Node.js – 用于生成跟踪数据并将其发送到 X-Ray 的 SDK

每个开发工具包均提供了将遥测数据发送到 X-Ray 服务的方法。然后，您可以使用 X-Ray 查看、筛选和获得对应用程序性能指标的洞察，从而发现问题和优化机会。

> **Important**
>
> X-Ray 和 Powertools for AWS Lambda SDK 是 AWS 提供的紧密集成的分析解决方案的一部分。ADOT Lambda Layers 是全行业通用的跟踪分析标准的一部分，该标准通常会收集更多数据，但可能不适用于所有使用案例。您可以使用任一解决方案在 X-Ray 中实现端到端跟踪。要了解有关如何在两者之间进行选择的更多信息，请参阅在 AWS Distro for Open Telemetry 和 X-Ray 开发工具包之间进行选择。

Sections

使用 ADOT 分析您的 Node.js 函数

ADOT 提供完全托管式 Lambda 层，这些层使用 OTel SDK，将收集遥测数据所需的一切内容打包起来。通过使用此层，您可以在不必修改任何函数代码的情况下，对您的 Lambda 函数进行分析。您

还可以将您的层配置为对 OTel 进行自定义初始化。有关更多信息，请参阅 ADOT 文档中的适用于 Lambda 上的 ADOT 收集器的自定义配置。

对于 Node.js 运行时，可以添加 适用于 ADOT Javascript 的 AWS 托管 Lambda 层以自动分析您的函数。有关如何添加此层的详细说明，请参阅 ADOT 文档中的 AWS Distro for OpenTelemetry Lambda 对 JavaScript 的支持。

使用 X-Ray SDK 分析您的 Node.js 函数

要记录有关您的 Lambda 函数对应用程序中的其他资源进行调用的详细信息，您还可以使用 AWS X-Ray SDK for Node.js。要获取开发工具包，请将 `aws-xray-sdk-core` 包添加到应用程序的依赖项中。

Example blank-nodejs/package.json

```
{
  "name": "blank-nodejs",
  "version": "1.0.0",
  "private": true,
  "devDependencies": {
    "jest": "29.7.0"
  },
  "dependencies": {
    "@aws-sdk/client-lambda": "3.345.0",
    "aws-xray-sdk-core": "3.5.3"
  },
  "scripts": {
    "test": "jest"
  }
}
```

要在 AWS SDK for JavaScript v3 中检测 AWS SDK 客户端，请使用 `captureAWSv3Client` 方法封装客户端实例。

Example blank-nodejs/function/index.js – 跟踪AWS开发工具包客户端

```
const AWSXRay = require('aws-xray-sdk-core');
const { LambdaClient, GetAccountSettingsCommand } = require('@aws-sdk/client-lambda');

// Create client outside of handler to reuse
const lambda = AWSXRay.captureAWSv3Client(new LambdaClient());
```

```
// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
  ...
```

Lambda 运行时设置一些环境变量来配置 X-Ray 开发工具包。例如，Lambda 将 AWS_XRAY_CONTEXT_MISSING 设置为 LOG_ERROR，以避免从 X-Ray 开发工具包引发运行时错误。要设置自定义上下文缺失策略，请覆盖函数配置中的环境变量以使其没有值，然后可以通过编程方式设置上下文缺失策略。

Example 初始化代码示例

```
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

有关更多信息，请参阅 the section called “配置环境变量”。

在添加正确的依赖项并进行必要的代码更改后，请通过 Lambda 控制台或 API 激活函数配置中的跟踪。

使用 Lambda 控制台激活跟踪

要使用控制台切换 Lambda 函数的活动跟踪，请按照以下步骤操作：

打开活跃跟踪

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration（配置），然后选择 Monitoring and operations tools（监控和操作工具）。
4. 选择编辑。
5. 在 X-Ray 下方，开启 Active tracing（活动跟踪）。
6. 选择保存。

使用 Lambda API 激活跟踪

借助 AWS CLI 或 AWS SDK 在 Lambda 函数上配置跟踪，请使用以下 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下示例 AWS CLI 命令对名为 my-function 的函数启用活跃跟踪。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

跟踪模式是发布函数版本时版本特定配置的一部分。您无法更改已发布版本上的跟踪模式。

使用 AWS CloudFormation 激活跟踪

要对 AWS CloudFormation 模板中的 `AWS::Lambda::Function` 资源激活跟踪，请使用 `TracingConfig` 属性。

Example function-inline.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

对于 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 资源，请使用 `Tracing` 属性。

Example template.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

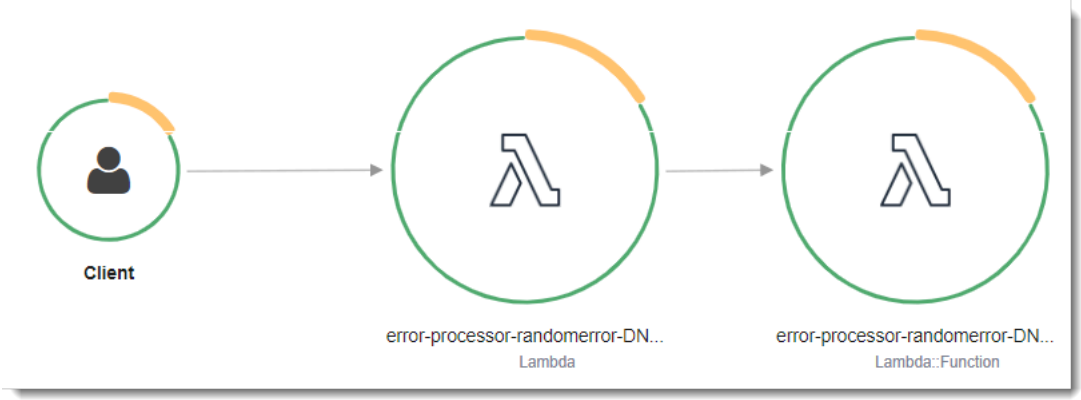
解释 X-Ray 跟踪

您的函数需要权限才能将跟踪数据上载到 X-Ray。在 Lambda 控制台中激活跟踪后，Lambda 会将所需权限添加到函数的执行角色。如果没有，请将 AWSXRayDaemonWriteAccess 策略添加到执行角色。

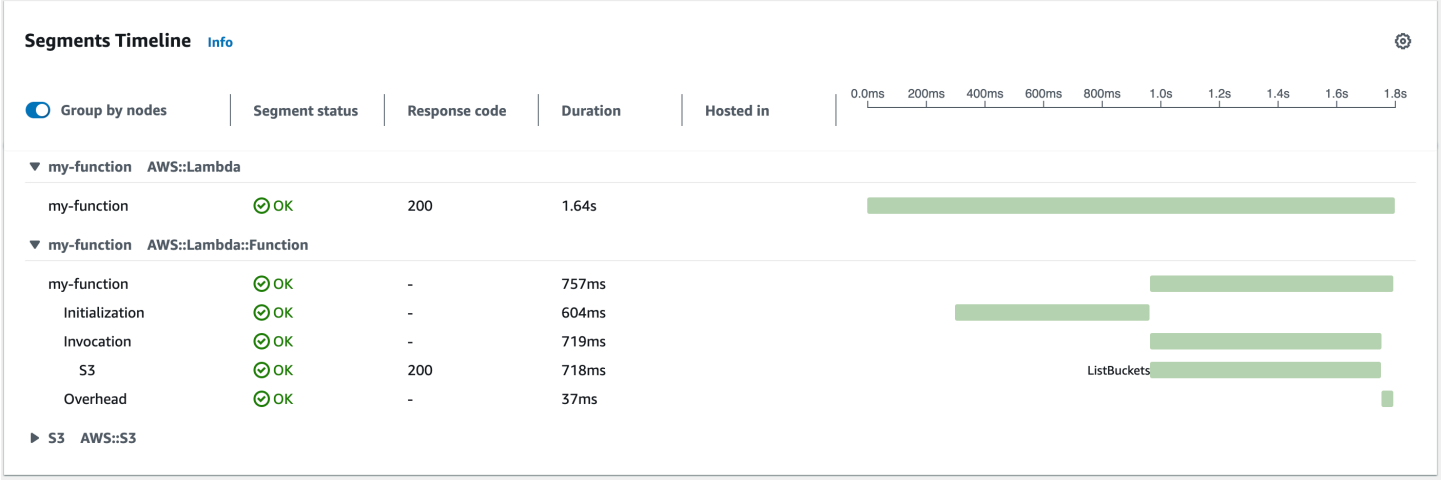
在配置活跃跟踪后，您可以通过应用程序观察特定请求。X-Ray 服务图将显示有关应用程序及其所有组件的信息。以下示例显示了具有两个函数的应用程序。主函数处理事件，有时会返回错误。位于顶部的第二个函数将处理第一个函数的日志组中显示的错误，并使用 AWS SDK 调用 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

X-Ray 无法跟踪对应用程序的所有请求。X-Ray 将应用采样算法确保跟踪有效，同时仍会提供所有请求的一个代表性样本。采样率是每秒 1 个请求和 5% 的其他请求。您无法为函数配置此 X-Ray 采样率。

在 X-Ray 中，跟踪记录有关由一个或多个服务处理的请求的信息。Lambda 会每个跟踪记录 2 个分段，这些分段将在服务图上创建两个节点。下图突出显示了这两个节点：



位于左侧的第一个节点表示接收调用请求的 Lambda 服务。第二个节点表示特定的 Lambda 函数。以下示例显示了一个包含这 2 个分段的跟踪。两者都命名为 my-function，但其中一个函数具有 AWS::Lambda 源，另一个则具有 AWS::Lambda::Function 源。如果 AWS::Lambda 分段显示错误，则表示 Lambda 服务存在问题。如果 AWS::Lambda::Function 分段显示错误，则说明函数存在问题。



此示例将展开 AWS::Lambda::Function 分段，以显示其三个子分段。

> **Note**
>
> AWS 目前正在实施对 Lambda 服务的更改。由于这些更改，您可能会看到 AWS 账户 中不同 Lambda 函数发出的系统日志消息和跟踪分段的结构和内容之间存在细微差异。
> 此处显示的示例跟踪说明了旧样式函数分段。以下段落介绍了新旧样式分段之间的差异。
> 这些更改将在未来几周内实施，除中国和 GovCloud 区域外，所有 AWS 区域 的函数都将过渡到使用新格式的日志消息和跟踪分段。

旧样式函数分段包含以下子分段：

- 初始化 – 表示加载函数和运行初始化代码所花费的时间。此子分段仅对由您的函数的每个实例处理的第一个事件显示。
- 调用 – 表示执行处理程序代码花费的时间。
- 开销 – 表示 Lambda 运行时为准备处理下一个事件而花费的时间。

新样式函数分段不包含 Invocation 子分段。而是将客户子分段直接附加到函数分段。有关新旧样式函数分段结构的更多信息，请参阅 the section called “了解 X-Ray 跟踪”。

您还可以分析 HTTP 客户端、记录 SQL 查询以及使用注释和元数据创建自定义子段。有关更多信息，请参阅 AWS X-Ray 开发人员指南中的 AWS X-Ray SDK for Node.js。

> **定价**
>
> 作为 AWS 免费套餐的组成部分，您可以每月免费使用 X-Ray 跟踪，但不能超过一定限制。超出该阈值后，X-Ray 会对跟踪存储和检索进行收费。有关更多信息，请参阅 AWS X-Ray 定价。

在层中存储运行时依赖项 (X-Ray SDK)

如果您使用 X-Ray 开发工具包来分析AWS开发工具包客户端和您的函数代码，则您的部署程序包可能会变得相当大。为了避免每次更新函数代码时上载运行时依赖项，请将 X-Ray SDK 打包到 Lambda 层中。

以下示例显示存储 AWS X-Ray SDK for Node.js 的 `AWS::Serverless::LayerVersion` 资源。

Example template.yml – 依赖项层

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-nodejs-lib
      Description: Dependencies for the blank sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - nodejs20.x
```

使用此配置，仅在更改运行时依赖项时您才会更新库层。由于函数部署软件包仅包含您的代码，因此可以帮助缩短上传时间。

为依赖项创建层需要更改构建才能在部署之前生成层存档。有关工作示例，请参阅 blank-nodejs 示例应用程序。

使用 TypeScript 构建 Lambda 函数

你可以使用 Node.js 运行时在 AWS Lambda 中运行 TypeScript 代码。由于 Node.js 不会在本机运行 TypeScript 代码，因此必须首先将 TypeScript 代码转换为 JavaScript。然后，可以使用 JavaScript 文件将您的函数代码部署到 Lambda。您的代码将在包含适用于 JavaScript 的 AWS SDK 的环境中运行，其中包含来自您管理的 AWS Identity and Access Management (IAM) 角色的凭证。要了解有关 Node.js 运行时随附的 SDK 版本的更多信息，请参阅 the section called “包含运行时的 SDK 版本”。

Lambda 支持以下 Node.js 运行时。

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
Node.js 20	`nodejs20.x`	Amazon Linux 2023	未计划	未计划	未计划
Node.js 18	`nodejs18.x`	Amazon Linux 2	2025 年 7 月 31 日	2025 年 9 月 1 日	2025 年 10 月 1 日

主题

- 设置 TypeScript 开发环境
- 定义采用 TypeScript 的 Lambda 函数处理程序
- 使用 .zip 文件归档部署在 Lambda 中转换的 TypeScript 代码
- 使用容器镜像在 Lambda 中部署转换后的 TypeScript 代码
- 使用 TypeScript Lambda 函数的层
- 使用 Lambda 上下文对象检索 TypeScript 函数信息
- TypeScript Lambda 函数日志记录和监控
- 追踪 AWS Lambda 中的 TypeScript 代码

设置 TypeScript 开发环境

您可以使用本地集成开发环境 (IDE)、文本编辑器或 AWS Cloud9 来编写 TypeScript 函数代码。您无法在 Lambda 控制台上创建 TypeScript 代码。

要转换您的 TypeScript 代码，请设置一个编译器，例如 esbuild 或者 Microsoft 的 TypeScript 编译器 (`tsc`)，后者与 TypeScript 分发版捆绑在一起。您可以使用 AWS Serverless Application Model (AWS

SAM) 或 AWS Cloud Development Kit (AWS CDK) 来简化 TypeScript 代码的构建和部署。两种工具都使用 esbuild 将 TypeScript 代码转换为 JavaScript。

在使用 esbuild 时，请注意以下事项：

- 有几个 TypeScript 注意事项。
- 您必须配置 TypeScript 转换设置，以匹配您计划使用的 Node.js 运行时。有关更多信息，请参阅 esbuild 文档中的目标。有关演示如何将 Lambda 支持的特定 Node.js 版本设为目标的 tsconfig.json 文件示例，请参阅 TypeScript GitHub 存储库。
- esbuild 不执行类型检查。要检查类型，请使用 `tsc` 编译器。运行 `tsc -noEmit` 或将 `"noEmit"` 参数添加到您的 tsconfig.json 文件，如以下示例中所示。这会将 `tsc` 配置为不发出 JavaScript 文件。在检查类型后，使用 esbuild 将 TypeScript 文件转换为 JavaScript。

Example tsconfig.json

```
{
 "compilerOptions": {
   "target": "es2020",
   "strict": true,
   "preserveConstEnums": true,
   "noEmit": true,
   "sourceMap": false,
   "module":"commonjs",
   "moduleResolution":"node",
   "esModuleInterop": true,
   "skipLibCheck": true,
   "forceConsistentCasingInFileNames": true,
   "isolatedModules": true,
 },
 "exclude": ["node_modules", "**/*.test.ts"]
}
```

定义采用 TypeScript 的 Lambda 函数处理程序

Lambda 函数处理程序是函数代码中处理事件的方法。当调用函数时，Lambda 运行处理程序方法。您的函数会一直运行，直到处理程序返回响应、退出或超时。

主题

Tescript 处理程序基础知识

Example TypeScript 处理程序

此示例函数记录事件对象的内容并返回日志的位置。请注意以下几点：

- 在 Lambda 函数中使用此代码之前，必须添加 @types/aws-lambda 程序包作为开发依赖项。此程序包包含 Lambda 的类型定义。安装 `@types/aws-lambda` 时，`import` 语句 (`import ... from 'aws-lambda'`) 会导入类型定义。它不导入 `aws-lambda` NPM 程序包，这是一个无关的第三方工具。有关更多信息，请参阅 DefinitelyTyped GitHub 存储库中的 aws-lambda。
- 本示例中的处理程序是 ES 模块，必须位于 `package.json` 文件中或使用 `.mjs` 文件扩展名进行指定。有关详细信息，请参阅 将函数处理程序指定为 ES 模块。

```
import { Handler } from 'aws-lambda';

export const handler: Handler = async (event, context) => {
    console.log('EVENT: \n' + JSON.stringify(event, null, 2));
    return context.logStreamName;
};
```

运行时会将参数传递到处理程序方法。第一个参数是 `event` 对象，它包含来自调用方的信息。调用程序在调用 Invoke 时将该信息作为 JSON 格式字符串传递，运行时将它转换为对象。当 AWS 服务调用您的函数时，事件结构因服务而异。对于 TypeScript，建议对事件对象使用类型注释。有关更多信息，请参阅 将类型用于事件对象。

第二个参数是 context 对象，它包含有关调用、函数和执行环境的信息。在前面的示例中，函数将从 context 对象获取日志流的名称，然后将其返回给调用方。

您也可以使用回调参数，这是一种可在非异步处理程序中进行调用以发送响应的函数。我们建议您使用 Async/await（异步/等待），而不使用 Callback（回调）。Async/await 具有更好的易读性、错误处理能力和效率。有关 Async/await 与 Callback 之间差异的更多信息，请参阅 使用回调。

使用异步/等待

如果您的代码执行异步任务，则使用 Async/await 模式来确保处理程序会完成运行。Async/await 是一种简洁、易读的 Node.js 异步代码编写方式，无需嵌套回调或链式承诺。使用 Async/await 时，您编写的代码看起来与同步代码类似，同时仍然是异步和非阻止式的。

`async` 关键字会将函数标记为异步，`await` 关键字会暂停函数的执行，直到 `Promise` 完成解析为止。

Example TypeScript 函数 - 异步

此示例使用 `fetch`，它支持 `nodejs18.x` 运行时。请注意以下几点：

- 在 Lambda 函数中使用此代码之前，必须添加 @types/aws-lambda 程序包作为开发依赖项。此程序包包含 Lambda 的类型定义。安装 `@types/aws-lambda` 时，`import` 语句 (`import ... from 'aws-lambda'`) 会导入类型定义。它不导入 `aws-lambda` NPM 程序包，这是一个无关的第三方工具。有关更多信息，请参阅 DefinitelyTyped GitHub 存储库中的 aws-lambda。
- 本示例中的处理程序是 ES 模块，必须位于 `package.json` 文件中或使用 `.mjs` 文件扩展名进行指定。有关详细信息，请参阅 将函数处理程序指定为 ES 模块。

```
import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
const url = 'https://aws.amazon.com/';
export const lambdaHandler = async (event: APIGatewayProxyEvent):
 Promise<APIGatewayProxyResult> => {
    try {
        // fetch is available with Node.js 18
        const res = await fetch(url);
        return {
            statusCode: res.status,
            body: JSON.stringify({
                message: await res.text(),
            }),
        };
    } catch (err) {
```

```
        console.log(err);
        return {
            statusCode: 500,
            body: JSON.stringify({
                message: 'some error happened',
            }),
        };
    }
};
```

使用回调

我们建议您使用 Async/await 来声明函数处理程序，而不是使用回调。Async/await 是更好的选择，原因有以下几点：

- 易读性：Async/await 代码比回调代码更易阅读和理解，回调代码很快就会变得难以理解，从而导致回调失败。
- 调试和错误处理：基于回调的代码可能非常难以调试。调用堆栈可能变得难以理解，并且很容易出现错误。使用 Async/await 时，您可以使用 try/catch 块来处理错误。
- 效率：回调通常需要在代码的不同部分之间切换。Async/await 可以减少上下文切换的数量，从而提高代码效率。

当您在处理程序中使用回调时，函数会一直执行，直到 事件循环 为空或函数超时为止。在完成所有事件循环任务之前，不会将响应发送给调用方。如果函数超时，则会返回 error。可以通过将 context.callbackWaitsForEmptyEventLoop 设置为 false，从而将运行时配置为立即发送响应。

callback 函数有两个参数：一个是 `Error`，一个是响应。响应对象必须与 `JSON.stringify` 兼容。

Example 包含回调的 TypeScript 函数

此示例函数接收来自 Amazon API Gateway 的事件、记录事件和上下文对象，然后向 API Gateway 返回响应。请注意以下几点：

- 在 Lambda 函数中使用此代码之前，必须添加 @types/aws-lambda 程序包作为开发依赖项。此程序包包含 Lambda 的类型定义。安装 `@types/aws-lambda` 时，`import` 语句 (`import ... from 'aws-lambda'`) 会导入类型定义。它不导入 `aws-lambda` NPM 程序包，这是一个无关的第三方工具。有关更多信息，请参阅 DefinitelyTyped GitHub 存储库中的 aws-lambda。
- 本示例中的处理程序是 ES 模块，必须位于 `package.json` 文件中或使用 `.mjs` 文件扩展名进行指定。有关详细信息，请参阅 将函数处理程序指定为 ES 模块。

```
import { Context, APIGatewayProxyCallback, APIGatewayEvent } from 'aws-lambda';

export const lambdaHandler = (event: APIGatewayEvent, context: Context, callback:
 APIGatewayProxyCallback): void => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    callback(null, {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    });
};
```

将类型用于事件对象

我们建议您不要将任何类型用于处理程序参数和返回类型，因为您将失去检查类型的能力。相反，请使用 sam local generate-event AWS Serverless Application Model CLI 命令或者使用来自 @types/aws-lambda 软件包的开源定义生成事件。

使用 sam local generate-event 命令生成事件

1. 生成 Amazon Simple Storage Service (Amazon S3) 代理事件。

   ```
   sam local generate-event s3 put >> S3PutEvent.json
   ```

2. 使用 quicktype 实用工具从 S3PutEvent.json 文件生成类型定义。

   ```
   npm install -g quicktype
   quicktype S3PutEvent.json -o S3PutEvent.ts
   ```

3. 在您的代码中使用生成的类型。

   ```
   import { S3PutEvent } from './S3PutEvent';

   export const lambdaHandler = async (event: S3PutEvent): Promise<void> => {
     event.Records.map((record) => console.log(record.s3.object.key));
   };
   ```

使用来自 @types/aws-lambda 软件包的开源定义生成事件

1. 添加 @types/aws-lambda 软件包作为开发依赖项。

```
npm install -D @types/aws-lambda
```

2. 在您的代码中使用这些类型。

```
import { S3Event } from "aws-lambda";

export const lambdaHandler = async (event: S3Event): Promise<void> => {
  event.Records.map((record) => console.log(record.s3.object.key));
};
```

Typescript Lambda 函数的代码最佳实践

在构建 Lambda 函数时，请遵循以下列表中的指南，采用最佳编码实践：

- 从核心逻辑中分离 Lambda 处理程序。这样您可以创建更容易进行单元测试的函数。在 Node.js 中可能如下所示：

```
exports.myHandler = function(event, context, callback) {
 var foo = event.foo;
 var bar = event.bar;
 var result = MyLambdaFunction (foo, bar);

 callback(null, result);
}

function MyLambdaFunction (foo, bar) {
 // MyLambdaFunction logic here
}
```

- 控制函数部署程序包中的依赖关系。AWS Lambda 执行环境包含许多库。对于 Node.js 和 Python 运行时，其中包括 AWS SDK。Lambda 会定期更新这些库，以支持最新的功能组合和安全更新。这些更新可能会使 Lambda 函数的行为发生细微变化。要完全控制您的函数所用的依赖项，请使用部署程序包来打包所有依赖项。
- 将依赖关系的复杂性降至最低。首选在执行环境启动时可以快速加载的更简单的框架。
- 将部署程序包大小精简为只包含运行时必要的部分。这样会减少调用前下载和解压缩部署程序包所需的时间。

- 利用执行环境重用来提高函数性能。连接软件开发工具包 (SDK) 客户端和函数处理程序之外的数据库，并在 `/tmp` 目录中本地缓存静态资产。由函数的同一实例处理的后续调用可重用这些资源。这样就可以通过缩短函数运行时间来节省成本。

  为了避免调用之间潜在的数据泄露，请不要使用执行环境来存储用户数据、事件或其他具有安全影响的信息。如果您的函数依赖于无法存储在处理程序的内存中的可变状态，请考虑为每个用户创建单独的函数或单独的函数版本。
- 使用 keep-alive 指令来维护持久连接。Lambda 会随着时间的推移清除空闲连接。在调用函数时尝试重用空闲连接会导致连接错误。要维护您的持久连接，请使用与运行时关联的 keep-alive 指令。有关示例，请参阅在 Node.js 中通过 Keep-Alive 重用连接。
- 使用环境变量将操作参数传递给函数。例如，您在写入 Amazon S3 存储桶时，不应对要写入的存储桶名称进行硬编码，而应将存储桶名称配置为环境变量。
- 避免在 Lambda 函数中使用递归调用，在这种情况下，函数会调用自己或启动可能再次调用该函数的进程。这可能会导致意想不到的函数调用量和升级成本。如果您看到意外的调用量，请立即将函数保留并发设置为 `0` 来限制对函数的所有调用，同时更新代码。
- Lambda 函数代码中不要使用非正式的非公有 API。对于 AWS Lambda 托管式运行时，Lambda 会定期为 Lambda 的内部 API 应用安全性和功能更新。这些内部 API 更新可能不能向后兼容，会导致意外后果，例如，假设您的函数依赖于这些非公有 API，则调用会失败。请参阅 API 参考以查看公开发布的 API 列表。
- 编写幂等代码。为您的函数编写幂等代码可确保以相同的方式处理重复事件。您的代码应该正确验证事件并优雅地处理重复事件。有关更多信息，请参阅如何使我的 Lambda 函数具有幂等性？。

使用 .zip 文件归档部署在 Lambda 中转换的 TypeScript 代码

在可以将 TypeScript 代码部署到 AWS Lambda 之前，您需要将其转换为 JavaScript。本页介绍了使用 .zip 文件存档构建 TypeScript 代码并将其部署到 Lambda 的三种方法：

- 使用 AWS Serverless Application Model（AWS SAM）
- 使用 AWS Cloud Development Kit (AWS CDK)
- 使用 AWS Command Line Interface (AWS CLI) 和 esbuild

AWS SAM 和 AWS CDK 可以简化 TypeScript 函数的构建和部署。AWS SAM 模板规范提供了一种简单而干净的语法，用于描述构成无服务器应用程序的 Lambda 函数、API、权限、配置和事件。AWS CDK 使您能够借助编程语言的强大表达能力，在云中构建可靠、可扩展且成本高效的应用程序。AWS CDK 适合中度到高度经验的 AWS 用户。AWS CDK 和 AWS SAM 两者均使用 esbuild 将 TypeScript 代码转换为 JavaScript。

使用 AWS SAM 将 TypeScript 代码部署到 Lambda

请按照以下步骤使用 AWS SAM 来下载、构建和部署示例 Hello World TypeScript 应用程序。此应用程序实现了基本的 API 后端。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，会调用 Lambda 函数。该函数将返回一条 `hello world` 消息。

> **Note**
>
> AWS SAM 使用 esbuild 从 TypeScript 代码创建 Node.js Lambda 函数。esbuild 支持目前处于公开预览状态。在公开预览期间，esbuild 支持可能会发生向后不兼容的变更。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- AWS CLI 版本 2
- AWS SAM CLI 版本 1.75 或更高版本
- Node.js 18.x

部署示例 AWS SAM 应用程序

1. 使用 Hello World TypeScript 模板初始化该应用程序。

```
sam init --app-template hello-world-typescript --name sam-app --package-type Zip --
runtime nodejs18.x
```

2. （可选）该示例应用程序包括常用工具的配置，例如用于代码检查的 esLlint 和用于单元测试的 Jest。要运行检查和测试命令，请执行以下操作：

```
cd sam-app/hello-world
npm install
npm run lint
npm run test
```

3. 构建应用程序。

```
cd sam-app
sam build
```

4. 部署应用程序。

```
sam deploy --guided
```

5. 按照屏幕上的提示操作。要接受交互式体验中提供的原定设置选项，请通过 Enter 进行响应。
6. 输出将显示 REST API 的端点。在浏览器中打开该端点，以测试函数。您应该会看到以下响应：

```
{"message":"hello world"}
```

7. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
sam delete
```

使用 AWS CDK 将 TypeScript 代码部署到 Lambda

请按照以下步骤使用 AWS CDK 来构建和部署示例 TypeScript 应用程序。此应用程序实现了基本的 API 后端。它由 API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，会调用 Lambda 函数。该函数将返回一条 `hello world` 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- AWS CLI 版本 2
- AWS CDK 版本 2
- Node.js 18.x
- Docker 或 esbuild

部署示例 AWS CDK 应用程序

1. 为您的新应用程序创建一个项目目录。

```
mkdir hello-world
cd hello-world
```

2. 初始化该应用程序。

```
cdk init app --language typescript
```

3. 添加 @types/aws-lambda 软件包作为开发依赖项。此程序包包含 Lambda 的类型定义。

```
npm install -D @types/aws-lambda
```

4. 打开 lib 目录。您应该会看到一个名为 hello-world-stack.ts 的文件。在此目录中创建两个新文件：hello-world.function.ts 和 hello-world.ts。
5. 打开 hello-world.function.ts，然后将以下代码添加到该文件。这是适用于 Lambda 函数的代码。

> **Note**
>
> `import` 语句从 @types/aws-lambda 中导入类型定义。它不导入 `aws-lambda` NPM 程序包，这是一个无关的第三方工具。有关更多信息，请参阅 DefinitelyTyped GitHub 存储库中的 aws-lambda。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
```

```
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. 打开 hello-world.ts，然后将以下代码添加到该文件。其中包含 NodejsFunction 构造（它将创建 Lambda 函数）和 LambdaRestApi 构造（它将创建 REST API）。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function');
    new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
  }
}
```

NodejsFunction 构造默认假设以下内容：

- 您的函数处理程序称为 handler。
- 包含函数代码的 .ts 文件 (hello-world.function.ts) 与包含构造的 .ts 文件 (hello-world.ts) 位于相同的目录中。该构造使用构造的 ID（“hello-world”）和 Lambda 处理程序文件的名称（“function”）来查找函数代码。例如，如果您的函数代码位于名为 hello-world.my-function.ts 的文件中，则 hello-world.ts 文件必须按如下所示引用该函数代码：

```
const helloFunction = new NodejsFunction(this, 'my-function');
```

您可以更改此行为并配置其他 esbuild 参数。有关更多信息，请参阅 AWS CDK API 参考中的配置 esbuild。

7. 打开 hello-world-stack.ts。这是定义您的 AWS CDK 堆栈的代码。使用以下代码替换该代码：

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
```

```
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

8. 从包含您的 cdk.json 文件的 hello-world 目录中，部署您的应用程序。

```
cdk deploy
```

9. AWS CDK 使用 esbuild 构建和打包 Lambda 函数，然后将该函数部署到 Lambda 运行时。输出将显示 REST API 的端点。在浏览器中打开该端点，以测试函数。您应该会看到以下响应：

```
{"message":"hello world"}
```

这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

使用 AWS CLI 和 esbuild 将 TypeScript 代码部署到 Lambda

以下示例演示如何使用 esbuild 和 AWS CLI 转换 TypeScript 代码并将其部署到 Lambda。esbuild 将生成一个包含所有依赖项的 JavaScript 文件。这是您需要添加到 .zip 归档中的唯一文件。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- AWS CLI 版本 2
- Node.js 18.x
- Lambda 函数的执行角色
- 对于 Windows 用户来说，这是一款压缩文件实用程序，例如 7zip。

部署示例函数

1. 在本地计算机上，为新函数创建项目目录。
2. 使用 npm 或您选择的软件包管理器创建一个新的 Node.js 项目。

```
npm init
```

3. 添加 @types/aws-lambda 和 esbuild 软件包作为开发依赖项。@types/aws-lambda 程序包包含 Lambda 的类型定义。

```
npm install -D @types/aws-lambda esbuild
```

4. 创建名为 index.ts 的新文件。将以下代码添加到该新文件中。这是适用于 Lambda 函数的代码。该函数将返回一条 hello world 消息。该函数不会创建任何 API Gateway 资源。

> **Note**
>
> import 语句从 @types/aws-lambda 中导入类型定义。它不导入 aws-lambda NPM 程序包，这是一个无关的第三方工具。有关更多信息，请参阅 DefinitelyTyped GitHub 存储库中的 aws-lambda。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  console.log(`Event: ${JSON.stringify(event, null, 2)}`);
  console.log(`Context: ${JSON.stringify(context, null, 2)}`);
  return {
      statusCode: 200,
      body: JSON.stringify({
          message: 'hello world',
      }),
   };
};
```

5. 将构建脚本添加到 package.json 文件。这会将 esbuild 配置为自动创建 .zip 部署软件包。有关更多信息，请参阅 esbuild 文档中的构建脚本。

   Linux and MacOS

```
"scripts": {
  "prebuild": "rm -rf dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
```

```
  "postbuild": "cd dist && zip -r index.zip index.js*"
},
```

Windows

在此示例中，"postbuild" 命令使用 7zip 实用程序创建您的 .zip 文件。使用您自己首选的 Windows zip 实用程序并根据需要修改该命令。

```
"scripts": {
  "prebuild": "del /q dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
  "postbuild": "cd dist && 7z a -tzip index.zip index.js*"
},
```

6. 构建软件包。

```
npm run build
```

7. 使用该 .zip 部署软件包创建 Lambda 函数。将突出显示的文本替换为您的执行角色的 Amazon 资源名称（ARN）。

```
aws lambda create-function --function-name hello-world --runtime "nodejs18.x" --
role arn:aws:iam::123456789012:role/lambda-ex --zip-file "fileb://dist/index.zip"
 --handler index.handler
```

8. 运行测试事件，以确认该函数会返回以下响应。如果您想使用 API Gateway 调用此函数，请创建和配置 REST API。

```
{
  "statusCode": 200,
  "body": "{\"message\":\"hello world\"}"
}
```

使用容器镜像在 Lambda 中部署转换后的 TypeScript 代码

您可以将 TypeScript 代码作为 Node.js 容器镜像部署到 AWS Lambda 函数中。AWS 将为 Node.js 提供基本镜像，以帮助您构建容器镜像。这些基本映像会预加载一个语言运行时和在 Lambda 上运行映像所需的其他组件。AWS 为每个基本映像提供 Dockerfile，以帮助构建容器映像。

如果使用社群或私有企业基本镜像，则必须将 Node.js 运行时接口客户端 (RIC) 添加到该基本镜像，以使其与 Lambda 兼容。

Lambda 提供了可用于本地测试的运行时系统接口仿真器。Node.js 的 AWS 基本映像包含运行时系统接口仿真器。如果您使用备用基本映像，例如 Alpine Linux 或 Debian 映像，则可将仿真器构建到映像中或将其安装在本地计算机上。

使用 Node.js 基本映像构建和打包 TypeScript 函数代码

先决条件

要完成本节中的步骤，您必须满足以下条件：

- AWS CLI 版本 2
- Docker
- Node.js 20.x

从基本映像创建映像

通过AWS基本映像为 Lambda 创建映像

1. 在本地计算机上，为新函数创建项目目录。
2. 使用 npm 或您选择的软件包管理器创建一个新的 Node.js 项目。

```
npm init
```

3. 添加 @types/aws-lambda 和 esbuild 软件包作为开发依赖项。`@types/aws-lambda` 程序包包含 Lambda 的类型定义。

```
npm install -D @types/aws-lambda esbuild
```

4. 将构建脚本添加到 `package.json` 文件。

```
  "scripts": {
```

```
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js"
}
```

5. 创建名为 `index.ts` 的新文件。将以下示例代码添加到该新文件。这是适用于 Lambda 函数的代码。该函数将返回一条 `hello world` 消息。

> **Note**
>
> `import` 语句从 @types/aws-lambda 中导入类型定义。它不导入 `aws-lambda` NPM 程序包，这是一个无关的第三方工具。有关更多信息，请参阅 DefinitelyTyped GitHub 存储库中的 aws-lambda。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. 使用以下配置创建一个新的 Dockerfile。

   - 将 `FROM` 属性设置为基本映像的 URI。
   - 设置 `CMD` 参数以指定 Lambda 函数处理程序。

   以下示例 Dockerfile 将使用多阶段构建。第一步将 TypeScript 代码转换为 JavaScript。第二步生成仅包含 JavaScript 文件和生产依赖项的容器映像。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 `root` 用户。

Example Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:20 as builder
WORKDIR /usr/app
COPY package.json index.ts  ./
RUN npm install
RUN npm run build

FROM public.ecr.aws/lambda/nodejs:20
WORKDIR ${LAMBDA_TASK_ROOT}
COPY --from=builder /usr/app/dist/* ./
CMD ["index.handler"]
```

7. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 `docker-image` 并为其提供 `test` 标签。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 该命令指定了 `--platform linux/amd64` 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 `--platform linux/arm64` 选项。

（可选）在本地测试映像

1. 使用 docker run 命令启动 Docker 映像。在此示例中，`docker-image` 是映像名称，`test` 是标签。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令会将映像作为容器运行，并在 `localhost:9000/2015-03-31/functions/function/invocations` 创建本地端点。

> **Note**
>
> 如果为 ARM64 指令集架构创建 Docker 映像，请务必使用 `--platform linux/arm64` 选项，而不是 `--platform linux/amd64` 选项。

2. 在新的终端窗口中，将事件发布到本地端点。

   Linux/macOS

   在 Linux 和 macOS 中，运行以下 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   在 PowerShell 中，运行以下 `Invoke-WebRequest` 命令：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

3. 获取容器 ID。

   ```
   docker ps
   ```

4. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 `111122223333` 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
```

```
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。
   - 将 `<ECRrepositoryUri>` 替换为复制的 `repositoryUri`。确保 URI 末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 `ImageUri`，指定之前的存储库 URI。确保 URI 末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，`--publish` 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

使用 TypeScript Lambda 函数的层

Lambda 层是包含补充代码或数据的 .zip 文件存档。层通常包含库依赖项、自定义运行时系统或配置文件。创建层涉及三个常见步骤：

1. 打包层内容。此步骤需要创建 .zip 文件存档，其中包含要在函数中使用的依赖项。
2. 在 Lambda 中创建层。
3. 将层添加到函数。

本主题包含有关如何正确打包并创建具有外部库依赖项的 Node.js Lambda 层的步骤和指南。此外，本主题还说明了如何将层与在 TypeScript 中编写的函数一起使用。

先决条件

要完成本部分中的步骤，您必须满足以下条件：

- Node.js 20 和 npm 软件包管理器。有关安装 Node.js 的更多信息，请参阅 Node.js 文档中的 Installing Node.js via package manager。
- AWS CLI 版本 2

在本主题中，我们引用了 aws-lambda-developer-guide GitHub 存储库中的 layer-nodejs 示例应用程序。此应用程序包含脚本，脚本会将 lodash 库打包到 Lambda 层中。`layer` 目录包含用于生成层的脚本。在使用来自层的依赖项的 `function-ts` 目录中，该应用程序还包含了 TypeScript 示例函数。创建层后，即可转换、部署并调用相应的函数来验证一切是否正常运行。本文档将演示如何使用 TypeScript 示例函数创建、打包、部署并测试该层。

该示例应用程序使用 Node.js 20 运行时。如果您在层中添加其他依赖项，则它们必须与 Node.js 20 兼容。

Node.js 层与 Lambda 运行时环境的兼容性

在 Node.js 层中打包代码时，需要指定与该代码兼容的 Lambda 运行时环境。要评测代码与运行时的兼容性，请考虑代码是为哪些版本的 Node.js、操作系统和指令集架构设计的。

Lambda Node.js 运行时指定其 Node.js 版本和操作系统。在本文档中，您将使用基于 AL2023 的 Node.js 20 运行时。有关运行时版本的更多信息，请参阅the section called “支持的运行时”。在创建 Lambda 函数时，您可以指定指令集架构。在本文档中，您将使用 `arm64` 架构。有关 Lambda 中的架构的更多信息，请参阅the section called “指令集（ARM/x86）”。

如果您使用软件包中提供的代码，每个软件包维护者都会独立定义其兼容性。大多数 Node.js 开发都是为了独立于操作系统和指令集架构而设计。此外，打破与新 Node.js 版本的不兼容性的情况并不常见。与评测软件包与 Node.js 版本、操作系统或指令集架构的兼容性相比，应该花更多的时间来评测软件包之间的兼容性。

有时，Node.js 包包含编译后的代码，这需要您考虑操作系统和指令集架构的兼容性。如果您确实需要评测软件包的代码兼容性，则需要检查软件包及其文档。NPM 中的软件包可以通过 `package.json` 清单文件的 `engines`、`os` 和 `cpu` 字段来指定其兼容性。有关 `package.json` 文件的更多信息，请参阅 NPM 文档中的 package.json。

Node.js 运行时的层路径

当您向函数添加层时，Lambda 会将层内容加载到执行环境中。如果您的层将依赖项打包到特定的文件夹路径中，Node.js 执行环境将识别这些模块，并且您可以从函数代码中引用这些模块。

为确保您的模块被拾取，请使用以下文件夹路径之一，将它们打包到层 .zip 文件中。这些文件存储在 `/opt` 中，并将文件夹路径加载到 `PATH` 环境变量中。

- `nodejs/node_modules`
- `nodejs/node`*X*`/node_modules`

例如，您在本教程中创建生成的层.zip 文件具有以下目录结构：

```
layer_content.zip
# nodejs
    # node20
        # node_modules
            # lodash
            # <other potential dependencies>
            # ...
```

您将把 lodash 库放在 `nodejs/node20/node_modules` 目录中。这可确保 Lambda 在函数调用期间可以找到该库。

打包层内容

在本示例中，您要将 `lodash` 库打包在一个层 .zip 文件中。完成以下步骤，安装并打包层内容。

安装并打包层内容

1. 克隆 GitHub 存储库中的 aws-lambda-developer-guide，其中包含 sample-apps/layer-nodejs 目录中需要的示例代码。

```
git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
```

2. 导航到 layer-nodejs 示例应用程序的 layer 目录。此目录包含用于正确创建并打包层的脚本。

```
cd aws-lambda-developer-guide/sample-apps/layer-nodejs/layer
```

3. 确保 package.json 文件列出 lodash。此文件定义了要包含在层中的依赖项。您可以更新此文件，纳入要包含在层中的任何依赖项。

> **Note**
>
> 在您的依赖项上传到 Lambda 层后，此步骤中使用的 package.json 不会存储起来，也不会与依赖项一起使用。它仅在层打包过程中使用，不像 Node.js 应用程序或已发布包中的文件那样指定运行命令和兼容性。

4. 确保您拥有运行 layer 目录中脚本的 shell 权限。

```
chmod 744 1-install.sh && chmod 744 2-package.sh
```

5. 使用以下命令运行 1-install.sh 脚本：

```
./1-install.sh
```

此脚本将运行 npm install，它会读取您的 package.json 并下载其中定义的依赖项。

Example 1-install.sh

```
npm install .
```

6. 使用以下命令运行 2-package.sh 脚本：

```
./2-package.sh
```

此脚本将内容从 node_modules 目录复制到名为 nodejs/node20 的新目录中，随后将 nodejs 目录的内容压缩到一个名为 layer_content.zip 的文件中。这便是层的 .zip 文件。您可以解压缩文件，验证是否包含正确的文件结构，如 the section called “Node.js 运行时的层路径”部分所示。

Example 2-package.sh

```
mkdir -p nodejs/node20
cp -r node_modules nodejs/node20/
zip -r layer_content.zip nodejs
```

创建层

获取在上一部分中生成的 layer_content.zip 文件，将其作为 Lambda 层上传。您可以使用 AWS Management Console 上传层，也可以通过 AWS Command Line Interface（AWS CLI）使用 Lambda API 上传层。上传层 .zip 文件时，在以下 PublishLayerVersion AWS CLI 命令中，将 nodejs20.x 指定为兼容的运行时系统，并将 arm64 指定为兼容的架构。

```
aws lambda publish-layer-version --layer-name node-lodash-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes nodejs20.x \
    --compatible-architectures "arm64"
```

注意响应中的 LayerVersionArn，与 arn:aws:lambda:us-east-1:*123456789012*:layer:node-lodash-layer:1 类似。在本教程的下一步中，在将层添加到函数时，您要用到此 Amazon 资源名称（ARN）。

将层添加到函数

部署在函数代码中使用 lodash 库的示例 Lambda 函数，然后附加创建的层。要使用在 TypeScript 中编写的函数代码创建 Lambda 函数，必须将 TypeScript 转换为 JavaScript，以供 Node.js 运行时使用。有关此过程的更多信息，请参阅 the section called “处理程序”。为了提高兼容性，使用层分发依赖项时，请使用 tsc 来转换 TypeScript 模块。如果要捆绑依赖项，请考虑使用 esbuild。有关使用 esbuild 进行捆绑的更多信息，请参阅the section called “部署 .zip 文件归档”。

要部署该函数，您需要一个执行角色。有关更多信息，请参阅 the section called “执行角色（函数访问其他资源的权限）”。如果目前没有执行角色，则按照可折叠部分中的步骤操作。若有，则跳至下一部分来部署函数。

（可选）创建执行角色

创建执行角色

1. 在 IAM 控制台中，打开 Roles（角色）页面。
2. 选择创建角色。
3. 创建具有以下属性的角色。

   - Trusted entity（可信任的实体）– Lambda。
   - Permissions（权限）– AWSLambdaBasicExecutionRole。
   - Role name（角色名称）– **lambda-role**。

   AWSLambdaBasicExecutionRole 策略具有函数将日志写入 CloudWatch Logs 所需的权限。

示例函数代码使用 lodash `_.findLastIndex` 方法来读取对象数组。它将对象与标准进行比较，以找到匹配项的索引。然后，它会返回 Lambda 响应中对象的索引和值。

```
import { Handler } from 'aws-lambda';
import * as _ from 'lodash';

type User = {
  user: string;
  active: boolean;
}

type UserResult = {
  statusCode: number;
  body: string;
}

const users: User[] = [
  { 'user': 'Carlos', 'active': true },
  { 'user': 'Gil-dong', 'active': false },
  { 'user': 'Pat', 'active': false }
];

export const handler: Handler<any, UserResult> = async (): Promise<UserResult> => {

  let out = _.findLastIndex(users, (user: User) => { return user.user == 'Pat'; });
  const response = {
```

```
    statusCode: 200,
    body: JSON.stringify(out + ", " + users[out].user),
  };
  return response;
};
```

部署 Lambda 函数

1. 导航到 layer-nodejs 示例应用程序的 function-ts/ 目录。如果当前在 layer-nodejs 示例应用程序的 layer/ 目录中，请运行以下命令：

```
cd ../function-ts
```

2. 使用以下命令安装 package.json 中列出的开发依赖项：

```
npm install
```

3. 运行 package.json 中定义的 build 任务，以转换函数代码并将其打包到 .zip 文件中。使用以下命令：

```
npm run build
```

4. 部署函数。在以下 AWS CLI 命令中，将 --role 参数替换为执行角色 ARN：

```
aws lambda create-function --function-name nodejs_function_with_layer \
    --runtime nodejs20.x \
    --architectures "arm64" \
    --handler index.handler \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://dist/index.zip
```

5. 将层附加到函数。在以下 AWS CLI 命令中，将 --layers 参数替换为之前记下的层版本 ARN：

```
aws lambda update-function-configuration --function-name nodejs_function_with_layer
 \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:nodejs-lodash-layer:1"
```

6. 调用函数，使用以下 AWS CLI 命令验证它是否可以正常工作：

```
aws lambda invoke --function-name nodejs_function_with_layer \
    --cli-binary-format raw-in-base64-out \
```

```
    --payload '{}' response.json
```

应看到类似如下内容的输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

输出 `response.json` 文件包含有关响应的详细信息。

（可选）清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除 Lambda 层

1. 打开 Lambda 控制台的 Layers page（层页面）。
2. 选择您创建的层。
3. 选择删除，然后再次选择删除。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择 Delete（删除）。

使用 Lambda 上下文对象检索 TypeScript 函数信息

当 Lambda 运行您的函数时，它会将上下文对象传递到处理程序。此对象提供的方法和属性包含有关调用、函数和执行环境的信息。

上下文方法

- `getRemainingTimeInMillis()` – 返回执行超时前剩余的毫秒数。

上下文属性

- `functionName` – Lambda 函数的名称。
- `functionVersion` – 函数的版本
- `invokedFunctionArn` – 用于调用函数的 Amazon Resource Name (ARN)。表明调用者是否指定了版本号或别名。
- `memoryLimitInMB` – 为函数分配的内存量。
- `awsRequestId` – 调用请求的标识符。
- `logGroupName` – 函数的日志组。
- `logStreamName` – 函数实例的日志流。
- `identity` – （移动应用程序）有关授权请求的 Amazon Cognito 身份的信息。
  - `cognitoIdentityId` – 经过身份验证的 Amazon Cognito 身份。
  - `cognitoIdentityPoolId` – 授权调用的 Amazon Cognito 身份池。
- `clientContext` – （移动应用程序）客户端应用程序提供给 Lambda 的客户端上下文。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`
  - `env.make`
  - `env.model`
  - `env.locale`

  - `Custom` – 由客户应用程序设置的自定义值。
- `callbackWaitsForEmptyEventLoop` – 设置为 false 可在回调运行时立即发送响应，而不是等待 Node.js 事件循环成空。如果为 false，则任何未完成的事件将在下次调用期间继续运行。

您可以使用 @types/aws-lambda npm 程序包来处理上下文对象。

Example index.ts 文件

以下示例函数记录了上下文信息并返回了日志的位置。

> **Note**
>
> 在 Lambda 函数中使用此代码之前，必须添加 @types/aws-lambda 程序包作为开发依赖项。此程序包包含 Lambda 的类型定义。安装 `@types/aws-lambda` 时，`import` 语句 (`import ... from 'aws-lambda'`) 会导入类型定义。它不导入 `aws-lambda` NPM 程序包，这是一个无关的第三方工具。有关更多信息，请参阅 DefinitelyTyped GitHub 存储库中的 aws-lambda。

```
import { Context } from 'aws-lambda';
export const lambdaHandler = async (event: string, context: Context): Promise<string>
 => {
  console.log('Remaining time: ', context.getRemainingTimeInMillis());
  console.log('Function name: ', context.functionName);
  return context.logStreamName;
};
```

TypeScript Lambda 函数日志记录和监控

AWS Lambda 将自动监控 Lambda 函数并将日志条目发送到 Amazon CloudWatch。您的 Lambda 函数带有一个 CloudWatch Logs 日志组以及函数的每个实例的日志流。Lambda 运行时系统环境会将每次调用的详细信息以及函数代码的其他输出发送到该日志流。有关 CloudWatch Logs 的更多信息，请参阅将 CloudWatch Logs 日志与 Lambda 结合使用。

要从函数代码输出日志，可以使用控制台对象上的方法。如需更详细的日志记录，可以使用任何写入 `stdout` 或 `stderr` 的日志记录库。

Sections

使用日志记录工具和库

Powertools for AWS Lambda（TypeScript）是一个开发人员工具包，可用于实施无服务器最佳实践并提高开发人员速度。日志记录实用程序提供经优化的 Lambda 日志记录程序，其中包含有关所有函数的函数上下文的附加信息，输出结构为 JSON。请使用该实用程序执行以下操作：

- 从 Lambda 上下文中捕获关键字段，冷启动并将日志记录输出结构化为 JSON
- 根据指示记录 Lambda 调用事件（默认情况下禁用）
- 通过日志采样仅针对一定百分比的调用输出所有日志（默认情况下禁用）
- 在任何时间点将其他键附加到结构化日志
- 使用自定义日志格式设置程序（自带格式设置程序），从而在与组织的日志记录 RFC 兼容的结构中输出日志

将 Powertools for AWS Lambda（TypeScript）和 AWS SAM 用于结构化日志记录

请按照以下步骤使用 AWS SAM 通过集成的 Powertools for AWS Lambda（TypeScript）模块来下载、构建和部署示例 Hello World TypeScript 应用程序。此应用程序实现了基本的 API 后端，并使用

Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 `hello world` 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Node.js 18.x 或更高版本
- AWS CLI 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS SAM 应用程序

1. 使用 Hello World TypeScript 模板初始化该应用程序。

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x
```

2. 构建应用程序。

```
cd sam-app && sam build
```

3. 部署应用程序。

```
sam deploy --guided
```

4. 按照屏幕上的提示操作。要在交互式体验中接受提供的默认选项，请按 `Enter`。

> **Note**
>
> 对于 HelloWorldFunction 可能没有定义授权，确定执行此操作吗？，确保输入 `y`。

5. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 调用 API 端点：

```
curl <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

7. 要获取该函数的日志，请运行 sam logs。有关更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 使用日志。

```
sam logs --stack-name sam-app
```

该日志输出类似于以下示例：

```
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.552000
 START RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Version: $LATEST
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.594000
 2022-08-31T09:33:10.557Z 70693159-7e94-4102-a2af-98a6343fb8fb
 INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
[{"Namespace":"sam-app","Dimensions":[["service"]],"Metrics":
[{"Name":"ColdStart","Unit":"Count"}]}]},"service":"helloWorld","ColdStart":1}
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.595000
 2022-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"level":"INFO","message":"This is an INFO log - sending HTTP 200 - hello world
 response","service":"helloWorld","timestamp":"2022-08-31T09:33:10.594Z"}
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.655000
 2022-08-31T09:33:10.655Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"_aws":{"Timestamp":1661938390655,"CloudWatchMetrics":[{"Namespace":"sam-
app","Dimensions":[["service"]],"Metrics":[]}]},"service":"helloWorld"}
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.754000 END
 RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.754000
 REPORT RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Duration: 201.55 ms Billed
 Duration: 202 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 252.42
 ms
XRAY TraceId: 1-630f2ad5-1de22b6d29a658a466e7ecf5 SegmentId: 567c116658fbf11a
 Sampled: true
```

8. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
sam delete
```

管理日志保留日期

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期结束后，日志将自动删除。要设置日志保留日期，请将以下内容添加到您的 AWS SAM 模板中：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
    # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

将 Powertools for AWS Lambda（TypeScript）和 AWS CDK 用于结构化日志记录

请按照以下步骤使用 AWS CDK 通过集成的 Powertools for AWS Lambda（TypeScript）模块来下载、构建和部署示例 Hello World TypeScript 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 `hello world` 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Node.js 18.x 或更高版本
- AWS CLI 版本 2
- AWS CDK 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS CDK 应用程序

1. 为您的新应用程序创建一个项目目录。

```
mkdir hello-world
cd hello-world
```

2. 初始化该应用程序。

```
cdk init app --language typescript
```

3. 添加 @types/aws-lambda 软件包作为开发依赖项。

```
npm install -D @types/aws-lambda
```

4. 安装 Powertools Logger 实用程序。

```
npm install @aws-lambda-powertools/logger
```

5. 打开 lib 目录。您应该会看到一个名为 hello-world-stack.ts 的文件。在此目录中创建两个新文件：hello-world.function.ts 和 hello-world.ts。
6. 打开 hello-world.function.ts，然后将以下代码添加到该文件。这是适用于 Lambda 函数的代码。

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Logger } from '@aws-lambda-powertools/logger';
const logger = new Logger();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  logger.info('This is an INFO log - sending HTTP 200 - hello world response');
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};
```

7. 打开 hello-world.ts，然后将以下代码添加到该文件。它包含 NodejsFunction 构造，该构造创建 Lambda 函数，为 Powertools 配置环境变量，并将日志保留日期设置为一周。它还包括创建 REST API 的 LambdaRestApi 构造。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { RetentionDays } from 'aws-cdk-lib/aws-logs';
import { CfnOutput } from 'aws-cdk-lib';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        Powertools_SERVICE_NAME: 'helloWorld',
        LOG_LEVEL: 'INFO',
      },
      logRetention: RetentionDays.ONE_WEEK,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. 打开 hello-world-stack.ts。这是定义您的 AWS CDK 堆栈的代码。使用以下代码替换该代码：

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. 转回项目目录。

```
cd hello-world
```

10. 部署您的应用程序。

```
cdk deploy
```

11. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

12. 调用 API 端点：

```
curl <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

13. 要获取该函数的日志，请运行 sam logs。有关更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 使用日志。

```
sam logs --stack-name HelloWorldStack
```

该日志输出类似于以下示例：

```
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.047000
 START RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Version: $LATEST
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.050000 {
"level": "INFO",
"message": "This is an INFO log - sending HTTP 200 - hello world response",
"service": "helloWorld",
"timestamp": "2022-08-31T14:48:37.048Z",
"xray_trace_id": "1-630f74c4-2b080cf77680a04f2362bcf2"
}
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.082000 END
 RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.082000
 REPORT RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Duration: 34.60 ms Billed
 Duration: 35 ms Memory Size: 128 MB Max Memory Used: 57 MB Init Duration: 173.48
 ms
```

14. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
cdk destroy
```

在 Lambda 控制台中查看日志

调用 Lambda 函数后，您可以使用 Lambda 控制台查看日志输出。

如果可以在嵌入式代码编辑器中测试代码，则可以在执行结果中找到日志。使用控制台测试功能调用函数时，可以在详细信息部分找到日志输出。

在 CloudWatch 控制台中查看日志

您可以使用 Amazon CloudWatch 控制台查看所有 Lambda 函数调用的日志。

使用 CloudWatch 控制台查看日志

1. 打开 CloudWatch 控制台的 Log groups（日志组页面）。
2. 选择您的函数 (/aws/lambda/***your-function-name***) 的日志组。
3. 创建日志流。

每个日志流对应一个函数实例。日志流会在您更新 Lambda 函数以及创建更多实例来处理多个并发调用时显示。要查找特定调用的日志，建议您使用 AWS X-Ray 检测函数。X-Ray 会在追踪中记录有关请求和日志流的详细信息。

追踪 AWS Lambda 中的 TypeScript 代码

Lambda 与 AWS X-Ray 集成，以帮助您跟踪、调试和优化 Lambda 应用程序。您可以在某个请求遍历应用程序中的资源（其中可能包括 Lambda 函数和其他 AWS 服务）时，使用 X-Ray 跟踪该请求。

要将跟踪数据发送到 X-Ray，您可以使用以下三个开发工具包库之一：

- 适用于 OpenTelemetry 的 AWS 发行版 (ADOT) – 一种安全、可供生产、支持 AWS 的 OpenTelemetry (OTel) SDK 的分发版本。
- 适用于 Node.js 的 AWS X-Ray SDK – 用于生成跟踪数据并将其发送到 X-Ray 的 SDK。
- Powertools for AWS Lambda（TypeScript） – 一个开发人员工具包，可用于实施无服务器最佳实践并提高开发人员速度。

每个开发工具包均提供了将遥测数据发送到 X-Ray 服务的方法。然后，您可以使用 X-Ray 查看、筛选和获得对应用程序性能指标的洞察，从而发现问题和优化机会。

> **Important**
>
> X-Ray 和 Powertools for AWS Lambda SDK 是 AWS 提供的紧密集成的分析解决方案的一部分。ADOT Lambda Layers 是全行业通用的跟踪分析标准的一部分，该标准通常会收集更多数据，但可能不适用于所有使用案例。您可以使用任一解决方案在 X-Ray 中实现端到端跟踪。要了解有关如何在两者之间进行选择的更多信息，请参阅在 AWS Distro for Open Telemetry 和 X-Ray 开发工具包之间进行选择。

Sections

将 Powertools for AWS Lambda（TypeScript）和 AWS SAM 用于跟踪

请按照以下步骤使用 AWS SAM 通过集成的 Powertools for AWS Lambda（TypeScript）模块来下载、构建和部署示例 Hello World TypeScript 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API

Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 `hello world` 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Node.js 18.x 或更高版本
- AWS CLI 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS SAM 应用程序

1. 使用 Hello World TypeScript 模板初始化该应用程序。

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x --no-tracing
```

2. 构建应用程序。

```
cd sam-app && sam build
```

3. 部署应用程序。

```
sam deploy --guided
```

4. 按照屏幕上的提示操作。要在交互式体验中接受提供的默认选项，请按 `Enter`。

> **Note**
>
> 对于 HelloWorldFunction 可能没有定义授权，确定执行此操作吗？，确保输入 y。

5. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 调用 API 端点：

```
curl <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

7. 要获取该函数的跟踪信息，请运行 sam traces。

```
sam traces
```

该跟踪输出类似于以下示例：

```
XRay Event [revision 1] at (2023-01-31T11:29:40.527000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.483s)
  - 0.425s - sam-app/Prod [HTTP: 200]
    - 0.422s - Lambda [HTTP: 200]
  - 0.406s - sam-app-HelloWorldFunction-Xyzv11a1bcde [HTTP: 200]
  - 0.172s - sam-app-HelloWorldFunction-Xyzv11a1bcde
    - 0.179s - Initialization
    - 0.112s - Invocation
      - 0.052s - ## app.lambdaHandler
        - 0.001s - ### MySubSegment
    - 0.059s - Overhead
```

8. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
sam delete
```

X-Ray 无法跟踪对应用程序的所有请求。X-Ray 将应用采样算法确保跟踪有效，同时仍会提供所有请求的一个代表性样本。采样率是每秒 1 个请求和 5% 的其他请求。您无法为函数配置此 X-Ray 采样率。

将 Powertools for AWS Lambda（TypeScript）和 AWS CDK 用于跟踪

请按照以下步骤使用 AWS CDK 通过集成的 Powertools for AWS Lambda（TypeScript）模块来下载、构建和部署示例 Hello World TypeScript 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API

Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 `hello world` 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Node.js 18.x 或更高版本
- AWS CLI 版本 2
- AWS CDK 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS Cloud Development Kit (AWS CDK) 应用程序

1. 为您的新应用程序创建一个项目目录。

```
mkdir hello-world
cd hello-world
```

2. 初始化该应用程序。

```
cdk init app --language typescript
```

3. 添加 @types/aws-lambda 软件包作为开发依赖项。

```
npm install -D @types/aws-lambda
```

4. 安装 Powertools Tracer 实用程序。

```
npm install @aws-lambda-powertools/tracer
```

5. 打开 lib 目录。您应该会看到一个名为 hello-world-stack.ts 的文件。在此目录中创建两个新文件：hello-world.function.ts 和 hello-world.ts。
6. 打开 hello-world.function.ts，然后将以下代码添加到该文件。这是适用于 Lambda 函数的代码。

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Tracer } from '@aws-lambda-powertools/tracer';
const tracer = new Tracer();
```

```
export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  // Get facade segment created by Lambda
  const segment = tracer.getSegment();

  // Create subsegment for the function and set it as active
  const handlerSegment = segment.addNewSubsegment(`## ${process.env._HANDLER}`);
  tracer.setSegment(handlerSegment);

  // Annotate the subsegment with the cold start and serviceName
  tracer.annotateColdStart();
  tracer.addServiceNameAnnotation();

  // Add annotation for the awsRequestId
  tracer.putAnnotation('awsRequestId', context.awsRequestId);
  // Create another subsegment and set it as active
  const subsegment = handlerSegment.addNewSubsegment('### MySubSegment');
  tracer.setSegment(subsegment);
  let response: APIGatewayProxyResult = {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
  // Close subsegments (the Lambda one is closed automatically)
  subsegment.close(); // (### MySubSegment)
  handlerSegment.close(); // (## index.handler)

  // Set the facade segment as active again (the one created by Lambda)
  tracer.setSegment(segment);
  return response;
};
```

7. 打开 hello-world.ts，然后将以下代码添加到该文件。它包含 NodejsFunction 构造，该构造创建 Lambda 函数，为 Powertools 配置环境变量，并将日志保留日期设置为一周。它还包括创建 REST API 的 LambdaRestApi 构造。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { CfnOutput } from 'aws-cdk-lib';
import { Tracing } from 'aws-cdk-lib/aws-lambda';
```

```
export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        POWERTOOLS_SERVICE_NAME: 'helloWorld',
      },
      tracing: Tracing.ACTIVE,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. 打开 hello-world-stack.ts。这是定义您的 AWS CDK 堆栈的代码。使用以下代码替换该代码：

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. 部署您的应用程序。

```
cd ..
cdk deploy
```

10. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

11. 调用 API 端点：

```
curl <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

12. 要获取该函数的跟踪信息，请运行 sam traces。

```
sam traces
```

该跟踪输出类似于以下示例：

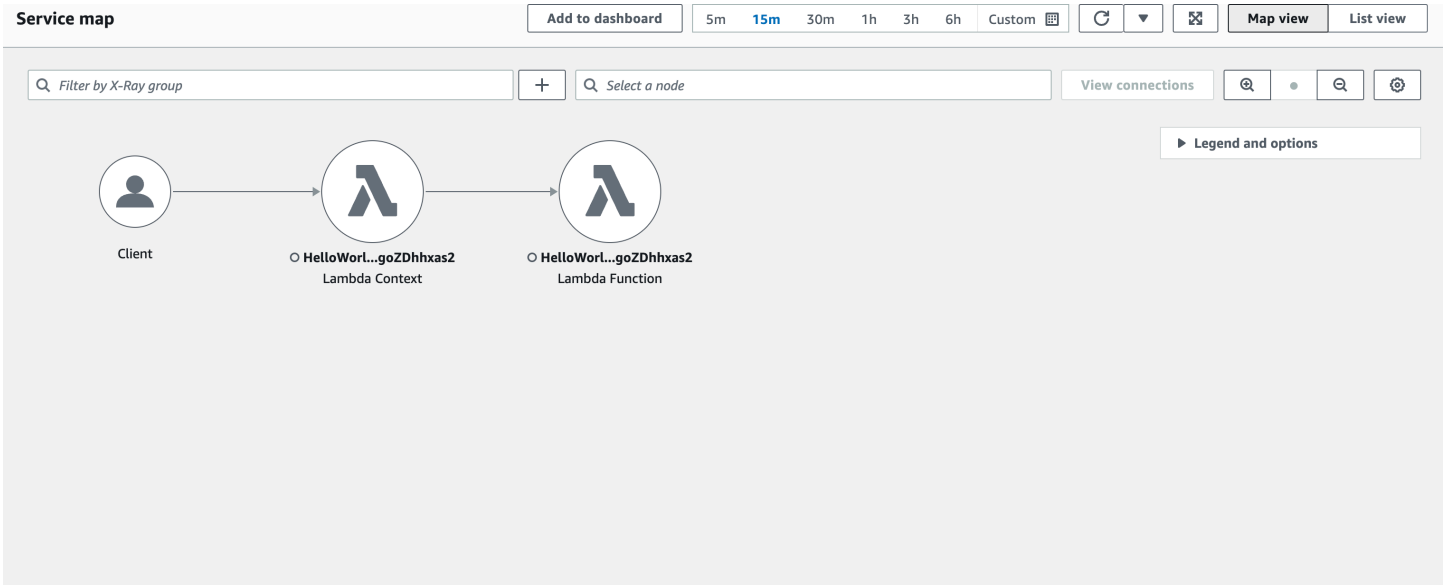
```
XRay Event [revision 1] at (2023-01-31T11:50:06.997000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.449s)
  - 0.350s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde [HTTP: 200]
  - 0.157s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde
    - 0.169s - Initialization
    - 0.058s - Invocation
      - 0.055s - ## index.handler
        - 0.000s - ### MySubSegment
    - 0.099s - Overhead
```

13. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
cdk destroy
```

解释 X-Ray 跟踪

在配置活跃跟踪后，您可以通过应用程序观察特定请求。X-Ray 跟踪图将显示有关应用程序及其所有组件的信息。以下示例显示了一个示例应用程序的跟踪：


Service map
Add to dashboard
5m
15m
30m
1h
3h
6h
Custom
Map view
List view
Filter by X-Ray group
Select a node
View connections
Legend and options
Client
HelloWorl...goZDhhxas2
Lambda Context
HelloWorl...goZDhhxas2
Lambda Function

使用 Python 构建 Lambda 函数

您可以在 AWS Lambda 中运行 Python。Lambda 为 Python 提供运行时运行您的代码来处理事件。您的代码在包含 SDK for Python (Boto3) 的环境中运行，其中包含来自您管理的 AWS Identity and Access Management (IAM) 角色的凭证。要了解有关 Python 运行时随附的 SDK 版本的更多信息，请参阅 the section called “包含运行时的 SDK 版本”。

Lambda 支持以下 Python 运行时。

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
Python 3.12	`python3.12`	Amazon Linux 2023	未计划	未计划	未计划
Python 3.11	`python3.11`	Amazon Linux 2	未计划	未计划	未计划
Python 3.10	`python3.10`	Amazon Linux 2	未计划	未计划	未计划
Python 3.9	`python3.9`	Amazon Linux 2	未计划	未计划	未计划

创建 Python 函数

1. 打开 Lambda 控制台。
2. 选择 Create function（创建函数）。
3. 配置以下设置：
   - 函数名称：输入函数名称。
   - 运行时系统：选择 Python 3.12。
4. 选择 Create function（创建函数）。
5. 要配置测试事件，请选择测试。
6. 对于事件名称，输入 **`test`**。
7. 选择 Save changes（保存更改）。

8. 要调用该函数，请选择 Test（测试）。

控制台将使用名为 `lambda_function` 的源文件创建一个 Lambda 函数。您可以在内置代码编辑器中编辑此文件并添加更多文件。要保存您的更改，请选择 Save（保存）。然后，要运行代码，请选择 Test（测试）。

您的 Lambda 函数附带了 CloudWatch Logs 日志组。函数运行时会将每次调用的详细信息发送到 CloudWatch Logs。该运行时会中继调用期间函数输出的任何日志。如果您的函数返回错误，则 Lambda 将为错误设置格式，并将其返回给调用方。

主题

包含运行时的 SDK 版本

Python 运行时中包含的 AWS SDK 版本取决于运行时版本和您的 AWS 区域。要查找您正在使用的运行时中包含的 SDK 的版本，请使用以下代码创建 Lambda 函数。

```
import boto3
import botocore

def lambda_handler(event, context):
   print(f'boto3 version: {boto3.__version__}')
   print(f'botocore version: {botocore.__version__}')
```

响应格式

在 Python 3.12 及更高版本的 Python 运行时系统中，函数作为其 JSON 响应的一部分返回 Unicode 字符。早期的 Python 运行时系统在响应中返回 Unicode 字符的转义序列。例如，在 Python 3.11 中，如果您返回诸如“こんにちは”之类的 Unicode 字符串，它会转义 Unicode 字符并返回“\u3053\u3093\u306b\u3061\u306f”。Python 3.12 运行时系统会返回原始的“こんにちは”。

使用 Unicode 响应可以减小 Lambda 响应的大小，从而更轻松地将较大的响应容纳到同步函数 6MB 的最大有效负载大小中。在前面的示例中，转义后的版本为 32 个字节，而 Unicode 字符串的转义版本为 17 个字节。

升级到 Python 3.12 后，可能需要调整代码以适应新的响应格式。如果调用方期望使用转义 Unicode，则必须在返回函数中添加代码以手动转义 Unicode，或者调整调用方以处理 Unicode 返回。

顺利关闭扩展程序

Python 3.12 及更高版本的 Python 运行时系统为带有外部扩展程序的函数提供了改进的顺利关闭功能。当 Lambda 关闭执行环境时，它会向运行时系统发送 `SIGTERM` 信号，然后将一个 `SHUTDOWN` 事件发送给每个注册的外部扩展。您可以在 Lambda 函数中捕获 `SIGTERM` 信号，并清理该函数创建的资源，例如数据库连接。

要了解有关执行环境生命周期的更多信息，请参阅 了解 Lambda 执行环境生命周期。有关如何顺利关闭扩展程序的示例，请参阅 AWS Samples GitHub Repository。

定义采用 Python 的 Lambda 函数处理程序

Lambda 函数处理程序是函数代码中处理事件的方法。当调用函数时，Lambda 运行处理程序方法。您的函数会一直运行，直到处理程序返回响应、退出或超时。

在 Python 中创建函数处理程序时，可以使用以下一般语法：

```
def handler_name(event, context):
    ...
    return some_value
```

主题

命名

在创建 Lambda 函数时指定的 Lambda 函数处理程序名称来自以下内容：

- Lambda 处理程序函数所在的文件名称。
- Python 处理程序函数的名称。

函数处理程序可随意命名；但是，在 Lambda 控制台上的默认名称为 `lambda_function.lambda_handler`。此功能处理程序名称将函数名称表示为 (`lambda_handler`)，存储处理程序代码的文件位于 (`lambda_function.py`)。

如果您在控制台中使用不同的文件名或函数处理程序名称创建函数，则必须编辑默认处理程序名称。

更改函数处理程序名称（控制台）

1. 打开 Lambda 控制台的函数页面，然后选择一个函数。
2. 选择节点选项卡。
3. 向下滚动到运行时设置窗格并选择编辑。

4. 在处理程序中，输入函数处理程序的新名称。
5. 选择保存。

工作方式

当 Lambda 调用函数处理程序时，Lambda 运行时会将两个参数传送给函数处理程序：

- 第一个参数是事件对象。事件是 JSON 格式的文档，其中包含要处理的 Lambda 函数的数据。Lambda 运行时将事件转换为一个对象，并将该对象传递给函数代码。它通常是 Python `dict` 类型。也可以是 `list`、`str`、`int`、`float`、或 `NoneType` 类型。

  事件对象包含来自调用服务的信息。在调用函数时，可以确定事件的结构和内容。当 AWS 服务调用函数时，该服务会定义事件结构。有关 AWS 服务中的事件的更多信息，请参阅 使用来自其 AWS 他服务的事件调用 Lambda。
- 第二个参数是 上下文对象。在运行时由 Lambda 将上下文对象传递给函数。此对象提供的方法和属性包含有关调用、函数和运行时环境的信息。

返回值

（可选）处理程序可返回值。返回值所发生的状态取决于调用函数的调用类型和服务。例如：

- 如果您使用 `RequestResponse` 调用类型（例如 同步调用 Lambda 函数），AWS Lambda 会将 Python 函数调用的结果返回到调用 Lambda 函数的客户端（在对调用请求的 HTTP 响应中，序列化为 JSON）。例如，AWS Lambda 控制台使用 `RequestResponse` 调用类型，因此当您使用控制台调用函数时，控制台将显示返回的值。
- 如果处理程序返回 `json.dumps` 无法序列化的对象，则运行时返回错误。
- 如果处理程序返回 None（就像不具有 `return` 语句的 Python 函数隐式执行的那样），则运行时返回 `null`。
- 如果您使用 `Event` 调用类型（一种异步调用），则该值将被丢弃。

> **Note**
>
> 在 Python 3.9 及更高版本中，Lambda 在错误响应中包含调用的 requestId。

示例

以下部分介绍了可以与 Lambda 结合使用的 Python 函数的示例。如果您使用 Lambda 控制台编写函数，无需附加 zip 归档文件即可运行本部分中的功能。这些函数使用标准 Python 库，这些库位于您选择的 Lambda 运行时中。有关更多信息，请参阅 ???。

返回消息

以下示例显示了名为 `lambda_handler` 的函数。该函数接受用户输入名字和姓氏，并从它接收为输入的事件返回包含数据的消息。

```
def lambda_handler(event, context):
    message = 'Hello {} {}!'.format(event['first_name'], event['last_name'])
    return {
        'message' : message
    }
```

您可以使用以下事件数据 调用函数：

```
{
    "first_name": "John",
    "last_name": "Smith"
}
```

响应显示作为输入传递的事件数据：

```
{
    "message": "Hello John Smith!"
}
```

解析响应

以下示例显示了名为 `lambda_handler` 的函数。该函数在运行时使用 Lambda 传递的事件数据。它解析 JSON 响应中 `AWS_REGION` 返回的 环境变量 。

```
import os
import json

def lambda_handler(event, context):
    json_region = os.environ['AWS_REGION']
```

```
    return {
        "statusCode": 200,
        "headers": {
            "Content-Type": "application/json"
        },
        "body": json.dumps({
            "Region ": json_region
        })
    }
```

您可以使用任何事件数据 调用函数：

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

Lambda 运行时会在初始化过程中设置多个环境变量。有关在运行时响应中返回的环境变量的更多信息，请参阅 使用 Lambda 环境变量配置代码中的值。

本示例中的函数取决于 Invoke API 的成功响应（200 中）。有关调用 API 状态的更多信息，请参阅调用响应语法。

返回计算

以下示例显示了名为 lambda_handler 的函数。该函数接受用户输入并将计算返回给用户。有关此示例的更多信息，请参阅 aws-doc-sdk-examples GitHub 存储库。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    ...
    result = None
    action = event.get('action')
    if action == 'increment':
        result = event.get('number', 0) + 1
        logger.info('Calculated result of %s', result)
    else:
```

```
        logger.error("%s is not a valid action.", action)

    response = {'result': result}
    return response
```

您可以使用以下事件数据 调用函数：

```
{
    "action": "increment",
    "number": 3
}
```

Python Lambda 函数的代码最佳实践

在构建 Lambda 函数时，请遵循以下列表中的指南，采用最佳编码实践：

- 从核心逻辑中分离 Lambda 处理程序。这样您可以创建更容易进行单元测试的函数。例如，在 Python 中，如下所示：

  ```
  def lambda_handler(event, context):
      foo = event['foo']
      bar = event['bar']

      result = my_lambda_function(foo, bar)

  def my_lambda_function(foo, bar):
      // MyLambdaFunction logic here
  ```

- 控制函数部署程序包中的依赖关系。AWS Lambda 执行环境包含许多库。对于 Node.js 和 Python 运行时，其中包括 AWS SDK。Lambda 会定期更新这些库，以支持最新的功能组合和安全更新。这些更新可能会使 Lambda 函数的行为发生细微变化。要完全控制您的函数所用的依赖项，请使用部署程序包来打包所有依赖项。
- 将依赖关系的复杂性降至最低。首选在执行环境启动时可以快速加载的更简单的框架。
- 将部署程序包大小精简为只包含运行时必要的部分。这样会减少调用前下载和解压缩部署程序包所需的时间。

- 利用执行环境重用来提高函数性能。连接软件开发工具包 (SDK) 客户端和函数处理程序之外的数据库，并在 `/tmp` 目录中本地缓存静态资产。由函数的同一实例处理的后续调用可重用这些资源。这样就可以通过缩短函数运行时间来节省成本。

为了避免调用之间潜在的数据泄露，请不要使用执行环境来存储用户数据、事件或其他具有安全影响的信息。如果您的函数依赖于无法存储在处理程序的内存中的可变状态，请考虑为每个用户创建单独的函数或单独的函数版本。

- 使用 keep-alive 指令来维护持久连接。Lambda 会随着时间的推移清除空闲连接。在调用函数时尝试重用空闲连接会导致连接错误。要维护您的持久连接，请使用与运行时关联的 keep-alive 指令。有关示例，请参阅在 Node.js 中通过 Keep-Alive 重用连接。
- 使用环境变量将操作参数传递给函数。例如，您在写入 Amazon S3 存储桶时，不应对要写入的存储桶名称进行硬编码，而应将存储桶名称配置为环境变量。
- 避免在 Lambda 函数中使用递归调用，在这种情况下，函数会调用自己或启动可能再次调用该函数的进程。这可能会导致意想不到的函数调用量和升级成本。如果您看到意外的调用量，请立即将函数保留并发设置为 0 来限制对函数的所有调用，同时更新代码。
- Lambda 函数代码中不要使用非正式的非公有 API。对于 AWS Lambda 托管式运行时，Lambda 会定期为 Lambda 的内部 API 应用安全性和功能更新。这些内部 API 更新可能不能向后兼容，会导致意外后果，例如，假设您的函数依赖于这些非公有 API，则调用会失败。请参阅 API 参考以查看公开发布的 API 列表。
- 编写幂等代码。为您的函数编写幂等代码可确保以相同的方式处理重复事件。您的代码应该正确验证事件并优雅地处理重复事件。有关更多信息，请参阅如何使我的 Lambda 函数具有幂等性？。

将 .zip 文件归档用于 Python Lambda 函数

AWS Lambda 函数的代码包含一个 .py 文件，其中包含函数的处理程序代码，以及代码所依赖的任何其他包和模块。要将此函数部署到 Lambda，您可以使用部署包。此包可以是 .zip 文件归档或容器映像。有关在 Python 中使用容器映像的更多信息，请参阅使用容器映像部署 Python Lambda 函数。

要创建 .zip 文件归档格式的部署包，可以使用命令行工具内置的 .zip 文件归档实用工具或任何其他 .zip 文件实用工具（例如 7zip）。以下各部分中显示的示例假设您在 Linux 或 macOS 环境中使用命令行 `zip` 工具。要在 Windows 中使用相同命令，您可以安装 Windows Subsystem for Linux，以获取 Windows 集成版本的 Ubuntu 和 Bash。

请注意，Lambda 使用 POSIX 文件权限，因此在创建 .zip 文件归档之前，您可能需要为部署包文件夹设置权限。

主题

Python 中的运行时系统依赖项

对于使用 Python 运行时系统的 Lambda 函数，依赖项可以是任何 Python 程序包或模块。使用 .zip 存档部署函数时，可以使用函数代码或使用 Lambda 层将这些依赖项添加到 .zip 文件中。层是可以包含其他代码或其他内容的单独的 .zip 文件。要了解在 Python 中使用 Lambda 层的更多信息，请参阅the section called “图层”。

Lambda Python 运行时系统包含 AWS SDK for Python (Boto3) 及其依赖项。Lambda 会在运行时系统中为您无法添加自定义依赖项的部署场景提供开发工具包。这些场景包括使用内置代码编辑器在控制台中创建函数，或者使用 AWS Serverless Application Model（AWS SAM）或 AWS CloudFormation 模板中的内联函数。

Lambda 会定期更新 Python 运行时系统中的库，以包含最新更新和安全补丁。如果函数使用运行时系统中包含的 Boto3 SDK 版本，但部署包包含开发工具包依赖项，可能会导致版本不一致问题。例如，

部署包可能包含开发工具包依赖项 urllib3。当 Lambda 在运行时系统中更新开发工具包时，运行时系统的新版本与部署包中的 urllib3 版本之间的兼容性问题可能会导致您的函数失败。

> **Important**
>
> 为了保持对依赖项的完全控制并避免可能的版本不一致问题，我们建议您将所有函数的依赖项添加到部署包中，即使它们的版本已包含在 Lambda 运行时系统中也是如此。这包括 Boto3 SDK。

要查找正在使用的运行时系统中包含哪个版本的 SDK for Python（Boto3），请参阅 the section called “包含运行时的 SDK 版本”。

在 AWS 责任共担模式下，您负责管理函数部署包中的所有依赖项。这包括应用更新和安全补丁。要更新函数部署包中的依赖项，请先创建一个新的 .zip 文件，然后将其上传到 Lambda 中。有关更多信息，请参阅 创建含依赖项的 .zip 部署包 和 使用 .zip 文件创建和更新 Python Lambda 函数。

创建不含依赖项的 .zip 部署包

如果您的函数代码没有依赖项，则 .zip 文件仅包含带有函数处理程序代码的 .py 文件。使用您的首选 zip 实用工具创建一个 .zip 文件，并将 .py 文件置于根目录中。如果 .py 文件不在 .zip 文件的根目录下，Lambda 将无法运行代码。

要了解如何部署 .zip 文件以创建新的 Lambda 函数或更新现有函数，请参阅 使用 .zip 文件创建和更新 Python Lambda 函数。

创建含依赖项的 .zip 部署包

如果函数代码依赖其他包或模块，可以使用函数代码或使用 Lambda 层将这些依赖项添加到 .zip 文件中。本部分中的说明向您展示了如何将依赖项包含在 .zip 部署包中。要让 Lambda 运行代码，必须将包含处理程序代码和所有函数依赖项的.py 文件安装在.zip 文件的根目录下。

假设函数代码保存在名为 `lambda_function.py` 的文件中。以下示例 CLI 命令将创建名为 `my_deployment_package.zip` 的 .zip 文件，其中包含函数代码及其依赖项。您可以将依赖项直接安装到项目目录中的文件夹，也可以使用 Python 虚拟环境。

要创建部署包（项目目录）

1. 导航到包含 `lambda_function.py` 源代码文件的项目目录。在此示例中，该目录名为 `my_function`。

```
cd my_function
```

2. 创建将在其中安装依赖项的名为 package 的新目录。

```
mkdir package
```

请注意，对于 .zip 部署包，Lambda 期望源代码及其依赖项全部位于 .zip 文件的根目录中。但是，直接在项目目录中安装依赖项可能会引入大量新文件和文件夹，使在 IDE 中导航变得困难。您可以在此目录中创建一个单独的 package 目录，以将依赖项与源代码分开。

3. 在 package 目录中安装依赖项。以下示例将使用 pip 从 Python 程序包索引中安装 Boto3 SDK。如果函数代码使用您自己创建的 Python 程序包，请将这类程序包保存在 package 目录中。

```
pip install --target ./package boto3
```

4. 创建包含已安装库在根目录中的 .zip 文件。

```
cd package
zip -r ../my_deployment_package.zip .
```

这样会在您的项目目录中生成一个 my_deployment_package.zip 文件。

5. 将 lambda_function.py 文件添加到 .zip 文件的根目录中。

```
cd ..
zip my_deployment_package.zip lambda_function.py
```

.zip 文件应采用扁平目录结构，将函数的处理程序代码和所有依赖项文件夹安装在根目录中，如下所示。

```
my_deployment_package.zip
|- bin
|  |-jp.py
|- boto3
|  |-compat.py
|  |-data
|  |-docs
...
|- lambda_function.py
```

如果包含函数的处理程序代码的 .py 文件不在 .zip 文件的根目录中，Lambda 将无法运行代码。

要创建部署包（虚拟环境）

1. 在项目目录中创建和激活虚拟环境。在此示例中，项目目录名为 my_function。

```
~$ cd my_function
~/my_function$ python3.12 -m venv my_virtual_env
~/my_function$ source ./my_virtual_env/bin/activate
```

2. 使用 pip 安装所需的库。下面的示例将安装 Boto3 SDK

```
(my_virtual_env) ~/my_function$ pip install boto3
```

3. 使用 pip show 在虚拟环境中查找 pip 安装依赖项的位置。

```
(my_virtual_env) ~/my_function$ pip show <package_name>
```

pip 在其中安装库的文件夹可能名为 site-packages 或 dist-packages。此文件夹可能位于 lib/python3.x 或 lib64/python3.x 目录中（其中 python3.x 代表正在使用的 Python 版本）。

4. 停用虚拟环境

```
(my_virtual_env) ~/my_function$ deactivate
```

5. 导航到包含使用 pip 安装了依赖项的目录，并在项目目录中创建一个 .zip 文件，将已安装的依赖项置于其根目录。在此示例中，pip 已在 my_virtual_env/lib/python3.12/site-packages 目录中安装了所需依赖项。

```
~/my_function$ cd my_virtual_env/lib/python3.12/site-packages
~/my_function/my_virtual_env/lib/python3.12/site-packages$ zip -r ../../../../
my_deployment_package.zip .
```

6. 导航到包含处理程序代码的 .py 文件所在的项目目录的根目录，然后将该文件添加到 .zip 程序包的根目录中。在此示例中，您的函数代码文件名为 lambda_function.py。

```
~/my_function/my_virtual_env/lib/python3.12/site-packages$ cd ../../../../
~/my_function$ zip my_deployment_package.zip lambda_function.py
```

依赖项搜索路径和包含运行时系统的库

在代码中使用 `import` 语句时，Python 运行时系统会搜索其搜索路径中的目录，直到找到相应模块或包。默认情况下，运行时系统搜索的第一个位置是解压缩并安装 .zip 部署包的目录 (`/var/task`)。如果部署包含有包含运行时系统的库的某个版本，则此版本将优先于运行时系统中包含的版本。部署包中的依赖项也优先于层中的依赖项。

当您向层添加依赖项时，Lambda 会将其提取到 `/opt/python/lib/python3.x/site-packages`（其中 `python3.x` 表示正在使用的运行时系统版本）或 `/opt/python` 中。在搜索路径中，这些目录优先于含有包含运行时系统的库和安装了 pip 的库的目录（`/var/runtime` 和 `/var/lang/lib/python3.x/site-packages`）。因此，函数层中的库优先于运行时系统中包含的版本。

> **Note**
>
> 在 Python 3.11 托管式运行时系统和基本映像中，AWS SDK 及其依赖项安装在 `/var/lang/lib/python3.11/site-packages` 目录中。

通过添加以下代码段，您可以查看 Lambda 函数的完整搜索路径。

```
import sys

search_path = sys.path
print(search_path)
```

> **Note**
>
> 由于部署包或层中的依赖项优先于包含运行时系统的库，因此如果您在包中包含 urllib3 等开发工具包依赖项而不包含开发工具包，可能会导致版本不一致问题。如果您要部署自己的 Boto3 依赖项版本，则还必须将 Boto3 作为依赖项部署到部署包中。我们建议您打包函数的所有依赖项，即使运行时系统中包含各种版本也是如此。

您还可以在 .zip 程序包内的单独文件夹中添加依赖项。例如，您可以将某个 Boto3 SDK 版本添加到 .zip 程序包中名为 `common` 的文件夹中。解压缩并安装 .zip 程序包后，此文件夹将放置在 `/var/task` 目录中。要在代码中使用 .zip 部署包中某个文件夹中的依赖项，请使用 `import from` 语句。例如，要使用 .zip 程序包中名为 `common` 的文件夹中的 Boto3 版本，请使用以下语句。

```
from common import boto3
```

使用 __pycache__ 文件夹

我们建议您不要在函数部署包中包含 `__pycache__` 文件夹。在具有不同架构或操作系统的生成计算机上编译的 Python 字节码可能与 Lambda 执行环境不兼容。

使用原生库创建 .zip 部署包

如果您的函数仅使用纯 Python 程序包和模块，则可以使用 `pip install` 命令在任何本地生成计算机上安装依赖项并创建 .zip 文件。许多流行的 Python 库（包括 NumPy 和 Pandas）都不是纯 Python 的，包含用 C 或 C++ 编写的代码。将包含 C/C++ 代码的库添加到部署包时，必须正确构建包以确保它与 Lambda 执行环境兼容。

Python 程序包索引（PyPI）上提供的大多数包都以“wheel”（.whl 文件）的形式提供。.whl 文件是一种 zip 文件，它包含已构建的分发，其中包含针对特定操作系统和指令集架构的预编译二进制文件。要使部署包与 Lambda 兼容，您需要为 Linux 操作系统和函数的指令集架构安装轮子。

有些包可能只能作为源分发提供。对于这些包，您需要自己编译和构建 C/C++ 组件。

要查看哪些分发可用于所需的包，请执行以下操作：

1. 在 Python 程序包索引主页上搜索程序包名称。
2. 选择要使用的包的版本。
3. 选择下载文件。

使用已构建分发（wheel）

要下载与 Lambda 兼容的 wheel，请使用 pip `--platform` 选项。

如果 Lambda 函数使用 x86_64 指令集架构，请运行以下 `pip install` 命令以在 `package` 目录中安装兼容的 wheel。将 `--python 3.x` 替换为正在使用的 Python 运行时系统版本。

```
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
```

```
<package_name>
```

如果函数使用的是 arm64 指令集架构，请运行以下命令。将 `--python 3.x` 替换为正在使用的 Python 运行时系统版本。

```
pip install \
--platform manylinux2014_aarch64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

使用源分发

如果包仅作为源分发提供，则需要自己构建 C/C++ 库。要使包与 Lambda 执行环境兼容，您需要在使用相同的 Amazon Linux 2 操作系统的环境中构建包。您可以通过在 Amazon EC2 Linux 实例中构建包来实现此目的。

要了解如何启动并连接到 Amazon EC2 Linux 实例，请参阅《适用于 Linux 实例的 Amazon EC2 用户指南》中的教程：Amazon EC2 Linux 实例入门。

使用 .zip 文件创建和更新 Python Lambda 函数

创建 .zip 部署包后，您可以用其创建新的 Lambda 函数或更新现有的 Lambda 函数。您可以使用 Lambda 控制台、AWS Command Line Interface 和 Lambda API 部署 .zip 程序包。您也可以使用 AWS Serverless Application Model（AWS SAM）和 AWS CloudFormation 创建和更新 Lambda 函数。

Lambda 的 .zip 部署包的最大大小为 250MB（已解压缩）。请注意，此限制适用于您上传的所有文件（包括任何 Lambda 层）的组合大小。

Lambda 运行时需要权限才能读取部署包中的文件。在 Linux 权限八进制表示法中，Lambda 对于不可执行文件（rw-r--r--）需要 644 个权限，对于目录和可执行文件需要 755 个权限（rwxr-xr-x）。

在 Linux 和 MacOS 中，使用 chmod 命令更改部署包中文件和目录的文件权限。例如，要为可执行文件提供正确的权限，请运行以下命令。

```
chmod 755 <filepath>
```

要在 Windows 中更改文件权限，请参阅 Microsoft Windows 文档中的 Set, View, Change, or Remove Permissions on an Object。

使用控制台通过 .zip 文件创建和更新函数

要创建新函数，必须先在控制台中创建该函数，然后上传您的 .zip 归档。要更新现有函数，请打开函数页面，然后按照相同的步骤添加更新的 .zip 文件。

如果您的 .zip 文件小于 50MB，则可以通过直接从本地计算机上传该文件来创建或更新函数。对于大于 50MB 的 .zip 文件，必须首先将您的程序包上传到 Amazon S3 存储桶。有关如何使用 AWS Management Console 将文件上传到 Amazon S3 存储桶的说明，请参阅 Amazon S3 入门。要使用 AWS CLI 上传文件，请参阅《AWS CLI 用户指南》中的移动对象。

> **Note**
>
> 您无法更改现有函数的部署包类型（.zip 或容器映像）。例如，您无法将容器映像函数转换为使用 .zip 文件归档。您必须创建新函数。

创建新函数（控制台）

1. 打开 Lambda 控制台的“函数”页面，然后选择创建函数。
2. 选择从头开始创作。
3. 在基本信息中，执行以下操作：
   a. 对于函数名称，输入函数的名称。
   b. 对于运行时系统，选择要使用的运行时系统。
   c. （可选）对于架构，选择要用于函数的指令集架构。默认架构为 x86_64。确保您的函数的 .zip 部署包与您选择的指令集架构兼容。
4. （可选）在 Permissions（权限）下，展开 Change default execution role（更改默认执行角色）。您可以创建新的执行角色，也可以使用现有角色。
5. 选择 Create function (创建函数)。Lambda 使用您选择的运行时系统创建基本“Hello world”函数。

从本地计算机上传 .zip 归档（控制台）

1. 在 Lambda 控制台的“函数”页面中，选择要为其上传 .zip 文件的函数。
2. 选择代码选项卡。

3. 在代码源窗格中，选择上传自。
4. 选择 .zip 文件。
5. 要上传 .zip 文件，请执行以下操作：
   a. 选择上传，然后在文件选择器中选择您的 .zip 文件。
   b. 选择打开。
   c. 选择保存。

从 Amazon S3 存储桶上传 .zip 归档（控制台）

1. 在 Lambda 控制台的“函数”页面中，选择要为其上传新 .zip 文件的函数。
2. 选择代码选项卡。
3. 在代码源窗格中，选择上传自。
4. 选择 Amazon S3 位置。
5. 粘贴 .zip 文件的 Amazon S3 链接 URL，然后选择保存。

使用控制台代码编辑器更新 .zip 文件函数

对于某些带有 .zip 部署包的函数，您可以使用 Lambda 控制台的内置代码编辑器直接更新函数代码。要使用此功能，函数必须满足以下条件：

- 函数必须使用一种解释性语言运行时系统（Python、Node.js 或 Ruby）
- 函数的部署包必须小于 50 MB（未压缩状态）。

带有容器映像部署包的函数的代码不能直接在控制台中编辑。

要使用控制台代码编辑器更新函数代码。

1. 打开 Lambda 控制台的“函数”页面，然后选择函数。
2. 选择代码选项卡。
3. 在代码源窗格中，选择源代码文件并在集成的代码编辑器中对其进行编辑。
4. 编辑完代码后，展开主侧栏中的部署部分，然后选择部署。

使用 AWS CLI 通过 .zip 文件创建和更新函数

您可以使用 AWS CLI 创建新函数或使用 .zip 文件更新现有函数。使用 create-function 和 update-function-code 命令部署 .zip 程序包。如果您的 .zip 文件小于 50MB，则可以从本地生成计算机上的文件位置上传 .zip 程序包。对于较大的文件，必须从 Amazon S3 存储桶上传 .zip 程序包。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

> **Note**
>
> 如果您使用 AWS CLI 从 Amazon S3 存储桶上传 .zip 文件，则该存储桶必须与您的函数位于同一个 AWS 区域 中。

要通过 AWS CLI 使用 .zip 文件创建新函数，则必须指定以下内容：

- 函数的名称 (`--function-name`)
- 函数的运行时系统 (`--runtime`)
- 函数的执行角色 (`--role`) 的 Amazon 资源名称（ARN）
- 函数代码 (`--handler`) 中处理程序方法的名称

还必须指定 .zip 文件的位置。如果 .zip 文件位于本地生成计算机上的文件夹中，请使用 `--zip-file` 选项指定文件路径，如以下示例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime python3.12 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

要指定 .zip 文件在 Amazon S3 存储桶中的位置，请使用 `--code` 选项，如以下示例命令所示。您只需对版本控制对象使用 `S3ObjectVersion` 参数。

```
aws lambda create-function --function-name myFunction \
--runtime python3.12 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

要使用 CLI 更新现有函数，请使用 `--function-name` 参数指定函数的名称。您还必须指定要用于更新函数代码的 .zip 文件的位置。如果 .zip 文件位于本地生成计算机上的文件夹中，请使用 `--zip-file` 选项指定文件路径，如以下示例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

要指定 .zip 文件在 Amazon S3 存储桶中的位置，请使用 `--s3-bucket` 和 `--s3-key` 选项，如以下示例命令所示。您只需对版本控制对象使用 `--s3-object-version` 参数。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

使用 Lambda API 通过 .zip 文件创建和更新函数

要使用 .zip 文件归档创建和更新函数，请使用以下 API 操作：

- CreateFunction
- UpdateFunctionCode

使用 AWS SAM 通过 .zip 文件创建和更新函数

AWS Serverless Application Model（AWS SAM）是一个工具包，可帮助简化在 AWS 上构建和运行无服务器应用程序的过程。您可以在 YAML 或 JSON 模板中为应用程序定义资源，并使用 AWS SAM 命令行界面（AWS SAM CLI）构建、打包和部署应用程序。当您通过 AWS SAM 模板构建 Lambda 函数时，AWS SAM 会使用您的函数代码和您指定的任何依赖项自动创建 .zip 部署包或容器映像。要了解有关使用 AWS SAM 构建和部署 Lambda 函数的更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 AWS SAM 入门。

您可以使用 AWS SAM 创建使用现有 .zip 文件归档的 Lambda 函数。要使用 AWS SAM 创建 Lambda 函数，您可以将 .zip 文件保存在 Amazon S3 存储桶或生成计算机上的本地文件夹中。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

在 AWS SAM 模板中，`AWS::Serverless::Function` 资源将指定 Lambda 函数。在此资源中，设置以下属性以创建使用 .zip 文件归档的函数：

- `PackageType` – 设置为 `Zip`
- `CodeUri` – 设置为函数代码的 Amazon S3 URI、本地文件夹的路径或 FunctionCode 对象

- `Runtime` – 设置为您选择的运行时系统

使用 AWS SAM，如果 .zip 文件大于 50MB，则不需要先将其上传到 Amazon S3 存储桶。AWS SAM 可以从本地生成计算机上的某个位置上传最大允许大小为 250MB（已解压缩）的 .zip 程序包。

要了解有关在 AWS SAM 中使用 .zip 文件部署函数的更多信息，请参阅《AWS SAM 开发人员指南》中的 AWS::Serverless::Function。

使用 AWS CloudFormation 通过 .zip 文件创建和更新函数

您可以使用 AWS CloudFormation 创建使用 .zip 文件归档的 Lambda 函数。要从 .zip 文件创建 Lambda 函数，必须先将您的文件上传到 Amazon S3 存储桶。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

对于 Node.js 和 Python 运行时系统，您还可以在 AWS CloudFormation 模板中提供内联源代码。然后 AWS CloudFormation 会在您构建函数时创建包含代码的 .zip 文件。

使用现有 .zip 文件

在 AWS CloudFormation 模板中，`AWS::Lambda::Function` 资源将指定 Lambda 函数。在此资源中，设置以下属性以创建使用 .zip 文件归档的函数：

- `PackageType` – 设置为 `Zip`
- `Code` – 在 `S3Bucket` 和 `S3Key` 字段中输入 Amazon S3 存储桶名称和 .zip 文件名。
- `Runtime` – 设置为您选择的运行时系统

从内联代码创建 .zip 文件

您可以在 AWS CloudFormation 模板中声明使用 Python 或 Node.js 内联编写的简单函数。由于代码嵌入在 YAML 或 JSON 中，因此您无法向部署包添加任何外部依赖关系。这意味着您的函数必须使用运行时系统中包含的 AWS SDK 版本。模板的要求（例如必须转义某些字符）也让使用 IDE 的语法检查和代码完成功能变得更加困难。这意味着模板可能需要额外的测试。由于这些限制，内联声明函数最适合用于不经常更改的非常简单的代码。

要从 Node.js 和 Python 运行时系统的内联代码创建 .zip 文件，请在模板的 `AWS::Lambda::Function` 资源中设置以下属性：

- `PackageType` – 设置为 `Zip`
- `Code` – 在 `ZipFile` 字段中输入函数代码

- `Runtime` – 设置为您选择的运行时系统

AWS CloudFormation 生成的 .zip 文件不能超过 4MB。要了解有关在 AWS CloudFormation 中使用 .zip 文件部署函数的更多信息，请参阅《AWS CloudFormation 用户指南》中的 AWS::Lambda::Function。

使用容器镜像部署 Python Lambda 函数

有三种方法可以为 Python Lambda 函数构建容器映像：

- 使用 Python 的 AWS 基本映像

  AWS 基本映像会预加载一个语言运行时系统、一个用于管理 Lambda 和函数代码之间交互的运行时系统接口客户端，以及一个用于本地测试的运行时系统接口仿真器。

- 使用 AWS 仅限操作系统的基础镜像

  AWS 仅限操作系统的运行时系统包含 Amazon Linux 发行版和运行时系统接口模拟器。这些镜像通常用于为编译语言（例如 Go 和 Rust）以及 Lambda 未提供基础映像的语言或语言版本（例如 Node.js 19）创建容器镜像。您也可以使用仅限操作系统的基础映像来实施自定义运行时系统。要使映像与 Lambda 兼容，您必须在映像中包含 Python 的运行时系统接口客户端。

- 使用非 AWS 基本映像

  您还可以使用其他容器注册表的备用基本映像，例如 Alpine Linux 或 Debian。您还可以使用您的组织创建的自定义映像。要使映像与 Lambda 兼容，您必须在映像中包含 Python 的运行时系统接口客户端。

> **Tip**
>
> 要缩短 Lambda 容器函数激活所需的时间，请参阅 Docker 文档中的使用多阶段构建。要构建高效的容器映像，请遵循编写 Dockerfiles 的最佳实践。

此页面介绍了如何为 Lambda 构建、测试和部署容器映像。

主题

Python AWS 基本映像

AWS为 Python 提供了以下基本映像：

标签	运行时	操作系统	Dockerfile	淘汰
3.12	Python 3.12	Amazon Linux 2023	GitHub 上适用于 Python 3.12 的 Dockerfile	未计划
3.11	Python 3.11	Amazon Linux 2	GitHub 上适用于 Python 3.11 的 Dockerfile	未计划
3.10	Python 3.10	Amazon Linux 2	GitHub 上适用于 Python 3.10 的 Dockerfile	未计划
3.9	Python 3.9	Amazon Linux 2	GitHub 上的适用于 Python 3.9 的 Dockerfile	未计划

Amazon ECR 存储库：gallery.ecr.aws/lambda/python

Python 3.12 及更高版本的基础映像基于 Amazon Linux 2023 最小容器映像。Python 3.8-3.11 基础映像基于 Amazon Linux 2 映像。与 Amazon Linux 2 相比，基于 AL2023 的映像具有多项优势，包括较小的部署占用空间以及 `glibc` 等更新版本的库。

基于 AL2023 的映像使用 `microdnf`（符号链接为 `dnf`）作为软件包管理器，而不是 Amazon Linux 2 中的默认软件包管理器 `yum`。`microdnf` 是 `dnf` 的独立实现。有关基于 AL2023 的映像中已包含软件包的列表，请参阅 Comparing packages installed on Amazon Linux 2023 Container Images 中的 Minimal Container 列。有关 AL2023 和 Amazon Linux 2 之间区别的更多信息，请参阅 AWS 计算博客上的 Introducing the Amazon Linux 2023 runtime for AWS Lambda。

> **Note**
>
> 要在本地运行基于 AL2023 的映像，包括使用 AWS Serverless Application Model（AWS SAM），您必须使用 Docker 版本 20.10.10 或更高版本。

基本映像中的依赖项搜索路径

在代码中使用 `import` 语句时，Python 运行时系统会搜索其搜索路径中的目录，直到找到相应模块或包。默认情况下，运行时系统会首先搜索 `{LAMBDA_TASK_ROOT}` 目录。如果映像含有包含运行时系统的库的某个版本，则此版本将优先于运行时系统中包含的版本。

搜索路径中的其他步骤取决于您使用的 Python Lambda 基本映像的版本：

- Python 3.11 及更高版本：包含运行时系统的库和安装了 pip 的库均安装在 `/var/lang/lib/python3.11/site-packages` 目录中。此目录优先于搜索路径中的 `/var/runtime`。您可以通过使用 pip 安装更新版本来覆盖 SDK。您可以使用 pip 来验证包含运行时系统的 SDK 及其依赖项是否与您安装的任何程序包兼容。
- Python 3.8-3.10：包含运行时系统的库安装在 `/var/runtime` 目录中。安装了 pip 的库安装在 `/var/lang/lib/python3.x/site-packages` 目录中。此 `/var/runtime` 目录优先于搜索路径中的 `/var/lang/lib/python3.x/site-packages`。

通过添加以下代码段，您可以查看 Lambda 函数的完整搜索路径。

```
import sys

search_path = sys.path
print(search_path)
```

使用 Python 的 AWS 基本映像

先决条件

要完成本节中的步骤，您必须满足以下条件：

- AWS CLI 版本 2
- Docker（Python 3.12 及更高版本基础映像的最低版本为 20.10.10）
- Python

从基本映像创建映像

要从 Python 的 AWS 的基本映像创建容器映像

1. 为项目创建一个目录，然后切换到该目录。

   ```
   mkdir example
   cd example
   ```

2. 创建名为 `lambda_function.py` 的新文件。您可以将以下示例函数代码添加到文件中进行测试，也可以使用您自己的函数代码。

Example Python 函数

```
import sys
def handler(event, context):
    return 'Hello from AWS Lambda using Python' + sys.version + '!'
```

3. 创建名为 `requirements.txt` 的新文件。如果您使用的是上一步中的示例函数代码，则可以将文件留空，因为没有依赖项。否则，请列出每个必需的库。例如，如果您的函数使用的是 AWS SDK for Python (Boto3)，则 `requirements.txt` 应如下所示：

   Example requirements.txt

   ```
   boto3
   ```

4. 使用以下配置创建一个新的 Dockerfile。

   - 将 `FROM` 属性设置为基本映像的 URI。
   - 使用 `COPY` 命令将函数代码和运行时系统依赖项复制到 `{LAMBDA_TASK_ROOT}`，此为 Lambda 定义的环境变量。
   - 将 `CMD` 参数设置为 Lambda 函数处理程序。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 `USER` 指令时默认为 `root` 用户。

   Example Dockerfile

   ```
   FROM public.ecr.aws/lambda/python:3.12

   # Copy requirements.txt
   COPY requirements.txt ${LAMBDA_TASK_ROOT}

   # Install the specified packages
   RUN pip install -r requirements.txt

   # Copy function code
   COPY lambda_function.py ${LAMBDA_TASK_ROOT}
   ```

```
# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.handler" ]
```

5. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 docker-image 并为其提供 test 标签。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 该命令指定了 --platform linux/amd64 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 --platform linux/arm64 选项。

（可选）在本地测试映像

1. 使用 docker run 命令启动 Docker 映像。在此示例中，docker-image 是映像名称，test 是标签。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令会将映像作为容器运行，并在 localhost:9000/2015-03-31/functions/function/invocations 创建本地端点。

> **Note**
>
> 如果为 ARM64 指令集架构创建 Docker 映像，请务必使用 --platform linux/*arm64* 选项，而不是 --platform linux/*amd64* 选项。

2. 在新的终端窗口中，将事件发布到本地端点。

Linux/macOS

在 Linux 和 macOS 中，运行以下 curl 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，运行以下 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 获取容器 ID。

```
docker ps
```

4. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 111122223333 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。

- 将 <ECRrepositoryUri> 替换为复制的 repositoryUri。确保 URI 末尾包含 :latest。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 :latest。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 ImageUri，指定之前的存储库 URI。确保 URI 末尾包含 :latest。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
```

```
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，`--publish` 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

将备用基本映像与运行时系统接口客户端配合使用

如果使用仅限操作系统的基础映像或者备用基础映像，则必须在映像中包括运行时系统接口客户端。运行时系统接口客户端可扩展 将 Lambda 运行时 API 用于自定义运行时，用于管理 Lambda 和函数代码之间的交互。

使用 pip 程序包管理器安装 Python 的运行时系统接口客户端：

```
pip install awslambdaric
```

您还可以从 GitHub 下载 Python 运行时接口客户端。

以下示例演示了如何使用非 AWS 基本映像为 Python 构建容器映像。示例 Dockerfile 使用官方 Python 基本映像。Dockerfile 包含 Python 的运行时系统接口客户端。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- AWS CLI 版本 2
- Docker
- Python

从备用基本映像创建映像

要从非 AWS 基本映像创建容器映像

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir example
cd example
```

2. 创建名为 `lambda_function.py` 的新文件。您可以将以下示例函数代码添加到文件中进行测试，也可以使用您自己的函数代码。

   Example Python 函数

```
import sys
def handler(event, context):
    return 'Hello from AWS Lambda using Python' + sys.version + '!'
```

3. 创建名为 `requirements.txt` 的新文件。如果您使用的是上一步中的示例函数代码，则可以将文件留空，因为没有依赖项。否则，请列出每个必需的库。例如，如果您的函数使用的是 AWS SDK for Python (Boto3)，则 `requirements.txt` 应如下所示：

   Example requirements.txt

```
boto3
```

4. 创建新 Dockerfile。以下 Dockerfile 使用官方 Python 基本映像而不是 AWS 基本映像。Dockerfile 包含运行时系统接口客户端，该客户端可使映像与 Lambda 兼容。以下示例 Dockerfile 将使用多阶段构建。

   - 将 `FROM` 属性设置为基本映像。
   - 将 `ENTRYPOINT` 设置为您希望 Docker 容器在启动时运行的模块。在本例中，模块为运行时系统接口客户端。
   - 将 `CMD` 设置为 Lambda 函数处理程序。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 `USER` 指令时默认为 `root` 用户。

Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM python:3.12 AS build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

# Install the function's dependencies
RUN pip install \
    --target ${FUNCTION_DIR} \
        awslambdaric

# Use a slim version of the base Python image to reduce the final image size
FROM python:3.12-slim

# Include global arg in this stage of the build
ARG FUNCTION_DIR
# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/local/bin/python", "-m", "awslambdaric" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "lambda_function.handler" ]
```

5. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 `docker-image` 并为其提供 `test` 标签。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 该命令指定了 `--platform linux/amd64` 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 `--platform linux/arm64` 选项。

（可选）在本地测试映像

使用运行时系统接口仿真器在本地测试映像。您可以将仿真器构建到映像中，也可以使用以下程序将其安装在本地计算机上。

在本地计算机上安装并运行运行时系统接口仿真器

1. 从项目目录中，运行以下命令以从 GitHub 下载运行时系统接口仿真器（x86-64 架构）并将其安装在本地计算机上。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   要安装 arm64 仿真器，请将上一条命令中的 GitHub 存储库 URL 替换为以下内容：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   ```

```
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

要安装 arm64 模拟器，请将 $downloadLink 替换为以下内容：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. 使用 docker run 命令启动 Docker 映像。请注意以下几点：

   - docker-image 是映像名称，test 是标签。
   - /usr/local/bin/python -m awslambdaric lambda_function.handler 是 ENTRYPOINT，后跟您 Dockerfile 中的 CMD。

   Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/python -m awslambdaric lambda_function.handler
```

   PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/local/bin/python -m awslambdaric lambda_function.handler
```

   此命令会将映像作为容器运行，并在 localhost:9000/2015-03-31/functions/function/invocations 创建本地端点。

   > **Note**
   >
   > 如果为 ARM64 指令集架构创建 Docker 映像，请务必使用 --platform linux/*arm64* 选项，而不是 --platform linux/*amd64* 选项。

3. 将事件发布到本地端点。

Linux/macOS

在 Linux 和 macOS 中，运行以下 `curl` 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，运行以下 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 获取容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 `111122223333` 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
```

```
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。
   - 将 `<ECRrepositoryUri>` 替换为复制的 `repositoryUri`。确保 URI 末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 `ImageUri`，指定之前的存储库 URI。确保 URI 末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，`--publish` 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

对于如何从 Alpine 基本镜像创建 Python 镜像的示例，请参阅AWS博客上的 Lambda 容器镜像支持。

使用 Python Lambda 函数的层

Lambda 层是包含补充代码或数据的 .zip 文件存档。层通常包含库依赖项、自定义运行时系统或配置文件。创建层涉及三个常见步骤：

1. 打包层内容。此步骤需要创建 .zip 文件存档，其中包含要在函数中使用的依赖项。
2. 在 Lambda 中创建层。
3. 将层添加到函数。

本主题包含有关如何正确打包并创建具有外部库依赖项的 Python Lambda 层的步骤和指南。

主题

先决条件

要完成本部分中的步骤，您必须满足以下条件：

- Python 3.11 和 pip 程序包安装程序
- AWS CLI 版本 2

在整个主题中，我们会引用 awsdocs GitHub 存储库中的 `layer-python` 示例应用程序。该应用程序包含用于下载依赖项并生成层的脚本。该应用程序还包含相应的函数，函数使用来自层的依赖项。创建层后，即可部署并调用相应的函数来验证一切是否正常运行。由于使用 Python 3.11 运行时系统来运行这些函数，因此这些层还必须与 Python 3.11 兼容。

在 `layer-python` 示例应用程序中，有两个示例：

- 第一个示例展示如何将 `requests` 库打包到 Lambda 层中。`layer/` 目录包含用于生成层的脚本。`function/` 目录包含示例函数，用于帮助测试该层是否正常工作。本教程的大部分内容将演示如何创建并打包该层。
- 第二个示例展示如何将 numpy 库打包到 Lambda 层中。`layer-numpy/` 目录包含用于生成层的脚本。`function-numpy/` 目录包含示例函数，用于帮助测试该层是否正常工作。有关如何创建并打包该层的示例，请参阅the section called “使用 manylinux Wheel 发行版”。

Python 层与 Amazon Linux 的兼容性

创建层的第一步是将所有层内容捆绑到.zip 文件存档中。由于 Lambda 函数在 Amazon Linux 上运行，因此层内容必须能够在 Linux 环境中编译和构建。

在 Python 中，除了源代码发行版外，大多数程序包还可作为 Wheel（`.whl` 文件）使用。每个 Wheel 都是一种已构建的发行版，支持 Python 版本、操作系统和机器指令集的特定组合。

Wheel 有助于确保层与 Amazon Linux 兼容。下载依赖项时，如果可能，请下载通用 Wheel。（默认情况下，如果通用 Wheel 可用，则 `pip` 将安装通用 Wheel。）通用 Wheel 包含 `any` 作为平台标签，表示与包括 Amazon Linux 在内的所有平台兼容。

在接下来的示例中，您要将 `requests` 库打包到 Lambda 层中。`requests` 库是可用作通用 Wheel 的示例程序包。

并非所有 Python 程序包都作为通用 Wheel 发行。例如，numpy 有多个 Wheel 发行版，每个版本都支持一组不同的平台。对于此类程序包，请下载 `manylinux` 发行版来确保与 Amazon Linux 兼容。有关如何打包此类层的详细说明，请参阅the section called “使用 manylinux Wheel 发行版”。

在极少数情况下，Python 程序包可能无法作为 Wheel 使用。如果只存在源代码发行版 (sdist)，则建议在基于 Amazon Linux 2023 基本容器映像的 Docker 环境中安装并打包依赖项。如果想包含以其他语言（例如 C/C++）编写的自定义库，也推荐使用这种方法。此方法在 Docker 中模仿 Lambda 执行环境，可确保非 Python 程序包依赖项与 Amazon Linux 兼容。

Python 运行时系统的层路径

当您向函数添加层时，Lambda 会将层内容加载到该执行环境的 `/opt` 目录中。对于每个 Lambda 运行时系统，PATH 变量都包括 `/opt` 目录中的特定文件夹路径。为确保 PATH 变量能够获取层内容，层 .zip 文件应在以下文件夹路径中具有依赖项：

- `python`

- python/lib/python3.*x*/site-packages

例如，您在本教程中创建生成的层.zip 文件具有以下目录结构：

```
layer_content.zip
# python
    # lib
        # python3.11
            # site-packages
                # requests
                # <other_dependencies> (i.e. dependencies of the requests package)
                # ...
```

requests 库位于 python/lib/python3.11/site-packages 目录中，位置正确。这可确保 Lambda 在函数调用期间可以找到该库。

打包层内容

在本示例中，您要将 Python requests 库打包到一个层 .zip 文件中。完成以下步骤，安装并打包层内容。

安装并打包层内容

1. 克隆 aws-lambda-developer-guide GitHub 存储库，其中包含 sample-apps/layer-python 目录中需要的示例代码。

   ```
   git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
   ```

2. 导航到 layer-python 示例应用程序的 layer 目录。此目录包含用于正确创建并打包层的脚本。

   ```
   cd aws-lambda-developer-guide/sample-apps/layer-python/layer
   ```

3. 检查 requirements.txt 文件。此文件定义了要包含在层中的依赖项，即 requests 库。您可以更新此文件，纳入要包含在层中的任何依赖项。

   Example requirements.txt

   ```
   requests==2.31.0
   ```

4. 确保拥有运行这两个脚本的权限。

```
chmod 744 1-install.sh && chmod 744 2-package.sh
```

5. 使用以下命令运行 1-install.sh 脚本：

```
./1-install.sh
```

此脚本使用 venv 来创建名为 create_layer 的 Python 虚拟环境，随后在 create_layer/lib/python3.11/site-packages 目录中安装所有必需的依赖项。

Example 1-install.sh

```
python3.11 -m venv create_layer
source create_layer/bin/activate
pip install -r requirements.txt
```

6. 使用以下命令运行 2-package.sh 脚本：

```
./2-package.sh
```

此脚本将内容从 create_layer/lib 目录复制到名为 python 的新目录中，随后将 python 目录的内容压缩到一个名为 layer_content.zip 的文件中。这便是层的 .zip 文件。您可以解压缩文件，验证是否包含正确的文件结构，如 the section called “Python 运行时系统的层路径”部分所示。

Example 2-package.sh

```
mkdir python
cp -r create_layer/lib python/
zip -r layer_content.zip python
```

创建层

在本部分，您会获取在上一部分中生成的 layer_content.zip 文件，将其作为 Lambda 层上传。您可以使用 AWS Management Console 上传层，也可以通过 AWS Command Line Interface（AWS CLI）使用 Lambda API 上传层。上传层 .zip 文件时，在以下 PublishLayerVersion AWS CLI 命令中，将 python3.11 指定为兼容的运行时系统，并将 arm64 指定为兼容的架构。

```
aws lambda publish-layer-version --layer-name python-requests-layer \
```

```
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes python3.11 \
    --compatible-architectures "arm64"
```

注意响应中的 `LayerVersionArn`，与 `arn:aws:lambda:us-east-1:123456789012:layer:python-requests-layer:1` 类似。在本教程的下一步中，在将层添加到函数时，您要用到此 Amazon 资源名称（ARN）。

将层添加到函数

在本部分，您要部署在函数代码中使用 `requests` 库的示例 Lambda 函数，然后附加该层。要部署该函数，您需要一个 the section called “执行角色（函数访问其他资源的权限）”。如果目前没有执行角色，则按照可折叠部分中的步骤操作。

（可选）创建执行角色

创建执行角色

1. 在 IAM 控制台中，打开 Roles（角色）页面。
2. 选择创建角色。
3. 创建具有以下属性的角色。

   - Trusted entity（可信任的实体）– Lambda。
   - Permissions（权限）– AWSLambdaBasicExecutionRole。
   - Role name（角色名称）– **lambda-role**。

   AWSLambdaBasicExecutionRole 策略具有函数将日志写入 CloudWatch Logs 所需的权限。

Lambda 函数代码会导入 `requests` 库，发出简单的 HTTP 请求，然后返回状态代码和正文。

```
import requests

def lambda_handler(event, context):
    print(f"Version of requests library: {requests.__version__}")
    request = requests.get('https://api.github.com/')
    return {
        'statusCode': request.status_code,
        'body': request.text
    }
```

部署 Lambda 函数

1. 导航到 function/ 目录。如果当前在 layer/ 目录中，请运行以下命令：

```
cd ../function
```

2. 使用以下命令创建 .zip 文件部署包：

```
zip my_deployment_package.zip lambda_function.py
```

3. 部署函数。在以下 AWS CLI 命令中，将 --role 参数替换为执行角色 ARN：

```
aws lambda create-function --function-name python_function_with_layer \
    --runtime python3.11 \
    --architectures "arm64" \
    --handler lambda_function.lambda_handler \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://my_deployment_package.zip
```

4. 接下来，将层附加到函数。在以下 AWS CLI 命令中，将 --layers 参数替换为之前记下的层版本 ARN：

```
aws lambda update-function-configuration --function-name python_function_with_layer
 \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:python-requests-layer:1"
```

5. 最后，尝试使用以下 AWS CLI 命令调用函数：

```
aws lambda invoke --function-name python_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

应看到类似如下内容的输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

输出 response.json 文件包含有关响应的详细信息。

（可选）清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除 Lambda 层

1. 打开 Lambda 控制台的 Layers page（层页面）。
2. 选择您创建的层。
3. 选择删除，然后再次选择删除。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择删除。

使用 **manylinux** Wheel 发行版

有时，要作为依赖项包含的程序包没有通用 Wheel（具体来说，就是没有 any 作为平台标签）。在这种情况下，请改为下载支持 manylinux 的 Wheel。此举可确保 Layer 库与 Amazon Linux 兼容。

numpy 是没有通用 Wheel 的程序包。如果要将 numpy 程序包包含在层中，则可以完成以下示例步骤来正确安装并打包层。

安装并打包层内容

1. 克隆 aws-lambda-developer-guide GitHub 存储库，其中包含 sample-apps/layer-python 目录中需要的示例代码。

```
git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
```

2. 导航到 layer-python 示例应用程序的 layer-numpy 目录。此目录包含用于正确创建并打包层的脚本。

```
cd aws-lambda-developer-guide/sample-apps/layer-python/layer-numpy
```

3. 检查 requirements.txt 文件。此文件定义了要包含在层中的依赖项，即 numpy 库。您可以指定与 Python 3.11、Amazon Linux 以及 x86_64 指令集兼容的 manylinux Wheel 发行版的 URL：

   Example requirements.txt

   ```
   https://files.pythonhosted.org/packages/3a/d0/
   edc009c27b406c4f9cbc79274d6e46d634d139075492ad055e3d68445925/numpy-1.26.4-cp311-
   cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
   ```

4. 确保拥有运行这两个脚本的权限。

   ```
   chmod 744 1-install.sh && chmod 744 2-package.sh
   ```

5. 使用以下命令运行 1-install.sh 脚本：

   ```
   ./1-install.sh
   ```

   此脚本使用 venv 来创建名为 create_layer 的 Python 虚拟环境，随后在 create_layer/lib/python3.11/site-packages 目录中安装所有必需的依赖项。在这种情况下，pip 命令有所不同，因为必须将 --platform 标签指定为 manylinux2014_x86_64。此操作告诉 pip 要安装正确的 manylinux Wheel，即便本地计算机使用的是 macOS 或 Windows 也是如此。

   Example 1-install.sh

   ```
   python3.11 -m venv create_layer
   source create_layer/bin/activate
   pip install -r requirements.txt --platform=manylinux2014_x86_64 --only-binary=:all:
    --target ./create_layer/lib/python3.11/site-packages
   ```

6. 使用以下命令运行 2-package.sh 脚本：

   ```
   ./2-package.sh
   ```

   此脚本将内容从 create_layer/lib 目录复制到名为 python 的新目录中，随后将 python 目录的内容压缩到一个名为 layer_content.zip 的文件中。这便是层的 .zip 文件。您可以解压缩文件，验证是否包含正确的文件结构，如 the section called “Python 运行时系统的层路径” 部分所示。

Example 2-package.sh

```
mkdir python
cp -r create_layer/lib python/
zip -r layer_content.zip python
```

要将该层上传到 Lambda，请使用以下 PublishLayerVersion AWS CLI 命令：

```
aws lambda publish-layer-version --layer-name python-numpy-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes python3.11 \
    --compatible-architectures "x86_64"
```

注意响应中的 `LayerVersionArn`，与 `arn:aws:lambda:us-east-1:123456789012:layer:python-numpy-layer:1` 类似。要验证层是否按预期运行，请在 `function-numpy` 目录中部署 Lambda 函数。

部署 Lambda 函数

1. 导航到 `function-numpy/` 目录。如果当前在 `layer-numpy/` 目录中，请运行以下命令：

   ```
   cd ../function-numpy
   ```

2. 检查函数代码。函数会导入 numpy 库，创建简单的 numpy 数组，然后返回虚拟状态代码和正文。

   ```
   import json
   import numpy as np

   def lambda_handler(event, context):

       x = np.arange(15, dtype=np.int64).reshape(3, 5)
       print(x)

       return {
           'statusCode': 200,
           'body': json.dumps('Hello from Lambda!')
       }
   ```

3. 使用以下命令创建 .zip 文件部署包：

```
zip my_deployment_package.zip lambda_function.py
```

4. 部署函数。在以下 AWS CLI 命令中，将 `--role` 参数替换为执行角色 ARN：

```
aws lambda create-function --function-name python_function_with_numpy \
    --runtime python3.11 \
    --handler lambda_function.lambda_handler \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://my_deployment_package.zip
```

5. 接下来，将层附加到函数。在以下 AWS CLI 命令中，将 `--layers` 参数替换为层版本 ARN：

```
aws lambda update-function-configuration --function-name python_function_with_numpy
 \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:python-requests-layer:1"
```

6. 最后，尝试使用以下 AWS CLI 命令调用函数：

```
aws lambda invoke --function-name python_function_with_numpy \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

应看到类似如下内容的输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

您可以检查函数日志，验证代码是否将 numpy 数组打印为标准输出。

使用 Lambda 上下文对象检索 Python 函数信息

当 Lambda 运行您的函数时，它会将上下文对象传递到处理程序。此对象提供的方法和属性包含有关调用、函数和执行环境的信息。有关如何将上下文对象传递到函数处理程序的更多信息，请参阅定义采用 Python 的 Lambda 函数处理程序。

上下文方法

- `get_remaining_time_in_millis` – 返回执行超时前剩余的毫秒数。

上下文属性

- `function_name` – Lambda 函数的名称。
- `function_version` – 函数的版本
- `invoked_function_arn` – 用于调用函数的 Amazon Resource Name (ARN)。表明调用者是否指定了版本号或别名。
- `memory_limit_in_mb` – 为函数分配的内存量。
- `aws_request_id` – 调用请求的标识符。
- `log_group_name` – 函数的日志组。
- `log_stream_name` – 函数实例的日志流。
- `identity` – （移动应用程序）有关授权请求的 Amazon Cognito 身份的信息。
  - `cognito_identity_id` – 经过身份验证的 Amazon Cognito 身份。
  - `cognito_identity_pool_id` – 授权调用的 Amazon Cognito 身份池。
- `client_context` – （移动应用程序）客户端应用程序提供给 Lambda 的客户端上下文。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `custom` – 由移动客户端应用程序设置的自定义值的 `dict`。
  - `env` – 由AWS开发工具包提供的环境信息的 `dict`。

适用于 Lambda 的 Powertools（Python）为 Lambda 上下文对象提供了接口定义。接口定义可用于类型提示，也可以用于进一步检查 Lambda 上下文对象的结构。有关接口定义，请参阅 GitHub 上 powertools-lambda-python 存储库中的 lambda_context.py。

以下示例显示记录上下文信息的处理程序函数。

Example handler.py

```
import time

def lambda_handler(event, context):
    print("Lambda function ARN:", context.invoked_function_arn)
    print("CloudWatch log stream name:", context.log_stream_name)
    print("CloudWatch log group name:",  context.log_group_name)
    print("Lambda Request ID:", context.aws_request_id)
    print("Lambda function memory limits in MB:", context.memory_limit_in_mb)
    # We have added a 1 second delay so you can see the time remaining in
 get_remaining_time_in_millis.
    time.sleep(1)
    print("Lambda time remaining in MS:", context.get_remaining_time_in_millis())
```

除了上面列出的选项，您还可以使用适用于 AWS 的 在 AWS Lambda 中检测 Python 代码 X-Ray 开发工具包来识别关键代码路径、跟踪其性能并收集数据以用于分析。

Python Lambda 函数日志记录和监控

AWS Lambda 将自动监控 Lambda 函数并将日志条目发送到 Amazon CloudWatch。您的 Lambda 函数带有一个 CloudWatch Logs 日志组以及函数的每个实例的日志流。Lambda 运行时系统环境会将每次调用的详细信息以及函数代码的其他输出发送到该日志流。有关 CloudWatch Logs 的更多信息，请参阅将 CloudWatch Logs 日志与 Lambda 结合使用。

要从函数代码输出日志，可以使用内置的 `logging` 模块。如需更详细的条目，可以使用任何写入 `stdout` 或 `stderr` 的日志记录库。

输出到日志

要将基本输出发送到日志，您可以使用函数中的 `print` 方法。以下示例记录了 CloudWatch Logs 日志组和流以及事件对象的值。

请注意，如果您的函数使用 Python `print` 语句输出日志，则 Lambda 只能以纯文本格式将日志输出发送到 CloudWatch Logs。要以结构化的 JSON 格式捕获日志，需要使用支持的日志记录库。请参阅the section called “在 Python 中使用 Lambda 高级日志记录控件”了解更多信息。

Example lambda_function.py

```
import os
def lambda_handler(event, context):
    print('## ENVIRONMENT VARIABLES')
    print(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    print(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    print('## EVENT')
    print(event)
```

Example 日志输出

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
/aws/lambda/my-function
2023/08/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 16 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
```

```
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

Python 运行时记录每次调用的 START、END 和 REPORT 行。REPORT 行包括以下数据：

REPORT 行数据字段

- RequestId – 调用的唯一请求 ID。
- Duration（持续时间）– 函数的处理程序方法处理事件所花费的时间。
- Billed Duration（计费持续时间）– 针对调用计费的时间量。
- Memory Size（内存大小）– 分配给函数的内存量。
- Max Memory Used（最大内存使用量）– 函数使用的内存量。如果调用共享执行环境，Lambda 会报告所有调用使用的最大内存。此行为可能会导致报告值高于预期。
- Init Duration（初始持续时间）– 对于提供的第一个请求，为运行时在处理程序方法外部加载函数和运行代码所花费的时间。
- XRAY TraceId – 对于追踪的请求，为 AWS X-Ray 追踪 ID。
- SegmentId – 对于追踪的请求，为 X-Ray 分段 ID。
- Sampled（采样）– 对于追踪的请求，为采样结果。

使用日志记录库

如需更详细的日志，请使用标准库中的日志记录模块，或任何写入 `stdout` 或 `stderr` 的第三方日志记录库。

对于支持的 Python 运行时系统，您可以选择以纯文本还是 JSON 格式捕获使用标准 `logging` 模块创建的日志。要了解更多信息，请参阅 the section called “在 Python 中使用 Lambda 高级日志记录控件”。

目前，所有 Python 运行时的默认日志格式均为纯文本。以下示例显示了如何在 CloudWatch Logs 中以纯文本格式捕获使用标准 `logging` 模块创建的日志输出。

```
import os
import logging
logger = logging.getLogger()
logger.setLevel("INFO")
```

```
def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES')
    logger.info(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    logger.info(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    logger.info('## EVENT')
    logger.info(event)
```

`logger` 的输出包括日志级别、时间戳和请求 ID。

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2023-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES
[INFO]  2023-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    /aws/
lambda/my-function
[INFO]  2023-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    2023/01/31/
[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d
[INFO]  2023-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT
[INFO]  2023-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}
END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 3 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

> **Note**
>
> 当函数的日志格式设置为纯文本时，Python 运行时系统的默认日志级别设置为 WARN。这意味着 Lambda 仅向 CloudWatch Logs 发送 WARN 及以下级别的日志输出。要更改默认日志级别，请使用 Python `logging` `setLevel()` 方法，如本示例代码所示。如果将函数的日志格式设置为 JSON，则建议使用 Lambda 高级日志记录控件来配置函数的日志级别，而不是通过在代码中设置日志级别。要了解更多信息，请参阅 the section called “在 Python 中使用日志级别筛选”。

在 Python 中使用 Lambda 高级日志记录控件

为了让您更好地控制如何捕获、处理和使用函数日志，您可以为支持的 Lambda Python 运行时系统配置以下日志记录选项：

- 日志格式 - 为函数日志选择纯文本或结构化的 JSON 格式
- 日志级别 - 对于 JSON 格式的日志，选择 Lambda 发送到 Amazon CloudWatch 的日志的详细信息级别，例如 ERROR、DEBUG 或 INFO
- 日志组 - 选择您的函数发送日志的目标 CloudWatch 日志组

有关这些日志记录选项的更多信息以及如何通过配置来使用函数的说明，请参阅 the section called “配置函数日志”。

要详细了解如何在 Python Lambda 函数中使用日志格式和日志级别选项，请参阅以下各节中的指南。

在 Python 中使用结构化的 JSON 日志

如果您为函数的日志格式选择 JSON，Lambda 会将 Python 标准日志记录库输出的日志作为结构化的 JSON 发送到 CloudWatch。每个 JSON 日志对象包含至少四个键值对和以下键：

- `"timestamp"` - 生成日志消息的时间
- `"level"` - 分配给消息的日志级别
- `"message"` - 日志消息的内容
- `"requestId"` - 函数调用的唯一请求 ID

Python `logging` 库还可以向此 JSON 对象添加其他键值对，例如 `"logger"`。

以下各节中的示例显示了当您将函数的日志格式配置为 JSON 时，如何在 CloudWatch Logs 中捕获使用 Python `logging` 库生成的日志输出。

请注意，如果您使用 `print` 方法生成基本日志输出（如 the section called “输出到日志” 中所述），则即使您将函数的日志格式配置为 JSON，Lambda 也会将这些输出捕获为纯文本。

使用 Python 日志记录库的标准 JSON 日志输出

以下示例代码段和日志输出显示了当函数的日志格式被设置为 JSON 时，如何在 CloudWatch Logs 中捕获使用 Python `logging` 库生成的标准日志输出。

Example Python 日志记录代码

```
import logging
logger = logging.getLogger()
```

```
def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON 日志记录

```
{
    "timestamp":"2023-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"Inside the handler function",
    "logger": "root",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

在 JSON 中记录额外的参数

当函数的日志格式设置为 JSON 时，通过使用 `extra` 关键字将 Python 词典传递到日志输出，您还可以使用标准 Python `logging` 库记录其他参数。

Example Python 日志记录代码

```
import logging

def lambda_handler(event, context):
    logging.info(
        "extra parameters example",
        extra={"a":"b", "b": [3]},
    )
```

Example JSON 日志记录

```
{
  "timestamp": "2023-11-02T15:26:28Z",
  "level": "INFO",
  "message": "extra parameters example",
  "logger": "root",
  "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
  "a": "b",
  "b": [
    3
  ]
}
```

在 JSON 中记录异常

以下代码段显示了当您将日志格式配置为 JSON 时，如何在函数的日志输出中捕获 Python 异常。请注意，使用 `logging.exception` 生成的日志输出被分配了日志级别 ERROR。

Example Python 日志记录代码

```
import logging

def lambda_handler(event, context):
    try:
        raise Exception("exception")
    except:
        logging.exception("msg")
```

Example JSON 日志记录

```
{
  "timestamp": "2023-11-02T16:18:57Z",
  "level": "ERROR",
  "message": "msg",
  "logger": "root",
  "stackTrace": [
    "  File \"/var/task/lambda_function.py\", line 15, in lambda_handler\n    raise
 Exception(\"exception\")\n"
  ],
  "errorType": "Exception",
  "errorMessage": "exception",
  "requestId": "3f9d155c-0f09-46b7-bdf1-e91dab220855",
  "location": "/var/task/lambda_function.py:lambda_handler:17"
}
```

使用其他日志记录工具的 JSON 结构化日志

如果您的代码已经使用其他日志记录库（例如 Powertools for AWS Lambda）来生成 JSON 结构化日志，则无需进行任何更改。AWS Lambda 不会对任何已采用 JSON 编码的日志进行双重编码。即使您将函数配置为使用 JSON 日志格式，您的日志输出也会以您定义的 JSON 结构显示在 CloudWatch 中。

以下示例显示了如何在 CloudWatch Logs 中捕获使用 Powertools for AWS Lambda 软件包生成的日志输出。无论您的函数的日志配置设置为 JSON 还是 TEXT，此日志输出的格式都相同。有关使用 Powertools for AWS Lambda 的更多信息，请参阅 the section called “将 Powertools for AWS

Lambda（Python）和 AWS SAM 用于结构化日志记录” 和 the section called “将 Powertools for AWS Lambda（Python）和 AWS CDK 用于结构化日志记录”

Example Python 日志记录代码段（使用 Powertools for AWS Lambda）

```
from aws_lambda_powertools import Logger

logger = Logger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON 日志记录（使用 Powertools for AWS Lambda）

```
{
    "level": "INFO",
    "location": "lambda_handler:7",
    "message": "Inside the handler function",
    "timestamp": "2023-10-31 22:38:21,010+0000",
    "service": "service_undefined",
    "xray_trace_id": "1-654181dc-65c15d6b0fecbdd1531ecb30"
}
```

在 Python 中使用日志级别筛选

通过配置日志级别筛选，您可以选择仅将特定日志记录级别或更低级别的日志发送到 CloudWatch Logs。要了解如何为您的函数配置日志级别筛选，请参阅 the section called “日志级别筛选”。

为了让 AWS Lambda 根据日志级别筛选应用程序日志，您的函数必须使用 JSON 格式的日志。您可以通过两种方式实现这一点：

- 使用标准 Python `logging` 库创建日志输出，并将您的函数配置为使用 JSON 日志格式。然后 AWS Lambda 会使用 the section called “在 Python 中使用结构化的 JSON 日志” 中所述的 JSON 对象中的“级别”键值对筛选日志输出。要了解如何配置函数的日志格式，请参阅 the section called “配置函数日志”。
- 使用其他日志记录库或方法在代码中创建 JSON 结构化日志，其中包含定义日志输出级别的“级别”键值对。例如，您可以使用 Powertools for AWS Lambda 通过代码生成 JSON 结构化日志输出。

  您也可以使用 print 语句输出包含日志级别标识符的 JSON 对象。以下 print 语句生成 JSON 格式的输出，其中日志级别设置为 INFO。如果您的函数的日志级别设置为 INFO、DEBUG 或 TRACE，则 AWS Lambda 会将 JSON 对象发送到 CloudWatch Logs。

```
print('{"msg":"My log message", "level":"info"}')
```

要让 Lambda 筛选函数的日志，还必须在 JSON 日志输出中包含一个 "timestamp" 键值对。必须以有效的 RFC 3339 时间戳格式指定时间。如果您未提供有效的时间戳，Lambda 将为日志分配 INFO 级别并为您添加时间戳。

在 Lambda 控制台中查看日志

调用 Lambda 函数后，您可以使用 Lambda 控制台查看日志输出。

如果可以在嵌入式代码编辑器中测试代码，则可以在执行结果中找到日志。使用控制台测试功能调用函数时，可以在详细信息部分找到日志输出。

在 CloudWatch 控制台中查看日志

您可以使用 Amazon CloudWatch 控制台查看所有 Lambda 函数调用的日志。

使用 CloudWatch 控制台查看日志

1. 打开 CloudWatch 控制台的 Log groups（日志组页面）。
2. 选择您的函数 (/aws/lambda/***your-function-name***) 的日志组。
3. 创建日志流。

每个日志流对应一个函数实例。日志流会在您更新 Lambda 函数以及创建更多实例来处理多个并发调用时显示。要查找特定调用的日志，建议您使用 AWS X-Ray 检测函数。X-Ray 会在追踪中记录有关请求和日志流的详细信息。

使用 AWS CLI 查看日志

AWS CLI 是一种开源工具，让您能够在命令行 Shell 中使用命令与 AWS 服务进行交互。要完成本节中的步骤，您必须拥有 AWS CLI 版本 2。

您可以通过 AWS CLI，使用 `--log-type` 命令选项检索调用的日志。响应包含一个 `LogResult` 字段，其中包含多达 4KB 来自调用的 base64 编码日志。

Example 检索日志 ID

以下示例说明如何从 `LogResult` 字段中检索名为 `my-function` 的函数的日志 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解码日志

在同一命令提示符下，使用 base64 实用程序解码日志。以下示例说明如何为 my-function 检索 base64 编码的日志。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

您应看到以下输出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64 实用程序在 Linux、macOS 和 Ubuntu on Windows 上可用。macOS 用户可能需要使用 base64 -D。

Example get-logs.sh 脚本

在同一命令提示符下，使用以下脚本下载最后五个日志事件。此脚本使用 sed 从输出文件中删除引号，并休眠 15 秒以等待日志可用。输出包括来自 Lambda 的响应，以及来自 get-log-events 命令的输出。

复制以下代码示例的内容并将其作为 `get-logs.sh` 保存在 Lambda 项目目录中。

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux（仅限）

在同一命令提示符下，macOS 和 Linux 用户可能需要运行以下命令以确保脚本可执行。

```
chmod -R 755 get-logs.sh
```

Example 检索最后五个日志事件

在同一命令提示符下，运行以下脚本以获取最后五个日志事件。

```
./get-logs.sh
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
```

```
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


删除日志

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期之后，日志将自动删除。

使用其他日志记录工具和库

Powertools for AWS Lambda（Python）是一个开发人员工具包，用于实施无服务器最佳实践并提高开发人员速度。日志记录实用程序提供经优化的 Lambda 日志记录程序，其中包含有关所有函数的函数上下文的附加信息，输出结构为 JSON。请使用该实用程序执行以下操作：

- 从 Lambda 上下文中捕获关键字段，冷启动并将日志记录输出结构化为 JSON
- 根据指示记录 Lambda 调用事件（默认情况下禁用）

- 通过日志采样仅针对一定百分比的调用输出所有日志（默认情况下禁用）
- 在任何时间点将其他键附加到结构化日志
- 使用自定义日志格式设置程序（自带格式设置程序），从而在与组织的日志记录 RFC 兼容的结构中输出日志

将 Powertools for AWS Lambda（Python）和 AWS SAM 用于结构化日志记录

请按照以下步骤使用 AWS SAM，通过集成的 Powertools for Python 模块来下载、构建和部署示例 Hello World Python 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 `hello world` 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Python 3.9
- AWS CLI 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS SAM 应用程序

1. 使用 Hello World Python 模板初始化该应用程序。

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. 构建应用程序。

```
cd sam-app && sam build
```

3. 部署应用程序。

```
sam deploy --guided
```

4. 按照屏幕上的提示操作。要在交互式体验中接受提供的默认选项，请按 Enter。

> **Note**
>
> 对于 HelloWorldFunction 可能没有定义授权，确定执行此操作吗？，确保输入 y。

5. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 调用 API 端点：

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

7. 要获取该函数的日志，请运行 sam logs。有关更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 使用日志。

```
sam logs --stack-name sam-app
```

该日志输出类似于以下示例：

```
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2023-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
  "message": "Hello world API - HTTP 200",
  "timestamp": "2023-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
```

```
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
```

```
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 17 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

8. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
sam delete
```

管理日志保留日期

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期结束后，日志将自动删除。要设置日志保留日期，请将以下内容添加到您的 AWS SAM 模板中：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
```

```
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

将 Powertools for AWS Lambda（Python）和 AWS CDK 用于结构化日志记录

请按照以下步骤使用 AWS CDK，通过集成的 Powertools for AWS Lambda（Python）模块来下载、构建和部署示例 Hello World Python 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 hello world 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Python 3.9
- AWS CLI 版本 2
- AWS CDK 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS CDK 应用程序

1. 为您的新应用程序创建一个项目目录。

   ```
   mkdir hello-world
   cd hello-world
   ```

2. 初始化该应用程序。

   ```
   cdk init app --language python
   ```

3. 安装 Python 依赖项。

   ```
   pip install -r requirements.txt
   ```

4. 在根文件夹下创建目录 lambda_function。

```
mkdir lambda_function
cd lambda_function
```

5. 创建文件 app.py 并将以下代码添加到该文件中。这是适用于 Lambda 函数的代码。

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. 打开 hello_world 目录。您应该会看到一个名为 hello_world_stack.py 的文件。

```
cd ..
cd hello_world
```

7. 打开 hello_world_stack.py，然后将以下代码添加到该文件中。其中包含 Lambda 构造函数，该构造函数可创建 Lambda 函数，为 Powertools 配置环境变量并将日志保留期设置为一周；还包含 ApiGatewayv1 构造函数，用于创建 REST API。

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
 constructor is in Alpha, o we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
aws_cdk.aws_lambda_python_alpha/README.html
            # Check all Powertools layers versions here: https://
docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
{self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_9,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
```

```
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. 部署您的应用程序。

```
cd ..
cdk deploy
```

9. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. 调用 API 端点：

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

11. 要获取该函数的日志，请运行 sam logs。有关更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 使用日志。

```
sam logs --stack-name HelloWorldStack
```

该日志输出类似于以下示例：

```
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2023-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
    "message": "Hello world API - HTTP 200",
  "timestamp": "2023-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
```

```
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 17 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

12. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
cdk destroy
```

Python 中的 AWS Lambda 函数测试

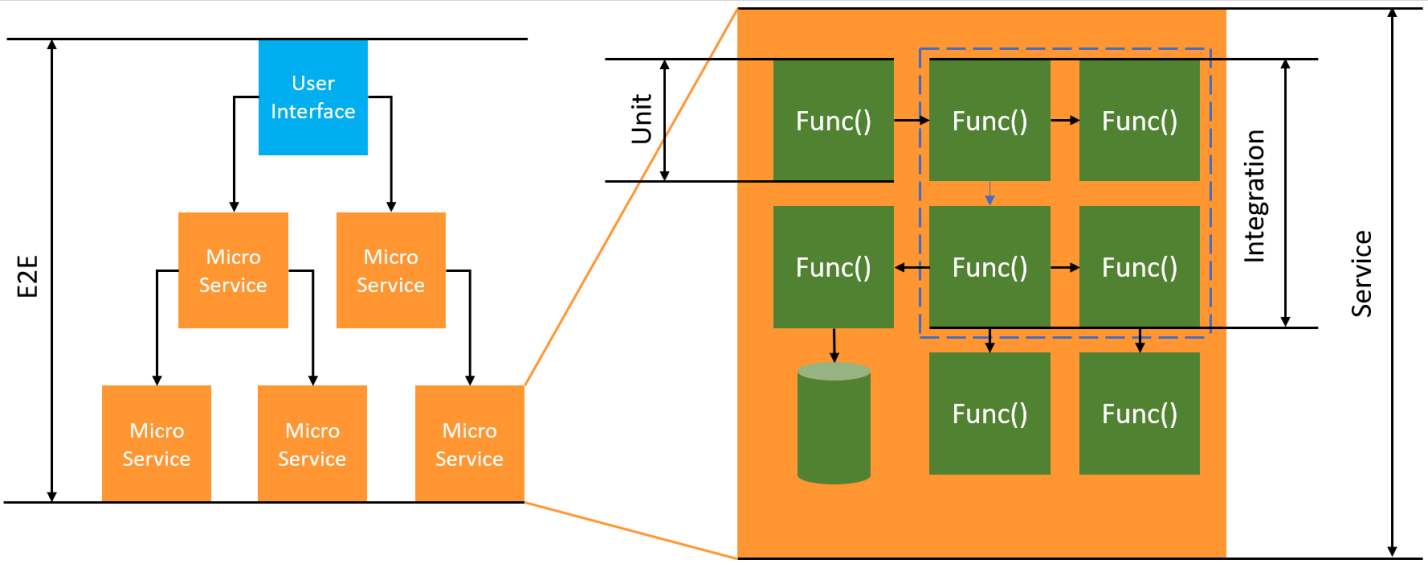
> **Note**
>
> 有关测试无服务器解决方案的技术和最佳实践的完整介绍，请参阅测试函数一章。

测试无服务器函数使用传统的测试类型和技术，但您还必须考虑对无服务器应用程序进行整体测试。基于云的测试将为您的函数和无服务器应用程序的质量提供最准确的衡量。

无服务器应用程序架构包括可通过 API 调用提供关键应用程序功能的托管服务。因此，您的开发周期应包括在函数与服务交互时验证功能的自动化测试。

如果您不创建基于云的测试，则可能会由于本地环境和已部署环境之间的差异而遇到问题。在将代码提升到下一个部署环境（例如 QA、暂存或生产）之前，您的持续集成过程应针对在云端预置的一套资源运行测试。

继续阅读本简短指南，了解无服务器应用程序的测试策略，或者访问无服务器测试示例存储库，深入了解特定于您所选语言和运行时系统的实用示例。



对于无服务器测试，您仍需要编写单元、集成和端到端测试。

- 单元测试 – 针对隔离代码块运行的测试。例如，验证业务逻辑以计算给定特定项目和目的地的配送费用。
- 集成测试 – 涉及通常在云环境中交互的两个或更多组件或服务的测试。例如，验证函数是否会处理队列中的事件。

- 端到端测试 – 验证整个应用程序行为的测试。例如，确保正确设置基础设施，并确保事件在服务之间按预期流动，以记录客户的订单。

测试无服务器应用程序

您通常会使用多种方法来测试无服务器应用程序代码，包括在云端进行测试、使用 Mock 进行测试，以及偶尔使用仿真器进行测试。

在云端进行测试

在云端进行测试对于各个阶段的测试（包括单元测试、集成测试和端到端测试）而言都很有价值。您可以针对部署在云端并与基于云的服务进行交互的代码运行测试。这种方法可以准确衡量代码的质量。

在云端调试 Lambda 函数的一种便捷方法是在控制台中使用测试事件。测试事件是函数的一个 JSON 输入。如果函数不需要输入，则事件可以是空 JSON 文档（{}）。控制台为各种服务集成提供示例事件。在控制台中创建事件后，您可以将其与团队共享，以简化测试并保持一致性。

> **Note**
>
> 在控制台中测试函数是一种快速开始的方法，但自动化测试周期可确保应用程序质量和开发速度。

测试工具

有一些工具和技术可以加速开发反馈循环。例如，AWS SAM Accelerate 和 AWS CDK 监视模式都减少了更新云环境所需的时间。

Moto 是一个用于 Mock 模拟 AWS 服务和资源的 Python 库，以便您无需修改或稍作修改，就能使用装饰器拦截和模拟响应，以此测试函数。

Powertools for AWS Lambda（Python）的验证功可提供装饰器，以便您可以验证 Python 函数的输入事件和输出响应。

有关更多信息，请阅读博文使用 Python 对 Lambda 进行单元测试和 Mock 模拟 AWS 服务。

要减少云部署迭代所涉及的延迟，请参阅 AWS Serverless Application Model（AWS SAM）Accelerate、AWS 云开发工具包（AWS CDK）监视模式。这些工具会监控基础设施和代码的变更情况，再通过创建增量更新并将其自动部署到云环境中来响应这些变更。

使用这些工具的示例可在 Python 测试示例代码存储库中找到。

在 AWS Lambda 中检测 Python 代码

Lambda 与 AWS X-Ray 集成，以帮助您跟踪、调试和优化 Lambda 应用程序。您可以在某个请求遍历应用程序中的资源（其中可能包括 Lambda 函数和其他 AWS 服务）时，使用 X-Ray 跟踪该请求。

要将跟踪数据发送到 X-Ray，您可以使用以下三个开发工具包库之一：

- 适用于 OpenTelemetry 的 AWS 发行版 (ADOT) – 一种安全、可供生产、支持 AWS 的 OpenTelemetry (OTel) SDK 的分发版本。
- AWS X-Ray SDK for Python – 用于生成跟踪数据并将其发送到 X-Ray 的 SDK
- Powertools for AWS Lambda（Python） – 一个开发人员工具包，用于实施无服务器最佳实践并提高开发人员速度。

每个开发工具包均提供了将遥测数据发送到 X-Ray 服务的方法。然后，您可以使用 X-Ray 查看、筛选和获得对应用程序性能指标的洞察，从而发现问题和优化机会。

> **Important**
>
> X-Ray 和 Powertools for AWS Lambda SDK 是 AWS 提供的紧密集成的分析解决方案的一部分。ADOT Lambda Layers 是全行业通用的跟踪分析标准的一部分，该标准通常会收集更多数据，但可能不适用于所有使用案例。您可以使用任一解决方案在 X-Ray 中实现端到端跟踪。要了解有关如何在两者之间进行选择的更多信息，请参阅在 AWS Distro for Open Telemetry 和 X-Ray 开发工具包之间进行选择。

Sections

将 Powertools for AWS Lambda（Python）和 AWS SAM 用于跟踪

请按照以下步骤使用 AWS SAM，通过集成的 Powertools for AWS Lambda（Python）模块来下载、构建和部署示例 Hello World Python 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 hello world 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Python 3.11
- AWS CLI 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS SAM 应用程序

1. 使用 Hello World Python 模板初始化该应用程序。

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.11 --no-tracing
```

2. 构建应用程序。

```
cd sam-app && sam build
```

3. 部署应用程序。

```
sam deploy --guided
```

4. 按照屏幕上的提示操作。要在交互式体验中接受提供的默认选项，请按 Enter。

> **Note**
>
> 对于 HelloWorldFunction 可能没有定义授权，确定执行此操作吗？，确保输入 y。

5. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 调用 API 端点：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

7. 要获取该函数的跟踪信息，请运行 sam traces。

```
sam traces
```

该跟踪输出类似于以下示例：

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
```

```
      - fault count(5XX): 0
      - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

8. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
sam delete
```

X-Ray 无法跟踪对应用程序的所有请求。X-Ray 将应用采样算法确保跟踪有效，同时仍会提供所有请求的一个代表性样本。采样率是每秒 1 个请求和 5% 的其他请求。您无法为函数配置此 X-Ray 采样率。

将 Powertools for AWS Lambda（Python）和 AWS CDK 用于跟踪

请按照以下步骤使用 AWS CDK，通过集成的 Powertools for AWS Lambda（Python）模块来下载、构建和部署示例 Hello World Python 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 hello world 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Python 3.11
- AWS CLI 版本 2
- AWS CDK 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS CDK 应用程序

1. 为您的新应用程序创建一个项目目录。

```
mkdir hello-world
cd hello-world
```

2. 初始化该应用程序。

```
cdk init app --language python
```

3. 安装 Python 依赖项。

```
pip install -r requirements.txt
```

4. 在根文件夹下创建目录 lambda_function。

```
mkdir lambda_function
cd lambda_function
```

5. 创建文件 app.py 并将以下代码添加到该文件中。这是适用于 Lambda 函数的代码。

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)
```

```
    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. 打开 hello_world 目录。您应该会看到一个名为 hello_world_stack.py 的文件。

```
cd ..
cd hello_world
```

7. 打开 hello_world_stack.py，然后将以下代码添加到该文件中。其中包含 Lambda 构造函数，该构造函数可创建 Lambda 函数，为 Powertools 配置环境变量并将日志保留期设置为一周；还包含 ApiGatewayv1 构造函数，用于创建 REST API。

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
```

```
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
 constructor is in Alpha, o we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
aws_cdk.aws_lambda_python_alpha/README.html
            # Check all Powertools layers versions here: https://
docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
{self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_11,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. 部署您的应用程序。

```
cd ..
cdk deploy
```

9. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. 调用 API 端点：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

11. 要获取该函数的跟踪信息，请运行 sam traces。

```
sam traces
```

该跟踪输出类似于以下示例：

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
```

```
    - fault count(5XX): 0
    - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

12. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
cdk destroy
```

使用 ADOT 分析您的 Python 函数

ADOT 提供完全托管式 Lambda 层，这些层使用 OTel SDK，将收集遥测数据所需的一切内容打包起来。通过使用此层，您可以在不必修改任何函数代码的情况下，对您的 Lambda 函数进行分析。您还可以将您的层配置为对 OTel 进行自定义初始化。有关更多信息，请参阅 ADOT 文档中的适用于 Lambda 上的 ADOT 收集器的自定义配置。

对于 Python 运行时，可以添加 AWS适用于 ADOT Python 的托管 Lambda 层以自动分析您的函数。此层适用于 arm64 和 x86_64 架构。有关如何添加此层的详细说明，请参阅 ADOT 文档中的适用于 OpenTelemetry 的 AWS 发行版对于 Python 的 Lambda 支持。

使用 X-Ray SDK 分析您的 Python 函数

要记录有关您的 Lambda 函数对应用程序中的其他资源进行调用的详细信息，您还可以使用 AWS X-Ray SDK for Python。要获取开发工具包，请将 `aws-xray-sdk` 包添加到应用程序的依赖项中。

Example requirements.txt

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

在函数代码中，可以通过使用 `aws_xray_sdk.core` 模块修补 `boto3` 库来分析 AWS SDK 客户端。

Example 函数 – 跟踪 AWS SDK 客户端

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

logger = logging.getLogger()
logger.setLevel(logging.INFO)
patch_all()

client = boto3.client('lambda')
client.get_account_settings()

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES\r' + jsonpickle.encode(dict(**os.environ)))
  ...
```

在添加正确的依赖项并进行必要的代码更改后，请通过 Lambda 控制台或 API 激活函数配置中的跟踪。

使用 Lambda 控制台激活跟踪

要使用控制台切换 Lambda 函数的活动跟踪，请按照以下步骤操作：

打开活跃跟踪

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration（配置），然后选择 Monitoring and operations tools（监控和操作工具）。
4. 选择编辑。
5. 在 X-Ray 下方，开启 Active tracing（活动跟踪）。
6. 选择保存。

使用 Lambda API 激活跟踪

借助 AWS CLI 或 AWS SDK 在 Lambda 函数上配置跟踪，请使用以下 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration

- CreateFunction

以下示例 AWS CLI 命令对名为 my-function 的函数启用活跃跟踪。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

跟踪模式是发布函数版本时版本特定配置的一部分。您无法更改已发布版本上的跟踪模式。

使用 AWS CloudFormation 激活跟踪

要对 AWS CloudFormation 模板中的 `AWS::Lambda::Function` 资源激活跟踪，请使用 `TracingConfig` 属性。

Example function-inline.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

对于 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 资源，请使用 `Tracing` 属性。

Example template.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

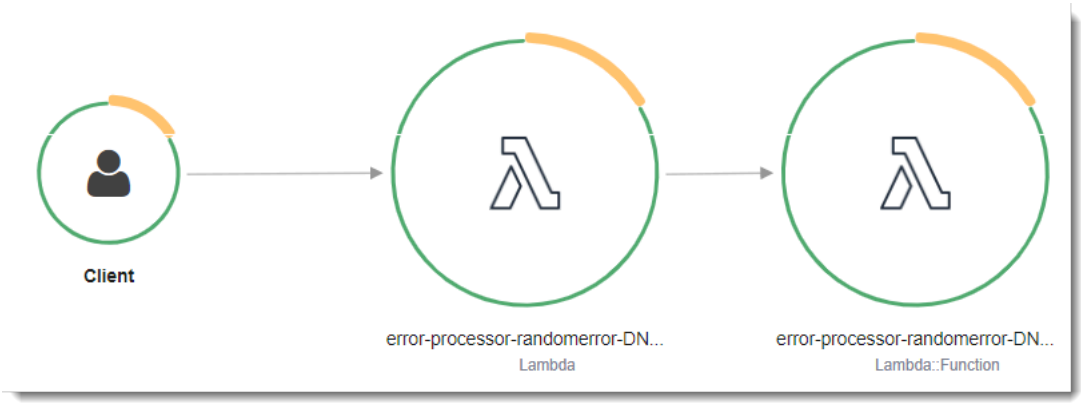
解释 X-Ray 跟踪

您的函数需要权限才能将跟踪数据上载到 X-Ray。在 Lambda 控制台中激活跟踪后，Lambda 会将所需权限添加到函数的执行角色。如果没有，请将 AWSXRayDaemonWriteAccess 策略添加到执行角色。

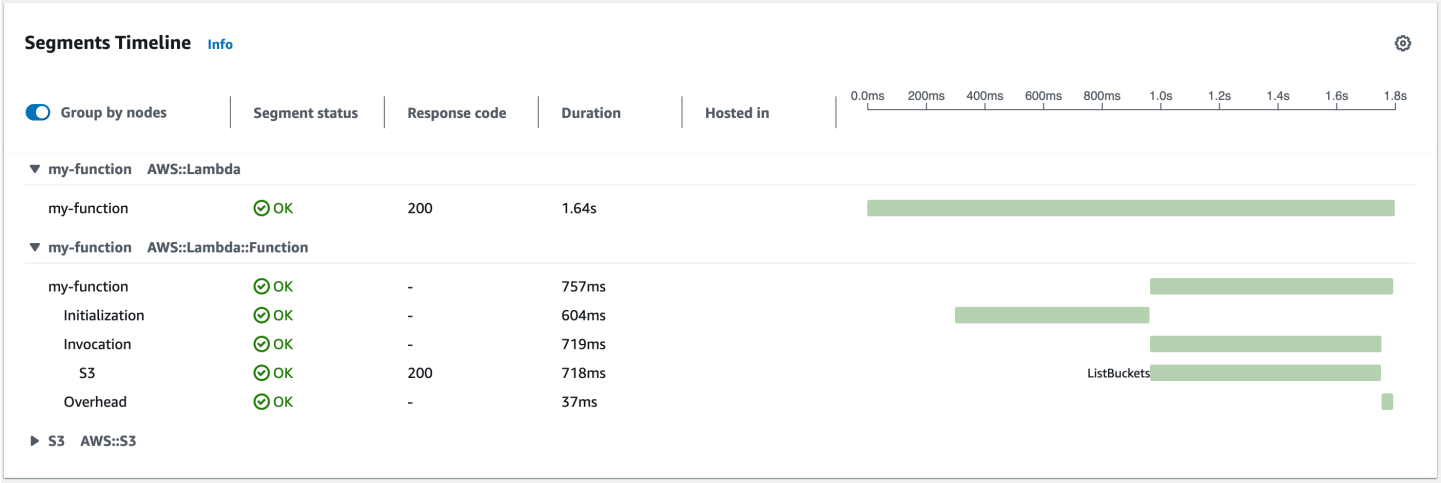
在配置活跃跟踪后，您可以通过应用程序观察特定请求。X-Ray 服务图将显示有关应用程序及其所有组件的信息。以下示例显示了具有两个函数的应用程序。主函数处理事件，有时会返回错误。位于顶部的第二个函数将处理第一个函数的日志组中显示的错误，并使用 AWS SDK 调用 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

X-Ray 无法跟踪对应用程序的所有请求。X-Ray 将应用采样算法确保跟踪有效，同时仍会提供所有请求的一个代表性样本。采样率是每秒 1 个请求和 5% 的其他请求。您无法为函数配置此 X-Ray 采样率。

在 X-Ray 中，跟踪记录有关由一个或多个服务处理的请求的信息。Lambda 会每个跟踪记录 2 个分段，这些分段将在服务图上创建两个节点。下图突出显示了这两个节点：



位于左侧的第一个节点表示接收调用请求的 Lambda 服务。第二个节点表示特定的 Lambda 函数。以下示例显示了一个包含这 2 个分段的跟踪。两者都命名为 my-function，但其中一个函数具有 `AWS::Lambda` 源，另一个则具有 `AWS::Lambda::Function` 源。如果 `AWS::Lambda` 分段显示错误，则表示 Lambda 服务存在问题。如果 `AWS::Lambda::Function` 分段显示错误，则说明函数存在问题。



此示例将展开 `AWS::Lambda::Function` 分段，以显示其三个子分段。

> **Note**
>
> AWS 目前正在实施对 Lambda 服务的更改。由于这些更改，您可能会看到 AWS 账户 中不同 Lambda 函数发出的系统日志消息和跟踪分段的结构和内容之间存在细微差异。
> 此处显示的示例跟踪说明了旧样式函数分段。以下段落介绍了新旧样式分段之间的差异。
> 这些更改将在未来几周内实施，除中国和 GovCloud 区域外，所有 AWS 区域 的函数都将过渡到使用新格式的日志消息和跟踪分段。

旧样式函数分段包含以下子分段：

- 初始化 – 表示加载函数和运行初始化代码所花费的时间。此子分段仅对由您的函数的每个实例处理的第一个事件显示。
- 调用 – 表示执行处理程序代码花费的时间。
- 开销 – 表示 Lambda 运行时为准备处理下一个事件而花费的时间。

新样式函数分段不包含 `Invocation` 子分段。而是将客户子分段直接附加到函数分段。有关新旧样式函数分段结构的更多信息，请参阅 the section called “了解 X-Ray 跟踪”。

您还可以分析 HTTP 客户端、记录 SQL 查询以及使用注释和元数据创建自定义子段。有关更多信息，请参阅 AWS X-Ray 开发人员指南中的 AWS X-Ray SDK for Python。

> **定价**
>
> 作为 AWS 免费套餐的组成部分，您可以每月免费使用 X-Ray 跟踪，但不能超过一定限制。超出该阈值后，X-Ray 会对跟踪存储和检索进行收费。有关更多信息，请参阅 AWS X-Ray 定价。

在层中存储运行时依赖项 (X-Ray SDK)

如果您使用 X-Ray 开发工具包来分析AWS开发工具包客户端和您的函数代码，则您的部署程序包可能会变得相当大。为了避免每次更新函数代码时上载运行时依赖项，请将 X-Ray SDK 打包到 Lambda 层中。

以下示例显示存储 AWS X-Ray SDK for Python 的 `AWS::Serverless::LayerVersion` 资源。

Example template.yml – 依赖项层

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-python-lib
      Description: Dependencies for the blank-python sample app.
      ContentUri: package/.
      CompatibleRuntimes:
        - python3.11
```

使用此配置，仅在更改运行时依赖项时您才会更新库层。由于函数部署软件包仅包含您的代码，因此可以帮助缩短上传时间。

为依赖项创建层需要更改构建才能在部署之前生成层存档。有关工作示例，请参阅 blank-python 示例应用程序。

使用 Ruby 构建 Lambda 函数

您可以在 AWS Lambda 中运行 Ruby 代码。Lambda 可为 Ruby 提供运行时，用于运行代码来处理事件。您的代码在包含 AWS SDK for Ruby 的环境中运行，其中包含来自您管理的 AWS Identity and Access Management (IAM) 角色的凭证。要了解有关 Ruby 运行时随附的 SDK 版本的更多信息，请参阅 the section called “包含运行时的 SDK 版本”。

Lambda 支持以下 Ruby 运行时：

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
Ruby 3.3	`ruby3.3`	Amazon Linux 2023	未计划	未计划	未计划
Ruby 3.2	`ruby3.2`	Amazon Linux 2	未计划	未计划	未计划

创建 Ruby 函数

1. 打开 Lambda 控制台。
2. 选择 Create function（创建函数）。
3. 配置以下设置：
   - 函数名称：输入函数名称。
   - 运行时系统：选择 Ruby 3.2。
4. 选择 Create function（创建函数）。
5. 要配置测试事件，请选择测试。
6. 对于事件名称，输入 **`test`**。
7. 选择 Save changes（保存更改）。
8. 要调用该函数，请选择 Test（测试）。

控制台将使用名为 `lambda_function.rb` 的源文件创建一个 Lambda 函数。您可以在内置代码编辑器中编辑此文件并添加更多文件。要保存您的更改，请选择 Save（保存）。然后，要运行代码，请选择 Test（测试）。

`lambda_function.rb` 文件会导出一个名为 `lambda_handler` 的函数，此函数将接受事件对象和上下文对象。这是 Lambda 在调用函数时调用的处理函数。Ruby 函数运行时从 Lambda 获取调用事件并将其传递到处理程序。在函数配置中，处理程序值为 `lambda_function.lambda_handler`。

保存函数代码时，Lambda 控制台会创建一个 .zip 文件归档部署包。在控制台外部开发函数代码时（使用 IDE），您需要创建部署程序包将代码上载到 Lambda 函数。

除了调用事件之外，函数运行时还将上下文对象传递给处理程序。上下文对象包含有关调用、函数和执行环境的其他信息。环境变量中提供了更多信息。

您的 Lambda 函数附带了 CloudWatch Logs 日志组。函数运行时会将每次调用的详细信息发送到 CloudWatch Logs。该运行时会中继调用期间函数输出的任何日志。如果您的函数返回错误，则 Lambda 将为错误设置格式，并将其返回给调用方。

主题

包含运行时的 SDK 版本

Ruby 运行时中包含的 AWS SDK 版本取决于运行时版本和您的 AWS 区域。适用于 Ruby 的 AWS SDK 采用模块化设计，并按照 AWS 服务各自独立。要查找您正在使用的运行时中包含的特定服务 Gem 的版本号，请使用以下格式的代码创建 Lambda 函数。用您的代码使用的服务 Gem 的名称替换 `aws-sdk-s3` 和 `Aws::S3`。

```
require 'aws-sdk-s3'

def lambda_handler(event:, context:)
  puts "Service gem version: #{Aws::S3::GEM_VERSION}"
  puts "Core version: #{Aws::CORE_GEM_VERSION}"
```

```
end
```

再启用一个 Ruby JIT（YJIT）

Ruby 3.2 运行时系统支持 YJIT，这是一款轻量级的极简 Ruby JIT 编译器。YJIT 提供的性能有显著提升，但消耗的内存也比 Ruby 解释器更多。建议将 YJIT 用于 Ruby on Rails 工作负载。

默认情况下，将不会启用 YJIT。要为 Ruby 3.2 函数启用 YJIT，请将 `RUBY_YJIT_ENABLE` 环境变量设置为 `1`。要确认已启用 YJIT，请打印 `RubyVM::YJIT.enabled?` 方法的结果。

Example ：确认已启用 YJIT

```
puts(RubyVM::YJIT.enabled?())
# => true
```

定义采用 Ruby 的 Lambda 函数处理程序

Lambda 函数处理程序是函数代码中处理事件的方法。当调用函数时，Lambda 运行处理程序方法。您的函数会一直运行，直到处理程序返回响应、退出或超时。

主题

Ruby 处理程序基础知识

在以下示例中，文件 `function.rb` 将定义一个名为 `handler` 的处理程序方法。该处理程序函数将选取两个对象作为输入并返回一个 JSON 文档。

Example function.rb

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

在您的函数配置中，`handler` 设置指示 Lambda 在何处查找处理程序。对于上述示例，此设置的正确值为 **`function.handler`**。它包含两个由点分隔的名称：文件的名和处理程序方法的名称。

您还可以在类中定义处理程序方法。以下示例在名为 `process` 的模块中的名为 `Handler` 的类上定义了一个名为 `LambdaFunctions` 的处理程序。

Example source.rb

```
module LambdaFunctions
  class Handler
    def self.process(event:,context:)
      "Hello!"
    end
  end
end
```

在本例中，处理程序设置为 **`source.LambdaFunctions::Handler.process`**。

该处理程序接受的两个对象是调用事件和上下文。事件是一个 Ruby 对象，包含由调用方提供的负载。如果该负载为 JSON 文档，则事件对象为 Ruby 哈希。否则，它是一个字符串。上下文对象具有一些方法和属性，它们提供了有关调用、函数和执行环境的信息。

函数处理程序在 Lambda 函数每次被调用时执行。处理程序外部的静态代码对该函数的每个实例执行一次。如果您的处理程序使用开发工具包客户端和数据库连接之类的资源，您可以在处理程序方法外部创建这些资源以对多个调用重复使用它们。

您的函数的每个实例都可以处理多个调用事件，但一次只能处理一个事件。在给定的任何时间，处理事件的实例数都是您的函数的并发。有关 Lambda 执行环境的更多信息，请参阅 了解 Lambda 执行环境生命周期。

Ruby Lambda 函数的代码最佳实践

在构建 Lambda 函数时，请遵循以下列表中的指南，采用最佳编码实践：

- 从核心逻辑中分离 Lambda 处理程序。这样您可以创建更容易进行单元测试的函数。例如，在 Ruby 中，如下所示：

```
def lambda_handler(event:, context:)
    foo = event['foo']
    bar = event['bar']

    result = my_lambda_function(foo:, bar:)

end

def my_lambda_function(foo:, bar:)
    // MyLambdaFunction logic here

end
```

- 控制函数部署程序包中的依赖关系。AWS Lambda 执行环境包含许多库。对于 Ruby 运行时，其中包括 AWS SDK。Lambda 会定期更新这些库，以支持最新的功能组合和安全更新。这些更新可能会使 Lambda 函数的行为发生细微变化。要完全控制您的函数所用的依赖项，请使用部署程序包来打包所有依赖项。
- 将依赖关系的复杂性降至最低。首选在执行环境启动时可以快速加载的更简单的框架。
- 将部署程序包大小精简为只包含运行时必要的部分。这样会减少调用前下载和解压缩部署程序包所需的时间。对于用 Ruby 编写的函数，请勿将整个 AWS SDK 库作为部署包的一部分上传，而是有选择地依赖于获取您所需的 SDK 组件的 gem（例如 DynamoDB 或 Amazon S3 SDK gem）。

- 利用执行环境重用来提高函数性能。连接软件开发工具包 (SDK) 客户端和函数处理程序之外的数据库，并在 `/tmp` 目录中本地缓存静态资产。由函数的同一实例处理的后续调用可重用这些资源。这样就可以通过缩短函数运行时间来节省成本。

  为了避免调用之间潜在的数据泄露，请不要使用执行环境来存储用户数据、事件或其他具有安全影响的信息。如果您的函数依赖于无法存储在处理程序的内存中的可变状态，请考虑为每个用户创建单独的函数或单独的函数版本。
- 使用 keep-alive 指令来维护持久连接。Lambda 会随着时间的推移清除空闲连接。在调用函数时尝试重用空闲连接会导致连接错误。要维护您的持久连接，请使用与运行时关联的 keep-alive 指令。有关示例，请参阅在 Node.js 中通过 Keep-Alive 重用连接。
- 使用环境变量将操作参数传递给函数。例如，您在写入 Amazon S3 存储桶时，不应对要写入的存储桶名称进行硬编码，而应将存储桶名称配置为环境变量。
- 避免在 Lambda 函数中使用递归调用，在这种情况下，函数会调用自己或启动可能再次调用该函数的进程。这可能会导致意想不到的函数调用量和升级成本。如果您看到意外的调用量，请立即将函数保留并发设置为 `0` 来限制对函数的所有调用，同时更新代码。
- Lambda 函数代码中不要使用非正式的非公有 API。对于 AWS Lambda 托管式运行时，Lambda 会定期为 Lambda 的内部 API 应用安全性和功能更新。这些内部 API 更新可能不能向后兼容，会导致意外后果，例如，假设您的函数依赖于这些非公有 API，则调用会失败。请参阅 API 参考以查看公开发布的 API 列表。
- 编写幂等代码。为您的函数编写幂等代码可确保以相同的方式处理重复事件。您的代码应该正确验证事件并优雅地处理重复事件。有关更多信息，请参阅如何使我的 Lambda 函数具有幂等性？。

使用 .zip 文件归档部署 Ruby Lambda 函数

AWS Lambda 函数的代码包含一个 .rb 文件，其中包含函数的处理程序代码，以及代码所依赖的任何其他依赖项（Gem）。要将此函数部署到 Lambda，您可以使用部署包。此包可以是 .zip 文件归档或容器映像。有关在 Ruby 中使用容器映像的更多信息，请参阅使用容器映像部署 Ruby Lambda 函数。

要创建 .zip 文件归档格式的部署包，可以使用命令行工具内置的 .zip 文件归档实用工具或任何其他 .zip 文件实用工具（例如 7zip）。以下各部分中显示的示例假设您在 Linux 或 macOS 环境中使用命令行 `zip` 工具。要在 Windows 中使用相同命令，您可以安装 Windows Subsystem for Linux，以获取 Windows 集成版本的 Ubuntu 和 Bash。

请注意，Lambda 使用 POSIX 文件权限，因此在创建 .zip 文件归档之前，您可能需要为部署包文件夹设置权限。

以下部分中的示例命令使用 Bundler 实用程序为部署包添加依赖项。要安装 Bundler，请运行以下命令：

```
gem install bundler
```

Sections

Ruby 中的依赖项

对于使用 Ruby 运行时系统的 Lambda 函数，依赖项可以是任何 Ruby Gem。使用 .zip 存档部署函数时，可以使用函数代码或使用 Lambda 层将这些依赖项添加到 .zip 文件中。层是可以包含其他代码或其他内容的单独的 .zip 文件。要了解有关使用 Lambda 层的更多信息，请参阅 Lambda 层。

Ruby 运行时包括 AWS SDK for Ruby。如果您的函数使用开发工具包，则无需将其与代码捆绑。不过，如果您希望保持对依赖项的完全控制权或使用特定版本的开发工具包，则可将其添加到函数的部署包中。您可以将开发工具包包含在.zip 文件中，也可以使用 Lambda 层进行添加。.zip 文件或 Lambda

层中的依赖项优先于运行时系统中包含的版本。要查找运行时系统版本中包含哪个版本的 SDK for Ruby，请参阅 the section called “包含运行时的 SDK 版本”。

在 AWS 责任共担模式下，您负责管理函数部署包中的所有依赖项。这包括应用更新和安全补丁。要更新函数部署包中的依赖项，请先创建一个新的 .zip 文件，然后将其上传到 Lambda 中。有关更多信息，请参阅 创建含依赖项的 .zip 部署包 和 使用 .zip 文件创建和更新 Ruby Lambda 函数。

创建不含依赖项的 .zip 部署包

如果您的函数代码没有依赖项，则 .zip 文件仅包含带有函数处理程序代码的 .rb 文件。使用首选 ZIP 实用工具创建一个 .zip 文件，并将 .rb 文件置于根目录中。如果 .rb 文件不在 .zip 文件的根目录下，Lambda 将无法运行代码。

要了解如何部署 .zip 文件以创建新的 Lambda 函数或更新现有函数，请参阅 使用 .zip 文件创建和更新 Ruby Lambda 函数。

创建含依赖项的 .zip 部署包

如果函数代码依赖其他 Ruby Gem，您可以使用函数代码将这些依赖项添加到 .zip 文件中，也可以使用 Lambda 层进行添加。本部分中的说明旨在向您展示如何将依赖项包含在 .zip 部署包中。有关如何将依赖项包含在层中的说明，请参阅 the section called “为依赖项创建 Ruby 层”。

假设函数代码保存在项目目录下名为 `lambda_function.rb` 的文件中。以下示例 CLI 命令将创建名为 `my_deployment_package.zip` 的 .zip 文件，其中包含函数代码及其依赖项。

创建部署包

1. 在项目目录中，创建 `Gemfile` 来指定其中的依赖项。

   ```
   bundle init
   ```

2. 使用首选文本编辑器对 `Gemfile` 进行编辑，从而指定函数的依赖项。例如，要使用 TZInfo Gem，请按照如下所示编辑 `Gemfile`。

   ```
   source "https://rubygems.org"
   gem "tzinfo"
   ```

3. 运行以下命令，将 `Gemfile` 中指定的 Gem 安装到项目目录中。此命令将 `vendor/bundle` 设置为 Gem 的默认安装路径。

   ```
   bundle config set --local path 'vendor/bundle' && bundle install
   ```

您应该可以看到类似于如下所示的输出内容。

```
Fetching gem metadata from https://rubygems.org/..........
Resolving dependencies...
Using bundler 2.4.13
Fetching tzinfo 2.0.6
Installing tzinfo 2.0.6
...
```

> **Note**
>
> 稍后想再次全局安装 Gem，请运行以下命令：
>
> ```
> bundle config set --local system 'true'
> ```

4. 创建 .zip 文件存档，其中包含带有函数处理程序代码以及您在上一步中安装的依赖项的 `lambda_function.rb` 文件。

```
zip -r my_deployment_package.zip lambda_function.rb vendor
```

您应该可以看到类似于如下所示的输出内容。

```
adding: lambda_function.rb (deflated 37%)
  adding: vendor/ (stored 0%)
  adding: vendor/bundle/ (stored 0%)
  adding: vendor/bundle/ruby/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/build_info/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/aws-eventstream-1.0.1.gem (deflated 36%)
...
```

为依赖项创建 Ruby 层

本部分中的说明旨在向您展示如何将依赖项包含在层中。有关如何将依赖项包含在部署包中的说明，请参阅 the section called “创建含依赖项的 .zip 部署包”。

当您向函数添加层时，Lambda 会将层内容加载到该执行环境的 /opt 目录中。对于每个 Lambda 运行时系统，PATH 变量都包括 /opt 目录中的特定文件夹路径。为确保 PATH 变量能够获取层内容，层 .zip 文件应在以下文件夹路径中具有依赖项：

- ruby/gems/2.7.0 (GEM_PATH)
- ruby/lib (RUBYLIB)

例如，层.zip 文件结构可能如下所示：

```
json.zip
# ruby/gems/2.7.0/
                | build_info
                | cache
                | doc
                | extensions
                | gems
                | # json-2.1.0
                # specifications
                  # json-2.1.0.gemspec
```

此外，Lambda 会自动检测 /opt/lib 目录中的任何库，以及 /opt/bin 目录中的任何二进制文件。为确保 Lambda 正确获取层内容，还可以创建包含以下结构的层：

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

打包层后，请参阅 the section called “创建和删除层” 和 the section called “添加层” 以完成层设置。

使用原生库创建 .zip 部署包

许多常见的 Ruby Gem（例如 nokogiri、nio4r 和 mysql），都包含用 C 语言编写的本机扩展。将包含 C 代码的库添加到部署包时，必须正确构建程序包，确保其与 Lambda 执行环境兼容。

对于生产应用程序，建议您使用 AWS Serverless Application Model（AWS SAM）来构建和部署代码。在 AWS SAM 中，使用 sam build --use-container 选项在类似 Lambda 的 Docker 容器中

构建函数。要了解有关使用 AWS SAM 来部署函数代码的更多信息，请参阅《AWS SAM 开发人员指南》中的 Building applications。

要在不使用 AWS SAM 的情况下创建包含带本机扩展的 Gem 的.zip 部署包，您也可以使用容器将依赖项捆绑到与 Lambda Ruby 运行时系统环境相同的环境中。要完成这些步骤，就必须在生成计算机上安装 Docker。要了解有关安装 Docker 的更多信息，请参阅 Install Docker Engine。

在 Docker 容器中创建.zip 部署包

1. 在本地生成计算机上创建文件夹来保存容器。在该文件夹中，创建一个名为 `dockerfile` 的文件，再将以下代码粘贴到该文件中。

   ```
   FROM public.ecr.aws/sam/build-ruby3.2:latest-x86_64
   RUN gem update bundler
   CMD "/bin/bash"
   ```

2. 在您创建 `dockerfile` 的文件夹中，运行以下命令来创建 Docker 容器。

   ```
   docker build -t awsruby32 .
   ```

3. 导航到项目目录，其中包含带有函数处理程序代码以及指定函数依赖项的 `Gemfile` 的 `.rb` 文件。在该目录中，运行以下命令启动 Lambda Ruby 容器。

   Linux/MacOS

   ```
   docker run --rm -it -v $PWD:/var/task -w /var/task awsruby32
   ```

   > **Note**
   >
   > 在 MacOS 中，您可能会看到一条警告，告知您所请求映像的平台与检测到的主机平台不匹配。请忽略该警告。

   Windows PowerShell

   ```
   docker run --rm -it -v ${pwd}:var/task -w /var/task awsruby32
   ```

   容器启动时，您应该会看到 bash 提示符。

```
bash-4.2#
```

4. 配置捆绑实用程序，将 Gemfile 中指定的 Gem 安装到本地 vendor/bundle 目录中并安装依赖项。

```
bash-4.2# bundle config set --local path 'vendor/bundle' && bundle install
```

5. 创建包含函数代码及其依赖项的 .zip 部署包。在此示例中，包含函数处理程序代码的文件名为 lambda_function.rb。

```
bash-4.2# zip -r my_deployment_package.zip lambda_function.rb vendor
```

6. 退出容器并返回到本地项目目录。

```
bash-4.2# exit
```

现在，您可以使用.zip 文件部署包来创建或更新 Lambda 函数。请参阅 使用 .zip 文件创建和更新 Ruby Lambda 函数。

使用 .zip 文件创建和更新 Ruby Lambda 函数

创建 .zip 部署包后，您可以用其创建新的 Lambda 函数或更新现有的 Lambda 函数。您可以使用 Lambda 控制台、AWS Command Line Interface 和 Lambda API 部署 .zip 程序包。您也可以使用 AWS Serverless Application Model（AWS SAM）和 AWS CloudFormation 创建和更新 Lambda 函数。

Lambda 的 .zip 部署包的最大大小为 250MB（已解压缩）。请注意，此限制适用于您上传的所有文件（包括任何 Lambda 层）的组合大小。

Lambda 运行时需要权限才能读取部署包中的文件。在 Linux 权限八进制表示法中，Lambda 对于不可执行文件（rw-r--r--）需要 644 个权限，对于目录和可执行文件需要 755 个权限（rwxr-xr-x）。

在 Linux 和 MacOS 中，使用 chmod 命令更改部署包中文件和目录的文件权限。例如，要为可执行文件提供正确的权限，请运行以下命令。

```
chmod 755 <filepath>
```

要在 Windows 中更改文件权限，请参阅 Microsoft Windows 文档中的 Set, View, Change, or Remove Permissions on an Object。

使用控制台通过 .zip 文件创建和更新函数

要创建新函数，必须先在控制台中创建该函数，然后上传您的 .zip 归档。要更新现有函数，请打开函数页面，然后按照相同的步骤添加更新的 .zip 文件。

如果您的 .zip 文件小于 50MB，则可以通过直接从本地计算机上传该文件来创建或更新函数。对于大于 50MB 的 .zip 文件，必须首先将您的程序包上传到 Amazon S3 存储桶。有关如何使用 AWS Management Console 将文件上传到 Amazon S3 存储桶的说明，请参阅 Amazon S3 入门。要使用 AWS CLI 上传文件，请参阅《AWS CLI 用户指南》中的移动对象。

> **Note**
>
> 您无法更改现有函数的部署包类型（.zip 或容器映像）。例如，您无法将容器映像函数转换为使用 .zip 文件归档。您必须创建新函数。

创建新函数（控制台）

1. 打开 Lambda 控制台的“函数”页面，然后选择创建函数。
2. 选择从头开始创作。
3. 在基本信息中，执行以下操作：
   a. 对于函数名称，输入函数的名称。
   b. 对于运行时系统，选择要使用的运行时系统。
   c. （可选）对于架构，选择要用于函数的指令集架构。默认架构为 x86_64。确保您的函数的 .zip 部署包与您选择的指令集架构兼容。
4. （可选）在 Permissions（权限）下，展开 Change default execution role（更改默认执行角色）。您可以创建新的执行角色，也可以使用现有角色。
5. 选择 Create function (创建函数)。Lambda 使用您选择的运行时系统创建基本“Hello world”函数。

从本地计算机上传 .zip 归档（控制台）

1. 在 Lambda 控制台的“函数”页面中，选择要为其上传 .zip 文件的函数。
2. 选择代码选项卡。

3. 在代码源窗格中，选择上传自。
4. 选择 .zip 文件。
5. 要上传 .zip 文件，请执行以下操作：
   a. 选择上传，然后在文件选择器中选择您的 .zip 文件。
   b. 选择打开。
   c. 选择保存。

从 Amazon S3 存储桶上传 .zip 归档（控制台）

1. 在 Lambda 控制台的“函数”页面中，选择要为其上传新 .zip 文件的函数。
2. 选择代码选项卡。
3. 在代码源窗格中，选择上传自。
4. 选择 Amazon S3 位置。
5. 粘贴 .zip 文件的 Amazon S3 链接 URL，然后选择保存。

使用控制台代码编辑器更新 .zip 文件函数

对于某些带有 .zip 部署包的函数，您可以使用 Lambda 控制台的内置代码编辑器直接更新函数代码。要使用此功能，函数必须满足以下条件：

- 函数必须使用一种解释性语言运行时系统（Python、Node.js 或 Ruby）
- 函数的部署包必须小于 50 MB（未压缩状态）。

带有容器映像部署包的函数的代码不能直接在控制台中编辑。

要使用控制台代码编辑器更新函数代码。

1. 打开 Lambda 控制台的“函数”页面，然后选择函数。
2. 选择代码选项卡。
3. 在代码源窗格中，选择源代码文件并在集成的代码编辑器中对其进行编辑。
4. 编辑完代码后，展开主侧栏中的部署部分，然后选择部署。

使用 AWS CLI 通过 .zip 文件创建和更新函数

您可以使用 AWS CLI 创建新函数或使用 .zip 文件更新现有函数。使用 create-function 和 update-function-code 命令部署 .zip 程序包。如果您的 .zip 文件小于 50MB，则可以从本地生成计算机上的文件位置上传 .zip 程序包。对于较大的文件，必须从 Amazon S3 存储桶上传 .zip 程序包。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

> **Note**
>
> 如果您使用 AWS CLI 从 Amazon S3 存储桶上传 .zip 文件，则该存储桶必须与您的函数位于同一个 AWS 区域 中。

要通过 AWS CLI 使用 .zip 文件创建新函数，则必须指定以下内容：

- 函数的名称 (`--function-name`)
- 函数的运行时系统 (`--runtime`)
- 函数的执行角色 (`--role`) 的 Amazon 资源名称（ARN）
- 函数代码 (`--handler`) 中处理程序方法的名称

还必须指定 .zip 文件的位置。如果 .zip 文件位于本地生成计算机上的文件夹中，请使用 `--zip-file` 选项指定文件路径，如以下示例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

要指定 .zip 文件在 Amazon S3 存储桶中的位置，请使用 `--code` 选项，如以下示例命令所示。您只需对版本控制对象使用 `S3ObjectVersion` 参数。

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

要使用 CLI 更新现有函数，请使用 `--function-name` 参数指定函数的名称。您还必须指定要用于更新函数代码的 .zip 文件的位置。如果 .zip 文件位于本地生成计算机上的文件夹中，请使用 `--zip-file` 选项指定文件路径，如以下示例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

要指定 .zip 文件在 Amazon S3 存储桶中的位置，请使用 `--s3-bucket` 和 `--s3-key` 选项，如以下示例命令所示。您只需对版本控制对象使用 `--s3-object-version` 参数。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

使用 Lambda API 通过 .zip 文件创建和更新函数

要使用 .zip 文件归档创建和更新函数，请使用以下 API 操作：

- CreateFunction
- UpdateFunctionCode

使用 AWS SAM 通过 .zip 文件创建和更新函数

AWS Serverless Application Model（AWS SAM）是一个工具包，可帮助简化在 AWS 上构建和运行无服务器应用程序的过程。您可以在 YAML 或 JSON 模板中为应用程序定义资源，并使用 AWS SAM 命令行界面（AWS SAM CLI）构建、打包和部署应用程序。当您通过 AWS SAM 模板构建 Lambda 函数时，AWS SAM 会使用您的函数代码和您指定的任何依赖项自动创建 .zip 部署包或容器映像。要了解有关使用 AWS SAM 构建和部署 Lambda 函数的更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 AWS SAM 入门。

您可以使用 AWS SAM 创建使用现有 .zip 文件归档的 Lambda 函数。要使用 AWS SAM 创建 Lambda 函数，您可以将 .zip 文件保存在 Amazon S3 存储桶或生成计算机上的本地文件夹中。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

在 AWS SAM 模板中，`AWS::Serverless::Function` 资源将指定 Lambda 函数。在此资源中，设置以下属性以创建使用 .zip 文件归档的函数：

- `PackageType` – 设置为 `Zip`
- `CodeUri` – 设置为函数代码的 Amazon S3 URI、本地文件夹的路径或 FunctionCode 对象

- `Runtime` – 设置为您选择的运行时系统

使用 AWS SAM，如果 .zip 文件大于 50MB，则不需要先将其上传到 Amazon S3 存储桶。AWS SAM 可以从本地生成计算机上的某个位置上传最大允许大小为 250MB（已解压缩）的 .zip 程序包。

要了解有关在 AWS SAM 中使用 .zip 文件部署函数的更多信息，请参阅《AWS SAM 开发人员指南》中的 AWS::Serverless::Function。

使用 AWS CloudFormation 通过 .zip 文件创建和更新函数

您可以使用 AWS CloudFormation 创建使用 .zip 文件归档的 Lambda 函数。要从 .zip 文件创建 Lambda 函数，必须先将您的文件上传到 Amazon S3 存储桶。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

在 AWS CloudFormation 模板中，`AWS::Lambda::Function` 资源将指定 Lambda 函数。在此资源中，设置以下属性以创建使用 .zip 文件归档的函数：

- `PackageType` – 设置为 `Zip`
- `Code` – 在 `S3Bucket` 和 `S3Key` 字段中输入 Amazon S3 存储桶名称和 .zip 文件名。
- `Runtime` – 设置为您选择的运行时系统

AWS CloudFormation 生成的 .zip 文件不能超过 4MB。要了解有关在 AWS CloudFormation 中使用 .zip 文件部署函数的更多信息，请参阅《AWS CloudFormation 用户指南》中的 AWS::Lambda::Function。

使用容器镜像部署 Ruby Lambda 函数

有三种方法可以为 Ruby Lambda 函数构建容器映像：

- 使用 Ruby 的 AWS 基本映像

  AWS 基本映像会预加载一个语言运行时系统、一个用于管理 Lambda 和函数代码之间交互的运行时系统接口客户端，以及一个用于本地测试的运行时系统接口仿真器。

- 使用 AWS 仅限操作系统的基础镜像

  AWS 仅限操作系统的运行时系统包含 Amazon Linux 发行版和运行时系统接口模拟器。这些镜像通常用于为编译语言（例如 Go 和 Rust）以及 Lambda 未提供基础映像的语言或语言版本（例如 Node.js 19）创建容器镜像。您也可以使用仅限操作系统的基础映像来实施自定义运行时系统。要使映像与 Ruby 兼容，您必须在映像中包含 Java 的运行时系统接口客户端。

- 使用非 AWS 基本映像

  您还可以使用其他容器注册表的备用基本映像，例如 Alpine Linux 或 Debian。您还可以使用您的组织创建的自定义映像。要使映像与 Ruby 兼容，您必须在映像中包含 Java 的运行时系统接口客户端。

> **Tip**
>
> 要缩短 Lambda 容器函数激活所需的时间，请参阅 Docker 文档中的使用多阶段构建。要构建高效的容器映像，请遵循编写 Dockerfiles 的最佳实践。

此页面介绍了如何为 Lambda 构建、测试和部署容器映像。

主题

Ruby AWS 基本映像

AWS 为 Ruby 提供了以下基本映像：

标签	运行时	操作系统	Dockerfile	淘汰
3.3	Ruby 3.3	Amazon Linux 2023	GitHub 上的适用于 Ruby 3.3 的 Dockerfile	未计划
3.2	Ruby 3.2	Amazon Linux 2	GitHub 上的适用于 Ruby 3.2 的 Dockerfile	未计划

Amazon ECR 存储库：gallery.ecr.aws/lambda/ruby

使用 Ruby 的 AWS 基本映像

先决条件

要完成本节中的步骤，您必须满足以下条件：

- AWS CLI 版本 2
- Docker
- Ruby

从基本映像创建映像

要创建适用于 Ruby 的容器映像

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir example
cd example
```

2. 创建名为 `Gemfile` 的新文件。您可以在此处列出应用程序所需的 RubyGems 软件包。AWS SDK for Ruby 可从 RubyGems 获得。您应该选择特定的 AWS 服务 Gem 进行安装。例如，要使用适用于 Lambda 的 Ruby Gem，Gemfile 应如下所示：

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

或者，aws-sdk Gem 包含所有可用的 AWS 服务 Gem。此 Gem 非常大。我们建议您仅在依赖许多 AWS 服务时使用它。

3. 使用 bundle 安装来安装 Gemfile 中指定的依赖项。

```
bundle install
```

4. 创建名为 lambda_function.rb 的新文件。您可以将以下示例函数代码添加到文件中进行测试，也可以使用您自己的函数代码。

Example Ruby 函数

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. 创建新 Dockerfile。以下示例 Dockerfile 使用 AWS 基本映像。此 Docerfile 使用以下配置：

- 将 FROM 属性设置为基本映像的 URI。
- 使用 COPY 命令将函数代码和运行时系统依赖项复制到 {LAMBDA_TASK_ROOT}，此为 Lambda 定义的环境变量。
- 将 CMD 参数设置为 Lambda 函数处理程序。

请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 root 用户。

Example Dockerfile

```
FROM public.ecr.aws/lambda/ruby:3.2

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
```

```
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 docker-image 并为其提供 test 标签。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 该命令指定了 --platform linux/amd64 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 --platform linux/arm64 选项。

（可选）在本地测试映像

1. 使用 docker run 命令启动 Docker 映像。在此示例中，docker-image 是映像名称，test 是标签。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令会将映像作为容器运行，并在 localhost:9000/2015-03-31/functions/function/invocations 创建本地端点。

> **Note**
>
> 如果为 ARM64 指令集架构创建 Docker 映像，请务必使用 --platform linux/*arm64* 选项，而不是 --platform linux/*amd64* 选项。

2. 在新的终端窗口中，将事件发布到本地端点。

Linux/macOS

在 Linux 和 macOS 中，运行以下 `curl` 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，运行以下 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 获取容器 ID。

```
docker ps
```

4. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 `111122223333` 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
```

```
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。
   - 将 `<ECRrepositoryUri>` 替换为复制的 `repositoryUri`。确保 URI 末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 `ImageUri`，指定之前的存储库 URI。确保 URI 末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，`--publish` 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

将备用基本映像与运行时系统接口客户端配合使用

如果使用仅限操作系统的基础映像或者备用基础映像，则必须在映像中包括运行时系统接口客户端。运行时系统接口客户端可扩展 将 Lambda 运行时 API 用于自定义运行时，用于管理 Lambda 和函数代码之间的交互。

使用 RubyGems.org 程序包管理器安装适用于 Ruby 的 Lambda 运行时系统接口客户端：

```
gem install aws_lambda_ric
```

您也可以从 GitHub 下载 Ruby 运行时接口客户端。运行时系统接口客户端支持 Ruby 版本 2.5.x 到 2.7.x。

以下示例演示了如何使用非 AWS 基本映像构建适用于 Ruby 的容器映像。示例 Dockerfile 使用官方 Ruby 基本映像。Docker 包含运行时系统接口客户端。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- AWS CLI 版本 2
- Docker
- Ruby

从备用基本映像创建映像

要使用备用基本映像创建适用于 Ruby 的容器映像

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir example
cd example
```

2. 创建名为 `Gemfile` 的新文件。您可以在此处列出应用程序所需的 RubyGems 软件包。AWS SDK for Ruby 可从 RubyGems 获得。您应该选择特定的 AWS 服务 Gem 进行安装。例如，要使用适用于 Lambda 的 Ruby Gem，Gemfile 应如下所示：

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

或者，aws-sdk Gem 包含所有可用的 AWS 服务 Gem。此 Gem 非常大。我们建议您仅在依赖许多 AWS 服务时使用它。

3. 使用 bundle 安装来安装 Gemfile 中指定的依赖项。

```
bundle install
```

4. 创建名为 `lambda_function.rb` 的新文件。您可以将以下示例函数代码添加到文件中进行测试，也可以使用您自己的函数代码。

Example Ruby 函数

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. 创建新 Dockerfile。以下 Dockerfile 使用 Ruby 基本映像而不是 AWS 基本映像。Dockerfile 包含适用于 Ruby 的运行时系统接口客户端，该客户端可使映像与 Lambda 兼容。或者，您可以将运行时系统接口客户端添加到应用程序的 Gemfile 中。

   - 将 FROM 属性设置为 Ruby 基本映像。
   - 为函数代码创建目录和指向该目录的环境变量。在本示例中，目录为 /var/task，会镜像 Lambda 执行环境。不过，您可以为函数代码选择任何目录，因为 Dockerfile 不使用 AWS 基础映像。
   - 将 ENTRYPOINT 设置为您希望 Docker 容器在启动时运行的模块。在本例中，模块为运行时系统接口客户端。
   - 将 CMD 参数设置为 Lambda 函数处理程序。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 root 用户。

   Example Dockerfile

```
FROM ruby:2.7

# Install the runtime interface client for Ruby
RUN gem install aws_lambda_ric

# Add the runtime interface client to the PATH
ENV PATH="/usr/local/bundle/bin:${PATH}"

# Create a directory for the Lambda function
ENV LAMBDA_TASK_ROOT=/var/task
RUN mkdir -p ${LAMBDA_TASK_ROOT}
```

```
WORKDIR ${LAMBDA_TASK_ROOT}

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "aws_lambda_ric" ]

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 `docker-image` 并为其提供 `test` 标签。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 该命令指定了 `--platform linux/amd64` 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 `--platform linux/arm64` 选项。

（可选）在本地测试映像

使用运行时系统接口仿真器在本地测试映像。您可以将仿真器构建到映像中，也可以使用以下程序将其安装在本地计算机上。

在本地计算机上安装并运行运行时系统接口仿真器

1. 从项目目录中，运行以下命令以从 GitHub 下载运行时系统接口仿真器（x86-64 架构）并将其安装在本地计算机上。

Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && \
    curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-
runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
    chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

要安装 arm64 仿真器，请将上一条命令中的 GitHub 存储库 URL 替换为以下内容：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

要安装 arm64 模拟器，请将 `$downloadLink` 替换为以下内容：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. 使用 docker run 命令启动 Docker 映像。请注意以下几点：

   - `docker-image` 是映像名称，`test` 是标签。
   - `aws_lambda_ric lambda_function.LambdaFunction::Handler.process` 是 `ENTRYPOINT`，后跟您 Dockerfile 中的 `CMD`。

   Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
```

```
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        aws_lambda_ric lambda_function.LambdaFunction::Handler.process
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    aws_lambda_ric lambda_function.LambdaFunction::Handler.process
```

此命令会将映像作为容器运行，并在 `localhost:9000/2015-03-31/functions/function/invocations` 创建本地端点。

> **Note**
>
> 如果为 ARM64 指令集架构创建 Docker 映像，请务必使用 `--platform linux/arm64` 选项，而不是 `--platform linux/amd64` 选项。

3. 将事件发布到本地端点。

Linux/macOS

在 Linux 和 macOS 中，运行以下 `curl` 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，运行以下 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 获取容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 111122223333 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。
   - 将 `<ECRrepositoryUri>` 替换为复制的 `repositoryUri`。确保 URI 末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 `ImageUri`，指定之前的存储库 URI。确保 URI 末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，`--publish` 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

使用 Ruby Lambda 函数的层

Lambda 层是包含补充代码或数据的 .zip 文件存档。层通常包含库依赖项、自定义运行时系统或配置文件。创建层涉及三个常见步骤：

1. 打包层内容。此步骤需要创建 .zip 文件存档，其中包含要在函数中使用的依赖项。
2. 在 Lambda 中创建层。
3. 将层添加到函数。

本主题包含有关如何正确打包并创建具有外部库依赖项的 Ruby Lambda 层的步骤和指南。

主题

先决条件

要完成本部分中的步骤，您必须满足以下条件：

- Ruby 3.3 与 gem 包安装程序一起分发。
- AWS CLI 版本 2

在整个主题中，我们会引用 awsdocs GitHub 存储库中的 `layer-ruby` 示例应用程序。该应用程序包含用于下载依赖项并生成层的脚本。该应用程序还包含相应的函数，函数使用来自层的依赖项。创建层后，即可部署并调用相应的函数来验证一切是否正常运行。由于使用 Ruby 3.3 运行时来运行这些函数，因此这些层还必须与 Ruby 3.3 兼容。

在 `layer-ruby` 示例应用程序中，您应将 `tzinfo` 库打包到 Lambda 层中。`layer/` 目录包含用于生成层的脚本。`function/` 目录包含示例函数，用于帮助测试该层是否正常工作。本教程的大部分内容将演示如何创建并打包该层。

Ruby 层与 Lambda 运行时环境的兼容性

在 Ruby 层中打包代码时，需要指定与该代码兼容的 Lambda 运行时环境。要评测代码与运行时的兼容性，请考虑代码是为哪些版本的 Ruby、操作系统和指令集架构设计的。

Lambda Ruby 运行时指定其 Ruby 版本和操作系统。在本文档中，您将使用基于 AL2023 的 Ruby 3.3 运行时。有关运行时版本的更多信息，请参阅the section called “支持的运行时”。在创建 Lambda 函数时，您可以指定指令集架构。在本文档中，您将使用 x86_64 架构。有关 Lambda 中的架构的更多信息，请参阅the section called “指令集（ARM/x86）”。

如果您使用软件包中提供的代码，每个软件包维护者都会独立定义其兼容性。大多数 Gem 完全使用 Ruby 编写，并且与任何使用兼容 Ruby 语言版本的运行时兼容。

有时，Gem 会使用名为扩展的 Ruby 功能来编译代码或在安装过程中包含预编译的代码。如果您依赖带有本机扩展的 Gem，则必须评估操作系统和指令集架构的兼容性。要评估 Gem 和 Ruby 运行时之间的兼容性，需要检查 Gem 及其文档。您可以通过检查 Gem 规范中是否定义了 `extensions`，来确定 Gem 是否使用扩展。Ruby 通过 `RUBY_PLATFORM` 全局常量识别其运行的平台。Lambda Ruby 运行时会将平台定义为 `aarch64-linux`（在 `arm64` 架构上运行时），或定义为 `x86_64-linux`（在 `x86_64` 架构上运行时）。无法保证可以检查 Gem 是否与这些平台兼容，但是部分 Gem 通过 `platform` Gem 规范属性声明了其支持的平台。

Ruby 运行时的层路径

当您向函数添加层时，Lambda 会将层内容加载到该执行环境的 `/opt` 目录中。对于每个 Lambda 运行时系统，`PATH` 变量都包括 `/opt` 目录中的特定文件夹路径。为确保 `PATH` 变量能够获取层内容，层 .zip 文件应在以下文件夹路径中具有依赖项：

- `ruby/gems/`*`x`*，其中 x 是运行时上的 Ruby 版本，例如 `3.3.0`。
- `ruby/lib/`

本文档使用 `ruby/gems/`*`x`* 路径。Lambda 要求此目录的内容与 Bundler 安装目录的结构相对应。将 Gem 依赖项与其他元数据子目录一起存储在层路径的 `/gems` 子目录中。例如，您在本教程中创建生成的层.zip 文件具有以下目录结构：

```
layer_content.zip
# ruby
    # gems
        # 3.3.0
```

```
            # gems
                # tzinfo-2.0.6
                # <other_dependencies> (i.e. dependencies of the tzinfo package)
                # ...
            # <metadata generated by bundle>
```

tzinfo 库位于 ruby/gems/3.3.0/ 目录中，位置正确。这可确保 Lambda 在函数调用期间可以找到该库。

打包层内容

在此示例中，您要将 Ruby tzinfo 库打包在一个层 .zip 文件中。完成以下步骤，安装并打包层内容。

安装并打包层内容

1. 克隆 aws-lambda-developer-guide GitHub 存储库，其中包含 sample-apps/layer-ruby 目录中需要的示例代码。

   ```
   git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
   ```

2. 导航到 layer-ruby 示例应用程序的 layer 目录。此目录包含用于正确创建并打包层的脚本。

   ```
   cd aws-lambda-developer-guide/sample-apps/layer-ruby/layer
   ```

3. 检查 Gemfile。此文件定义了要包含在层中的依赖项，即 tzinfo 库。您可以更新此文件，纳入要包含在层中的任何依赖项。

   Example Gemfile

   ```
   source "https://rubygems.org"

   gem "tzinfo"
   ```

4. 确保拥有运行这两个脚本的权限。

   ```
   chmod 744 1-install.sh && chmod 744 2-package.sh
   ```

5. 使用以下命令运行 1-install.sh 脚本：

   ```
   ./1-install.sh
   ```

此脚本将捆绑程序配置为在项目目录中安装依赖项。随后在 vendor/bundle/ 目录中安装所有必需的依赖项。

Example 1-install.sh

```
bundle config set --local path 'vendor/bundle'
bundle install
```

6. 使用以下命令运行 2-package.sh 脚本：

```
./2-package.sh
```

此脚本将内容从 vendor/bundle 目录复制到名为 ruby 的新目录中，随后将 ruby 目录的内容压缩到一个名为 layer_content.zip 的文件中。这便是层的 .zip 文件。您可以解压缩文件，验证是否包含正确的文件结构，如 the section called “Ruby 运行时的层路径”部分所示。

Example 2-package.sh

```
mkdir -p ruby/gems/3.3.0
cp -r vendor/bundle/ruby/3.3.0/* ruby/gems/3.3.0/
zip -r layer_content.zip ruby
```

创建层

在本部分，您会获取在上一部分中生成的 layer_content.zip 文件，将其作为 Lambda 层上传。您可以使用 AWS Management Console 上传层，也可以通过 AWS Command Line Interface（AWS CLI）使用 Lambda API 上传层。上传层 .zip 文件时，在以下 PublishLayerVersion AWS CLI 命令中，将 ruby3.3 指定为兼容的运行时系统，并将 arm64 指定为兼容的架构。

```
aws lambda publish-layer-version --layer-name ruby-requests-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes ruby3.3 \
    --compatible-architectures "arm64"
```

注意响应中的 LayerVersionArn，与 arn:aws:lambda:us-east-1:*123456789012*:layer:ruby-requests-layer:1 类似。在本教程的下一步中，在将层添加到函数时，您要用到此 Amazon 资源名称（ARN）。

将层添加到函数

在本部分，您要部署在函数代码中使用 `tzinfo` 库的示例 Lambda 函数，然后附加该层。要部署该函数，您需要一个 the section called “执行角色（函数访问其他资源的权限）”。如果目前没有执行角色，则按照可折叠部分中的步骤操作。

（可选）创建执行角色

创建执行角色

1. 在 IAM 控制台中，打开 Roles（角色）页面。
2. 选择创建角色。
3. 创建具有以下属性的角色。
   - Trusted entity（可信任的实体）– Lambda。
   - Permissions（权限）– AWSLambdaBasicExecutionRole。
   - Role name（角色名称）– **`lambda-role`**。

   AWSLambdaBasicExecutionRole 策略具有函数将日志写入 CloudWatch Logs 所需的权限。

Lambda 函数代码导入 `tzinfo` 库，然后返回状态码和本地化的日期字符串。

```
require 'json'
require 'tzinfo'

def lambda_handler(event:, context:)
    tz = TZInfo::Timezone.get('America/New_York')
    { statusCode: 200, body: tz.to_local(Time.utc(2018, 2, 1, 12, 30, 0)) }
end
```

部署 Lambda 函数

1. 导航到 `function/` 目录。如果当前在 `layer/` 目录中，请运行以下命令：

   ```
   cd ../function
   ```

2. 使用以下命令创建 .zip 文件部署包：

```
zip my_deployment_package.zip lambda_function.rb
```

3. 部署函数。在以下 AWS CLI 命令中，将 `--role` 参数替换为执行角色 ARN：

```
aws lambda create-function --function-name ruby_function_with_layer \
    --runtime ruby3.3 \
    --architectures "arm64" \
    --handler lambda_function.lambda_handler \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://my_deployment_package.zip
```

4. 接下来，将层附加到函数。在以下 AWS CLI 命令中，将 `--layers` 参数替换为之前记下的层版本 ARN：

```
aws lambda update-function-configuration --function-name ruby_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:ruby-requests-layer:1"
```

5. 最后，尝试使用以下 AWS CLI 命令调用函数：

```
aws lambda invoke --function-name ruby_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

应看到类似如下内容的输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

输出 `response.json` 文件包含有关响应的详细信息。

（可选）清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除 Lambda 层

1. 打开 Lambda 控制台的 Layers page（层页面）。
2. 选择您创建的层。
3. 选择删除，然后再次选择删除。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择 Delete（删除）。

使用 Lambda 上下文对象检索 Ruby 函数信息

当 Lambda 运行您的函数时，它会将上下文对象传递到处理程序。此对象提供的方法和属性包含有关调用、函数和执行环境的信息。

上下文方法

- `get_remaining_time_in_millis` – 返回执行超时前剩余的毫秒数。

上下文属性

- `function_name` – Lambda 函数的名称。
- `function_version` – 函数的版本
- `invoked_function_arn` – 用于调用函数的 Amazon Resource Name (ARN)。表明调用者是否指定了版本号或别名。
- `memory_limit_in_mb` – 为函数分配的内存量。
- `aws_request_id` – 调用请求的标识符。
- `log_group_name` – 函数的日志组。
- `log_stream_name` – 函数实例的日志流。
- `deadline_ms` – 执行超时的日期（Unix 时间格式，以毫秒为单位）。
- `identity` –（移动应用程序）授权请求的 Amazon Cognito 身份的相关信息。
- `client_context` –（移动应用程序）客户端应用程序提供给 Lambda 的客户端上下文。

Ruby Lambda 函数日志记录和监控

AWS Lambda 将代表您自动监控 Lambda 函数并将日志记录发送至 Amazon CloudWatch。您的 Lambda 函数带有一个 CloudWatch Logs 日志组以及函数的每个实例的日志流。Lambda 运行时环境会将每个调用的详细信息发送到日志流，然后中继函数代码的日志和其他输出。有关更多信息，请参阅将 CloudWatch Logs 日志与 Lambda 结合使用。

本页旨在介绍如何从 Lambda 函数的代码生成日志输出，并使用 AWS Command Line Interface、Lambda 控制台或 CloudWatch 控制台访问日志。

Sections

创建返回日志的函数

要从函数代码输出日志，您可以使用 `puts` 语句或使用写入到 `stdout` 或 `stderr` 的任何日志记录库。以下示例记录环境变量和事件对象的值。

Example lambda_function.rb

```
# lambda_function.rb

def handler(event:, context:)
    puts "## ENVIRONMENT VARIABLES"
    puts ENV.to_a
    puts "## EVENT"
    puts event.to_a
end
```

Example 日志格式

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
```

```
environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 16 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

Ruby 运行时记录每次调用的 START、END 和 REPORT 行。报告行提供了以下详细信息：

REPORT 行数据字段

- RequestId – 调用的唯一请求 ID。
- Duration（持续时间）– 函数的处理程序方法处理事件所花费的时间。
- Billed Duration（计费持续时间）– 针对调用计费的时间量。
- Memory Size（内存大小）– 分配给函数的内存量。
- Max Memory Used（最大内存使用量）– 函数使用的内存量。如果调用共享执行环境，Lambda 会报告所有调用使用的最大内存。此行为可能会导致报告值高于预期。
- Init Duration（初始持续时间）– 对于提供的第一个请求，为运行时在处理程序方法外部加载函数和运行代码所花费的时间。
- XRAY TraceId – 对于追踪的请求，为 AWS X-Ray 追踪 ID。
- SegmentId – 对于追踪的请求，为 X-Ray 分段 ID。
- Sampled（采样）– 对于追踪的请求，为采样结果。

如需更详细的日志，请使用 the section called “使用 Ruby 日志记录库”。

在 Lambda 控制台中查看日志

调用 Lambda 函数后，您可以使用 Lambda 控制台查看日志输出。

如果可以在嵌入式代码编辑器中测试代码，则可以在执行结果中找到日志。使用控制台测试功能调用函数时，可以在详细信息部分找到日志输出。

在 CloudWatch 控制台中查看日志

您可以使用 Amazon CloudWatch 控制台查看所有 Lambda 函数调用的日志。

使用 CloudWatch 控制台查看日志

1. 打开 CloudWatch 控制台的 Log groups（日志组页面）。
2. 选择您的函数 (/aws/lambda/***your-function-name***) 的日志组。
3. 创建日志流。

每个日志流对应一个函数实例。日志流会在您更新 Lambda 函数以及创建更多实例来处理多个并发调用时显示。要查找特定调用的日志，建议您使用 AWS X-Ray 检测函数。X-Ray 会在追踪中记录有关请求和日志流的详细信息。

使用 AWS Command Line Interface（AWS CLI）查看日志

AWS CLI 是一种开源工具，让您能够在命令行 Shell 中使用命令与 AWS 服务进行交互。要完成本节中的步骤，您必须拥有 AWS CLI 版本 2。

您可以通过 AWS CLI，使用 `--log-type` 命令选项检索调用的日志。响应包含一个 `LogResult` 字段，其中包含多达 4KB 来自调用的 base64 编码日志。

Example 检索日志 ID

以下示例说明如何从 `LogResult` 字段中检索名为 `my-function` 的函数的日志 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解码日志

在同一命令提示符下，使用 `base64` 实用程序解码日志。以下示例说明如何为 `my-function` 检索 base64 编码的日志。

```
aws lambda invoke --function-name my-function out --log-type Tail \
```

```
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

您应看到以下输出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

`base64` 实用程序在 Linux、macOS 和 Ubuntu on Windows 上可用。macOS 用户可能需要使用 `base64 -D`。

Example get-logs.sh 脚本

在同一命令提示符下，使用以下脚本下载最后五个日志事件。此脚本使用 `sed` 从输出文件中删除引号，并休眠 15 秒以等待日志可用。输出包括来自 Lambda 的响应，以及来自 `get-log-events` 命令的输出。

复制以下代码示例的内容并将其作为 `get-logs.sh` 保存在 Lambda 项目目录中。

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux（仅限）

在同一命令提示符下，macOS 和 Linux 用户可能需要运行以下命令以确保脚本可执行。

```
chmod -R 755 get-logs.sh
```

Example 检索最后五个日志事件

在同一命令提示符下，运行以下脚本以获取最后五个日志事件。

```
./get-logs.sh
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
```

```
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

删除日志

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期之后，日志将自动删除。

使用 Ruby 日志记录库

Ruby 日志记录库可返回易于读取的简化日志。使用日志记录实用程序输出与函数相关的详细信息、消息和错误代码。

```
# lambda_function.rb

require 'logger'

def handler(event:, context:)
  logger = Logger.new($stdout)
  logger.info('## ENVIRONMENT VARIABLES')
  logger.info(ENV.to_a)
  logger.info('## EVENT')
  logger.info(event)
  event.to_a
end
```

`logger` 的输出包括日志级别、时间戳和请求 ID。

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES

[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125
    environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
```

```
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}

END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 3 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

在 AWS Lambda 中检测 Ruby 代码

Lambda 与 AWS X-Ray 集成使您能够跟踪、调试和优化 Lambda 应用程序。在请求遍历应用程序中的资源（从前端 API 到后端的存储和数据库）时，您可以使用 X-Ray 跟踪请求。只需将 X-Ray 开发工具包库添加到构建配置中，就可以记录您的函数对AWS服务进行的任何调用的错误和延迟。

在配置活跃跟踪后，您可以通过应用程序观察特定请求。X-Ray 服务图将显示有关应用程序及其所有组件的信息。以下示例显示了具有两个函数的应用程序。主函数处理事件，有时会返回错误。位于顶部的第二个函数将处理第一个函数的日志组中显示的错误，并使用 AWS SDK 调用 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

要使用控制台切换 Lambda 函数的活动跟踪，请按照以下步骤操作：

打开活跃跟踪

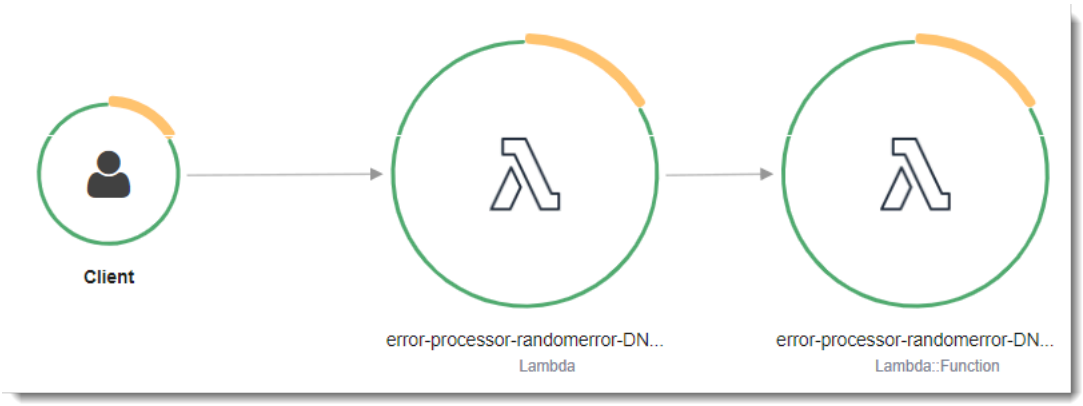
1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration（配置），然后选择 Monitoring and operations tools（监控和操作工具）。
4. 选择编辑。
5. 在 X-Ray 下方，开启 Active tracing（活动跟踪）。
6. 选择 Save（保存）。

> **定价**
>
> 作为 AWS 免费套餐的组成部分，您可以每月免费使用 X-Ray 跟踪，但不能超过一定限制。超出该阈值后，X-Ray 会对跟踪存储和检索进行收费。有关更多信息，请参阅 AWS X-Ray 定价。

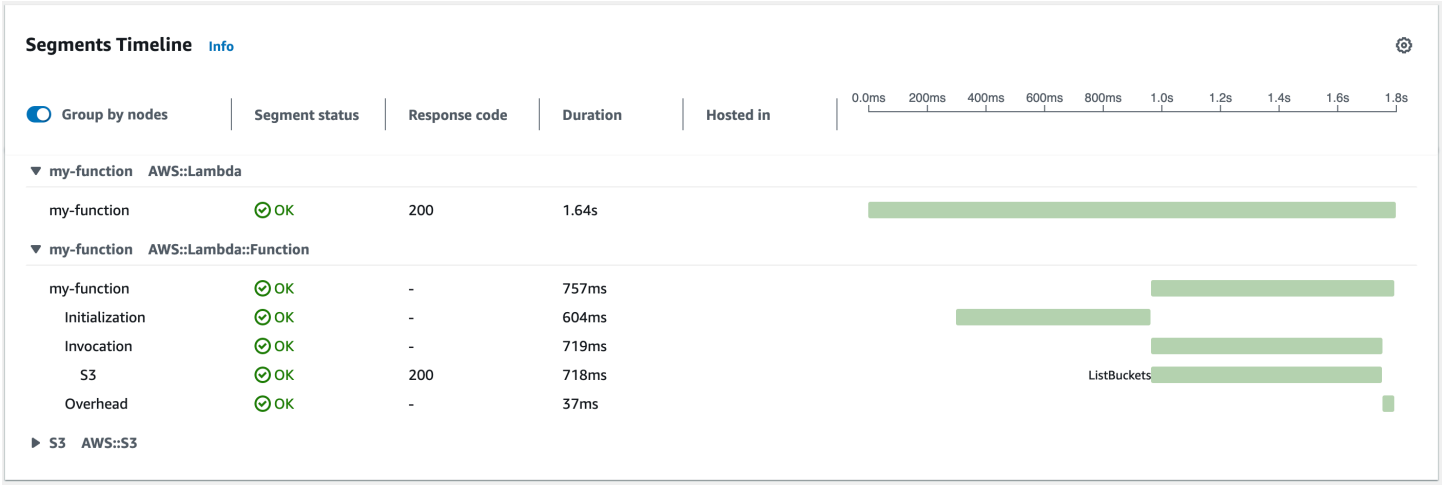
您的函数需要权限才能将跟踪数据上载到 X-Ray。在 Lambda 控制台中激活跟踪后，Lambda 会将所需权限添加到函数的执行角色。如果没有，请将 AWSXRayDaemonWriteAccess 策略添加到执行角色。

X-Ray 无法跟踪对应用程序的所有请求。X-Ray 将应用采样算法确保跟踪有效，同时仍会提供所有请求的一个代表性样本。采样率是每秒 1 个请求和 5% 的其他请求。您无法为函数配置此 X-Ray 采样率。

在 X-Ray 中，跟踪记录有关由一个或多个服务处理的请求的信息。Lambda 会每个跟踪记录 2 个分段，这些分段将在服务图上创建两个节点。下图突出显示了这两个节点：



位于左侧的第一个节点表示接收调用请求的 Lambda 服务。第二个节点表示特定的 Lambda 函数。以下示例显示了一个包含这 2 个分段的跟踪。两者都命名为 my-function，但其中一个函数具有 `AWS::Lambda` 源，另一个则具有 `AWS::Lambda::Function` 源。如果 `AWS::Lambda` 分段显示错误，则表示 Lambda 服务存在问题。如果 `AWS::Lambda::Function` 分段显示错误，则说明函数存在问题。



此示例将展开 `AWS::Lambda::Function` 分段，以显示其三个子分段。

> **Note**
>
> AWS 目前正在实施对 Lambda 服务的更改。由于这些更改，您可能会看到 AWS 账户 中不同 Lambda 函数发出的系统日志消息和跟踪分段的结构和内容之间存在细微差异。
> 此处显示的示例跟踪说明了旧样式函数分段。以下段落介绍了新旧样式分段之间的差异。
> 这些更改将在未来几周内实施，除中国和 GovCloud 区域外，所有 AWS 区域 的函数都将过渡到使用新格式的日志消息和跟踪分段。

旧样式函数分段包含以下子分段：

- 初始化 – 表示加载函数和运行初始化代码所花费的时间。此子分段仅对由您的函数的每个实例处理的第一个事件显示。
- 调用 – 表示执行处理程序代码花费的时间。
- 开销 – 表示 Lambda 运行时为准备处理下一个事件而花费的时间。

新样式函数分段不包含 `Invocation` 子分段。而是将客户子分段直接附加到函数分段。有关新旧样式函数分段结构的更多信息，请参阅 the section called “了解 X-Ray 跟踪”。

您可以使用处理程序代码来记录元数据并跟踪下游调用。要记录有关您的处理程序对其他资源和服务进行调用的详细信息，请使用 X-Ray SDK for Ruby。要获取开发工具包，请将 `aws-xray-sdk` 包添加到应用程序的依赖项中。

Example blank-ruby/function/Gemfile

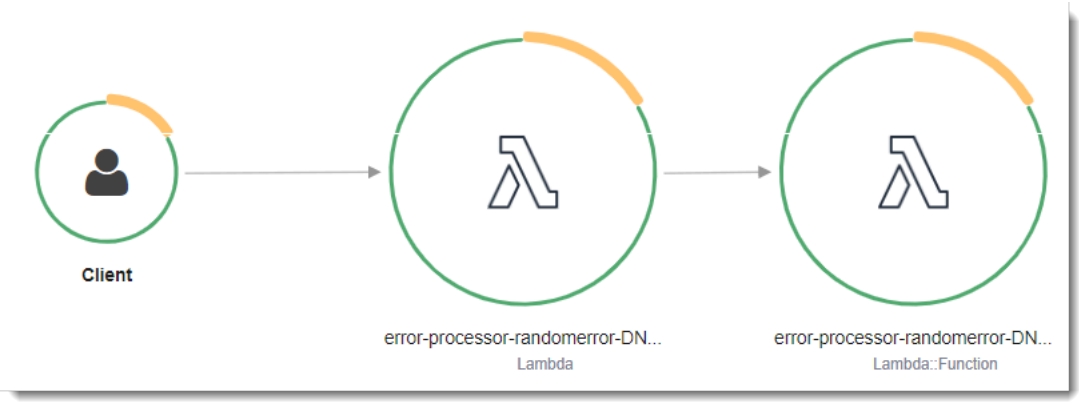
```
# Gemfile
source 'https://rubygems.org'

gem 'aws-xray-sdk', '0.11.4'
gem 'aws-sdk-lambda', '1.39.0'
gem 'test-unit', '3.3.5'
```

要检测 AWS 开发工具包客户端，需要在初始化代码中创建客户端后运行 `aws-xray-sdk/lambda` 模块。

Example blank-ruby/function/lambda_function.rb – 跟踪AWS开发工具包客户端

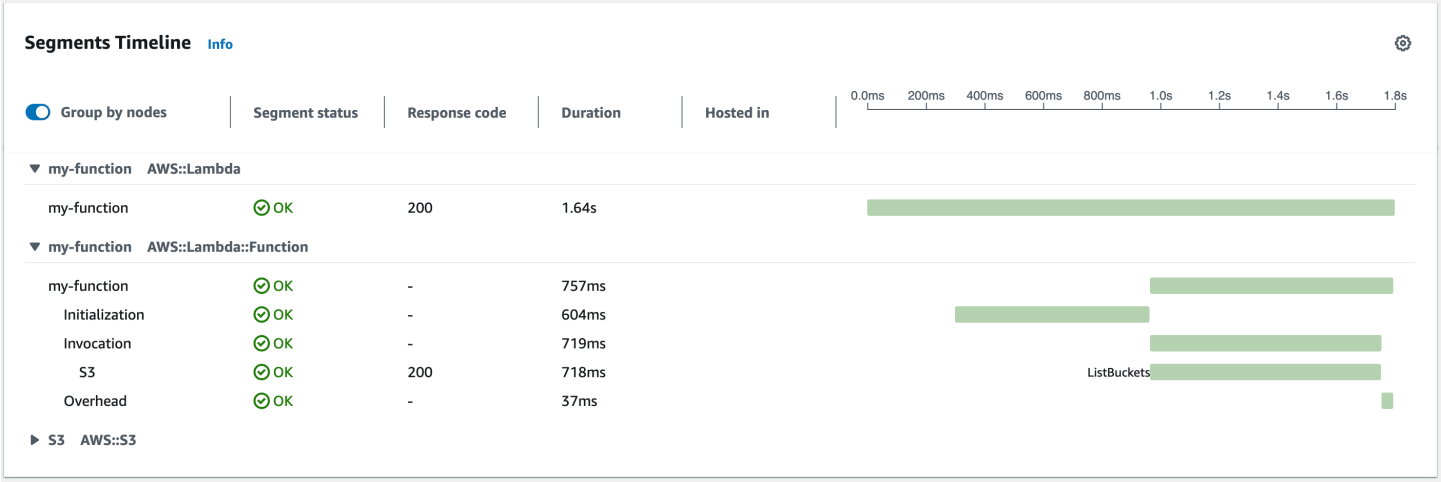
```
# lambda_function.rb
require 'logger'
require 'json'
require 'aws-sdk-lambda'
$client = Aws::Lambda::Client.new()
$client.get_account_settings()

require 'aws-xray-sdk/lambda'

def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  ...
```

在 X-Ray 中，跟踪记录有关由一个或多个服务处理的请求的信息。Lambda 会每个跟踪记录 2 个分段，这些分段将在服务图上创建两个节点。下图突出显示了这两个节点：



位于左侧的第一个节点表示接收调用请求的 Lambda 服务。第二个节点表示特定的 Lambda 函数。以下示例显示了一个包含这 2 个分段的跟踪。两者都命名为 my-function，但其中一个函数具有

AWS::Lambda 源，另一个则具有 AWS::Lambda::Function 源。如果 AWS::Lambda 分段显示错误，则表示 Lambda 服务存在问题。如果 AWS::Lambda::Function 分段显示错误，则说明函数存在问题。



此示例将展开 AWS::Lambda::Function 分段，以显示其三个子分段。

> **Note**
>
> AWS 目前正在实施对 Lambda 服务的更改。由于这些更改，您可能会看到 AWS 账户 中不同 Lambda 函数发出的系统日志消息和跟踪分段的结构和内容之间存在细微差异。
> 此处显示的示例跟踪说明了旧样式函数分段。以下段落介绍了新旧样式分段之间的差异。
> 这些更改将在未来几周内实施，除中国和 GovCloud 区域外，所有 AWS 区域 的函数都将过渡到使用新格式的日志消息和跟踪分段。

旧样式函数分段包含以下子分段：

- 初始化 – 表示加载函数和运行初始化代码所花费的时间。此子分段仅对由您的函数的每个实例处理的第一个事件显示。
- 调用 – 表示执行处理程序代码花费的时间。
- 开销 – 表示 Lambda 运行时为准备处理下一个事件而花费的时间。

新样式函数分段不包含 Invocation 子分段。而是将客户子分段直接附加到函数分段。有关新旧样式函数分段结构的更多信息，请参阅 the section called “了解 X-Ray 跟踪”。

您还可以分析 HTTP 客户端、记录 SQL 查询以及使用注释和元数据创建自定义子段。有关更多信息，请参阅 AWS X-Ray 开发人员指南中的 X-Ray SDK for Ruby。

小节目录

使用 Lambda API 启用活动跟踪

要使用 AWS CLI 或 AWS 开发工具包管理跟踪配置，请使用以下 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下示例 AWS CLI 命令对名为 my-function 的函数启用活跃跟踪。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

跟踪模式是发布函数版本时版本特定配置的一部分。您无法更改已发布版本上的跟踪模式。

使用 AWS CloudFormation 启用主动跟踪

要对 AWS CloudFormation 模板中的 `AWS::Lambda::Function` 资源激活跟踪，请使用 `TracingConfig` 属性。

Example function-inline.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

对于 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 资源，请使用 `Tracing` 属性。

Example template.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

在层中存储运行时依赖项

如果您使用 X-Ray 开发工具包来分析AWS开发工具包客户端和您的函数代码，则您的部署程序包可能会变得相当大。为了避免每次更新函数代码时上载运行时依赖项，请将 X-Ray SDK 打包到 Lambda 层中。

以下示例显示存储 X-Ray SDK for Ruby 的 `AWS::Serverless::LayerVersion` 资源。

Example template.yml – 依赖项层

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-ruby-lib
      Description: Dependencies for the blank-ruby sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - ruby2.5
```

使用此配置，仅在更改运行时依赖项时您才会更新库层。由于函数部署软件包仅包含您的代码，因此可以帮助缩短上传时间。

为依赖项创建层需要更改构建才能在部署之前生成层存档。有关工作示例，请参阅 blank-ruby 示例应用程序。

使用 Java 构建 Lambda 函数

您可以在 AWS Lambda 中运行 Java 代码。Lambda 为 Java 提供运行时，运行您的代码来处理事件。您的代码在 Amazon Linux 环境中运行，该环境包含来自您所管理的 AWS Identity and Access Management (IAM) 的角色的 AWS 凭证。

Lambda 支持以下 Java 运行时。

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
Java 21	`java21`	Amazon Linux 2023	未计划	未计划	未计划
Java 17	`java17`	Amazon Linux 2	未计划	未计划	未计划
Java 11	`java11`	Amazon Linux 2	未计划	未计划	未计划
Java 8	`java8.al2`	Amazon Linux 2	未计划	未计划	未计划

Lambda 提供以下适用于 Java 函数的库：

- com.amazonaws:aws-lambda-java-core（必需）– 定义处理程序方法接口和运行时传递给处理程序的上下文对象。如果您定义自己的输入类型，则这是您唯一需要的库。
- com.amazonaws:aws-lambda-java-events – 调用 Lambda 函数服务的事件的输入类型。
- com.amazonaws:aws-lambda-java-log4j2 – Apache Log4j 2 的 Appender 库，可用于将当前调用的请求 ID 添加到函数日志中。
- 适用于 Java 2.0 的AWS开发工具包：适用于 Java 编程语言的官方AWS开发工具包。

> **Important**
>
> 请勿使用 JDK API 的私有组件，例如私有字段、方法或类。非公共 API 组件可能会在任何更新中更改或删除，从而导致您的应用程序中断。

创建 Java 函数

1. 打开 Lambda 控制台。
2. 选择 Create function（创建函数）。
3. 配置以下设置：
   - 函数名称：输入函数名称。
   - 运行时系统：选择 Java 21。
4. 选择 Create function（创建函数）。
5. 要配置测试事件，请选择测试。
6. 对于事件名称，输入 **`test`**。
7. 选择 Save changes（保存更改）。
8. 要调用该函数，请选择 Test（测试）。

控制台会创建具有名为 `Hello` 的处理程序类的 Lambda 函数。由于 Java 是编译语言，因此您无法在 Lambda 控制台中查看或编辑源代码，但可以修改源代码的配置、调用源代码以及配置触发器。

> **Note**
>
> 要在本地环境中开始应用程序开发，请部署本指南的 GitHub 存储库中提供的其中一个示例应用程序。

`Hello` 类具有一个名为 `handleRequest` 的函数，此函数接受事件对象和上下文对象。这是 Lambda 在调用函数时调用的处理函数。Java 函数运行时从 Lambda 获取调用事件并将其传递到处理程序。在函数配置中，处理程序值为 `example.Hello::handleRequest`。

要更新函数的代码，您需要创建一个部署程序包，这是一个包含函数代码的 .zip 文件归档。随着函数开发的进行，您需要将函数代码存储在源代码控制中、添加库和实现部署自动化。首先，通过命令行创建部署程序包并更新代码。

除了调用事件之外，函数运行时还将上下文对象传递给处理程序。上下文对象包含有关调用、函数和执行环境的其他信息。环境变量中提供了更多信息。

您的 Lambda 函数附带了 CloudWatch Logs 日志组。函数运行时会将每次调用的详细信息发送到 CloudWatch Logs。该运行时会中继调用期间函数输出的任何日志。如果您的函数返回错误，则 Lambda 将为错误设置格式，并将其返回给调用方。

主题

- 定义采用 Java 的 Lambda 函数处理程序
- 使用 .zip 或 JAR 文件归档部署 Java Lambda 函数
- 使用容器镜像部署 Java Lambda 函数
- 使用 Java Lambda 函数的层
- 使用 Lambda SnapStart 提高启动性能
- 自定义 Lambda Java 函数的序列化
- 自定义 Lambda 函数的 Java 运行时启动行为
- 使用 Lambda 上下文对象检索 Java 函数信息
- Java Lambda 函数日志记录和监控
- 在 AWS Lambda 中检测 Java 代码
- AWS Lambda 的 Java 示例应用程序

定义采用 Java 的 Lambda 函数处理程序

Lambda 函数处理程序是函数代码中处理事件的方法。当调用函数时，Lambda 运行处理程序方法。您的函数会一直运行，直到处理程序返回响应、退出或超时。

本指南的 GitHub 存储库提供了易于部署的示例应用程序，用于演示各种处理程序类型。有关详细信息，请参阅本主题的末尾。

Sections

示例处理程序：Java 17 运行时系统

在如下 Java 17 示例中，名为 `HandlerIntegerJava17` 的类定义名为 `handleRequest` 的处理程序方法。处理程序方法接受以下输入：

- `IntegerRecord`，即表示事件数据的自定义 Java 记录。在此示例中，我们对 `IntegerRecord` 进行如下定义：

  ```
  record IntegerRecord(int x, int y, String message) {
  }
  ```

- 上下文对象，其提供的方法和属性包含有关调用、函数和执行环境的信息。

假设我们要编写一个函数，从输入 `IntegerRecord` 中记录 `message` 并返回 `x` 和 `y` 的总和。函数代码如下：

Example HandlerIntegerJava17.java

```
package example;
```

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

// Handler value: example.HandlerInteger
public class HandlerIntegerJava17 implements RequestHandler<IntegerRecord, Integer>{

  @Override
  /*
   * Takes in an InputRecord, which contains two integers and a String.
   * Logs the String, then returns the sum of the two Integers.
   */
  public Integer handleRequest(IntegerRecord event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    logger.log("String found: " + event.message());
    return event.x() + event.y();
  }
}

record IntegerRecord(int x, int y, String message) {
}
```

您通过在函数的配置上设置处理程序参数来指定您想要 Lambda 调用哪个方法。您可以按如下格式来表达处理程序：

- *package*.*Class*::*method* – 完整格式。例如：example.Handler::handleRequest。
- *package*.*Class*：实施处理程序接口的类的缩写格式。例如：example.Handler。

当 Lambda 调用您的处理程序时，Lambda 运行时系统将以 JSON 格式的字符串接收事件，并将其转换为对象。对于前述示例，示例事件可能如下：

Example event.json

```
{
  "x": 1,
  "y": 20,
  "message": "Hello World!"
}
```

您可以保存此文件并使用以下 AWS Command Line Interface（CLI）命令在本地测试自己的函数：

```
aws lambda invoke --function-name function_name --payload file://event.json out.json
```

示例处理程序：Java 11 及以下运行时系统

Lambda 支持 Java 17 和更高版本运行时系统中的记录。在所有 Java 运行时系统中，您可以使用类来表示事件数据。以下示例将整数列表和上下文对象作为输入，并返回列表中所有整数的总和。

Example Handler.java

在以下示例中，名为 `Handler` 的类定义名为 `handleRequest` 的处理程序方法。处理程序方法接受一个事件和上下文对象作为输入并返回一个字符串。

Example HandlerList.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.List;

// Handler value: example.HandlerList
public class HandlerList implements RequestHandler<List<Integer>, Integer>{

  @Override
  /*
   * Takes a list of Integers and returns its sum.
   */
  public Integer handleRequest(List<Integer> event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    logger.log("EVENT TYPE: " + event.getClass().toString());
    return event.stream().mapToInt(Integer::intValue).sum();
  }
}
```

如需更多示例，请参阅示例处理程序代码。

初始化代码

在首次调用您的函数之前，Lambda 会在初始化阶段运行您的静态代码和类构造函数。在初始化期间创建的资源在调用之间保留在内存中，处理程序可以重复使用这些资源数千次。而后，您可以在主处理程序方法外部添加初始化代码，以便节省计算时间并跨多个调用重用资源。

在以下示例中，客户端初始化代码位于主处理程序方法之外。运行时系统会在函数提供自己的第一个事件之前初始化客户端。后续事件的提供速度则要快得多，因为 Lambda 不需要再次初始化客户端。

Example Handler.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.Map;

import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.GetAccountSettingsResponse;
import software.amazon.awssdk.services.lambda.model.LambdaException;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String> {

    private static final LambdaClient lambdaClient = LambdaClient.builder().build();

    @Override
    public String handleRequest(Map<String,String> event, Context context) {

        LambdaLogger logger = context.getLogger();
        logger.log("Handler invoked");

        GetAccountSettingsResponse response = null;
        try {
            response = lambdaClient.getAccountSettings();
        } catch(LambdaException e) {
            logger.log(e.getMessage());
        }
        return response != null ? "Total code size for your account is " +
 response.accountLimit().totalCodeSize() + " bytes" : "Error";
    }
```

```
}
```

选择输入和输出类型

您可以在处理程序方法的签名中指定事件映射到的对象类型。在上述示例中，Java 运行时将事件反序列化为实现 `Map<String,String>` 接口的类型。字符串到字符串映射适用于平面事件，如下所示：

Example Event.json – 天气数据

```
{
  "temperatureK": 281,
  "windKmh": -3,
  "humidityPct": 0.55,
  "pressureHPa": 1020
}
```

但是，每个字段的值必须是字符串或数字。如果事件包含具有对象作为值的字段，则运行时无法对该字段进行反序列化并返回错误。

选择与函数处理的事件数据一起使用的输入类型。您可以使用基本类型、泛型类型或明确定义的类型。

输入类型

- `Integer` `Long`、`Double`、等 – 事件是一个没有其他格式的数字（例如，`3.5`）。运行时将值转换为指定类型的对象。
- `String` – 事件是一个 JSON 字符串，包括引号（例如，`"My string."`）。运行时将值（不带引号）转换为 `String` 对象。
- *`Type`*、`Map<String,`*`Type`*`>` 等 – 事件是一个 JSON 对象。运行时将其反序列化为指定类型或接口的对象。
- `List<Integer>` `List<String>`、`List<Object>`、等 – 事件是一个 JSON 数组。运行时将其反序列化为指定类型或接口的对象。
- `InputStream` – 事件是任何 JSON 类型。运行时将文档的字节流传递给处理程序而不进行修改。您可以对输入进行反序列化并将输出写到输出流。
- 库类型 – 对于AWS服务发送的事件，请使用 aws-lambda-java-events 库中的类型。

如果您定义了自己的输入类型，它应该是可反序列化的、可变的普通旧 Java 对象 (POJO)，对事件中的每个字段具有默认的构造函数和属性。事件中未映射到属性的键以及未包含在事件中的属性将被删除而不显示错误。

输出类型可以是对象或 void。运行时将返回值序列化为文本。如果输出是具有字段的对象，运行时会将其序列化为 JSON 文档。如果它是包装原始值的类型，则运行时返回该值的文本表示形式。

处理程序接口

aws-lambda-java-core 库为处理程序方法定义了两个接口。使用提供的接口简化处理程序配置，并在编译时验证处理程序方法签名。

- com.amazonaws.services.lambda.runtime.RequestHandler
- com.amazonaws.services.lambda.runtime.RequestStreamHandler

RequestHandler 接口是一个泛型类型，它采用两个参数：输入类型和输出类型。这两种类型都必须是对象。使用此接口时，Java 运行时会将事件反序列化为具有输入类型的对象，并将输出序列化为文本。当内置序列化与输入和输出类型配合使用时，请使用此接口。

Example Handler.java – 处理程序接口

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String>{
  @Override
  public String handleRequest(Map<String,String> event, Context context)
```

要使用您自己的序列化，请实现 RequestStreamHandler 接口。使用此接口，Lambda 将向您的处理程序传递输入流和输出流。处理程序从输入流读取字节，写到输出流，并返回 void。

以下 Java 21 示例展示了如何使用 Lambda 函数处理订单。该示例使用缓冲读取器和写入器类型来处理输入和输出流，并展示了如何定义要在函数中使用的自定义 Java 记录。

Example HandlerStream.java

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestStreamHandler;
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.List;
```

```
public class HandlerStream implements RequestStreamHandler {

    private static final ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public void handleRequest(InputStream input, OutputStream output, Context context)
 throws IOException {
        Order order = objectMapper.readValue(input, Order.class);

        processOrder(order);
        OrderAccepted orderAccepted = new OrderAccepted(order.orderId);

        objectMapper.writeValue(output, orderAccepted);
    }

    private void processOrder(Order order) {
        // business logic
    }

    public record Order(@JsonProperty("orderId") String orderId, @JsonProperty("items")
 List<Item> items) { }

    public record Item(@JsonProperty("name") String name, @JsonProperty("quantity")
 Integer quantity) { }

    public record OrderAccepted(@JsonProperty("orderId") String orderId) { }
}
```

Java Lambda 函数的代码最佳实践

在构建 Lambda 函数时，请遵循以下列表中的指南，采用最佳编码实践：

- 从核心逻辑中分离 Lambda 处理程序。这样您可以创建更容易进行单元测试的函数。
- 控制函数部署程序包中的依赖关系。AWS Lambda 执行环境包含许多库。Lambda 会定期更新这些库，以支持最新的功能组合和安全更新。这些更新可能会使 Lambda 函数的行为发生细微变化。要完全控制您的函数所用的依赖项，请使用部署程序包来打包所有依赖项。
- 将依赖关系的复杂性降至最低。首选在执行环境启动时可以快速加载的更简单的框架。例如，首选更简单的 Java 依赖关系注入 (IoC) 框架，如 Dagger 或 Guice，而不是更复杂的 Spring Framework。
- 将部署程序包大小精简为只包含运行时必要的部分。这样会减少调用前下载和解压缩部署程序包所需的时间。对于用 Java 编写的函数，请勿将整个 AWS SDK 库作为部署包的一部分上传，而是要根

据所需的模块有选择地挑选软件开发工具包中的组件（例如 DynamoDB、Simple Storage Service (Amazon S3) 软件开发工具包模块和 Lambda 核心库）。

- 利用执行环境重用来提高函数性能。连接软件开发工具包 (SDK) 客户端和函数处理程序之外的数据库，并在 `/tmp` 目录中本地缓存静态资产。由函数的同一实例处理的后续调用可重用这些资源。这样就可以通过缩短函数运行时间来节省成本。

  为了避免调用之间潜在的数据泄露，请不要使用执行环境来存储用户数据、事件或其他具有安全影响的信息。如果您的函数依赖于无法存储在处理程序的内存中的可变状态，请考虑为每个用户创建单独的函数或单独的函数版本。
- 使用 keep-alive 指令来维护持久连接。Lambda 会随着时间的推移清除空闲连接。在调用函数时尝试重用空闲连接会导致连接错误。要维护您的持久连接，请使用与运行时关联的 keep-alive 指令。有关示例，请参阅在 Node.js 中通过 Keep-Alive 重用连接。
- 使用环境变量将操作参数传递给函数。例如，您在写入 Amazon S3 存储桶时，不应对要写入的存储桶名称进行硬编码，而应将存储桶名称配置为环境变量。
- 避免在 Lambda 函数中使用递归调用，在这种情况下，函数会调用自己或启动可能再次调用该函数的进程。这可能会导致意想不到的函数调用量和升级成本。如果您看到意外的调用量，请立即将函数保留并发设置为 `0` 来限制对函数的所有调用，同时更新代码。
- Lambda 函数代码中不要使用非正式的非公有 API。对于 AWS Lambda 托管式运行时，Lambda 会定期为 Lambda 的内部 API 应用安全性和功能更新。这些内部 API 更新可能不能向后兼容，会导致意外后果，例如，假设您的函数依赖于这些非公有 API，则调用会失败。请参阅 API 参考以查看公开发布的 API 列表。
- 编写幂等代码。为您的函数编写幂等代码可确保以相同的方式处理重复事件。您的代码应该正确验证事件并优雅地处理重复事件。有关更多信息，请参阅如何使我的 Lambda 函数具有幂等性？。

- 避免使用 Java DNS 缓存。Lambda 函数已经缓存了 DNS 响应。如果您使用了另一个 DNS 缓存，则可能会出现连接超时。

  `java.util.logging.Logger` 类可以间接启用 JVM DNS 缓存。要覆盖默认设置，请在初始化 `logger` 之前将 networkaddress.cache.ttl 设置为 0。例如：

```
public class MyHandler {
  // first set TTL property
  static{
   java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
  }
```

```
 // then instantiate logger
  var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
}
```

- 将依赖关系 `.jar` 文件置于单独的 /lib 目录中，可减少 Lambda 解压缩部署程序包（用 Java 编写）所需的时间。这样比将函数的所有代码置于具有大量 `.class` 文件的同一 jar 中要快。有关说明，请参阅使用 .zip 或 JAR 文件归档部署 Java Lambda 函数：

示例处理程序代码

本指南的 GitHub 存储库包括演示如何使用各种处理程序类型和接口的示例应用程序。每个示例应用程序都包含用于轻松部署和清理的脚本、一个 AWS SAM 模板和支持资源。

Java 中的 Lambda 应用程序示例

- java17-examples：这是一种 Java 函数，演示如何使用 Java 记录来表示输入事件数据对象。
- java-basic – 具有单元测试和变量日志记录配置的最小 Java 函数的集合。
- java-events – Java 函数的集合，其中包含用于处理来自 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis 等各种服务的事件的框架代码。这些函数使用最新版本的 aws-lambda-events 库（3.0.0 及更新版本）。这些示例不需要 AWS 开发工具包作为依赖项。
- s3-java – 此 Java 函数可处理来自 Amazon S3 的通知事件，并使用 Java 类库（JCL）从上传的图像文件创建缩略图。
- 自定义序列化 –如何使用 fastJson、Gson、Moshi 和 jackson-jr 等常用库实现自定义序列化的示例。
- 使用 API Gateway 调用 Lambda 函数 – Java 函数，用于扫描包含员工信息的 Amazon DynamoDB 表。然后，该函数使用 Amazon Simple Notification Service 向员工发送短信，祝贺他们工作周年纪念日快乐。此示例使用 API Gateway 调用函数。

`java-events` 和 `s3-java` 应用程序将 AWS 服务事件作为输入并返回字符串。`java-basic` 应用程序包括几种类型的处理程序：

- Handler.java – 以 `Map<String,String>` 作为输入。
- HandlerInteger.java – 以 `Integer` 作为输入。
- HandlerList.java – 以 `List<Integer>` 作为输入。
- HandlerStream.java – 以 `InputStream` 和 `OutputStream` 作为输入。
- HandlerString.java – 以 `String` 作为输入。
- HandlerWeatherData.java – 以自定义类型作为输入。

要测试不同的处理程序类型，只需更改 AWS SAM 模板中的处理程序值即可。有关详细说明，请参阅示例应用程序的自述文件。

使用 .zip 或 JAR 文件归档部署 Java Lambda 函数

您的 AWS Lambda 函数代码由脚本或编译的程序及其依赖项组成。您可以使用部署程序包将函数代码部署到 Lambda。Lambda 支持两种类型的部署程序包：容器镜像和 .zip 文件归档。

本页将介绍如何将部署程序包创建为 .zip 文件或 Jar 文件，然后使用该部署程序包通过 AWS Lambda (AWS Command Line Interface) 将函数代码部署到 AWS CLI。

Sections

先决条件

AWS CLI 是一种开源工具，让您能够在命令行 Shell 中使用命令与 AWS 服务进行交互。要完成本节中的步骤，您必须拥有 AWS CLI 版本 2。

工具和库

Lambda 提供以下适用于 Java 函数的库：

- com.amazonaws:aws-lambda-java-core（必需）– 定义处理程序方法接口和运行时传递给处理程序的上下文对象。如果您定义自己的输入类型，则这是您唯一需要的库。
- com.amazonaws:aws-lambda-java-events – 调用 Lambda 函数服务的事件的输入类型。
- com.amazonaws:aws-lambda-java-log4j2 – Apache Log4j 2 的 Appender 库，可用于将当前调用的请求 ID 添加到函数日志中。
- 适用于 Java 2.0 的AWS开发工具包 – 适用于 Java 编程语言的官方AWS开发工具包。

这些库可通过 Maven 中央存储库获得。将它们添加到您的构建定义中，如下所示：

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
      <version>1.5.1</version>
    </dependency>
  </dependencies>
```

要创建部署程序包，请将函数代码和依赖项编译成单个 .zip 文件或 Java 存档 (JAR) 文件。对于 Gradle，请使用 `Zip` 构建类型。对于 Apache Maven，请使用 Maven Shade 插件。要上传部署包，请使用 Lambda 控制台、Lambda API 或 AWS Serverless Application Model（AWS SAM）。

> **Note**
>
> 为了减小部署程序包的大小，请将函数的依赖项打包到层中。层可让您独立管理依赖项，可供多个函数使用，并可与其他账户共享。有关更多信息，请参阅 Lambda 层。

使用 Gradle 构建部署程序包

要创建包含 Gradle 中函数代码和依赖项的部署包，请使用 `Zip` 构建类型。这是来自完整示例 build.gradle 文件的示例：

Example build.gradle – 构建任务

```
task buildZip(type: Zip) {
    into('lib') {
        from(jar)
        from(configurations.runtimeClasspath)
    }
}
```

此构建配置在 `build/distributions` 目录中生成部署程序包。在 `into('lib')` 语句中，`jar` 任务将包含主类的 jar 归档组装到名为 `lib` 的文件夹中。此外，`configurations.runtimeClassPath` 任务将依赖项库从构建的类路径复制到同一 `lib` 文件夹中。

Example build.gradle – 依赖项

```
dependencies {
    ...
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    implementation 'org.apache.logging.log4j:log4j-api:2.17.1'
    implementation 'org.apache.logging.log4j:log4j-core:2.17.1'
    runtimeOnly 'org.apache.logging.log4j:log4j-slf4j18-impl:2.17.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
    ...
}
```

Lambda 按 Unicode 字母顺序加载 JAR 文件。如果 `lib` 目录中的多个 JAR 文件包含相同的类，则使用第一个。可以使用以下 shell 脚本来识别重复类：

Example test-zip.sh

```
mkdir -p expanded
unzip path/to/my/function.zip -d expanded
find ./expanded/lib -name '*.jar' | xargs -n1 zipinfo -1 | grep '.*.class' | sort |
 uniq -c | sort
```

为依赖项创建 Java 层

> **Note**
>
> 在 Java 等编译语言中将层与函数结合使用，不一定会产生与使用 Python 等解释性语言的相同效果。由于 Java 是一种编译语言，因此函数仍然需要在初始化阶段将所有共享程序集手动加载到内存中，而这可能会增加冷启动时间。我们建议您改为在编译时包含任何共享代码，以充分利用内置编译器的优化。

本部分中的说明旨在向您展示如何将依赖项包含在层中。有关如何将依赖项包含在部署包中的说明，请参阅 the section called “使用 Gradle 构建部署程序包” 或 the section called “使用 Maven 构建部署程序包”。

当您向函数添加层时，Lambda 会将层内容加载到该执行环境的 `/opt` 目录中。对于每个 Lambda 运行时系统，PATH 变量都包括 `/opt` 目录中的特定文件夹路径。为确保 PATH 变量能够获取层内容，层 .zip 文件应在以下文件夹路径中具有依赖项：

- `java/lib` (CLASSPATH)

例如，层.zip 文件结构可能如下所示：

```
jackson.zip
# java/lib/jackson-core-2.2.3.jar
```

此外，Lambda 会自动检测 `/opt/lib` 目录中的任何库，以及 `/opt/bin` 目录中的任何二进制文件。为确保 Lambda 正确获取层内容，还可以创建包含以下结构的层：

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

打包层后，请参阅 the section called “创建和删除层” 和 the section called “添加层” 以完成层设置。

使用 Maven 构建部署程序包

要使用 Maven 构建部署程序包，请使用 Maven Shade 插件。该插件创建一个包含编译的函数代码及其所有依赖项的 JAR 文件。

Example pom.xml – 插件配置

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

要构建部署程序包，请使用 `mvn package` 命令。

```
[INFO] Scanning for projects...
[INFO] -----------------------< com.example:java-maven >-----------------------
[INFO] Building java-maven-function 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ java-maven ---
[INFO] Building jar: target/java-maven-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-shade-plugin:3.2.2:shade (default) @ java-maven ---
[INFO] Including com.amazonaws:aws-lambda-java-core:jar:1.2.2 in the shaded jar.
[INFO] Including com.amazonaws:aws-lambda-java-events:jar:3.11.1 in the shaded jar.
[INFO] Including joda-time:joda-time:jar:2.6 in the shaded jar.
[INFO] Including com.google.code.gson:gson:jar:2.8.6 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing target/java-maven-1.0-SNAPSHOT.jar with target/java-maven-1.0-
SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
```

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  8.321 s
[INFO] Finished at: 2020-03-03T09:07:19Z
[INFO] ------------------------------------------------------------------------
```

此命令在 `target` 目录中生成 JAR 文件。

> **Note**
>
> 如果您使用的是多版本 JAR（MRJAR），则必须在 `lib` 目录中包含 MRJAR（即由 Maven Shade 插件生成的着色 JAR），并在将部署包上传到 Lambda 之前对其进行压缩。否则，Lambda 可能无法正确解压 JAR 文件，从而导致 `MANIFEST.MF` 文件被忽略。

如果您使用 Appender 库 (`aws-lambda-java-log4j2`)，还必须为 Maven Shade 插件配置一个转换器。转换器库合并同时出现在 Appender 库和 Log4j 中的缓存文件的版本。

Example pom.xml – 具有 Log4j 2 Appender 的插件配置

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <transformers>
                <transformer
implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCacheFile
                </transformer>
              </transformers>
            </configuration>
          </execution>
        </executions>
```

```
        <dependencies>
          <dependency>
            <groupId>com.github.edwgiz</groupId>
            <artifactId>maven-shade-plugin.log4j2-cachefile-transformer</artifactId>
            <version>2.13.0</version>
          </dependency>
        </dependencies>
      </plugin>
```

使用 Lambda 控制台上传部署包

要创建新函数，必须先在控制台中创建该函数，然后上传您的 .zip 或 JAR 文件。要更新现有函数，请打开函数页面，然后按照相同的步骤添加更新的 .zip 或 JAR 文件。

如果您的部署包文件小于 50MB，则可以通过直接从本地计算机上传该文件来创建或更新函数。对于大于 50MB 的 .zip 或 JAR 文件，必须首先将您的程序包上传到 Amazon S3 存储桶。有关如何使用 AWS Management Console 将文件上传到 Amazon S3 存储桶的说明，请参阅 Amazon S3 入门。要使用 AWS CLI 上传文件，请参阅《AWS CLI 用户指南》中的移动对象。

> **Note**
>
> 您无法更改现有函数的部署包类型（.zip 或容器映像）。例如，您无法将容器映像函数转换为使用 .zip 文件归档。您必须创建新函数。

创建新函数（控制台）

1. 打开 Lambda 控制台的“函数”页面，然后选择创建函数。
2. 选择从头开始创作。
3. 在基本信息中，执行以下操作：
   a. 对于函数名称，输入函数的名称。
   b. 对于运行时系统，选择要使用的运行时系统。
   c. （可选）对于架构，选择要用于函数的指令集架构。默认架构为 x86_64。确保您的函数的 .zip 部署包与您选择的指令集架构兼容。
4. （可选）在 Permissions（权限）下，展开 Change default execution role（更改默认执行角色）。您可以创建新的执行角色，也可以使用现有角色。
5. 选择 Create function (创建函数)。Lambda 使用您选择的运行时系统创建基本“Hello world”函数。

从本地计算机上传 .zip 或 JAR 归档（控制台）

1. 在 Lambda 控制台的“函数”页面中，选择要为其上传 .zip 或 JAR 文件的函数。
2. 选择代码选项卡。
3. 在代码源窗格中，选择上传自。
4. 选择 .zip 或 .jar 文件。
5. 要上传 .zip 或 JAR 文件，请执行以下操作：
   a. 选择上传，然后在文件选择器中选择您的 .zip 或 JAR 文件。
   b. 选择打开。
   c. 选择保存。

从 Amazon S3 存储桶上传 .zip 或 JAR 归档（控制台）

1. 在 Lambda 控制台的“函数”页面中，选择要为其上传新 .zip 或 JAR 文件的函数。
2. 选择代码选项卡。
3. 在代码源窗格中，选择上传自。
4. 选择 Amazon S3 位置。
5. 粘贴 .zip 文件的 Amazon S3 链接 URL，然后选择保存。

使用 AWS CLI 上传部署包

您可以使用 AWS CLI 创建新函数或使用 .zip 或 JAR 文件更新现有函数。使用 create-function 和 update-function-code 命令部署 .zip 或 JAR 程序包。如果您的文件小于 50MB，则可以从本地生成计算机上的文件位置上传程序包。对于较大的文件，必须从 Amazon S3 存储桶上传 .zip 或 JAR 程序包。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

> **Note**
>
> 如果您使用 AWS CLI 从 Amazon S3 存储桶上传 .zip 或 JAR 文件，则该存储桶必须与您的函数位于同一个 AWS 区域中。

要通过 AWS CLI 使用 .zip 或 JAR 文件创建新函数，则必须指定以下内容：

- 函数的名称 (--function-name)
- 函数的运行时系统 (--runtime)
- 函数的执行角色 (--role) 的 Amazon 资源名称（ARN）
- 函数代码 (--handler) 中处理程序方法的名称

还必须指定 .zip 或 JAR 文件的位置。如果 .zip 或 JAR 文件位于本地生成计算机上的文件夹中，请使用 --zip-file 选项指定文件路径，如以下示例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime java21 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

要指定 .zip 文件在 Amazon S3 存储桶中的位置，请使用 --code 选项，如以下示例命令所示。您只需对版本控制对象使用 S3ObjectVersion 参数。

```
aws lambda create-function --function-name myFunction \
--runtime java21 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

要使用 CLI 更新现有函数，请使用 --function-name 参数指定函数的名称。您还必须指定要用于更新函数代码的 .zip 文件的位置。如果 .zip 文件位于本地生成计算机上的文件夹中，请使用 --zip-file 选项指定文件路径，如以下示例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

要指定 .zip 文件在 Amazon S3 存储桶中的位置，请使用 --s3-bucket 和 --s3-key 选项，如以下示例命令所示。您只需对版本控制对象使用 --s3-object-version 参数。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

使用 AWS SAM 上传部署程序包

您可以使用 AWS SAM 自动部署函数代码、配置和依赖项。AWS SAM 是 AWS CloudFormation 的一个扩展，它提供用于定义无服务器应用程序的简化语法。以下示例模板在 Gradle 使用的 `build/distributions` 目录中定义了一个包含部署程序包的函数。

Example template.yml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/java-basic.zip
      Handler: example.Handler
      Runtime: java21
      Description: Java function
      MemorySize: 512
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
        - AWSLambdaVPCAccessExecutionRole
      Tracing: Active
```

要创建函数，请使用 `package` 和 `deploy` 命令。这些命令是对 AWS CLI 的自定义。它们包装其他命令以将部署程序包上载到 Amazon S3，使用对象 URI 重写模板，然后更新函数的代码。

以下示例脚本运行 Gradle 构建并上传其创建的部署程序包。它在您第一次运行它时创建一个 AWS CloudFormation 堆栈。如果堆栈已经存在，脚本会更新它。

Example deploy.sh

```
#!/bin/bash
set -eo pipefail
aws cloudformation package --template-file template.yml --s3-bucket MY_BUCKET --output-
template-file out.yml
```

```
aws cloudformation deploy --template-file out.yml --stack-name java-basic --
capabilities CAPABILITY_NAMED_IAM
```

有关完整的工作示例，请参阅以下示例应用程序。

Java 中的 Lambda 应用程序示例

- java17-examples：这是一种 Java 函数，演示如何使用 Java 记录来表示输入事件数据对象。
- java-basic – 具有单元测试和变量日志记录配置的最小 Java 函数的集合。
- java-events – Java 函数的集合，其中包含用于处理来自 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis 等各种服务的事件的框架代码。这些函数使用最新版本的 aws-lambda-events 库（3.0.0 及更新版本）。这些示例不需要 AWS 开发工具包作为依赖项。
- s3-java – 此 Java 函数可处理来自 Amazon S3 的通知事件，并使用 Java 类库（JCL）从上传的图像文件创建缩略图。
- 自定义序列化 –如何使用 fastJson、Gson、Moshi 和 jackson-jr 等常用库实现自定义序列化的示例。
- 使用 API Gateway 调用 Lambda 函数 – Java 函数，用于扫描包含员工信息的 Amazon DynamoDB 表。然后，该函数使用 Amazon Simple Notification Service 向员工发送短信，祝贺他们工作周年纪念日快乐。此示例使用 API Gateway 调用函数。

使用容器镜像部署 Java Lambda 函数

有三种方法可以为 Java Lambda 函数构建容器映像：

- 使用 Java 的 AWS 基本映像

  AWS 基本映像会预加载一个语言运行时系统、一个用于管理 Lambda 和函数代码之间交互的运行时系统接口客户端，以及一个用于本地测试的运行时系统接口仿真器。

- 使用 AWS 仅限操作系统的基础镜像

  AWS 仅限操作系统的运行时系统包含 Amazon Linux 发行版和运行时系统接口模拟器。这些镜像通常用于为编译语言（例如 Go 和 Rust）以及 Lambda 未提供基础映像的语言或语言版本（例如 Node.js 19）创建容器镜像。您也可以使用仅限操作系统的基础映像来实施自定义运行时系统。要使映像与 Lambda 兼容，您必须在映像中包含 Java 的运行时系统接口客户端。

- 使用非 AWS 基本映像

  您还可以使用其他容器注册表的备用基本映像，例如 Alpine Linux 或 Debian。您还可以使用您的组织创建的自定义映像。要使映像与 Lambda 兼容，您必须在映像中包含 Java 的运行时系统接口客户端。

> **Tip**
>
> 要缩短 Lambda 容器函数激活所需的时间，请参阅 Docker 文档中的使用多阶段构建。要构建高效的容器映像，请遵循编写 Dockerfiles 的最佳实践。

此页面介绍了如何为 Lambda 构建、测试和部署容器映像。

主题

Java AWS 基本映像

AWS 为 Java 提供了以下基本镜像：

标签	运行时	操作系统	Dockerfile	淘汰
21	Java 21	Amazon Linux 2023	GitHub 上的适用于 Java 21 的 Dockerfile	未计划
17	Java 17	Amazon Linux 2	GitHub 上的适用于 Java 17 的 Dockerfile	未计划
11	Java 11	Amazon Linux 2	GitHub 上的适用于 Java 11 的 Dockerfile	未计划
8.al2	Java 8	Amazon Linux 2	GitHub 上的适用于 Java 8 的 Dockerfile	未计划

Amazon ECR 存储库：gallery.ecr.aws/lambda/java

Java 21 及更高版本的基础映像基于 Amazon Linux 2023 最小容器映像。早期的基础映像使用 Amazon Linux 2。与 Amazon Linux 2 相比，AL2023 具有多项优势，包括较小的部署占用空间以及 `glibc` 等更新版本的库。

基于 AL2023 的映像使用 `microdnf`（符号链接为 `dnf`）作为软件包管理器，而不是 Amazon Linux 2 中的默认软件包管理器 `yum`。`microdnf` 是 `dnf` 的独立实现。有关基于 AL2023 的映像中已包含软件包的列表，请参阅 Comparing packages installed on Amazon Linux 2023 Container Images 中的 Minimal Container 列。有关 AL2023 和 Amazon Linux 2 之间区别的更多信息，请参阅 AWS 计算博客上的 Introducing the Amazon Linux 2023 runtime for AWS Lambda。

> **Note**
>
> 要在本地运行基于 AL2023 的映像，包括使用 AWS Serverless Application Model（AWS SAM），您必须使用 Docker 版本 20.10.10 或更高版本。

使用 Java 的 AWS 基本映像

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Java（例如，Amazon Corretto）

- Docker（Java 21 及更高版本基础映像的最低版本为 20.10.10）
- Apache Maven 或 Gradle
- AWS CLI 版本 2

从基本映像创建映像

Maven

1. 运行以下命令，以使用适用于 Lambda 的原型创建 Maven 项目。以下参数为必需参数：

   - 服务 – 在 Lambda 函数中使用的 AWS 服务 客户端。有关可用源列表，请参阅 GitHub 上的 aws-sdk-java-v2/services。
   - 区域 – 要在其中创建 Lambda 函数的 AWS 区域。
   - groupId – 应用程序的完整程序包命名空间。
   - artifactId – 项目名称。这将成为项目的目录的名称。

   在 Linux 和 macOS 中，运行以下命令：

   ```
   mvn -B archetype:generate \
      -DarchetypeGroupId=software.amazon.awssdk \
      -DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
      -DgroupId=com.example.myapp \
      -DartifactId=myapp
   ```

   在 PowerShell 中，运行以下命令：

   ```
   mvn -B archetype:generate `
      "-DarchetypeGroupId=software.amazon.awssdk" `
      "-DarchetypeArtifactId=archetype-lambda" "-Dservice=s3" "-Dregion=US_WEST_2"
    `
      "-DgroupId=com.example.myapp" `
      "-DartifactId=myapp"
   ```

   适用于 Lambda 的 Maven 原型已预配置为使用 Java SE 8 进行编译，并包含对 AWS SDK for Java 的依赖项。如果使用其他原型或其他方法创建项目，则必须为 Maven 配置 Java 编译器并将开发工具包声明为依赖项。

2. 打开 *myapp*/src/main/java/com/example/*myapp* 目录，找到 App.java 文件。这是适用于 Lambda 函数的代码。您可以使用提供的示例代码进行测试，也可以将其替换为您自己的代码。
3. 导航回项目的根目录，然后创建一个具有以下配置的新 Dockerfile：

   - 将 FROM 属性设置为基本映像的 URI。
   - 将 CMD 参数设置为 Lambda 函数处理程序。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 root 用户。

   Example Dockerfile

   ```
   FROM public.ecr.aws/lambda/java:21

   # Copy function code and runtime dependencies from Maven layout
   COPY target/classes ${LAMBDA_TASK_ROOT}
   COPY target/dependency/* ${LAMBDA_TASK_ROOT}/lib/

   # Set the CMD to your handler (could also be done as a parameter override
    outside of the Dockerfile)
   CMD [ "com.example.myapp.App::handleRequest" ]
   ```

4. 编译项目并收集运行时系统依赖项。

   ```
   mvn compile dependency:copy-dependencies -DincludeScope=runtime
   ```

5. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 docker-image 并为其提供 test 标签。

   ```
   docker build --platform linux/amd64 -t docker-image:test .
   ```

   > **Note**
   >
   > 该命令指定了 --platform linux/amd64 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 --platform linux/arm64 选项。

Gradle

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir example
cd example
```

2. 运行以下命令来让 Gradle 在环境中的 example 目录中生成新的 Java 应用程序项目。在选择构建脚本 DSL 中，选择 2: Groovy。

```
gradle init --type java-application
```

3. 打开 */example*/app/src/main/java/*example* 目录，找到 App.java 文件。这是适用于 Lambda 函数的代码。您可以使用以下示例代码进行测试，也可以将其替换为您自己的代码。

   Example App.java

```
package com.example;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
public class App implements RequestHandler<Object, String> {
    public String handleRequest(Object input, Context context) {
        return "Hello world!";
    }
}
```

4. 打开 build.gradle文件。如果您使用的是上一步中的示例函数代码，请将 build.gradle 的内容替换为以下内容。如果您使用的是自己的函数代码，请根据需要修改 build.gradle 文件。

   Example build.gradle (Groovy DSL)

```
plugins {
  id 'java'
}
group 'com.example'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
  mavenCentral()
}
dependencies {
```

```
  implementation 'com.amazonaws:aws-lambda-java-core:1.2.1'
}
jar {
  manifest {
      attributes 'Main-Class': 'com.example.App'
  }
}
```

5. 第 2 步中的 `gradle init` 命令还在 `app/test` 目录中生成了一个虚拟测试用例。在本教程中，请删除 `/test` 目录，从而跳过运行测试。
6. 构建 项目。

```
gradle build
```

7. 在项目的根目录 (`/example`) 中，创建一个具有以下配置的 Dockerfile：
   - 将 FROM 属性设置为基本映像的 URI。
   - 使用 COPY 命令将函数代码和运行时系统依赖项复制到 {LAMBDA_TASK_ROOT}，此为 Lambda 定义的环境变量。
   - 将 CMD 参数设置为 Lambda 函数处理程序。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 `root` 用户。

   Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Gradle layout
COPY app/build/classes/java/main ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.App::handleRequest" ]
```

8. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 `docker-image` 并为其提供 `test` 标签。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 该命令指定了 `--platform linux/amd64` 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 `--platform linux/arm64` 选项。

（可选）在本地测试映像

1. 使用 docker run 命令启动 Docker 映像。在此示例中，`docker-image` 是映像名称，`test` 是标签。

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   此命令会将映像作为容器运行，并在 `localhost:9000/2015-03-31/functions/function/invocations` 创建本地端点。

   > **Note**
   >
   > 如果为 ARM64 指令集架构创建 Docker 映像，请务必使用 `--platform linux/arm64` 选项，而不是 `--platform linux/amd64` 选项。

2. 在新的终端窗口中，将事件发布到本地端点。

   Linux/macOS

   在 Linux 和 macOS 中，运行以下 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

PowerShell

在 PowerShell 中，运行以下 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 获取容器 ID。

```
docker ps
```

4. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 111122223333 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。
   - 将 `<ECRrepositoryUri>` 替换为复制的 `repositoryUri`。确保 URI 末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 `ImageUri`，指定之前的存储库 URI。确保 URI 末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，`--publish` 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

将备用基本映像与运行时系统接口客户端配合使用

如果使用仅限操作系统的基础映像或者备用基础映像，则必须在映像中包括运行时系统接口客户端。运行时系统接口客户端可扩展 将 Lambda 运行时 API 用于自定义运行时，用于管理 Lambda 和函数代码之间的交互。

在 Dockerfile 中安装 Java 的运行时系统接口客户端，或作为项目中的依赖项安装。例如，要使用 Maven 程序包管理器安装运行时系统接口客户端，请将以下内容添加到您的 `pom.xml` 文件中：

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
    <version>2.3.2</version>
</dependency>
```

有关程序包的详细信息，请参阅 Maven Central 存储库中的 AWS Lambda Java 运行时系统接口客户端。您还可以在 AWS Lambda Java 支持库 GitHub 存储库中查看运行时系统接口客户端源代码。

以下示例演示了如何使用 Amazon Corretto 映像为 Java 构建容器映像。Amazon Corretto 是开放 Java 开发工具包（OpenJDK）的免费、多平台、生产就绪型分发版。Maven 项目将运行时系统接口客户端作为依赖项包括在内。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Java（例如，Amazon Corretto）
- Docker

- Apache Maven
- AWS CLI 版本 2

从备用基本映像创建映像

1. 创建 Maven 项目。以下参数为必需参数：

   - groupId – 应用程序的完整程序包命名空间。
   - artifactId – 项目名称。这将成为项目的目录的名称。

   Linux/macOS

   ```
   mvn -B archetype:generate \
      -DarchetypeArtifactId=maven-archetype-quickstart \
      -DgroupId=example \
      -DartifactId=myapp \
      -DinteractiveMode=false
   ```

   PowerShell

   ```
   mvn -B archetype:generate `
      -DarchetypeArtifactId=maven-archetype-quickstart `
      -DgroupId=example `
      -DartifactId=myapp `
      -DinteractiveMode=false
   ```

2. 打开项目目录。

   ```
   cd myapp
   ```

3. 打开 pom.xml 文件并将相应内容替换为以下内容。此文件将 aws-lambda-java-runtime-interface-client 作为依赖项包括在内。或者，您可以在 Dockerfile 中安装运行时系统接口客户端。但是，最简单的方法是将库作为依赖项包含在内。

   ```
   <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
   www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
   maven-v4_0_0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>example</groupId>
   ```

```
  <artifactId>hello-lambda</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello-lambda</name>
  <url>http://maven.apache.org</url>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
      <version>2.3.2</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>3.1.2</version>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

4. 打开 *myapp*/src/main/java/com/example/*myapp* 目录，找到 App.java 文件。这是适用于 Lambda 函数的代码。使用以下内容替换相应代码。

   Example 函数处理程序

```
package example;
```

```
public class App {
    public static String sayHello() {
        return "Hello world!";
    }
}
```

5. 第 1 步中的 `mvn -B archetype:generate` 命令还会在 `src/test` 目录中生成了一个虚拟测试用例。在本教程中，请完整删除生成的 `/test` 目录，从而跳过运行测试。
6. 导航回项目的根目录，然后创建一个新 Dockerfile。以下示例 Dockerfile 使用 Amazon Corretto 映像。Amazon Corretto 是 OpenJDK 的免费、多平台、生产就绪型分发版。

   - 将 FROM 属性设置为基本映像的 URI。
   - 将 ENTRYPOINT 设置为您希望 Docker 容器在启动时运行的模块。在本例中，模块为运行时系统接口客户端。
   - 将 CMD 参数设置为 Lambda 函数处理程序。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 `root` 用户。

   Example Dockerfile

```
FROM public.ecr.aws/amazoncorretto/amazoncorretto:21 as base

# Configure the build environment
FROM base as build
RUN yum install -y maven
WORKDIR /src

# Cache and copy dependencies
ADD pom.xml .
RUN mvn dependency:go-offline dependency:copy-dependencies

# Compile the function
ADD . .
RUN mvn package

# Copy the function artifact and dependencies onto a clean base
FROM base
WORKDIR /function
```

```
COPY --from=build /src/target/dependency/*.jar ./
COPY --from=build /src/target/*.jar ./

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/bin/java", "-cp", "./*",
 "com.amazonaws.services.lambda.runtime.api.client.AWSLambda" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "example.App::sayHello" ]
```

7. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 docker-image 并为其提供 test 标签。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 该命令指定了 --platform linux/amd64 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 --platform linux/arm64 选项。

（可选）在本地测试映像

使用运行时系统接口仿真器在本地测试映像。您可以将仿真器构建到映像中，也可以使用以下程序将其安装在本地计算机上。

在本地计算机上安装并运行运行时系统接口仿真器

1. 从项目目录中，运行以下命令以从 GitHub 下载运行时系统接口仿真器（x86-64 架构）并将其安装在本地计算机上。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   要安装 arm64 仿真器，请将上一条命令中的 GitHub 存储库 URL 替换为以下内容：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

要安装 arm64 模拟器，请将 $downloadLink 替换为以下内容：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. 使用 docker run 命令启动 Docker 映像。请注意以下几点：

   - docker-image 是映像名称，test 是标签。
   - /usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda example.App::sayHello 是 ENTRYPOINT，后跟您 Dockerfile 中的 CMD。

   Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/bin/java -cp './*'
 com.amazonaws.services.lambda.runtime.api.client.AWSLambda
 example.App::sayHello
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/bin/java -cp './*'
 com.amazonaws.services.lambda.runtime.api.client.AWSLambda
 example.App::sayHello
```

此命令会将映像作为容器运行，并在 `localhost:9000/2015-03-31/functions/function/invocations` 创建本地端点。

> **Note**
>
> 如果为 ARM64 指令集架构创建 Docker 映像，请务必使用 `--platform linux/arm64` 选项，而不是 `--platform linux/amd64` 选项。

3. 将事件发布到本地端点。

Linux/macOS

在 Linux 和 macOS 中，运行以下 `curl` 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，运行以下 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 获取容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 111122223333 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。
   - 将 `<ECRrepositoryUri>` 替换为复制的 `repositoryUri`。确保 URI 末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 `ImageUri`，指定之前的存储库 URI。确保 URI 末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，--publish 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

使用 Java Lambda 函数的层

Lambda 层是包含补充代码或数据的 .zip 文件存档。层通常包含库依赖项、自定义运行时系统或配置文件。创建层涉及三个常见步骤：

1. 打包层内容。此步骤需要创建 .zip 文件存档，其中包含要在函数中使用的依赖项。
2. 在 Lambda 中创建层。
3. 将层添加到函数。

本主题包含有关如何正确打包并创建具有外部库依赖项的 Java Lambda 层的步骤和指南。

主题

先决条件

要完成本部分中的步骤，您必须满足以下条件：

- Java 21
- Apache Maven 3.8.6 或更高版本
- AWS CLI 版本 2

> **Note**
>
> 确保 Maven 引用的 Java 版本与要部署的函数的 Java 版本相同。例如，若要部署 Java 21 函数，`mvn -v` 命令应在输出中列出 Java 版本 21：
>
> ```
> Apache Maven 3.8.6
> ...
> ```

```
Java version: 21.0.2, vendor: Oracle Corporation, runtime: /Library/Java/
JavaVirtualMachines/jdk-21.jdk/Contents/Home
...
```

在整个主题中，我们会引用 awsdocs GitHub 存储库中的 layer-java 示例应用程序。该应用程序包含用于下载依赖项并生成层的脚本。该应用程序还包含相应的函数，函数使用来自层的依赖项。创建层后，即可部署并调用相应的函数来验证一切是否正常运行。由于使用 Java 21 运行时系统来运行这些函数，因此这些层还必须与 Java 21 兼容。

layer-java 示例应用程序在两个子目录中包含一个示例。layer 目录包含用于定义层依赖项的 pom.xml 文件，以及用于生成层的脚本。function 目录包含示例函数，用于帮助测试该层是否正常工作。本教程将演示如何创建并打包该层。

Java 层与 Amazon Linux 的兼容性

创建层的第一步是将所有层内容捆绑到.zip 文件存档中。由于 Lambda 函数在 Amazon Linux 上运行，因此层内容必须能够在 Linux 环境中编译和构建。

Java 代码采用独立于平台的设计，即便本地计算机不使用 Linux 环境，您也可以在本地计算机上打包层。将 Java 层上传到 Lambda 后，Java 层仍将与 Amazon Linux 兼容。

Java 运行时系统的层路径

当您向函数添加层时，Lambda 会将层内容加载到该执行环境的 /opt 目录中。对于每个 Lambda 运行时系统，PATH 变量都包括 /opt 目录中的特定文件夹路径。为确保 PATH 变量能够获取层内容，层 .zip 文件应在以下文件夹路径中具有依赖项：

- java/lib

例如，您在本教程中创建生成的层.zip 文件具有以下目录结构：

```
layer_content.zip
# java
    # lib
        # layer-java-layer-1.0-SNAPSHOT.jar
```

layer-java-layer-1.0-SNAPSHOT.jar JAR 文件（包含所有必需依赖项的 uber-jar）位于 java/lib 目录中，位置正确。这可确保 Lambda 在函数调用期间可以找到该库。

打包层内容

本示例将以下两个 Java 库打包到单个 JAR 文件中：

- aws-lambda-java-core：一组在 AWS Lambda 中用于处理 Java 的最小接口定义
- Jackson：一套特别适用于处理 JSON 的热门数据处理工具。

完成以下步骤，安装并打包层内容。

安装并打包层内容

1. 克隆 aws-lambda-developer-guide GitHub 存储库，其中包含 sample-apps/layer-java 目录中需要的示例代码。

```
git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
```

2. 导航到 layer-java 示例应用程序的 layer 目录。此目录包含用于正确创建并打包层的脚本。

```
cd aws-lambda-developer-guide/sample-apps/layer-java/layer
```

3. 检查 pom.xml 文件。在 <dependencies> 部分，定义要包含在层中的依赖项，即 aws-lambda-java-core 和 jackson-databind 库。您可以更新此文件，纳入要包含在层中的任何依赖项。

   Example pom.xml

```
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-lambda-java-core</artifactId>
        <version>1.2.3</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.17.0</version>
    </dependency>
</dependencies>
```

> **Note**
>
> pom.xml 文件的 <build> 部分包含两个插件。maven-compiler-plugin 可编译源代码。maven-shade-plugin 可将构件打包到一个 uber-jar 中。

4. 确保拥有运行这两个脚本的权限。

```
chmod 744 1-install.sh && chmod 744 2-package.sh
```

5. 使用以下命令运行 1-install.sh 脚本：

```
./1-install.sh
```

此脚本在当前目录中运行 mvn clean install。这会在 target/ 目录中创建包含所有必需依赖项的 uber-jar。

Example 1-install.sh

```
mvn clean install
```

6. 使用以下命令运行 2-package.sh 脚本：

```
./2-package.sh
```

此脚本会创建正确打包层内容所需的 java/lib 目录结构。然后，其会将 uber-jar 从 /target 目录复制到新创建的 java/lib 目录中。最后，此脚本会将 java 目录的内容压缩到名为 layer_content.zip 的文件中。这便是层的 .zip 文件。您可以解压缩文件，验证是否包含正确的文件结构，如 the section called “Java 运行时系统的层路径”部分所示。

Example 2-package.sh

```
mkdir java
mkdir java/lib
cp -r target/layer-java-layer-1.0-SNAPSHOT.jar java/lib/
zip -r layer_content.zip java
```

创建层

在本部分，您会获取在上一部分中生成的 `layer_content.zip` 文件，将其作为 Lambda 层上传。您可以使用 AWS Management Console 上传层，也可以通过 AWS Command Line Interface（AWS CLI）使用 Lambda API 上传层。上传层 .zip 文件时，在以下 PublishLayerVersion AWS CLI 命令中，将 `java21` 指定为兼容的运行时系统，并将 `arm64` 指定为兼容的架构。

```
aws lambda publish-layer-version --layer-name java-jackson-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes java21 \
    --compatible-architectures "arm64"
```

注意响应中的 `LayerVersionArn`，与 `arn:aws:lambda:us-east-1:123456789012:layer:java-jackson-layer:1` 类似。在本教程的下一步中，在将层添加到函数时，您要用到此 Amazon 资源名称（ARN）。

将层添加到函数

在本部分，您要部署在函数代码中使用 Jackson 库的示例 Lambda 函数，然后附加该层。要部署该函数，您需要一个 the section called “执行角色（函数访问其他资源的权限）”。如果目前没有执行角色，则按照可折叠部分中的步骤操作。

（可选）创建执行角色

创建执行角色

1. 在 IAM 控制台中，打开 Roles（角色）页面。
2. 选择创建角色。
3. 创建具有以下属性的角色。
   - Trusted entity（可信任的实体）– Lambda。
   - Permissions（权限）– AWSLambdaBasicExecutionRole。
   - Role name（角色名称）– **`lambda-role`**。

   AWSLambdaBasicExecutionRole 策略具有函数将日志写入 CloudWatch Logs 所需的权限。

Lambda 函数代码接受 Map<String, String> 作为输入，并使用 Jackson 将输入写为 JSON 字符串，然后再将其转换为预定义的 F1Car Java 对象。最后，函数使用 F1Car 对象中的字段来构造函数返回的字符串。

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.IOException;
import java.util.Map;

public class Handler {

    public String handleRequest(Map<String, String> input, Context context) throws
 IOException {
        // Parse the input JSON
        ObjectMapper objectMapper = new ObjectMapper();
        F1Car f1Car = objectMapper.readValue(objectMapper.writeValueAsString(input),
 F1Car.class);

        StringBuilder finalString = new StringBuilder();
        finalString.append(f1Car.getDriver());
        finalString.append(" is a driver for team ");
        finalString.append(f1Car.getTeam());
        return finalString.toString();
    }
}
```

部署 Lambda 函数

1. 导航到 function/ 目录。如果当前在 layer/ 目录中，请运行以下命令：

   ```
   cd ../function
   ```

2. 使用以下 Maven 命令构建项目：

   ```
   mvn package
   ```

   此命令在名为 layer-java-function-1.0-SNAPSHOT.jar 的 target/ 目录中生成 JAR 文件。

3. 部署函数。在以下 AWS CLI 命令中，将 `--role` 参数替换为执行角色 ARN：

```
aws lambda create-function --function-name java_function_with_layer \
    --runtime java21 \
    --architectures "arm64" \
    --handler example.Handler::handleRequest \
    --timeout 30 \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://target/layer-java-function-1.0-SNAPSHOT.jar
```

4. 接下来，将层附加到函数。在以下 AWS CLI 命令中，将 `--layers` 参数替换为之前记下的层版本 ARN：

```
aws lambda update-function-configuration --function-name java_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:java-jackson-layer:1"
```

5. 最后，尝试使用以下 AWS CLI 命令调用函数：

```
aws lambda invoke --function-name java_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "driver": "Max Verstappen", "team": "Red Bull" }' response.json
```

应看到类似如下内容的输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

这表示函数能够使用 Jackson 依赖项来正确执行函数。您可以检查输出 `response.json` 文件是否包含正确的返回字符串：

```
"Max Verstappen is a driver for team Red Bull"
```

（可选）清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除 Lambda 层

1. 打开 Lambda 控制台的 Layers page（层页面）。
2. 选择您创建的层。
3. 选择删除，然后再次选择删除。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择 Delete（删除）。

使用 Lambda SnapStart 提高启动性能

适用于 Java 的 Lambda SnapStart 可以将延迟敏感型应用程序的启动性能提高多达 10 倍，而无需支付额外费用，通常也无需更改您的函数代码。导致启动延迟的最大因素（通常称为冷启动时间）是指 Lambda 在初始化函数上花费的时间，其中包括加载函数代码、启动运行时以及初始化函数代码。

通过 SnapStart，Lambda 可以在您发布函数版本时初始化您的函数。Lambda 会对已初始化的执行环境的内存和磁盘状态创建 Firecracker microVM 快照、加密该快照并对其进行缓存以实现低延迟访问。当您首次调用某个函数版本时，以及随着调用的纵向扩展，Lambda 会从缓存的快照恢复新的执行环境，而不是从头开始初始化，从而减少了启动延迟。

> **Important**
>
> 如果您的应用程序要求状态唯一，则必须评估您的函数代码并验证其能否在快照操作后快速恢复。有关更多信息，请参阅 使用 Lambda SnapStart 处理唯一性。

主题

支持的功能和限制

SnapStart 支持 Java 11 和更高版本的 Java 托管运行时系统。不支持其他托管运行时系统（例如 `nodejs20.x` 和 `python3.12`）、仅限操作系统的运行时系统 和容器映像。

SnapStart 不支持预置并发、Amazon Elastic File System（Amazon EFS）或大于 512MB 的短暂存储。

要使用 SnapStart，您可以使用 Lambda 控制台、AWS Command Line Interface（AWS CLI）、Lambda API、AWS SDK for Java、AWS CloudFormation、AWS Serverless Application Model（AWS SAM）和 AWS Cloud Development Kit (AWS CDK)。有关更多信息，请参阅 激活和管理 Lambda SnapStart。

> **Note**
>
> 您只能对已发布的函数版本以及与版本关联的别名使用 SnapStart。不能对函数的未发布版本（$LATEST）使用 SnapStart。

支持的区域

以下 AWS 区域 提供 SnapStart：

- 美国东部（弗吉尼亚州北部）
- 美国东部（俄亥俄州）
- 美国西部（加利福尼亚北部）
- 美国西部（俄勒冈州）
- 非洲（开普敦）
- 亚太地区（香港）
- 亚太地区（孟买）
- 亚太地区（海得拉巴）
- Asia Pacific (Tokyo)
- 亚太地区 (首尔)
- 亚太地区（大阪）
- 亚太地区（新加坡）
- 亚太地区（悉尼）

- 亚太地区（雅加达）
- 亚太地区（墨尔本）
- 加拿大（中部）
- 欧洲地区（斯德哥尔摩）
- 欧洲地区（法兰克福）
- 欧洲（苏黎世）
- 欧洲地区（爱尔兰）
- 欧洲地区（伦敦）
- Europe (Paris)
- 欧洲地区（米兰）
- 欧洲（西班牙）
- 中东（阿联酋）
- 中东（巴林）
- 南美洲（圣保罗）

兼容性注意事项

通过 SnapStart，Lambda 可以使用单个快照作为多个执行环境的初始状态。如果您的函数在初始化阶段使用以下任何内容，则可能需要在使用 SnapStart 之前进行一些更改：

独特性

如果您的初始化代码生成了唯一内容，而此内容包含在快照中，则跨执行环境重复使用该内容时，该内容可能不唯一。若要在使用 SnapStart 时保持唯一性，则必须在初始化后生成唯一的内容。其中包括唯一 ID、唯一密钥以及用于生成伪随机性的熵。要了解如何还原唯一性，请参阅 使用 Lambda SnapStart 处理唯一性。

网络连接

当 Lambda 从快照恢复您的函数后，无法保证您的函数在初始化阶段所建立连接的状态不变。验证您的网络连接状态，并在必要时重新建立连接。在大多数情况下，AWS 开发工具包建立的网络连接会自动恢复。关于其他连接，请查看最佳实践。

临时数据

有些函数会在初始化阶段下载或初始化暂存数据，例如临时凭证或缓存的时间戳。使用暂存数据之前，请在函数处理程序中刷新此数据，即使不使用 SnapStart 也是如此。

SnapStart 定价

SnapStart 不会产生额外成本。您需要根据对您的函数发出的请求数量、您的代码运行所用时间以及为您的函数配置的内存来支付相关费用。持续时间从代码开始运行开始计算，直到代码返回或以其他方式结束为止，并向上取整为最接近的 1 毫秒。

持续时间费用适用于在函数处理程序中运行的代码、在处理程序之外声明的初始化代码、运行时（JVM）加载所用时间以及在运行时挂钩中运行的任何代码。有关 Lambda 如何计算持续时间的更多信息，请参阅 监控 Lambda SnapStart。

对于配置了 SnapStart 的函数，Lambda 会定期回收执行环境并重新运行您的初始化代码。为了实现韧性，Lambda 会在多个区域中创建快照。每次 Lambda 在另一个区域重新运行您的初始化代码时都会收取费用。有关 Lambda 如何计算费用的更多信息，请参阅 AWS Lambda 定价。

比较 Lambda SnapStart 和预置并发

当函数纵向扩展时，Lambda SnapStart 和预置并发都可以减少冷启动和异常延迟。SnapStart 可帮助您将启动性能提高多达 10 倍，而无需支付额外费用。预置并发可使函数保持初始化状态，并做好准备在几十毫秒内做出响应。配置预置并发会让您的 AWS 账户 产生费用。如果您的应用程序有严格的冷启动延迟要求，请使用预置并发。您不能对同一个函数版本同时使用 SnapStart 和预置并发。

> **Note**
>
> SnapStart 在与大规模函数调用搭配使用时效果最佳。不经常调用的函数可能无法获得相同的性能改进。

其他资源

除了阅读本章中的其他主题外，我们还建议您尝试通过 AWS Lambda SnapStart 更快速入门研讨会并观看来自 AWS re:Invent 2022 的快速冷启动 Java 函数会议。

激活和管理 Lambda SnapStart

若要使用 SnapStart，请对新的或现有 Lambda 函数激活 SnapStart。然后，发布并调用一个函数版本。

主题

激活 SnapStart（控制台）

为函数激活 SnapStart

1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数的名称。
3. 选择 Configuration（配置），然后选择 General configuration（常规配置）。
4. 在 General configuration（常规配置）窗格上，选择 Edit（编辑）。
5. 在 Edit basic settings（编辑基本设置）页面上，为 SnapStart 选择 Published versions（已发布版本）。
6. 选择保存。
7. 发布函数版本。Lambda 会初始化您的代码，创建初始化执行环境的快照，然后缓存快照以实现低延迟访问。
8. 调用函数版本。

激活 SnapStart（AWS CLI）

为现有函数激活 SnapStart

1. 通过运行带有 --snap-start 选项的 update-function-configuration 命令来更新函数配置。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --snap-start ApplyOn=PublishedVersions
```

2. 使用 publish-version 发布函数版本。

```
aws lambda publish-version \
  --function-name my-function
```

3. 通过运行 get-function-configuration 命令并指定版本号，确认已为相应函数版本激活 SnapStart。以下示例指定版本 1。

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

如果其响应显示 OptimizationStatus 为 On 且 State（状态）为 Active，则表示 SnapStart 已激活，并且已为指定函数版本生成快照。

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
 },
 "State": "Active",
```

4. 通过运行 invoke 命令并指定版本来调用相应的函数版本。以下示例调用版本 1。

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

创建新的函数时激活 SnapStart

1. 通过运行带有 --snap-start 选项的 create-function 命令来创建函数。对于 --role，指定您的执行角色的 Amazon 资源名称（ARN）。

```
aws lambda create-function \
  --function-name my-function \
  --runtime "java21" \
  --zip-file fileb://my-function.zip \
  --handler my-function.handler \
  --role arn:aws:iam::111122223333:role/lambda-ex \
  --snap-start ApplyOn=PublishedVersions
```

2. 使用 publish-version 命令创建版本。

```
aws lambda publish-version \
  --function-name my-function
```

3. 通过运行 get-function-configuration 命令并指定版本号，确认已为相应函数版本激活 SnapStart。以下示例指定版本 1。

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

如果其响应显示 OptimizationStatus 为 On 且 State（状态）为 Active，则表示 SnapStart 已激活，并且已为指定函数版本生成快照。

```
"SnapStart": {
     "ApplyOn": "PublishedVersions",
     "OptimizationStatus": "On"
  },
  "State": "Active",
```

4. 通过运行 invoke 命令并指定版本来调用相应的函数版本。以下示例调用版本 1。

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

激活 SnapStart（API）

激活 SnapStart

1. 请执行以下操作之一：
   - 通过使用 CreateFunction API 操作并指定 SnapStart 参数，创建已激活 SnapStart 的新函数。
   - 通过使用 UpdateFunctionConfiguration 操作并指定 SnapStart 参数，为现有函数激活 SnapStart。
2. 使用 PublishVersion 操作发布函数版本。Lambda 会初始化您的代码，为已初始化的执行环境创建快照，然后缓存该快照以实现低延迟访问。
3. 通过使用 GetFunctionConfiguration 操作确认已为该函数版本激活 SnapStart。指定版本号以确认已为该版本激活 SnapStart。如果其响应显示 OptimizationStatus 为 On 且 State（状态）为 `Active`，则表示 SnapStart 已激活，并且已为指定函数版本生成快照。

```
"SnapStart": {
        "ApplyOn": "PublishedVersions",
        "OptimizationStatus": "On"
    },
    "State": "Active",
```

4. 使用 invoke 操作调用相应函数版本。

Lambda SnapStart 和函数状态

使用 SnapStart 时可能会出现以下函数状态。Lambda 定期回收执行环境并为使用 SnapStart 配置的函数重新运行初始化代码时，也可能出现这些状态。

- `Pending` – Lambda 正在初始化您的代码并为已初始化的执行环境创建快照。在该函数版本上运行的任何调用或其他 API 操作都将失败。
- `Active` – 快照创建已完成，您可以调用此函数。若要使用 SnapStart，您必须调用已发布的函数版本，而不是未发布的版本（$LATEST）。
- `Inactive` – 函数版本已 14 天未被调用。当函数版本变为 `Inactive` 时，Lambda 会删除快照。如果您在 14 天后调用函数版本，Lambda 会返回 `SnapStartNotReadyException` 响应并开始初始化新的快照。函数版本达到 `Active` 状态之后再调用该函数。`Inactive` 状态也可能在 Lambda 对执行环境进行定期回收时出现。在这种情况下，如果您的函数无法初始化，则该函数可能进入 `Inactive` 状态。
- `Failed` – Lambda 在运行初始化代码或创建快照时出现错误。

更新快照

Lambda 为每个已发布的函数版本创建快照。若要更新快照，请发布新的函数版本。Lambda 使用最新的运行时和安全补丁自动更新您的快照。

将 SnapStart 与 AWS SDK for Java 结合使用

为了从您的函数调用 AWS 开发工具包，Lambda 通过代入函数的执行角色来生成一组临时凭证。这些凭证在函数调用期间可用作环境变量。您无需直接在代码中为开发工具包提供凭证。默认情况下，凭证提供程序链会按顺序检查每个可以设置凭证的位置，然后选择第一个可用位置，通常是环境变量（`AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY` 和 `AWS_SESSION_TOKEN`）。

> **Note**
>
> SnapStart 激活后，Java 运行时会自动使用容器凭证（`AWS_CONTAINER_CREDENTIALS_FULL_URI` 和 `AWS_CONTAINER_AUTHORIZATION_TOKEN`），而非访问密钥环境变量。这样可以防止凭证在函数还原之前过期。

将 SnapStart 与 AWS CloudFormation、AWS SAM 和 AWS CDK 结合使用

- AWS CloudFormation：在您的模板中声明 SnapStart 实体。
- AWS Serverless Application Model (AWS SAM)：在您的模板中声明 SnapStart 属性。
- AWS Cloud Development Kit (AWS CDK)：使用 SnapStartProperty 类型。

删除快照

在以下情况下 Lambda 会删除快照：

- 您删除函数或函数版本。
- 您在 14 天内未调用该函数版本。14 天未被调用后，函数版本将转换为非活动状态。如果您在 14 天后调用函数版本，Lambda 会返回 `SnapStartNotReadyException` 响应并开始初始化新的快照。函数版本达到活动状态之后再调用该函数。

Lambda 根据《一般数据保护条例》（GDPR）删除与已删除的快照相关的所有资源。

使用 Lambda SnapStart 处理唯一性

调用在 SnapStart 函数上纵向扩展时，Lambda 使用单个初始化的快照来恢复多个执行环境。如果您的初始化代码生成了快照中包含的唯一内容，则跨执行环境重复使用该内容时，该内容可能不是唯一的。若要在使用 SnapStart 时保持唯一性，则必须在初始化后生成唯一的内容。这包括唯一的 ID、唯一的密钥和用于生成伪随机性的熵。

我们推荐使用以下最佳实践来帮助您保持代码的唯一性。Lambda 还提供了开源 SnapStart 扫描工具，以帮助检查假设唯一性的代码。如果您在初始化阶段生成唯一的数据，则可以使用运行时挂钩来还原唯一性。使用运行时挂钩，您可以在 Lambda 拍摄快照之前立即运行特定代码，也可以在 Lambda 从快照恢复函数后立即运行特定代码。

避免在初始化期间保存依赖唯一性的状态

在函数的初始化阶段，避免缓存旨在保持唯一性的数据，例如生成用于日志记录的唯一 ID。相反，我们建议您在函数处理程序中生成唯一的数据或使用运行时挂钩。

Example – 在函数处理程序中生成唯一的 ID

以下示例演示如何在函数处理程序中生成 UUID。

```
import java.util.UUID;
  public class Handler implements RequestHandler<String, String> {
    private static UUID uniqueSandboxId = null;
    @Override
    public String handleRequest(String event, Context context) {
      if (uniqueSandboxId == null)
        uniqueSandboxId = UUID.randomUUID();
      System.out.println("Unique Sandbox Id: " + uniqueSandboxId);
      return "Hello, World!";
    }
  }
```

使用加密安全的伪随机数生成器（CSPRNG）

如果您的应用程序依赖随机性，我们建议您使用加密安全的随机数生成器（CSPRNG）。Java 的 Lambda 托管式运行时包括两个内置的 CSPRNG（OpenSSL 1.0.2 和 `java.security.SecureRandom`），它们可以使用 SnapStart 自动保持随机性。始终从 `/dev/random` 或 `/dev/urandom` 获取随机数的软件也通过 SnapStart 保持随机性。

AWS 加密库会自动保持 SnapStart 的随机性，从下表中指定的最低版本开始。如果您将这些库与 Lambda 函数一起使用，请确保使用以下最低版本或更高版本：

Library	支持的最低版本（x86）	支持的最低版本（ARM）
AWS libcrypto（AWS-LC）	1.16.0	1.30.0
AWS libcrypto FIPS	2.0.13	2.0.13

如果通过以下库将上述加密库与 Lambda 函数打包为可传递依赖项，请确保使用以下最低版本或更高版本：

Library	支持的最低版本（x86）	支持的最低版本（ARM）
AWS SDK for Java 2.x	2.23.20	2.26.12
AWS Common Runtime for Java	0.29.8	0.29.25
Amazon Corretto 加密提供商	2.4.1	2.4.1
Amazon Corretto 加密提供商 FIPS	2.4.1	2.4.1

Example – java.security.SecureRandom

以下示例使用 `java.security.SecureRandom`，即使函数从快照中恢复，它也会生成唯一的数字序列。

```
import java.security.SecureRandom;
  public class Handler implements RequestHandler<String, String> {
    private static SecureRandom rng = new SecureRandom();
    @Override
    public String handleRequest(String event, Context context) {
      for (int i = 0; i < 10; i++) {
        System.out.println(rng.next());
      }
      return "Hello, World!";
    }
```

```
}
```

SnapStart 扫描工具

Lambda 提供扫描工具，以帮助您检查假设唯一性的代码。SnapStart 扫描工具是一个开源 SpotBugs 插件，该工具根据一组规则运行静态分析。该扫描工具有助于识别可能打破有关唯一性假设的潜在代码实施。有关安装说明和扫描工具执行的检查列表，请参阅 GitHub 上的 aws-lambda-snapstart-java-rules 存储库。

要详细了解如何使用 SnapStart 处理唯一性，请参阅 AWS 计算博客上的通过 AWS Lambda SnapStart 加快启动速度。

在 Lambda 函数快照之前或之后实现代码

在 Lambda 创建快照之前或 Lambda 还原快照后，使用运行时挂钩实施代码。运行时挂钩作为开源检查点协调还原（CRaC）项目的一部分提供。CRaC 正在开发开放 Java 开发工具包（OpenJDK）。有关如何将 CRaC 与参考应用程序结合使用的示例，请参阅 GitHub 上的 CRaC 存储库。CRaC 使用三个主要元素：

- `Resource` – 具有两种方法的接口，`beforeCheckpoint()` 和 `afterRestore()`。使用这些方法实施要在快照之前和还原之后运行的代码。
- `Context <R extends Resource>` – 若要接收检查点和还原的通知，`Resource` 必须注册到 `Context`。
- `Core` – 协调服务，该服务通过静态方法 `Core.getGlobalContext()` 提供默认全局 `Context`。

有关 `Context` 和 `Resource` 的更多信息，请参阅 CRaC 文档中的 Package org.crac（程序包 org.crac）。

使用以下步骤通过 org.crac 程序包实施运行时挂钩。Lambda 运行时包含自定义的 CRaC 上下文实施现，该实施会在检查点检查之前和还原后调用运行时挂钩。

步骤 1：更新构建配置

将 `org.crac` 依赖项添加到构建配置中。下面的示例使用了 Gradle。有关其他构建系统的示例，请参阅 Apache Maven 文档。

```
dependencies {
    compile group: 'com.amazonaws', name: 'aws-lambda-java-core', version: '1.2.1'
    # All other project dependecies go here:
    # ...
    # Then, add the org.crac dependency:
 implementation group: 'org.crac', name: 'crac', version: '1.4.0'
}
```

第 2 步：更新 Lambda 处理程序

Lambda 函数处理程序是函数代码中处理事件的方法。当调用函数时，Lambda 运行处理程序方法。您的函数会一直运行，直到处理程序返回响应、退出或超时。

有关更多信息，请参阅 定义采用 Java 的 Lambda 函数处理程序。

以下示例处理程序介绍如何在检查点检查（beforeCheckpoint()）之前和还原（afterRestore()）之后运行代码。该处理程序还将 Resource 注册到运行时管理的全局 Context 中。

> **Note**
>
> Lambda 创建快照时，初始化代码最多可以运行 15 分钟。时间限制为 130 秒或配置的函数超时（最大 900 秒），以较高者为准。您的 beforeCheckpoint() 运行时挂钩计入初始化代码时限。Lambda 还原快照时，必须加载运行时（JVM），并且 afterRestore() 运行时挂钩必须在超时限制（10 秒）内完成。否则，您将收到 SnapStartTimeoutException。

```
...
  import org.crac.Resource;
  import org.crac.Core;
  ...
public class CRaCDemo implements RequestStreamHandler, Resource {
    public CRaCDemo() {
      Core.getGlobalContext().register(this);
    }
    public String handleRequest(String name, Context context) throws IOException {
      System.out.println("Handler execution");
      return "Hello " + name;
    }
    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("Before checkpoint");
    }
    @Override
    public void afterRestore(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("After restore");
```

Context 仅对注册对象保持 WeakReference。如果 Resource 是垃圾回收，则不会运行运行时挂钩。您的代码必须保持对 Resource 的强引用，以保证运行时挂钩正常运行。

以下是两个应避免的模式示例：

Example – 没有强引用的对象

```
Core.getGlobalContext().register( new MyResource() );
```

Example – 匿名类的对象

```
Core.getGlobalContext().register( new Resource() {

   @Override
   public void afterRestore(Context<? extends Resource> context) throws Exception {
    // ...
   }

   @Override
   public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
    // ...
   }

} );
```

相反，保持强引用。在以下示例中，注册的资源不是垃圾回收的，运行时挂钩可以持续运行。

Example – 具有强引用的对象

```
Resource myResource = new MyResource(); // This reference must be maintained to prevent
 the registered resource from being garbage collected
Core.getGlobalContext().register( myResource );
```

监控 Lambda SnapStart

您可以使用 Amazon CloudWatch、AWS X-Ray 和 使用遥测 API 访问扩展的实时遥测数据 监控 Lambda SnapStart 函数。

> **Note**
>
> `AWS_LAMBDA_LOG_GROUP_NAME` 和 `AWS_LAMBDA_LOG_STREAM_NAME` 环境变量在 Lambda SnapStart 函数中不可用。

适用于 SnapStart 的 CloudWatch

SnapStart 函数的 CloudWatch 日志流格式存在一些差异：

- 初始化日志 – 创建新的执行环境时，`REPORT` 不会包含 `Init Duration` 字段。这是因为 Lambda 在您创建版本时而不是在函数调用期间初始化 SnapStart 函数。对于 SnapStart 函数，`Init Duration` 字段在 `INIT_REPORT` 记录中。此记录显示 Init 阶段 的持续时间详细信息，包括任何 `beforeCheckpoint` 运行时挂钩的持续时间。
- 调用日志 – 创建新的执行环境时，`REPORT` 会包括 `Restore Duration` 和 `Billed Restore Duration` 字段：
  - `Restore Duration`：Lambda 恢复快照、加载运行时（JVM）和运行任何 `afterRestore` 运行时挂钩所花费的时间。恢复快照的过程可能包含在 MicroVM 之外的活动上花费的时间。`Restore Duration` 中报告了此时间。
  - `Billed Restore Duration`：Lambda 加载运行时（JVM）和运行任何 `afterRestore` 挂钩所花费的时间。您无需为还原快照所花费的时间付费。

> **Note**
>
> 持续时间费用适用于在函数处理程序中运行的代码、在处理程序之外声明的初始化代码、运行时（JVM）加载所需的时间以及在运行时挂钩中运行的任何代码。有关更多信息，请参阅 SnapStart 定价。

冷启动持续时间为 `Restore Duration` + `Duration` 的总和。

以下示例为 Lambda Insights 查询，该查询返回 SnapStart 函数的延迟百分位数。有关 Lambda Insights 查询的更多信息，请参阅 使用查询排除函数故障的示例工作流程。

```
filter @type = "REPORT"
  | parse @log /\d+:\/aws\/lambda\/(?<function>.*)/
  | parse @message /Restore Duration: (?<restoreDuration>.*?) ms/
  | stats
count(*) as invocations,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 50) as p50,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 90) as p90,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99) as p99,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99.9) as p99.9
group by function, (ispresent(@initDuration) or ispresent(restoreDuration)) as
 coldstart
  | sort by coldstart desc
```

适用于 SnapStart 的 X-Ray 活动跟踪

您可以使用 X-Ray 来跟踪向 Lambda SnapStart 函数发送的请求。SnapStart 函数的 X-Ray 子分段存在一些差异：

- SnapStart 函数没有 `Initialization` 子分段。
- `Restore` 子分段会显示 Lambda 恢复快照、加载运行时系统（JVM）和运行任何 `afterRestore` 运行时系统挂钩所花费的时间。恢复快照的过程可能包含在 MicroVM 之外的活动上花费的时间。该时间在 `Restore` 子分段中报告。您无需为在 microVM 之外还原快照所花费的时间付费。

SnapStart 的遥测 API 事件

Lambda 将以下 SnapStart 事件发送到 遥测 API：

- `platform.restoreStart` – 显示 `Restore` 阶段的开始时间。
- `platform.restoreRuntimeDone` – 显示 `Restore` 阶段是否成功。当运行时发送 `restore/next` 运行时 API 请求时，Lambda 会发送此消息。存在三种可能的状态：成功、失败和超时。
- `platform.restoreReport` – 显示 `Restore` 阶段的持续时间以及该阶段已计费的毫秒数。

Amazon API Gateway 和函数 URL 指标

如果您使用 API Gateway 创建 Web API，则可以使用 IntegrationLatency 指标来测量端到端延迟（API Gateway 将请求转发到后端和从后端收到响应之间的时间）。

如果使用 Lambda 函数 URL，则可以使用 UrlRequestLatency 指标来测量端到端延迟（函数 URL 收到请求和函数 URL 返回响应之间的时间）。

Lambda SnapStart 的安全模型

Lambda SnapStart 支持静态加密。Lambda 使用 AWS KMS key 加密快照。默认情况下，Lambda 使用 AWS 托管式密钥。如果此默认行为适合您的工作流，那么您无需设置任何其他内容。否则，您可以使用 create-function 中的 `--kms-key-arn` 选项或 update-function-configuration 命令来提供 AWS KMS 客户管理型密钥。这样做可能是为了控制 KMS 密钥的轮换，或者是为了满足组织管理 KMS 密钥的要求。客户托管式密钥产生标准 AWS KMS 费用。有关更多信息，请参阅AWS Key Management Service 定价。

删除 SnapStart 函数或函数版本时，对该函数或函数版本的所有 `Invoke` 请求将失败。Lambda 会自动删除 14 天未被调用的快照。Lambda 根据《一般数据保护条例》（GDPR）删除与已删除的快照相关的所有资源。

最大限度地提高 Lambda SnapStart 的性能

主题

性能优化

> **Note**
>
> SnapStart 在与大规模函数调用搭配使用时效果最佳。不经常调用的函数可能无法获得相同的性能改进。

为了最大限度地发挥 SnapStart 的优势，我们建议您在初始化代码中预加载导致启动延迟的类，而不是在函数处理程序中加载。这会将与大量类加载相关的延迟移出调用路径，从而优化了 SnapStart 的启动性能。

如果您在初始化期间无法预加载类，那么我们建议您使用虚拟调用预加载类。为此，请更新函数处理程序代码，如 AWS Labs GitHub 存储库上宠物商店函数的以下示例所示。

```
private static SpringLambdaContainerHandler<AwsProxyRequest, AwsProxyResponse> handler;
  static {
      try {
          handler =
 SpringLambdaContainerHandler.getAwsProxyHandler(PetStoreSpringAppConfig.class);

          // Use the onStartup method of the handler to register the custom filter
          handler.onStartup(servletContext -> {
              FilterRegistration.Dynamic registration =
 servletContext.addFilter("CognitoIdentityFilter", CognitoIdentityFilter.class);
              registration.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST),
 false, "/*");
          });

          // Send a fake Amazon API Gateway request to the handler to load classes
 ahead of time
          ApiGatewayRequestIdentity identity = new ApiGatewayRequestIdentity();
          identity.setApiKey("foo");
          identity.setAccountId("foo");
```

```
            identity.setAccessKey("foo");

            AwsProxyRequestContext reqCtx = new AwsProxyRequestContext();
            reqCtx.setPath("/pets");
            reqCtx.setStage("default");
            reqCtx.setAuthorizer(null);
            reqCtx.setIdentity(identity);

            AwsProxyRequest req = new AwsProxyRequest();
            req.setHttpMethod("GET");
            req.setPath("/pets");
            req.setBody("");
            req.setRequestContext(reqCtx);

            Context ctx = new TestContext();
            handler.proxy(req, ctx);


        } catch (ContainerInitializationException e) {
            // if we fail here. We re-throw the exception to force another cold start
            e.printStackTrace();
            throw new RuntimeException("Could not initialize Spring framework", e);
        }
    }
```

联网最佳实践

当 Lambda 从快照恢复您的函数时，无法保证您的函数在初始化阶段建立的连接的状态。在大多数情况下，AWS 开发工具包建立的网络连接会自动恢复。对于其他连接，我们建议您遵循以下最佳实践。

重新建立网络连接

函数从快照恢复时，请务必重新建立网络连接。我们建议您在函数处理程序中重新建立网络连接。或者，您可以使用 `afterRestore` 运行时挂钩。

不要使用主机名作为唯一的执行环境标识符

我们建议不要使用 `hostname` 将执行环境标识为应用程序中的唯一节点或容器。利用 SnapStart，使用单个快照作为多个执行环境的初始状态，并且所有执行环境都为 `InetAddress.getLocalHost()` 返回相同的 `hostname` 值。对于需要唯一执行环境标识或 `hostname` 值的应用程序，我们建议您在函数处理程序中生成唯一的 ID。或者，使用 `afterRestore` 运行时挂接生成唯一的 ID，然后使用该唯一 ID 作为执行环境的标识符。

避免将连接绑定到固定源端口

我们建议您避免将网络连接绑定到固定源端口。函数从快照恢复时，会重新建立连接，绑定到固定源端口的网络连接可能会失败。

避免使用 Java DNS 缓存

Lambda 函数已经缓存了 DNS 响应。如果您将另一个 DNS 缓存与 SnapStart 结合使用，则函数从快照恢复时可能会出现连接超时。

`java.util.logging.Logger` 类可以间接启用 JVM DNS 缓存。要覆盖默认设置，请在初始化 `logger` 之前将 networkaddress.cache.ttl 设置为 0。例如：

```
public class MyHandler {
  // first set TTL property
  static{
   java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
  }
 // then instantiate logger
  var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
}
```

为了防止 Java 11 运行时出现 `UnknownHostException` 故障，建议将 `networkaddress.cache.negative.ttl` 设置为 0。在 Java 17 及更高版本的运行时中，不必执行此步骤。您可以使用 `AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0` 环境变量为 Lambda 函数设置此属性。

禁用 JVM DNS 缓存并不能禁用 Lambda 的托管式 DNS 缓存。

排查 Lambda Java 函数的 SnapStart 错误

本页内容旨在解决使用 Lambda SnapStart 时出现的常见问题，包括快照创建错误、超时错误和内部服务错误。

SnapStartNotReadyException

错误：调用 Invoke20150331 操作时出现错误 (SnapStartNotReadyException)：Lambda 正在初始化函数。只要函数状态变为 ACTIVE，即可加以调用。

常见原因

在尝试调用处于 `Inactive` 状态的函数版本时，即会发生此错误。如果 14 天未调用函数版本或者 Lambda 定期回收执行环境，函数版本将变为 `Inactive`。

解决方案

函数版本达到 `Active` 状态之后再调用该函数。

SnapStartTimeoutException

问题：在尝试调用 SnapStart 函数版本时收到 `SnapStartTimeoutException`。

常见原因

在还原阶段，Lambda 会恢复 Java 运行时并运行任何 `afterRestore()` 运行时钩子。如果 `afterRestore()` 运行时钩子的运行时间超过 10 秒，则 `Restore` 阶段会超时，并且在尝试调用该函数时会出现错误。网络连接和凭证问题也可能导致 `Restore` 阶段超时。

解决方案

查看函数的 CloudWatch 日志，了解在还原阶段发生的超时错误。确保所有 `afterRestore()` 钩子在不到 10 秒的时间内完成。

Example CloudWatch 日志

```
{ "cause": "Lambda couldn't restore the snapshot within the timeout limit. (Service:
 Lambda, Status Code: 408, Request ID: 11a222c3-410f-427c-ab22-931d6bcbf4f2)", "error":
 "Lambda.SnapStartTimeoutException"}
```

500 内部服务错误

错误：Lambda 无法创建新快照，因为已达到并发快照创建上限。

常见原因

500 错误是 Lambda 服务自身的内部错误，而非函数或代码的问题。这些错误通常间歇性发生。

解决方案

尝试再次发布该函数版本。

401 未经授权

错误：会话令牌或标头密钥错误

常见原因

在 Lambda SnapStart 中使用 AWS Systems Manager Parameter Store 和 AWS Secrets Manager 扩展时会发生此错误。

解决方案

AWS Systems Manager Parameter Store 和 AWS Secrets Manager 与 SnapStart 不兼容。该扩展会在函数初始化期间生成用于与 AWS Secrets Manager 通信的凭证，但该凭证在与 SnapStart 一起使用时会导致凭证过期错误。

UnknownHostException

错误：无法执行 HTTP 请求：`abc.us-east-1.amazonaws.com` 的证书与任何主题备用名称都不匹配。

常见原因

Lambda 函数已经缓存了 DNS 响应。如果您将另一个 DNS 缓存与 SnapStart 结合使用，则函数从快照恢复时可能会出现连接超时。

解决方案

为了防止 Java 11 运行时出现 `UnknownHostException` 故障，建议将 `networkaddress.cache.negative.ttl` 设置为 0。在 Java 17 及更高版本的运行时中，不必执行此步骤。您可以使用 `AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0` 环境变量为 Lambda 函数设置此属性。

快照创建失败

错误：AWS Lambda 无法调用 SnapStart 函数。如果此错误仍然存在，请检查函数的 CloudWatch 日志，看是否存在初始化错误。

解决方案

查看函数的 Amazon CloudWatch 日志，了解 `beforeCheckpoint()` 运行时钩子是否超时。有时候，发布新的函数版本也可以解决该问题。

快照创建延迟

问题：在发布新的函数版本时，该函数会长时间保持 `Pending` 状态。

常见原因

Lambda 创建快照时，初始化代码最多可以运行 15 分钟。时间限制为 130 秒或配置的函数超时（最大 900 秒），以较高者为准。

如果函数已附加到 VPC，Lambda 可能还需要在函数变为 `Active` 之前创建网络接口。如果在函数处于 `Pending` 状态时尝试调用函数版本，可能会出现 409 `ResourceConflictException`。如果使用 Amazon API Gateway 端点调用该函数，API Gateway 中可能会出现 500 错误。

解决方案

至少等待 15 分钟让函数版本初始化，然后再调用函数。

自定义 Lambda Java 函数的序列化

Lambda Java 托管式运行时支持 JSON 事件的自定义序列化。自定义序列化可以简化代码并有可能提高性能。

主题

何时使用自定义序列化

调用 Lambda 函数时，需要将输入事件数据反序列化为 Java 对象，并且需要将函数的输出序列化回可以作为函数响应返回的格式。Lambda Java 托管式运行时提供默认的序列化和反序列化功能，非常适合处理来自各种 AWS 服务的事件有效载荷，例如 Amazon API Gateway 和 Amazon Simple Queue Service（Amazon SQS）。要在函数中使用这些服务集成事件，请向项目中添加 aws-java-lambda-events 依赖项。此 AWS 库包含表示这些服务集成事件的 Java 对象。

您也可以使用自己的对象来表示传递给 Lambda 函数的事件 JSON。托管式运行时会尝试将 JSON 序列化为对象的新实例，且该新实例具有其默认行为。如果默认序列化器无用例所需的行为，请使用自定义序列化。

例如，假设函数处理程序需要 `Vehicle` 类作为输入，并且此类具有以下结构：

```
public class Vehicle {
    private String vehicleType;
    private long vehicleId;
}
```

但是，JSON 事件有效载荷如下所示：

```
{
    "vehicle-type": "car",
    "vehicleID": 123
}
```

在这种情况下，托管式运行时中的默认序列化需要 JSON 属性名称与驼峰式大小写的 Java 类属性名称（`vehicleType`、`vehicleId`）相匹配。JSON 事件中的属性名称并非为驼峰式大小写（`vehicle-type`、`vehicleID`），因此必须使用自定义序列化。

实现自定义序列化

使用服务提供程序接口加载您选择的序列化器，而非托管式运行时的默认序列化逻辑。您可以使用标准 `RequestHandler` 接口将 JSON 事件有效载荷直接序列化为 Java 对象。

要在 Lambda Java 函数中使用自定义序列化

1. 添加 aws-lambda-java-core（作为依赖项）。此库包括 CustomPojoSerializer 接口，以及其他用于在 Lambda 中使用 Java 的接口定义。
2. 在项目的 `src/main/META-INF/services/` 目录中创建名为 `com.amazonaws.services.lambda.runtime.CustomPojoSerializer` 的文件。
3. 在此文件中，指定自定义序列化器实现（该实现必须实现 `CustomPojoSerializer` 接口）的完全限定名称。例如：

   ```
   com.mycompany.vehicles.CustomLambdaSerialzer
   ```

4. 实现 `CustomPojoSerializer` 接口以提供自定义序列化逻辑。
5. 在 Lambda 函数中使用标准 `RequestHandler` 接口。托管式运行时将使用自定义序列化程序。

有关如何使用 fastJson、Gson、Moshi 和 jackson-jr 等常用库实现自定义序列化的更多示例，请参阅《AWS GitHub repository》中的 custom-serialization 示例。

测试自定义序列化

测试函数，确保序列化和反序列化逻辑按预期运行。您可以使用 AWS Serverless Application Model 命令行接口（AWS SAMCLI）来模拟 Lambda 有效载荷的调用。在引入自定义序列化器时，此举可以帮助您快速测试和迭代函数。

1. 使用要调用函数的 JSON 事件有效载荷创建文件，然后调用 AWS SAM CLI。
2. 运行 sam local invoke 命令，在本地调用函数。例如：

   ```
   sam local invoke -e src/test/resources/event.json
   ```

有关更多信息，请参阅 Locally invoke Lambda functions with AWS SAM。

自定义 Lambda 函数的 Java 运行时启动行为

本页将介绍特定于 AWS Lambda 中 Java 函数的设置。您可以使用这些设置来自定义 Java 运行时系统启动行为。这样可以减少总体函数延迟并提高总体函数性能，而无需修改任何代码。

Sections

了解 JAVA_TOOL_OPTIONS 环境变量

在 Java 中，Lambda 支持 `JAVA_TOOL_OPTIONS` 环境变量以在 Lambda 中设置其他命令行变量。您可以通过多种方式使用此环境变量，例如自定义分层编译设置。下一个示例演示了如何针对此使用案例使用 `JAVA_TOOL_OPTIONS` 环境变量。

示例：自定义分层编译设置

分层编译是 Java 虚拟机（JVM）的一项功能。您可以使用特定的分层编译设置来充分利用 JVM 的即时（JIT）编译器。通常，C1 编译器经过优化，可缩短启动时间。C2 编译器经过优化以获得最佳整体性能，但它也会占用更多内存，并且需要更长的时间才能实现。

分层编译有 5 个不同级别。在级别 0 上，JVM 解释 Java 字节代码。在级别 4 上，JVM 使用 C2 编译器来分析在应用程序启动期间收集的分析数据。随着时间的推移，它会监控代码使用情况以确定最佳优化。

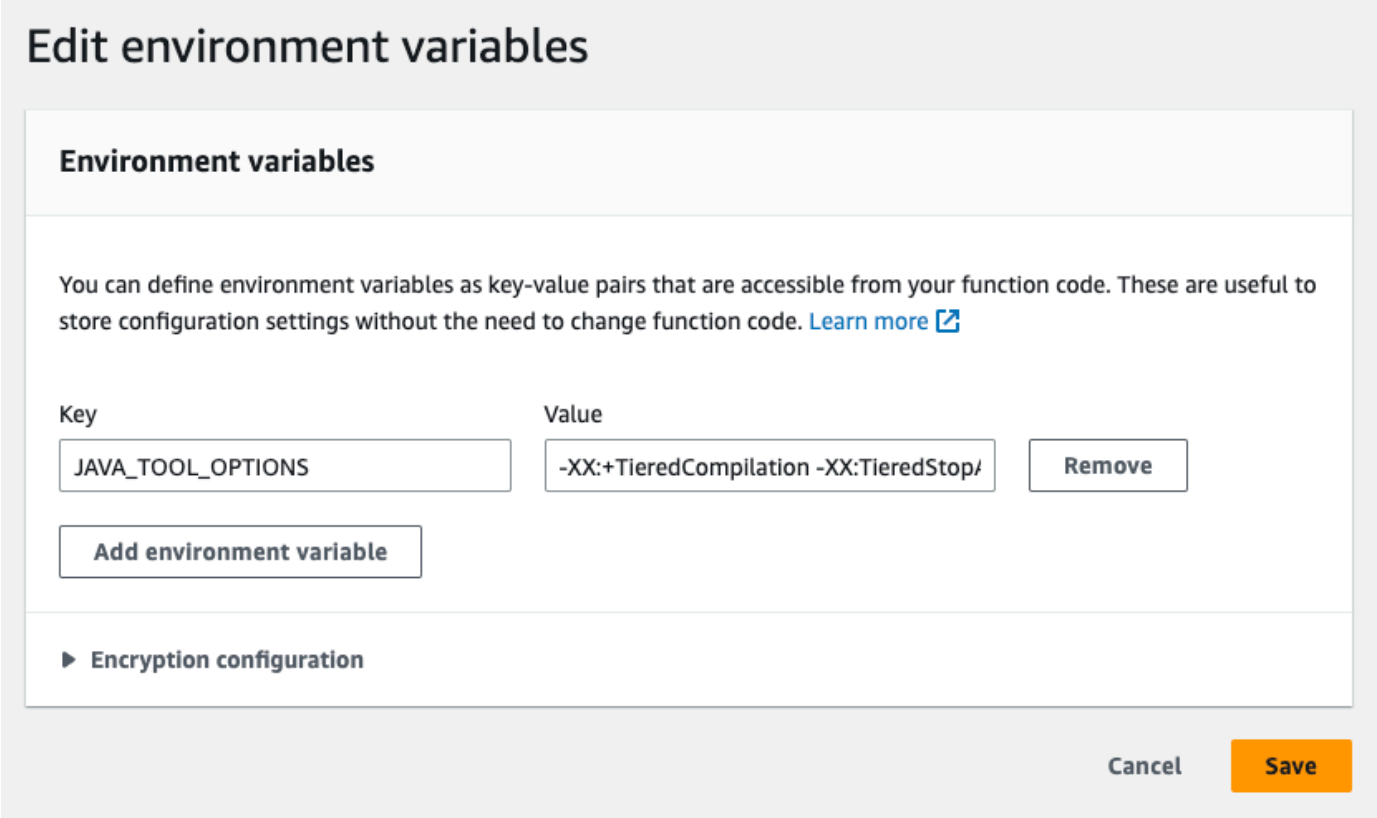
自定义分层编译级别可以帮助您减少 Java 函数冷启动延迟。例如，将分层编译级别设置为 1 以让 JVM 使用 C1 编译器。该编译器可以快速生成优化原生代码，但它不生成任何分析数据，也从不使用 C2 编译器。

在 Java 17 运行时系统中，用于分层编译的 JVM 标志默认设置为在级别 1 停止。对于 Java 11 及以下的运行时系统，可以通过执行如下步骤将分层编译级别设置为 1：

自定义分层编译设置（控制台）

1. 在 Lambda 控制台中打开函数页面。
2. 选择要为其自定义分层编译的 Java 函数。
3. 选择配置选项卡，然后从左侧菜单中选择环境变量。
4. 选择编辑。
5. 选择 Add environment variable (添加环境变量)。

6. 对于键，输入 JAVA_TOOL_OPTIONS。对于值，输入 -XX:+TieredCompilation -XX:TieredStopAtLevel=1。



7. 选择保存。

> **Note**
>
> 您也可以使用 Lambda SnapStart 来缓解冷启动问题。SnapStart 使用执行环境的缓存快照来显著提高启动性能。有关 SnapStart 的功能、限制和支持的区域的更多信息，请参阅 使用 Lambda SnapStart 提高启动性能。

示例：使用 JAVA_TOOL_OPIONS 自定义 GC 行为

Java 11 运行时系统使用串行垃圾收集器（GC）实现垃圾回收。默认情况下，Java 17 运行时系统也使用串行 GC。不过，在 Java 17 中，您也可以使用 JAVA_TOOL_OPTIONS 环境变量来更改默认 GC。您可以在 Parallel GC 和 Shenandoah GC 之间进行选择。

例如，如果工作负载会使用更多内存和多个 CPU，则考虑使用 Parallel GC 来获得更好的性能。为此，您可以将以下内容附加到 JAVA_TOOL_OPTIONS 环境变量的值中：

```
-XX:+UseParallelGC
```

使用 Lambda 上下文对象检索 Java 函数信息

当 Lambda 运行您的函数时，它会将上下文对象传递到处理程序。此对象提供的方法和属性包含有关调用、函数和执行环境的信息。

上下文方法

- `getRemainingTimeInMillis()` – 返回执行超时前剩余的毫秒数。
- `getFunctionName()` – 返回 Lambda 函数的名称。
- `getFunctionVersion()` – 返回函数的版本。
- `getInvokedFunctionArn()` – 返回用于调用函数的 Amazon 资源名称（ARN）。表明调用者是否指定了版本号或别名。
- `getMemoryLimitInMB()` – 返回为函数分配的内存量。
- `getAwsRequestId()` – 返回调用请求的标识符。
- `getLogGroupName()` – 返回函数的日志组。
- `getLogStreamName()` – 返回函数实例的日志流。
- `getIdentity()` –（移动应用程序）授权请求的 Amazon Cognito 身份的相关信息。
- `getClientContext()` –（移动应用程序）返回客户端应用程序提供给 Lambda 的客户端上下文。
- `getLogger()` – 返回函数的记录器对象。

以下示例显示一个使用上下文对象访问 Lambda 记录器的函数。

Example Handler.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.Map;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, Void>{

  @Override
  public Void handleRequest(Map<String,String> event, Context context)
  {
```

```
    LambdaLogger logger = context.getLogger();
    logger.log("EVENT TYPE: " + event.getClass());
    return null;
  }
}
```

该函数在返回 null 之前记录传入事件的类类型。

Example 日志输出

```
EVENT TYPE: class java.util.LinkedHashMap
```

上下文对象的接口在 aws-lambda-java-core 库中可用。您可以实现此接口来创建用于测试的上下文类。以下示例显示一个上下文类，该类返回大多数属性的虚拟值和一个有效的测试记录器。

Example src/test/java/example/TestContext.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.CognitoIdentity;
import com.amazonaws.services.lambda.runtime.ClientContext;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class TestContext implements Context{

  public TestContext() {}
  public String getAwsRequestId(){
    return new String("495b12a8-xmpl-4eca-8168-160484189f99");
  }
  public String getLogGroupName(){
    return new String("/aws/lambda/my-function");
  }
  public String getLogStreamName(){
    return new String("2020/02/26/[$LATEST]704f8dxmpla04097b9134246b8438f1a");
  }
  public String getFunctionName(){
    return new String("my-function");
  }
  public String getFunctionVersion(){
    return new String("$LATEST");
  }
  public String getInvokedFunctionArn(){
```

```
    return new String("arn:aws:lambda:us-east-2:123456789012:function:my-function");
  }
  public CognitoIdentity getIdentity(){
    return null;
  }
  public ClientContext getClientContext(){
    return null;
  }
  public int getRemainingTimeInMillis(){
    return 300000;
  }
  public int getMemoryLimitInMB(){
    return 512;
  }
  public LambdaLogger getLogger(){
    return new TestLogger();
  }

}
```

有关日志记录的更多信息，请参阅Java Lambda 函数日志记录和监控。

示例应用程序中的上下文

本指南的 GitHub 存储库包括演示如何使用上下文对象的示例应用程序。每个示例应用程序都包含用于轻松部署和清理的脚本、一个 AWS Serverless Application Model (AWS SAM) 模板和支持资源。

Java 中的 Lambda 应用程序示例

- java17-examples：这是一种 Java 函数，演示如何使用 Java 记录来表示输入事件数据对象。
- java-basic – 具有单元测试和变量日志记录配置的最小 Java 函数的集合。
- java-events – Java 函数的集合，其中包含用于处理来自 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis 等各种服务的事件的框架代码。这些函数使用最新版本的 aws-lambda-events 库（3.0.0 及更新版本）。这些示例不需要 AWS 开发工具包作为依赖项。
- s3-java – 此 Java 函数可处理来自 Amazon S3 的通知事件，并使用 Java 类库（JCL）从上传的图像文件创建缩略图。
- 自定义序列化 –如何使用 fastJson、Gson、Moshi 和 jackson-jr 等常用库实现自定义序列化的示例。
- 使用 API Gateway 调用 Lambda 函数 – Java 函数，用于扫描包含员工信息的 Amazon DynamoDB 表。然后，该函数使用 Amazon Simple Notification Service 向员工发送短信，祝贺他们工作周年纪念日快乐。此示例使用 API Gateway 调用函数。

Java Lambda 函数日志记录和监控

AWS Lambda 将自动监控 Lambda 函数并将日志条目发送到 Amazon CloudWatch。您的 Lambda 函数带有一个 CloudWatch Logs 日志组以及函数的每个实例的日志流。Lambda 运行时系统环境会将每次调用的详细信息以及函数代码的其他输出发送到该日志流。有关 CloudWatch Logs 的更多信息，请参阅将 CloudWatch Logs 日志与 Lambda 结合使用。

要从函数代码输出日志，您可以使用 java.lang.System 的方法，或使用写入到 stdout 或 stderr 的任何日志记录模块。

Sections

创建返回日志的函数

要从函数代码输出日志，您可以使用 java.lang.System 的方法，或使用写入到 `stdout` 或 `stderr` 的任何日志记录模块。aws-lambda-java-core 库提供一个名为 `LambdaLogger` 的记录器类，您可以从上下文对象访问该类。记录器类支持多行日志。

以下示例使用上下文对象提供的 `LambdaLogger` 记录器。

Example Handler.java

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Object, String>{
  Gson gson = new GsonBuilder().setPrettyPrinting().create();
  @Override
  public String handleRequest(Object event, Context context)
```

```
  {
    LambdaLogger logger = context.getLogger();
    String response = new String("SUCCESS");
    // log execution details
    logger.log("ENVIRONMENT VARIABLES: " + gson.toJson(System.getenv()));
    logger.log("CONTEXT: " + gson.toJson(context));
    // process event
    logger.log("EVENT: " + gson.toJson(event));
    return response;
  }
}
```

Example 日志格式

```
START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
ENVIRONMENT VARIABLES:
{
    "_HANDLER": "example.Handler",
    "AWS_EXECUTION_ENV": "AWS_Lambda_java8",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "512",
    ...
}
CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
EVENT:
{
  "records": [
    {
      "messageId": "19dd0b57-xmpl-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
       ...
    }
  ]
}
END RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0
REPORT RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Duration: 198.50 ms Billed
 Duration: 200 ms Memory Size: 512 MB Max Memory Used: 90 MB Init Duration: 524.75 ms
```

Java 运行时记录每次调用的 START、END 和 REPORT 行。报告行提供了以下详细信息：

REPORT 行数据字段

- RequestId – 调用的唯一请求 ID。
- Duration（持续时间）– 函数的处理程序方法处理事件所花费的时间。
- Billed Duration（计费持续时间）– 针对调用计费的时间量。
- Memory Size（内存大小）– 分配给函数的内存量。
- Max Memory Used（最大内存使用量）– 函数使用的内存量。如果调用共享执行环境，Lambda 会报告所有调用使用的最大内存。此行为可能会导致报告值高于预期。
- Init Duration（初始持续时间）– 对于提供的第一个请求，为运行时在处理程序方法外部加载函数和运行代码所花费的时间。
- XRAY TraceId – 对于追踪的请求，为 AWS X-Ray 追踪 ID。
- SegmentId – 对于追踪的请求，为 X-Ray 分段 ID。
- Sampled（采样）– 对于追踪的请求，为采样结果。

在 Java 中使用 Lambda 高级日志记录控件

为了更好地控制捕获、处理和使用函数日志的方式，您可以为支持的 Java 运行时系统配置以下日志记录选项：

- 日志格式 - 为函数日志选择纯文本或结构化的 JSON 格式
- 日志级别 - 对于 JSON 格式的日志，选择 Lambda 发送到 CloudWatch 的日志的详细信息级别，例如 ERROR、DEBUG 或 INFO
- 日志组 - 选择您的函数发送日志的目标 CloudWatch 日志组

有关这些日志记录选项的更多信息以及如何通过配置来使用函数的说明，请参阅 the section called “配置函数日志”。

要在 Java Lambda 函数中使用日志格式和日志级别选项，请参阅以下各节中的指南。

在 Java 中使用结构化的 JSON 日志格式

如果您为函数的日志格式选择 JSON，Lambda 将使用 `LambdaLogger` 类将日志输出作为结构化的 JSON 发送到 CloudWatch。每个 JSON 日志对象包含至少四个键值对和以下键：

- `"timestamp"` - 生成日志消息的时间

- `"level"` - 分配给消息的日志级别
- `"message"` - 日志消息的内容
- `"AWSrequestId"` - 函数调用的唯一请求 ID

根据您使用的日志记录方法，以 JSON 格式捕获的函数日志输出还可能包含其他键值对。

要为使用 `LambdaLogger` 记录器创建的日志分配级别，您需要在日志记录命令中提供 `LogLevel` 参数，如以下示例所示。

Example Java 日志记录代码

```
LambdaLogger logger = context.getLogger();
logger.log("This is a debug log", LogLevel.DEBUG);
```

此示例代码输出的日志将在 CloudWatch Logs 中被捕获，如下所示：

Example JSON 日志记录

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"DEBUG",
    "message":"This is a debug log",
    "AWSrequestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

如果您没有为日志输出分配级别，Lambda 将自动为其分配 INFO 级别。

如果您的代码已经使用其他日志记录库来生成 JSON 结构化日志，则无需进行任何更改。Lambda 不会对任何已采用 JSON 编码的日志进行双重编码。即使您将函数配置为使用 JSON 日志格式，您的日志输出也会以您定义的 JSON 结构显示在 CloudWatch 中。

在 Java 中使用日志级别筛选

为了让 AWS Lambda 根据日志级别筛选应用程序日志，您的函数必须使用 JSON 格式的日志。您可以通过两种方式实现这一点：

- 使用标准 `LambdaLogger` 创建日志输出，并将您的函数配置为使用 JSON 日志格式。然后，Lambda 使用 the section called “在 Java 中使用结构化的 JSON 日志格式” 中所述的 JSON 对象中的“级别”键值对筛选您的日志输出。要了解如何配置函数的日志格式，请参阅 the section called “配置函数日志”。

- 使用其他日志记录库或方法在代码中创建 JSON 结构化日志，其中包含定义日志输出级别的“级别”键值对。您可以使用任何可将 JSON 日志写入 `stdout` 或 `stderr` 的日志记录库。例如，您可以使用 Powertools for AWS Lambda 或 Log4j2 软件包通过代码生成 JSON 结构化日志输出。要了解更多信息，请参阅 the section called “将 Powertools for AWS Lambda（Java）和 AWS SAM 用于结构化日志记录” 和 the section called “使用 Log4j2 和 SLF4J 实现高级日志记录”。

将函数配置为使用日志级别筛选时，您必须从以下选项中选择希望 Lambda 发送到 CloudWatch Logs 的日志级别：

日志级别	标准使用情况
TRACE（最详细）	用于跟踪代码执行路径的最精细信息
调试	系统调试的详细信息
信息	记录函数正常运行情况的消息
警告	有关潜在错误的消息，如果不加以解决，这些错误可能会导致意外行为
错误	有关会阻碍代码按预期执行的问题的消息
FATAL（最简略）	有关导致应用程序停止运行的严重错误的消息

要让 Lambda 筛选函数的日志，还必须在 JSON 日志输出中包含一个 `"timestamp"` 键值对。必须以有效的 RFC 3339 时间戳格式指定时间。如果您未提供有效的时间戳，Lambda 将为日志分配 INFO 级别并为您添加时间戳。

Lambda 仅将选定级别及更低级别的系统日志发送到 CloudWatch。例如，如果您将日志级别配置为 WARN，Lambda 将发送与 WARN、ERROR 和 FATAL 级别相对应的日志。

使用 Log4j2 和 SLF4J 实现高级日志记录

> **Note**
>
> AWS Lambda 的托管式运行时或基本容器映像中不包括 Log4j2。因此，这些不受 CVE-2021-44228、CVE-2021-45046 和 CVE-2021-45105 中描述的问题影响。

对于客户的函数包含受影响的 Log4j2 版本的情况，我们对 Lambda Java 托管式运行时和基本容器映像应用了更改，以帮助缓解 CVE-2021-44228、CVE-2021-45046 和 CVE-2021-45105 中描述的问题。由于这一更改，使用 Log4J2 的客户可能会看到额外的日志条目，类似于“`Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)`”。Log4J2 输出中引用 jndi 映射器的任何日志字符串都将替换为“`Patched JndiLookup::lookup()`”。
无论是否有这一更改，我们强烈建议其函数中包括 Log4j2 的所有客户更新到最新版本。具体来说，在函数中使用 aws-lambda-java-log4j2 库的客户应更新到版本 1.5.0（或更高版本），然后重新部署函数。此版本将底层 Log4j2 实用程序依赖项更新为版本 2.17.0（或更高版本）。更新后的 aws-lambda-java-log4j2 二进制文件可在 Maven 存储库中获取，其源代码可在 Github 中获取。
最后，请注意，在任何情况下都不应使用任何与 aws-lambda-java-log4j（v1.0.0 或 1.0.1）相关的库。这些库与 log4j 的 1.x 版本相关，该版本已于 2015 年终止使用。这些库不受支持、未维护、未修补，并且存在已知的安全漏洞。

要自定义日志输出、在单元测试期间支持日志记录以及记录 AWS 开发工具包调用，请将 Apache Log4j2 与 SLF4J 结合使用。Log4j 是 Java 程序的日志库，这些程序使您能够配置日志级别和使用 Appender 库。SLF4J 是一个 Facade 库，可让您更改您使用的库，而不更改函数代码。

要将请求 ID 添加到函数的日志中，请使用 aws-lambda-java-log4j2 库中的 Appender。

Example src/main/resources/log4j2.xml – Appender 配置

```
<Configuration>
  <Appenders>
    <Lambda name="Lambda" format="${env:AWS_LAMBDA_LOG_FORMAT:-TEXT}">
       <LambdaTextFormat>
         <PatternLayout>
             <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1} - %m%n </
pattern>
         </PatternLayout>
       </LambdaTextFormat>
       <LambdaJSONFormat>
         <JsonTemplateLayout eventTemplateUri="classpath:LambdaLayout.json" />
       </LambdaJSONFormat>
     </Lambda>
   </Appenders>
   <Loggers>
     <Root level="${env:AWS_LAMBDA_LOG_LEVEL:-INFO}">
```

```
      <AppenderRef ref="Lambda"/>
    </Root>
    <Logger name="software.amazon.awssdk" level="WARN" />
    <Logger name="software.amazon.awssdk.request" level="DEBUG" />
  </Loggers>
</Configuration>
```

您可以通过在 <LambdaTextFormat> 和 <LambdaJSONFormat> 标签下指定布局来决定如何为纯文本或 JSON 输出配置 Log4j2 日志。

在此示例中，使用文本模式，每行都会在前面加上日期、时间、请求 ID、日志级别和类名。在 JSON 模式下，<JsonTemplateLayout> 要结合与 aws-lambda-java-log4j2 库一起提供的配置使用。

SLF4J 是一个用于在 Java 代码中进行日志记录的 Facade 库。在函数代码中，您可以使用 SLF4J 记录器工厂，通过适用于日志级别（如 info() 和 warn()）的方法来检索记录器。在构建配置中，您可以在类路径中包含日志记录库和 SLF4J 适配器。通过更改构建配置中的库，您可以在不更改函数代码的情况下更改记录器类型。从适用于 Java 的开发工具包中捕获日志需要使用 SLF4J。

在以下示例代码中，处理程序类使用 SLF4J 检索记录器。

Example src/main/java/example/HandlerS3.java：使用 SLF4J 进行日志记录

```
package example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;

import static org.apache.logging.log4j.CloseableThreadContext.put;


public class HandlerS3 implements RequestHandler<S3Event, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerS3.class);

    @Override
    public String handleRequest(S3Event event, Context context) {
        for(var record : event.getRecords()) {
            try (var loggingCtx = put("awsRegion", record.getAwsRegion())) {
```

```
                loggingCtx.put("eventName", record.getEventName());
                loggingCtx.put("bucket", record.getS3().getBucket().getName());
                loggingCtx.put("key", record.getS3().getObject().getKey());

                logger.info("Handling s3 event");
            }
        }

        return "Ok";
    }
}
```

此代码产生类似下面的日志输出：

Example 日志格式

```
{
    "timestamp": "2023-11-15T16:56:00.815Z",
    "level": "INFO",
    "message": "Handling s3 event",
    "logger": "example.HandlerS3",
    "AWSRequestId": "0bced576-3936-4e5a-9dcd-db9477b77f97",
    "awsRegion": "eu-south-1",
    "bucket": "java-logging-test-input-bucket",
    "eventName": "ObjectCreated:Put",
    "key": "test-folder/"
}
```

构建配置使用 Lambda Appender 和 SLF4J 适配器上的运行时系统依赖项以及 Log4j2 上的实现依赖项。

Example build.gradle – 日志记录依赖项

```
dependencies {
    ...
    'com.amazonaws:aws-lambda-java-log4j2:[1.6.0,)',
    'com.amazonaws:aws-lambda-java-events:[3.11.3,)',
    'org.apache.logging.log4j:log4j-layout-template-json:[2.17.1,)',
    'org.apache.logging.log4j:log4j-slf4j2-impl:[2.19.0,)',
    ...
}
```

在本地运行代码进行测试时，带有 Lambda 记录器的上下文对象不可用，并且没有供 Lambda Appender 使用的请求 ID 。有关测试配置示例，请参阅下一节中的示例应用程序。

使用其他日志记录工具和库

Powertools for AWS Lambda（Java）是一个开发人员工具包，用于实施无服务器最佳实践并提高开发人员速度。日志记录实用程序提供经优化的 Lambda 日志记录程序，其中包含有关所有函数的函数上下文的附加信息，输出结构为 JSON。请使用该实用程序执行以下操作：

- 从 Lambda 上下文中捕获关键字段，冷启动并将日志记录输出结构化为 JSON
- 根据指示记录 Lambda 调用事件（默认情况下禁用）
- 通过日志采样仅针对一定百分比的调用输出所有日志（默认情况下禁用）
- 在任何时间点将其他键附加到结构化日志
- 使用自定义日志格式设置程序（自带格式设置程序），从而在与组织的日志记录 RFC 兼容的结构中输出日志

将 Powertools for AWS Lambda（Java）和 AWS SAM 用于结构化日志记录

请按照以下步骤使用 AWS SAM，通过集成的 Powertools for AWS Lambda（Java）~ 模块来下载、构建和部署示例 Hello World Java 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 `hello world` 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Java 11
- AWS CLI 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS SAM 应用程序

1. 使用 Hello World Java 模板初始化该应用程序。

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. 构建应用程序。

```
cd sam-app && sam build
```

3. 部署应用程序。

```
sam deploy --guided
```

4. 按照屏幕上的提示操作。要在交互式体验中接受提供的默认选项，请按 Enter。

> **Note**
>
> 对于 HelloWorldFunction 可能没有定义授权，确定执行此操作吗？，确保输入 y。

5. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 调用 API 端点：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

7. 要获取该函数的日志，请运行 sam logs。有关更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 使用日志。

```
sam logs --stack-name sam-app
```

该日志输出类似于以下示例：

```
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.095000
 INIT_START Runtime Version: java:11.v15    Runtime Version ARN: arn:aws:lambda:eu-
central-1::runtime:0a25e3e7a1cc9ce404bc435eeb2ad358d8fa64338e618d0c224fe509403583ca
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.114000
 Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.793000
 Transforming org/apache/logging/log4j/core/lookup/JndiLookup
 (lambdainternal.CustomerClassLoader@1a6c5a9e)
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:35.252000
 START RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Version: $LATEST
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.531000 {
  "_aws": {
    "Timestamp": 1675416276051,
    "CloudWatchMetrics": [
      {
        "Namespace": "sam-app-powerools-java",
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ],
        "Dimensions": [
          [
            "Service",
            "FunctionName"
          ]
        ]
      }
    ]
  },
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "FunctionName": "sam-app-HelloWorldFunction-y9Iu1FLJJBGD",
  "functionVersion": "$LATEST",
  "ColdStart": 1.0,
  "Service": "service_undefined",
  "logStreamId": "2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.974000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.AWSXRayRecorder <init>
```

```
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.config.DaemonConfiguration <init>
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000
 INFO: Environment variable AWS_XRAY_DAEMON_ADDRESS is set. Emitting to daemon on
 address XXXX.XXXX.XXXX.XXXX:2000.
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.331000
 09:24:37.294 [main] INFO  helloworld.App - {"version":null,"resource":"/
hello","path":"/hello/","httpMethod":"GET","headers":{"Accept":"*/
*","CloudFront-Forwarded-Proto":"https","CloudFront-Is-Desktop-
Viewer":"true","CloudFront-Is-Mobile-Viewer":"false","CloudFront-Is-
SmartTV-Viewer":"false","CloudFront-Is-Tablet-Viewer":"false","CloudFront-
Viewer-ASN":"16509","CloudFront-Viewer-Country":"IE","Host":"XXXX.execute-
api.eu-central-1.amazonaws.com","User-Agent":"curl/7.86.0","Via":"2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)","X-Amz-
Cf-Id":"t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q==","X-
Amzn-Trace-Id":"Root=1-63dcd2d1-25f90b9d1c753a783547f4dd","X-Forwarded-
For":"XX.XXX.XXX.XX, XX.XXX.XXX.XX","X-Forwarded-Port":"443","X-
Forwarded-Proto":"https"},"multiValueHeaders":{"Accept":["*/
*"],"CloudFront-Forwarded-Proto":["https"],"CloudFront-Is-Desktop-Viewer":
["true"],"CloudFront-Is-Mobile-Viewer":["false"],"CloudFront-Is-SmartTV-
Viewer":["false"],"CloudFront-Is-Tablet-Viewer":["false"],"CloudFront-Viewer-
ASN":["16509"],"CloudFront-Viewer-Country":["IE"],"Host":["XXXX.execute-
api.eu-central-1.amazonaws.com"],"User-Agent":["curl/7.86.0"],"Via":["2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)"],"X-Amz-
Cf-Id":["t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q=="],"X-
Amzn-Trace-Id":["Root=1-63dcd2d1-25f90b9d1c753a783547f4dd"],"X-Forwarded-
For":["XXX, XXX"],"X-Forwarded-Port":["443"],"X-Forwarded-Proto":
["https"]},"queryStringParameters":null,"multiValueQueryStringParameters":null,"pathParamet
{"accountId":"XXX","stage":"Prod","resourceId":"at73a1","requestId":"ba09ecd2-
acf3-40f6-89af-fad32df67597","operationName":null,"identity":
{"cognitoIdentityPoolId":null,"accountId":null,"cognitoIdentityId":null,"caller":null,"apiK
hello","httpMethod":"GET","apiId":"XXX","path":"/Prod/
hello/","authorizer":null},"body":null,"isBase64Encoded":false}
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.351000
 09:24:37.351 [main] INFO  helloworld.App - Retrieving https://
checkip.amazonaws.com
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.313000 {
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "xray_trace_id": "1-63dcd2d1-25f90b9d1c753a783547f4dd",
  "functionVersion": "$LATEST",
  "Service": "service_undefined",
  "logStreamId": "2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
```

```
  "executionEnvironment": "AWS_Lambda_java11"
}
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000 END
 RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000
 REPORT RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765    Duration: 4118.98 ms
 Billed Duration: 4119 ms    Memory Size: 512 MB    Max Memory Used: 152 MB    Init
 Duration: 1155.47 ms
XRAY TraceId: 1-63dcd2d1-25f90b9d1c753a783547f4dd    SegmentId: 3a028fee19b895cb
 Sampled: true
```

8. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
sam delete
```

管理日志保留日期

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期结束后，日志将自动删除。要设置日志保留日期，请将以下内容添加到您的 AWS SAM 模板中：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

在 Lambda 控制台中查看日志

调用 Lambda 函数后，您可以使用 Lambda 控制台查看日志输出。

如果可以在嵌入式代码编辑器中测试代码，则可以在执行结果中找到日志。使用控制台测试功能调用函数时，可以在详细信息部分找到日志输出。

在 CloudWatch 控制台中查看日志

您可以使用 Amazon CloudWatch 控制台查看所有 Lambda 函数调用的日志。

使用 CloudWatch 控制台查看日志

1. 打开 CloudWatch 控制台的 Log groups（日志组页面）。
2. 选择您的函数 (/aws/lambda/***your-function-name***) 的日志组。
3. 创建日志流。

每个日志流对应一个函数实例。日志流会在您更新 Lambda 函数以及创建更多实例来处理多个并发调用时显示。要查找特定调用的日志，建议您使用 AWS X-Ray 检测函数。X-Ray 会在追踪中记录有关请求和日志流的详细信息。

使用 AWS Command Line Interface（AWS CLI）查看日志

AWS CLI 是一种开源工具，让您能够在命令行 Shell 中使用命令与 AWS 服务进行交互。要完成本节中的步骤，您必须拥有 AWS CLI 版本 2。

您可以通过 AWS CLI，使用 `--log-type` 命令选项检索调用的日志。响应包含一个 `LogResult` 字段，其中包含多达 4KB 来自调用的 base64 编码日志。

Example 检索日志 ID

以下示例说明如何从 `LogResult` 字段中检索名为 `my-function` 的函数的日志 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解码日志

在同一命令提示符下，使用 base64 实用程序解码日志。以下示例说明如何为 my-function 检索 base64 编码的日志。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

您应看到以下输出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64 实用程序在 Linux、macOS 和 Ubuntu on Windows 上可用。macOS 用户可能需要使用 base64 -D。

Example get-logs.sh 脚本

在同一命令提示符下，使用以下脚本下载最后五个日志事件。此脚本使用 sed 从输出文件中删除引号，并休眠 15 秒以等待日志可用。输出包括来自 Lambda 的响应，以及来自 get-log-events 命令的输出。

复制以下代码示例的内容并将其作为 get-logs.sh 保存在 Lambda 项目目录中。

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
```

```
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux（仅限）

在同一命令提示符下，macOS 和 Linux 用户可能需要运行以下命令以确保脚本可执行。

```
chmod -R 755 get-logs.sh
```

Example 检索最后五个日志事件

在同一命令提示符下，运行以下脚本以获取最后五个日志事件。

```
./get-logs.sh
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
```

```
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

删除日志

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期之后，日志将自动删除。

日志记录代码示例

本指南的 GitHub 存储库包括演示如何使用各种日志记录配置的示例应用程序。每个示例应用程序都包含用于轻松部署和清理的脚本、一个 AWS SAM 模板和支持资源。

Java 中的 Lambda 应用程序示例

- java17-examples：这是一种 Java 函数，演示如何使用 Java 记录来表示输入事件数据对象。
- java-basic – 具有单元测试和变量日志记录配置的最小 Java 函数的集合。
- java-events – Java 函数的集合，其中包含用于处理来自 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis 等各种服务的事件的框架代码。这些函数使用最新版本的 aws-lambda-events 库（3.0.0 及更新版本）。这些示例不需要 AWS 开发工具包作为依赖项。
- s3-java – 此 Java 函数可处理来自 Amazon S3 的通知事件，并使用 Java 类库（JCL）从上传的图像文件创建缩略图。
- 自定义序列化 –如何使用 fastJson、Gson、Moshi 和 jackson-jr 等常用库实现自定义序列化的示例。
- 使用 API Gateway 调用 Lambda 函数 – Java 函数，用于扫描包含员工信息的 Amazon DynamoDB 表。然后，该函数使用 Amazon Simple Notification Service 向员工发送短信，祝贺他们工作周年纪念日快乐。此示例使用 API Gateway 调用函数。

java-basic 示例应用程序显示支持日志记录测试的最小日志记录配置。处理程序代码使用上下文对象提供的 LambdaLogger 记录器。对于测试，应用程序使用一个自定义 TestLogger 类，此类实现带有 Log4j2 记录器的 LambdaLogger 接口。它使用 SLF4J 作为 Facade 以与AWS开发工具包兼容。从构建输出中排除日志记录库，以使部署程序包保持较小。

在 AWS Lambda 中检测 Java 代码

Lambda 与 AWS X-Ray 集成，以帮助您跟踪、调试和优化 Lambda 应用程序。您可以在某个请求遍历应用程序中的资源（其中可能包括 Lambda 函数和其他 AWS 服务）时，使用 X-Ray 跟踪该请求。

要将跟踪数据发送到 X-Ray，您可以使用以下两个软件开发工具包 (SDK) 库之一：

- 适用于 OpenTelemetry 的 AWS 发行版 (ADOT) – 一种安全、可供生产、支持 AWS 的 OpenTelemetry (OTel) SDK 的分发版本。
- AWS X-Ray SDK for Java – 用于生成跟踪数据并将其发送到 X-Ray 的 SDK
- Powertools for AWS Lambda（Java） – 一个开发人员工具包，用于实施无服务器最佳实践并提高开发人员速度。

每个开发工具包均提供了将遥测数据发送到 X-Ray 服务的方法。然后，您可以使用 X-Ray 查看、筛选和获得对应用程序性能指标的洞察，从而发现问题和优化机会。

> **Important**
>
> X-Ray 和 Powertools for AWS Lambda SDK 是 AWS 提供的紧密集成的分析解决方案的一部分。ADOT Lambda Layers 是全行业通用的跟踪分析标准的一部分，该标准通常会收集更多数据，但可能不适用于所有使用案例。您可以使用任一解决方案在 X-Ray 中实现端到端跟踪。要了解有关如何在两者之间进行选择的更多信息，请参阅在 AWS Distro for Open Telemetry 和 X-Ray 开发工具包之间进行选择。

Sections

- 示例应用程序中的 X-Ray 跟踪 (X-Ray SDK)

将 Powertools for AWS Lambda（Java）和 AWS SAM 用于跟踪

请按照以下步骤使用 AWS SAM，通过集成的 Powertools for AWS Lambda（Java）模块来下载、构建和部署示例 Hello World Java 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 `hello world` 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Java 11
- AWS CLI 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS SAM 应用程序

1. 使用 Hello World Java 模板初始化该应用程序。

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. 构建应用程序。

```
cd sam-app && sam build
```

3. 部署应用程序。

```
sam deploy --guided
```

4. 按照屏幕上的提示操作。要在交互式体验中接受提供的默认选项，请按 Enter。

> **Note**
>
> 对于 HelloWorldFunction 可能没有定义授权，确定执行此操作吗？，确保输入 y。

5. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 调用 API 端点：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

7. 要获取该函数的跟踪信息，请运行 sam traces。

```
sam traces
```

该跟踪输出类似于以下示例：

```
New XRay Service Graph
  Start time: 2023-02-03 14:31:48+01:00
  End time: 2023-02-03 14:31:48+01:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-y9Iu1FLJJBGD -
 Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 5.587
  Reference Id: 1 - client - sam-app-HelloWorldFunction-y9Iu1FLJJBGD - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
```

```
    - total response time: 0

XRay Event [revision 3] at (2023-02-03T14:31:48.500000) with id
 (1-63dd0cc4-3c869dec72a586875da39777) and duration (5.603s)
 - 5.587s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD [HTTP: 200]
 - 4.053s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD
   - 1.181s - Initialization
   - 4.037s - Invocation
     - 1.981s - ## handleRequest
       - 1.840s - ## getPageContents
   - 0.000s - Overhead
```

8. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
sam delete
```

将 Powertools for AWS Lambda（Java）和 AWS CDK 用于跟踪

请按照以下步骤使用 AWS CDK，通过集成的 Powertools for AWS Lambda（Java）模块来下载、构建和部署示例 Hello World Java 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 hello world 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Java 11
- AWS CLI 版本 2
- AWS CDK 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS CDK 应用程序

1. 为您的新应用程序创建一个项目目录。

```
mkdir hello-world
```

```
cd hello-world
```

2. 初始化该应用程序。

```
cdk init app --language java
```

3. 使用以下命令创建 Maven 项目：

```
mkdir app
cd app
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. 打开 `hello-world\app\Function` 目录中的 `pom.xml`，并将现有代码替换为以下代码，其中包括 Powertools 的依赖项和 Maven 插件。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>helloworld</groupId>
  <artifactId>Function</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Function</name>
  <url>http://maven.apache.org</url>
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <log4j.version>2.17.2</log4j.version>
</properties>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-tracing</artifactId>
            <version>1.3.0</version>
```

```
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-metrics</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-logging</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.2</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-events</artifactId>
            <version>3.11.1</version>
        </dependency>
  </dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>aspectj-maven-plugin</artifactId>
            <version>1.14.0</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
                <complianceLevel>${maven.compiler.target}</complianceLevel>
                <aspectLibraries>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-tracing</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-metrics</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-logging</artifactId>
```

```
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer

 implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCache
                                </transformer>
                            </transformers>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
                            <finalName>function</finalName>

                        </configuration>
                    </execution>
                </executions>
                <dependencies>
                    <dependency>
                        <groupId>com.github.edwgiz</groupId>
                        <artifactId>maven-shade-plugin.log4j2-cachefile-
transformer</artifactId>
                        <version>2.15</version>
                    </dependency>
                </dependencies>
        </plugin>
```

```
    </plugins>
</build>
</project>
```

5. 创建 hello-world\app\src\main\resource 目录并为日志配置创建 log4j.xml。

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. 打开 log4j.xml 并添加以下代码。

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Appenders>
        <Console name="JsonAppender" target="SYSTEM_OUT">
            <JsonTemplateLayout
 eventTemplateUri="classpath:LambdaJsonLayout.json" />
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="JsonLogger" level="INFO" additivity="false">
            <AppenderRef ref="JsonAppender"/>
        </Logger>
        <Root level="info">
            <AppenderRef ref="JsonAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

7. 打开 hello-world\app\Function\src\main\java\helloworld 目录中的 App.java，并将现有代码替换为以下代码。这是适用于 Lambda 函数的代码。

```
package helloworld;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;
```

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.metrics.Metrics;
import software.amazon.lambda.powertools.tracing.CaptureMode;
import software.amazon.lambda.powertools.tracing.Tracing;

import static software.amazon.lambda.powertools.tracing.CaptureMode.*;

/**
 * Handler for requests to Lambda function.
 */
public class App implements RequestHandler<APIGatewayProxyRequestEvent,
 APIGatewayProxyResponseEvent> {
    Logger log = LogManager.getLogger(App.class);


    @Logging(logEvent = true)
    @Tracing(captureMode = DISABLED)
    @Metrics(captureColdStart = true)
    public APIGatewayProxyResponseEvent handleRequest(final
 APIGatewayProxyRequestEvent input, final Context context) {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        headers.put("X-Custom-Header", "application/json");

        APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent()
                .withHeaders(headers);
        try {
            final String pageContents = this.getPageContents("https://
checkip.amazonaws.com");
            String output = String.format("{ \"message\": \"hello world\",
 \"location\": \"%s\" }", pageContents);

            return response
                    .withStatusCode(200)
                    .withBody(output);
        } catch (IOException e) {
            return response
                    .withBody("{}")
```

```
                    .withStatusCode(500);
        }
    }
    @Tracing(namespace = "getPageContents")
    private String getPageContents(String address) throws IOException {
        log.info("Retrieving {}", address);
        URL url = new URL(address);
        try (BufferedReader br = new BufferedReader(new
 InputStreamReader(url.openStream()))) {
            return br.lines().collect(Collectors.joining(System.lineSeparator()));
        }
    }
}
```

8. 打开 hello-world\src\main\java\com\myorg 目录中的 HelloWorldStack.java，并将现有代码替换为以下代码。此代码将使用 Lambda 构造函数和 ApiGatewayv2 构造函数创建 REST API 和 Lambda 函数。

```
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.apigatewayv2.alpha.*;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegration;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegrationProps
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.FunctionProps;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Tracing;
import software.amazon.awscdk.services.logs.RetentionDays;
import software.amazon.awscdk.services.s3.assets.AssetOptions;
import software.constructs.Construct;

import java.util.Arrays;
import java.util.List;

import static java.util.Collections.singletonList;
import static software.amazon.awscdk.BundlingOutput.ARCHIVED;

public class HelloWorldStack extends Stack {
    public HelloWorldStack(final Construct scope, final String id) {
```

```
        this(scope, id, null);
    }

    public HelloWorldStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        List<String> functionPackagingInstructions = Arrays.asList(
                "/bin/sh",
                "-c",
                "cd Function " +
                        "&& mvn clean install " +
                        "&& cp /asset-input/Function/target/function.jar /asset-
output/"
        );
        BundlingOptions.Builder builderOptions = BundlingOptions.builder()
                .command(functionPackagingInstructions)
                .image(Runtime.JAVA_11.getBundlingImage())
                .volumes(singletonList(
                        // Mount local .m2 repo to avoid download all the
 dependencies again inside the container
                        DockerVolume.builder()
                                .hostPath(System.getProperty("user.home") +
 "/.m2/")
                                .containerPath("/root/.m2/")
                                .build()
                ))
                .user("root")
                .outputType(ARCHIVED);

        Function function = new Function(this, "Function", FunctionProps.builder()
                .runtime(Runtime.JAVA_11)
                .code(Code.fromAsset("app", AssetOptions.builder()
                        .bundling(builderOptions
                                .command(functionPackagingInstructions)
                                .build())
                        .build()))
                .handler("helloworld.App::handleRequest")
                .memorySize(1024)
                .tracing(Tracing.ACTIVE)
                .timeout(Duration.seconds(10))
                .logRetention(RetentionDays.ONE_WEEK)
                .build());
```

```
        HttpApi httpApi = new HttpApi(this, "sample-api", HttpApiProps.builder()
                .apiName("sample-api")
                .build());

        httpApi.addRoutes(AddRoutesOptions.builder()
                .path("/")
                .methods(singletonList(HttpMethod.GET))
                .integration(new HttpLambdaIntegration("function", function,
 HttpLambdaIntegrationProps.builder()
                        .payloadFormatVersion(PayloadFormatVersion.VERSION_2_0)
                        .build()))
                .build());

        new CfnOutput(this, "HttpApi", CfnOutputProps.builder()
                .description("Url for Http Api")
                .value(httpApi.getApiEndpoint())
                .build());
    }
}
```

9. 打开 hello-world 目录中的 pom.xml，并将现有代码替换为以下代码。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.myorg</groupId>
    <artifactId>hello-world</artifactId>
    <version>0.1</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <cdk.version>2.70.0</cdk.version>
        <constructs.version>[10.0.0,11.0.0)</constructs.version>
        <junit.version>5.7.1</junit.version>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
```

```
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.0.0</version>
                <configuration>
                    <mainClass>com.myorg.HelloWorldApp</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <!-- AWS Cloud Development Kit -->
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>aws-cdk-lib</artifactId>
            <version>${cdk.version}</version>
        </dependency>
        <dependency>
            <groupId>software.constructs</groupId>
            <artifactId>constructs</artifactId>
            <version>${constructs.version}</version>
        </dependency>
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter</artifactId>
          <version>${junit.version}</version>
          <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
```

```
            <artifactId>apigatewayv2-integrations-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
    </dependencies>
</project>
```

10. 确保您位于 hello-world 目录中然后部署应用程序。

```
cdk deploy
```

11. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. 调用 API 端点：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

13. 要获取该函数的跟踪信息，请运行 sam traces。

```
sam traces
```

该跟踪输出类似于以下示例：

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
```

```
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

14. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
cdk destroy
```

使用 ADOT 分析您的 Java 函数

ADOT 提供完全托管式 Lambda 层，这些层使用 OTel SDK，将收集遥测数据所需的一切内容打包起来。通过使用此层，您可以在不必修改任何函数代码的情况下，对您的 Lambda 函数进行分析。您还可以将您的层配置为对 OTel 进行自定义初始化。有关更多信息，请参阅 ADOT 文档中的适用于 Lambda 上的 ADOT 收集器的自定义配置。

对于 Java 运行时，您可以在两个层之间选择使用：

- 适用于 ADOT Java 的 AWS 托管式 Lambda 层（自动分析代理）- 此层将在启动时自动转换您的函数代码，以收集跟踪数据。有关如何与 ADOT Java 代理配合使用此层的详细说明，请参阅适用于 OpenTelemetry 的 AWS 发行版对于 Java 的 Lambda 支持（自动分析代理）。
- 适用于 ADOT Java 的 AWS 托管式 Lambda 层 - 此层还可为 Lambda 函数提供内置分析，但其需要手动对代码进行一些更改，才能初始化 OTel SDK。有关如何使用此层的详细说明，请参阅 ADOT 文档中的适用于 OpenTelemetry 的 AWS 发行版对于 Java 的 Lambda 支持。

使用 X-Ray SDK 分析您的 Java 函数

要记录关于您的函数对应用程序中的其他资源和服务进行调用的数据，您可以将适用于 Java 的 X-Ray SDK 添加到您的构建配置中。以下示例显示了一种 Gradle 构建配置，其中包括激活 AWS SDK for Java 2.x 客户端自动分析的库。

Example build.gradle – 跟踪依赖项

```
dependencies {
    implementation platform('software.amazon.awssdk:bom:2.16.1')
    implementation platform('com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0')
    ...
    implementation 'com.amazonaws:aws-xray-recorder-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2-instrumentor'
    ...
}
```

在添加正确的依赖项并进行必要的代码更改后，请通过 Lambda 控制台或 API 激活函数配置中的跟踪。

使用 Lambda 控制台激活跟踪

要使用控制台切换 Lambda 函数的活动跟踪，请按照以下步骤操作：

打开活跃跟踪

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration（配置），然后选择 Monitoring and operations tools（监控和操作工具）。
4. 选择编辑。

5. 在 X-Ray 下方，开启 Active tracing（活动跟踪）。
6. 选择保存。

使用 Lambda API 激活跟踪

借助 AWS CLI 或 AWS SDK 在 Lambda 函数上配置跟踪，请使用以下 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下示例 AWS CLI 命令对名为 my-function 的函数启用活跃跟踪。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

跟踪模式是发布函数版本时版本特定配置的一部分。您无法更改已发布版本上的跟踪模式。

使用 AWS CloudFormation 激活跟踪

要对 AWS CloudFormation 模板中的 `AWS::Lambda::Function` 资源激活跟踪，请使用 `TracingConfig` 属性。

Example function-inline.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

对于 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 资源，请使用 `Tracing` 属性。

Example template.yml – 跟踪配置

```
Resources:
```

```
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

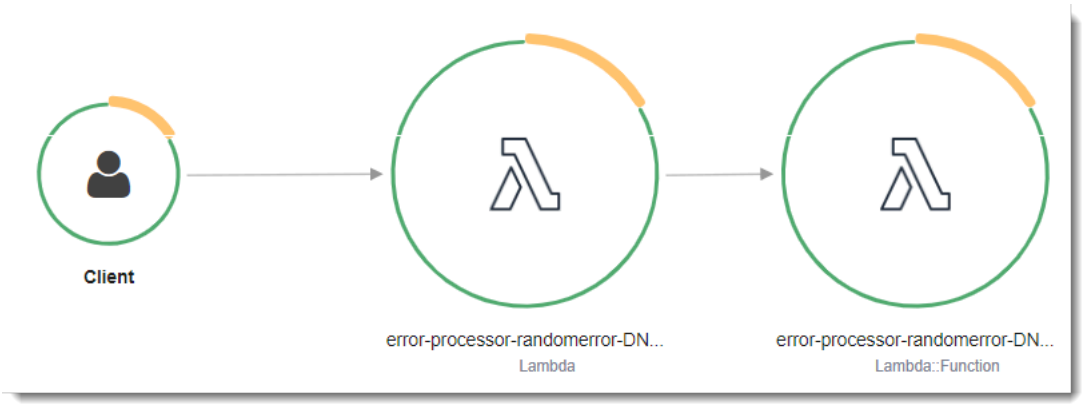
解释 X-Ray 跟踪

您的函数需要权限才能将跟踪数据上载到 X-Ray。在 Lambda 控制台中激活跟踪后，Lambda 会将所需权限添加到函数的执行角色。如果没有，请将 AWSXRayDaemonWriteAccess 策略添加到执行角色。

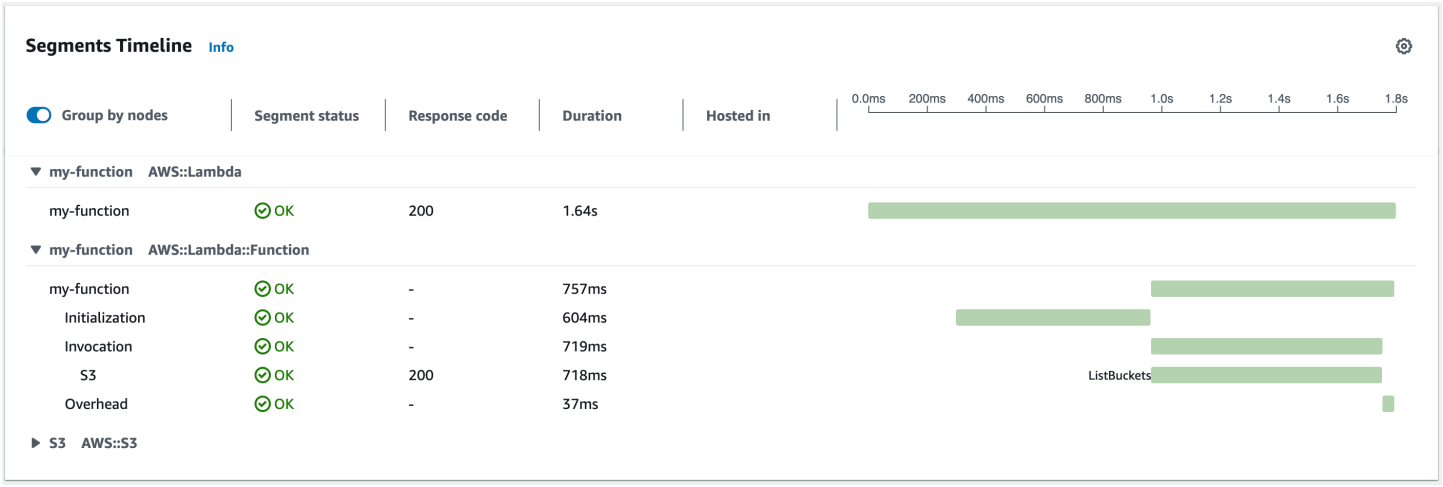
在配置活跃跟踪后，您可以通过应用程序观察特定请求。X-Ray 服务图将显示有关应用程序及其所有组件的信息。以下示例显示了具有两个函数的应用程序。主函数处理事件，有时会返回错误。位于顶部的第二个函数将处理第一个函数的日志组中显示的错误，并使用 AWS SDK 调用 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

X-Ray 无法跟踪对应用程序的所有请求。X-Ray 将应用采样算法确保跟踪有效，同时仍会提供所有请求的一个代表性样本。采样率是每秒 1 个请求和 5% 的其他请求。您无法为函数配置此 X-Ray 采样率。

在 X-Ray 中，跟踪记录有关由一个或多个服务处理的请求的信息。Lambda 会每个跟踪记录 2 个分段，这些分段将在服务图上创建两个节点。下图突出显示了这两个节点：



位于左侧的第一个节点表示接收调用请求的 Lambda 服务。第二个节点表示特定的 Lambda 函数。以下示例显示了一个包含这 2 个分段的跟踪。两者都命名为 my-function，但其中一个函数具有 AWS::Lambda 源，另一个则具有 AWS::Lambda::Function 源。如果 AWS::Lambda 分段显示错误，则表示 Lambda 服务存在问题。如果 AWS::Lambda::Function 分段显示错误，则说明函数存在问题。



此示例将展开 AWS::Lambda::Function 分段，以显示其三个子分段。

> **Note**
>
> AWS 目前正在实施对 Lambda 服务的更改。由于这些更改，您可能会看到 AWS 账户 中不同 Lambda 函数发出的系统日志消息和跟踪分段的结构和内容之间存在细微差异。
> 此处显示的示例跟踪说明了旧样式函数分段。以下段落介绍了新旧样式分段之间的差异。
> 这些更改将在未来几周内实施，除中国和 GovCloud 区域外，所有 AWS 区域 的函数都将过渡到使用新格式的日志消息和跟踪分段。

旧样式函数分段包含以下子分段：

- 初始化 – 表示加载函数和运行初始化代码所花费的时间。此子分段仅对由您的函数的每个实例处理的第一个事件显示。
- 调用 – 表示执行处理程序代码花费的时间。
- 开销 – 表示 Lambda 运行时为准备处理下一个事件而花费的时间。

新样式函数分段不包含 `Invocation` 子分段。而是将客户子分段直接附加到函数分段。有关新旧样式函数分段结构的更多信息，请参阅 the section called “了解 X-Ray 跟踪”。

> **Note**
>
> Lambda SnapStart 函数还包括一个 `Restore` 子分段。`Restore` 子分段会显示 Lambda 恢复快照、加载运行时系统（JVM）和运行任何 `afterRestore` 运行时系统挂钩所花费的时间。恢复快照的过程可能包含在 MicroVM 之外的活动上花费的时间。该时间在 `Restore` 子分段中报告。您无需为在 microVM 之外还原快照所花费的时间付费。

您还可以分析 HTTP 客户端、记录 SQL 查询以及使用注释和元数据创建自定义子段。有关更多信息，请参阅 AWS X-Ray 开发人员指南中的 AWS X-Ray SDK for Java。

> **定价**
>
> 作为 AWS 免费套餐的组成部分，您可以每月免费使用 X-Ray 跟踪，但不能超过一定限制。超出该阈值后，X-Ray 会对跟踪存储和检索进行收费。有关更多信息，请参阅 AWS X-Ray 定价。

在层中存储运行时依赖项 (X-Ray SDK)

如果您使用 X-Ray 开发工具包来分析AWS开发工具包客户端和您的函数代码，则您的部署程序包可能会变得相当大。为了避免每次更新函数代码时上载运行时依赖项，请将 X-Ray SDK 打包到 Lambda 层中。

以下示例显示了存储适用于 Java 的 AWS SDK for Java 和 X-Ray SDK 的 `AWS::Serverless::LayerVersion` 资源。

Example template.yml – 依赖项层

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/blank-java.zip
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-java-lib
      Description: Dependencies for the blank-java sample app.
      ContentUri: build/blank-java-lib.zip
      CompatibleRuntimes:
        - java21
```

使用此配置，仅在更改运行时依赖项时您才会更新库层。由于函数部署软件包仅包含您的代码，因此可以帮助缩短上传时间。

为依赖项创建层要求更改构建配置才能在部署之前生成层存档。有关工作示例，请参阅 GitHub 上的 java-basic 示例应用程序。

示例应用程序中的 X-Ray 跟踪 (X-Ray SDK)

本指南的 GitHub 存储库包括演示如何使用 X-Ray 跟踪的示例应用程序。每个示例应用程序都包含用于轻松部署和清理的脚本、一个 AWS SAM 模板和支持资源。

Java 中的 Lambda 应用程序示例

- java17-examples：这是一种 Java 函数，演示如何使用 Java 记录来表示输入事件数据对象。
- java-basic – 具有单元测试和变量日志记录配置的最小 Java 函数的集合。
- java-events – Java 函数的集合，其中包含用于处理来自 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis 等各种服务的事件的框架代码。这些函数使用最新版本的 aws-lambda-events 库（3.0.0 及更新版本）。这些示例不需要 AWS 开发工具包作为依赖项。
- s3-java – 此 Java 函数可处理来自 Amazon S3 的通知事件，并使用 Java 类库（JCL）从上传的图像文件创建缩略图。

- 自定义序列化 –如何使用 fastJson、Gson、Moshi 和 jackson-jr 等常用库实现自定义序列化的示例。
- 使用 API Gateway 调用 Lambda 函数 – Java 函数，用于扫描包含员工信息的 Amazon DynamoDB 表。然后，该函数使用 Amazon Simple Notification Service 向员工发送短信，祝贺他们工作周年纪念日快乐。此示例使用 API Gateway 调用函数。

所有示例应用程序都为 Lambda 功能启用了活动跟踪。例如，`s3-java` 应用程序显示 AWS SDK for Java 2.x 客户端的自动分析、用于测试的段管理、自定义子段以及使用 Lambda 层存储运行时依赖项。

AWS Lambda 的 Java 示例应用程序

本指南的 GitHub 存储库提供了演示如何在 AWS Lambda 使用 Java 的示例应用程序。每个示例应用程序都包含用于轻松部署和清理的脚本、一个 AWS CloudFormation 模板和支持资源。

Java 中的 Lambda 应用程序示例

- java17-examples：这是一种 Java 函数，演示如何使用 Java 记录来表示输入事件数据对象。
- java-basic – 具有单元测试和变量日志记录配置的最小 Java 函数的集合。
- java-events – Java 函数的集合，其中包含用于处理来自 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis 等各种服务的事件的框架代码。这些函数使用最新版本的 aws-lambda-events 库（3.0.0 及更新版本）。这些示例不需要 AWS 开发工具包作为依赖项。
- s3-java – 此 Java 函数可处理来自 Amazon S3 的通知事件，并使用 Java 类库（JCL）从上传的图像文件创建缩略图。
- 自定义序列化 –如何使用 fastJson、Gson、Moshi 和 jackson-jr 等常用库实现自定义序列化的示例。
- 使用 API Gateway 调用 Lambda 函数 – Java 函数，用于扫描包含员工信息的 Amazon DynamoDB 表。然后，该函数使用 Amazon Simple Notification Service 向员工发送短信，祝贺他们工作周年纪念日快乐。此示例使用 API Gateway 调用函数。

在 Lambda 上运行常见 Java 框架

- spring-cloud-function-samples：此示例来自 Spring，展示了如何使用 Spring Cloud Function 框架创建 AWS Lambda 函数。
- 无服务器 Spring Boot 应用程序演示：该示例展示了如何在带有 SnapStart 和不带有 SnapStart 的托管式 Java 运行时系统中设置典型的 Spring Boot 应用程序，或者如何使用自定义运行时系统设置为 GraalVM 本机映像。
- 无服务器 Micronaut 应用程序演示：该示例展示了如何在带有 SnapStart 和不带有 SnapStart 的托管式 Java 运行时系统中使用 Micronaut，或者如何使用自定义运行时系统设置为 GraalVM 本机映像。在《Micronaut/Lambda 指南》中了解更多信息。
- 无服务器 Quarkus 应用程序演示：该示例展示了如何在带有 SnapStart 和不带有 SnapStart 的托管式 Java 运行时系统中使用 Quarkus，或者如何使用自定义运行时系统设置为 GraalVM 本机映像。在《Quarkus/Lambda 指南》和《Quarkus/SnapStart 指南》中了解更多信息。

如果您刚接触用 Java 编写的 Lambda 函数，不妨先尝试 `java-basic` 示例。要开始使用 Lambda 事件源，请参阅 `java-events` 示例。这两种示例都会显示 Lambda 的 Java 库、环境变量、AWS SDK

和 AWS X-Ray SDK 的使用情况。这些示例需要最少的设置，并且可以在不到一分钟的时间内从命令行部署。

使用 Go 构建 Lambda 函数

Go 的实施方式与其他托管式运行时系统不同。由于 Go 本机编译为可执行的二进制文件，因此它不需要专用的语言运行时。使用仅限操作系统的运行时（provided 运行时系列）将 Go 函数部署到 Lambda。

主题

Go 运行时系统支持

Lambda 的 Go 1.x 托管式运行时已被弃用。如果您具有使用 Go 1.x 运行时的函数，则必须将函数迁移到 provided.al2023 或 provided.al2。与 go1.x 相比，provided.al2023 和 provided.al2 运行时系统具有多种优势，包括支持 arm64 架构（AWS Graviton2 处理器）、二进制文件更小以及调用时间稍快。

此迁移无需更改任何代码。唯一需要进行的更改涉及如何构建部署包以及使用哪个运行时系统来创建函数。有关更多信息，请参阅 AWS Compute Blog 上的 Migrating AWS Lambda functions from the Go1.x runtime to the custom runtime on Amazon Linux 2。

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
仅限操作系统的运行时系统	provided.al2023	Amazon Linux 2023	未计划	未计划	未计划
仅限操作系统的运行时系统	provided.al2	Amazon Linux 2	未计划	未计划	未计划

工具和库

Lambda 为 Go 运行时提供了以下工具和库：

- 适用于 Go 的 AWS 开发工具包：适用于 Go 编程语言的官方 AWS 开发工具包。
- github.com/aws/aws-lambda-go/lambda：适用于 Go 的 Lambda 编程模型的实现。AWS Lambda 使用此程序包调用您的处理程序。
- github.com/aws/aws-lambda-go/lambdacontext：用于访问上下文对象中的上下文信息的帮助程序。
- github.com/aws/aws-lambda-go/events：此库提供常见事件源集成的类型定义。
- github.com/aws/aws-lambda-go/cmd/build-lambda-zip：此工具可用于在 Windows 上创建 .zip 文件存档。

有关更多信息，请参阅 GitHub 上的 aws-lambda-go。

Lambda 为 Go 运行时提供了以下示例应用程序：

Go 中的 Lambda 应用程序示例

- go-al2：返回公有 IP 地址的 hello world 函数。此应用程序使用 `provided.al2` 自定义运行时系统。
- blank-go – 此 Go 函数显示 Lambda 的 Go 库、日志记录、环境变量和 AWS SDK 的使用情况。此应用程序使用 `go1.x` 运行时系统。

定义采用 Go 的 Lambda 函数处理程序

Lambda 函数处理程序是函数代码中处理事件的方法。当调用函数时，Lambda 运行处理程序方法。您的函数会一直运行，直到处理程序返回响应、退出或超时。

本页介绍如何使用采用 Go 的 Lambda 函数处理程序，包括项目设置、命名约定和最佳实践。本页还包括 Go Lambda 函数的示例，该函数接收订单信息，生成文本文件收据，然后将此文件放入 Amazon Simple Storage Service（S3）存储桶中。有关如何在编写函数后部署函数的信息，请参阅the section called “部署 .zip 文件归档”或the section called “部署容器镜像”。

主题

设置 Go 处理程序项目

在 Go 中编写的 Lambda 函数被编写为 Go 可执行文件。与初始化任何其他 Go 项目的方法相同，您可以使用以下 `go mod init` 命令初始化 Go Lambda 函数项目：

```
go mod init example-go
```

此处，`example-go` 表示模块名称。您可以使用任意值替换。此命令将初始化项目，并生成列出项目依赖项的 `go.mod` 文件。

使用 `go get` 命令将任何外部依赖项添加到项目中。例如，在采用 Go 的 Lambda 函数中，需要包含 github.com/aws/aws-lambda-go/lambda 包，该包将实现适用于 Go 的 Lambda 编程模型。使用以下 `go get` 命令安装此包：

```
go get github.com/aws/aws-lambda-go
```

函数代码应该位于 Go 文件中。在以下示例中，我们将此文件命名为 `main.go`。在此文件中，您可以在处理程序方法以及调用此处理程序的 `main()` 函数中实现核心函数逻辑。

示例 Go Lambda 函数代码

以下示例 Go Lambda 函数代码接收有关订单的信息，生成文本文件接收，并将此文件放入 Amazon S3 存储桶中。

Example **`main.go`** Lambda 函数

```
package main

import (
 "context"
 "encoding/json"
 "fmt"
 "log"
 "os"
 "strings"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

type Order struct {
 OrderID string  `json:"order_id"`
 Amount  float64 `json:"amount"`
 Item    string  `json:"item"`
}

var (
 s3Client *s3.Client
)

func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Fatalf("unable to load SDK config, %v", err)
```

```
 }

 s3Client = s3.NewFromConfig(cfg)
}

func uploadReceiptToS3(ctx context.Context, bucketName, key, receiptContent string)
 error {
 _, err := s3Client.PutObject(ctx, &s3.PutObjectInput{
  Bucket: &bucketName,
  Key:    &key,
  Body:   strings.NewReader(receiptContent),
 })
 if err != nil {
  log.Printf("Failed to upload receipt to S3: %v", err)
  return err
 }
 return nil
}

func handleRequest(ctx context.Context, event json.RawMessage) error {
 // Parse the input event
 var order Order
 if err := json.Unmarshal(event, &order); err != nil {
  log.Printf("Failed to unmarshal event: %v", err)
  return err
 }

 // Access environment variables
 bucketName := os.Getenv("RECEIPT_BUCKET")
 if bucketName == "" {
  log.Printf("RECEIPT_BUCKET environment variable is not set")
  return fmt.Errorf("missing required environment variable RECEIPT_BUCKET")
 }

 // Create the receipt content and key destination
 receiptContent := fmt.Sprintf("OrderID: %s\nAmount: $%.2f\nItem: %s",
  order.OrderID, order.Amount, order.Item)
 key := "receipts/" + order.OrderID + ".txt"

 // Upload the receipt to S3 using the helper method
 if err := uploadReceiptToS3(ctx, bucketName, key, receiptContent); err != nil {
  return err
 }
```

```
 log.Printf("Successfully processed order %s and stored receipt in S3 bucket %s",
 order.OrderID, bucketName)
 return nil
}

func main() {
 lambda.Start(handleRequest)
}
```

此 main.go 文件包含以下代码部分：

- package main：在 Go 中，包含 func main() 函数的包必须始终名为 main。
- import 数据块：使用此数据块来包含 Lambda 函数所需的库。
- type Order struct {} 数据块：在此 Go 结构中定义预期输入事件的形状。
- var () 数据块：使用此数据块定义您将在 Lambda 函数中使用的任何全局变量。
- func init() {}：在此 init() 方法中，包含您希望 Lambda 在初始化阶段运行的任何代码。
- func uploadReceiptToS3(...) {}：这是主 handleRequest 处理程序方法引用的帮助程序方法。
- func handleRequest(ctx context.Context, event json.RawMessage) error {}：这是包含主应用程序逻辑的主处理程序方法。
- func main() {}：这是 Lambda 处理程序必需的入口点。lambda.Start() 方法的参数是主处理程序方法。

要此函数正常运行，其执行角色必须允许 s3:PutObject 操作。此外，请确保您定义了 RECEIPT_BUCKET 环境变量。成功调用后，Amazon S3 存储桶应包含接收文件。

处理程序命名约定

对于采用 Go 的 Lambda 函数，处理程序可以使用任何名称。在此示例中，处理程序方法被命名为 handleRequest。要在代码中引用处理程序值，可以使用 _HANDLER 环境变量。

对于使用 .zip 部署包的 Go 函数，包含函数代码的可执行文件必须命名为 bootstrap。此外，bootstrap 文件必须位于 .zip 文件的根目录中。对于使用容器映像的 Go 函数，可执行文件可以使用任何名称。

定义和访问输入事件对象

JSON 是 Lambda 函数最常用且最标准的输入格式。在此示例中，该函数需要类似于下方的输入：

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

在使用采用 Go 的 Lambda 函数时，您可以将预期输入事件的形状定义为 Go 结构。在此示例中，我们定义结构来表示 `Order`：

```
type Order struct {
    OrderID string  `json:"order_id"`
    Amount  float64 `json:"amount"`
    Item    string  `json:"item"`
}
```

此结构与预期的输入形状相匹配。定义结构后，您可以编写处理程序签名，该签名采用与 encoding/json 标准库兼容的通用 JSON 类型。然后，您可以使用 func Unmarshal 函数将其反序列化到结构中。处理程序的前几行对此进行了说明：

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    // Parse the input event
    var order Order
    if err := json.Unmarshal(event, &order); err != nil {
        log.Printf("Failed to unmarshal event: %v", err)
        return err
    ...
}
```

反序列化后，您可以访问 `order` 变量的字段。例如，`order.OrderID` 从原始输入中检索 `"order_id"` 的值。

> **Note**
>
> 该 `encoding/json` 包只能访问导出的字段。若要导出，事件结构中的字段名称必须大写。

访问和使用 Lambda 上下文对象

Lambda 上下文对象包含有关调用、函数和执行环境的信息。在此例中，我们在处理程序签名中将此变量声明为 `ctx`：

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    ...
}
```

`ctx context.Context` 输入是函数处理程序中的可选参数。有关所接受处理程序签名的更多信息，请参阅the section called “Go 处理程序的有效处理程序签名”。

如果您使用 AWS SDK 调用其他服务，则多个关键区域中均需要上下文对象。例如，要正确初始化 SDK 客户端，您可以使用上下文对象加载正确的 AWS SDK 配置，如下所示：

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
```

SDK 调用自身时可能需要上下文对象作为输入。例如，该 `s3Client.PutObject` 调用接受上下文对象作为其第一个参数：

```
// Upload the receipt to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

除了 AWS SDK 请求之外，您还可以使用上下文对象进行函数监控。有关上下文对象的更多信息，请参阅the section called “上下文”。

Go 处理程序的有效处理程序签名

在 Go 中构建 Lambda 函数处理程序时，您有多个选项，但您必须遵守以下规则：

- 处理程序必须为函数。
- 处理程序可能需要 0 到 2 个参数。如果有两个参数，则第一个参数必须实现 `context.Context`。
- 处理程序可能返回 0 到 2 个参数。如果有一个返回值，则它必须实现 `error`。如果有两个返回值，则第二个值必须实现 `error`。

下面列出了有效的处理程序签名。`TIn` 和 `TOut` 表示类型与 encoding/json 标准库兼容。有关更多信息，请参阅 func Unmarshal，以了解如何反序列化这些类型。

-
```
func ()
```

- `func () error`
- `func () (TOut, error)`
- `func (TIn) error`
- `func (TIn) (TOut, error)`
- `func (context.Context) error`
- `func (context.Context) (TOut, error)`
- `func (context.Context, TIn) error`
- `func (context.Context, TIn) (TOut, error)`

在处理程序中使用 AWS SDK for Go v2

通常，您将使用 Lambda 函数与其他 AWS 资源进行交互或对其进行更新。与此类资源最简单的交互方法是使用 AWS SDK for Go v2。

> **Note**
>
> AWS SDK for Go（v1）处于维护模式，其支持的终止日期为 2025 年 7 月 31 日。我们建议您继续仅使用 AWS SDK for Go v2。

要向函数添加 SDK 依赖项，请使用适用于所需特定 SDK 客户端的 `go get` 命令。在上述示例代码中，我们使用了 `config` 库和 `s3` 库。在包含 `go.mod` 和 `main.go` 文件的目录中，通过运行以下命令添加这些依赖项：

```
go get github.com/aws/aws-sdk-go-v2/config
go get github.com/aws/aws-sdk-go-v2/service/s3
```

然后，相应地向函数的导入数据块中导入依赖项：

```
import (
```

```
    ...
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"
)
```

在处理程序中使用 SDK 时，请使用正确的设置配置客户端。最简单的方法是使用默认的凭证提供程序链。此示例说明了加载此配置的一种方法：

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    if err != nil {
        log.Printf("Failed to load AWS SDK config: %v", err)
        return err
    }
```

将此配置加载到 `cfg` 变量后，您可以将此变量传入至客户端实例化中。示例代码会按如下方法对 Amazon S3 客户端进行实例化处理：

```
// Create an S3 client
    s3Client := s3.NewFromConfig(cfg)
```

在此示例中，我们在 `init()` 函数中初始化了 Amazon S3 客户端，以免每次调用函数时都必须对其进行初始化。但问题在于，Lambda 无权在 `init()` 函数中访问上下文对象。作为解决方法，您可以模拟初始化阶段的 `context.TODO()` 传入占位符。稍后，使用客户端进行调用时，传入完整的上下文对象。此解决方法也在 the section called “在 AWS SDK 客户端初始化和调用中使用上下文” 中进行了介绍。

配置并初始化 SDK 客户端后，您可以使用它与其他 AWS 服务进行交互。示例代码按如下方式调用 Amazon S3 `PutObject` API：

```
_, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
    Bucket: &bucketName,
    Key:    &key,
    Body:   strings.NewReader(receiptContent),
})
```

评估环境变量

在处理程序代码中，您可以使用 `os.Getenv()` 方法引用任何环境变量。在此示例中，我们使用以下代码行引用已定义的 `RECEIPT_BUCKET` 环境变量：

```
// Access environment variables
    bucketName := os.Getenv("RECEIPT_BUCKET")
    if bucketName == "" {
        log.Printf("RECEIPT_BUCKET environment variable is not set")
        return fmt.Errorf("missing required environment variable RECEIPT_BUCKET")
    }
```

使用全局状态

为避免每次调用函数时创建新资源，您可以在 Lambda 函数的处理程序代码之外声明并修改全局变量。您可以在 `var` 数据块或语句中定义这些全局变量。此外，处理程序可能要声明 `init()` 函数，该函数在初始化阶段执行。`init` 方法与在 AWS Lambda 中的行为方式相同，正如在标准 Go 程序中一样。

Go Lambda 函数的代码最佳实践

在构建 Lambda 函数时，请遵循以下列表中的指南，采用最佳编码实践：

- 从核心逻辑中分离 Lambda 处理程序。这样您可以创建更容易进行单元测试的函数。
- 将依赖关系的复杂性降至最低。首选在执行环境启动时可以快速加载的更简单的框架。
- 将部署包大小精简为只包含运行时必要的部分。这样会减少调用前下载和解压缩部署程序包所需的时间。

- 利用执行环境重用来提高函数性能。连接软件开发工具包 (SDK) 客户端和函数处理程序之外的数据库，并在 `/tmp` 目录中本地缓存静态资产。由函数的同一实例处理的后续调用可重用这些资源。这样就可以通过缩短函数运行时间来节省成本。

  为了避免调用之间潜在的数据泄露，请不要使用执行环境来存储用户数据、事件或其他具有安全影响的信息。如果您的函数依赖于无法存储在处理程序的内存中的可变状态，请考虑为每个用户创建单独的函数或单独的函数版本。
- 使用 keep-alive 指令来维护持久连接。Lambda 会随着时间的推移清除空闲连接。在调用函数时尝试重用空闲连接会导致连接错误。要维护您的持久连接，请使用与运行时关联的 keep-alive 指令。有关示例，请参阅在 Node.js 中通过 Keep-Alive 重用连接。
- 使用环境变量将操作参数传递给函数。例如，您在写入 Amazon S3 存储桶时，不应对要写入的存储桶名称进行硬编码，而应将存储桶名称配置为环境变量。

- 避免在 Lambda 函数中使用递归调用，在这种情况下，函数会调用自己或启动可能再次调用该函数的进程。这可能会导致意想不到的函数调用量和升级成本。如果您看到意外的调用量，请立即将函数保留并发设置为 0 来限制对函数的所有调用，同时更新代码。
- Lambda 函数代码中不要使用非正式的非公有 API。对于 AWS Lambda 托管式运行时，Lambda 会定期为 Lambda 的内部 API 应用安全性和功能更新。这些内部 API 更新可能不能向后兼容，会导致意外后果，例如，假设您的函数依赖于这些非公有 API，则调用会失败。请参阅 API 参考以查看公开发布的 API 列表。
- 编写幂等代码。为您的函数编写幂等代码可确保以相同的方式处理重复事件。您的代码应该正确验证事件并优雅地处理重复事件。有关更多信息，请参阅如何使我的 Lambda 函数具有幂等性？。

使用 Lambda 上下文对象检索 Go 函数信息

在 Lambda，此上下文对象提供的方法和属性包含有关调用、函数和执行环境的信息。当 Lambda 运行您的函数时，它会将上下文对象传递到处理程序。要在处理程序中使用上下文对象，您可以选择将其声明为处理程序的输入参数。如果您想在处理程序中进行以下操作，则上下文对象为必需项：

- 您需要访问上下文对象提供的任何全局变量、方法或属性。如 the section called “访问调用上下文信息” 中所示，这些方法和属性对于确定调用函数的实体或测量函数调用时间等任务非常有用。
- 您需要使用 AWS SDK for Go 来调用其他服务。上下文对象是大多数调用的重要输入参数。有关更多信息，请参阅 the section called “在 AWS SDK 客户端初始化和调用中使用上下文”。

主题

上下文对象中支持的变量、方法和属性

Lambda 上下文库提供以下全局变量、方法和属性。

全局变量

- `FunctionName` – Lambda 函数的名称。
- `FunctionVersion` – 函数的版本。
- `MemoryLimitInMB` – 为函数分配的内存量。
- `LogGroupName` – 函数的日志组。
- `LogStreamName` – 函数实例的日志流。

上下文方法

- `Deadline` – 返回执行超时的日期（Unix 时间格式，以毫秒为单位）。

上下文属性

- `InvokedFunctionArn` – 用于调用函数的 Amazon Resource Name (ARN)。表明调用者是否指定了版本号或别名。
- `AwsRequestID` – 调用请求的标识符。
- `Identity` – (移动应用程序) 有关授权请求的 Amazon Cognito 身份的信息。
- `ClientContext` – (移动应用程序) 客户端应用程序提供给 Lambda 的客户端上下文。

访问调用上下文信息

Lambda 函数可以访问有关其环境和调用请求的元数据。这可以在程序包上下文出访问。如果您的处理程序将 `context.Context` 作为参数包含在内，则 Lambda 会将有关您的函数的信息插入上下文的 `Value` 属性。请注意，您需要导入 `lambdacontext` 库才能访问 `context.Context` 对象的内容。

```
package main

import (
        "context"
        "log"
        "github.com/aws/aws-lambda-go/lambda"
        "github.com/aws/aws-lambda-go/lambdacontext"
)

func CognitoHandler(ctx context.Context) {
        lc, _ := lambdacontext.FromContext(ctx)
        log.Print(lc.Identity.CognitoIdentityPoolID)
}

func main() {
        lambda.Start(CognitoHandler)
}
```

在上述示例中，`lc` 是用于使用 context 对象捕获的信息的变量，`log.Print(lc.Identity.CognitoIdentityPoolID)` 将输出该信息 (在本例中为 CognitoIdentityPoolID)。

以下示例介绍了如何使用上下文对象来监控您的 Lambda 函数完成任务所需的时间。这让您能够分析性能期望并相应地调整您的函数代码 (如果需要)。

```
package main
```

```
import (
        "context"
        "log"
        "time"
        "github.com/aws/aws-lambda-go/lambda"
)

func LongRunningHandler(ctx context.Context) (string, error) {

        deadline, _ := ctx.Deadline()
        deadline = deadline.Add(-100 * time.Millisecond)
        timeoutChannel := time.After(time.Until(deadline))

        for {

                select {

                case <- timeoutChannel:
                        return "Finished before timing out.", nil

                default:
                        log.Print("hello!")
                        time.Sleep(50 * time.Millisecond)
                }
        }
}

func main() {
        lambda.Start(LongRunningHandler)
}
```

在 AWS SDK 客户端初始化和调用中使用上下文

如果处理程序需要使用 AWS SDK for Go 来调用其他服务，请将上下文对象作为处理程序的输入包括在内。在 AWS 中，最佳实践是在大多数 AWS SDK 调用中传入上下文对象。例如，Amazon S3 `PutObject` 调用接受上下文对象 (`ctx`) 作为其第一个参数：

```
// Upload an object to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

要正确初始化 SDK 客户端，您还可以在将配置对象传递给客户端之前，使用上下文对象加载正确的配置：

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    ...
    s3Client = s3.NewFromConfig(cfg)
```

如果您想在主处理程序之外初始化 SDK 客户端（即在初始化阶段），则可以传入占位符上下文对象：

```
func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 ...
 s3Client = s3.NewFromConfig(cfg)
}
```

如果您以这种方式初始化客户端，请确保从主处理程序向 SDK 调用中传入正确的上下文对象。

使用 .zip 文件归档部署 Go Lambda 函数

您的 AWS Lambda 函数代码由脚本或编译的程序及其依赖项组成。您可以使用部署程序包将函数代码部署到 Lambda。Lambda 支持两种类型的部署程序包：容器镜像和 .zip 文件归档。

本页将介绍如何创建 .zip 文件作为 Go 运行时系统的部署包，然后使用 .zip 文件通过 AWS Management Console、AWS Command Line Interface（AWS CLI）和 AWS Serverless Application Model（AWS SAM）将函数代码部署到 AWS Lambda。

请注意，Lambda 使用 POSIX 文件权限，因此在创建 .zip 文件归档之前，您可能需要为部署包文件夹设置权限。

Sections

在 macOS 和 Linux 上创建 .zip 文件

以下步骤显示如何使用 `go build` 命令编译可执行文件并为 Lambda 创建 .zip 文件部署包。在编译代码之前，请确保您已从 GitHub 安装 lambda 程序包。此模块提供运行时系统接口的实现，用于管理 Lambda 与函数代码之间的交互。要下载此库，请运行以下命令。

```
go get github.com/aws/aws-lambda-go/lambda
```

如果您的函数使用 AWS SDK for Go，请下载标准的开发工具包模块集以及应用程序所需的任何 AWS 服务 API 客户端。要了解如何安装适用于 Go 的开发工具包，请参阅 AWS SDK for Go V2 入门。

使用提供的运行时系列

Go 的实施方式与其他托管式运行时系统不同。由于 Go 本机编译为可执行的二进制文件，因此它不需要专用的语言运行时。使用仅限操作系统的运行时（`provided` 运行时系列）将 Go 函数部署到 Lambda。

创建 .zip 部署包（macOS/Linux）

1. 在包含应用程序的 `main.go` 文件的项目目录中，编译可执行文件。请注意以下几点：
   - 可执行文件必须命名为 `bootstrap`。有关更多信息，请参阅 处理程序命名约定。

- 设置目标指令集架构。仅 OS 运行时支持 arm64 和 x86_64。
- 您可以使用可选的 `lambda.norpc` 标签排除 lambda 库的远程过程调用（RPC）组件。只有在使用已弃用的 Go 1.x 运行时系统时才需要 RPC 组件。排除 RPC 会减小部署包的大小。

对于 arm64 架构：

```
GOOS=linux GOARCH=arm64 go build -tags lambda.norpc -o bootstrap main.go
```

对于 x86_64 架构：

```
GOOS=linux GOARCH=amd64 go build -tags lambda.norpc -o bootstrap main.go
```

2. （可选）您可能需要使用 Linux 上的 `CGO_ENABLED=0` 编译程序包：

```
GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o bootstrap -tags lambda.norpc
 main.go
```

此命令为标准 C 库 (`libc`) 版本创建稳定的二进制程序包，这在 Lambda 和其他设备上可能有所不同。

3. 通过将可执行文件打包为 .zip 文件来创建部署程序包。

```
zip myFunction.zip bootstrap
```

> **Note**
>
> `bootstrap` 文件必须位于 .zip 文件的根目录中。

4. 创建函数。请注意以下几点：

- 二进制文件必须命名为 `bootstrap`，但处理程序名称可以是任何名称。有关更多信息，请参阅处理程序命名约定。
- 仅在使用 arm64 时，必须使用 `--architectures` 选项。默认值为 x86_64。
- 对于 `--role`，指定执行角色的 Amazon 资源名称（ARN）。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
```

```
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

在 Windows 上创建 .zip 文件

以下步骤演示如何从 GitHub 下载用于 Windows 的 build-lambda-zip 工具、如何编译可执行文件，以及如何创建 .zip 部署包。

> **Note**
>
> 如果您尚未完成此操作，则必须安装 git，然后将 `git` 可执行文件添加到您的 Windows %PATH% 环境变量。

在编译代码之前，请确保您已从 GitHub 安装 lambda 库。要下载此库，请运行以下命令。

```
go get github.com/aws/aws-lambda-go/lambda
```

如果您的函数使用 AWS SDK for Go，请下载标准的开发工具包模块集以及应用程序所需的任何 AWS 服务 API 客户端。要了解如何安装适用于 Go 的开发工具包，请参阅 AWS SDK for Go V2 入门。

使用提供的运行时系列

Go 的实施方式与其他托管式运行时系统不同。由于 Go 本机编译为可执行的二进制文件，因此它不需要专用的语言运行时。使用仅限操作系统的运行时（`provided` 运行时系列）将 Go 函数部署到 Lambda。

创建 .zip 部署包（Windows）

1. 从 GitHub 下载 build-lambda-zip 工具。

   ```
   go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
   ```

2. 使用来自 `GOPATH` 的工具创建 .zip 文件。如果您有 Go 的默认安装，则该工具通常在 `%USERPROFILE%\Go\bin` 中。否则，请导航到安装 Go 运行时的位置，然后执行以下任一操作：

cmd.exe

在 cmd.exe 中，根据目标指令集架构运行以下任一命令。仅 OS 运行时支持 arm64 和 x86_64。

您可以使用可选的 `lambda.norpc` 标签排除 lambda 库的远程过程调用（RPC）组件。只有在使用已弃用的 Go 1.x 运行时系统时才需要 RPC 组件。排除 RPC 会减小部署包的大小。

Example – 对于 x86_64 架构

```
set GOOS=linux
set GOARCH=amd64
set CGO_ENABLED=0
go build -tags lambda.norpc -o bootstrap main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

Example – 对于 arm64 架构

```
set GOOS=linux
set GOARCH=arm64
set CGO_ENABLED=0
go build -tags lambda.norpc -o bootstrap main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

PowerShell

在 PowerShell 中，根据目标指令集架构运行以下任一命令。仅 OS 运行时支持 arm64 和 x86_64。

您可以使用可选的 `lambda.norpc` 标签排除 lambda 库的远程过程调用（RPC）组件。只有在使用已弃用的 Go 1.x 运行时系统时才需要 RPC 组件。排除 RPC 会减小部署包的大小。

对于 x86_64 架构：

```
$env:GOOS = "linux"
$env:GOARCH = "amd64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

对于 arm64 架构：

```
$env:GOOS = "linux"
$env:GOARCH = "arm64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

3. 创建函数。请注意以下几点：

   - 二进制文件必须命名为 `bootstrap`，但处理程序名称可以是任何名称。有关更多信息，请参阅处理程序命名约定。
   - 仅在使用 arm64 时，必须使用 `--architectures` 选项。默认值为 x86_64。
   - 对于 `--role`，指定执行角色的 Amazon 资源名称（ARN）。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

使用 .zip 文件创建和更新 Go Lambda 函数

创建 .zip 部署包后，您可以用其创建新的 Lambda 函数或更新现有的 Lambda 函数。您可以使用 Lambda 控制台、AWS Command Line Interface 和 Lambda API 部署 .zip 程序包。您也可以使用 AWS Serverless Application Model（AWS SAM）和 AWS CloudFormation 创建和更新 Lambda 函数。

Lambda 的 .zip 部署包的最大大小为 250MB（已解压缩）。请注意，此限制适用于您上传的所有文件（包括任何 Lambda 层）的组合大小。

Lambda 运行时需要权限才能读取部署包中的文件。在 Linux 权限八进制表示法中，Lambda 对于不可执行文件（rw-r--r--）需要 644 个权限，对于目录和可执行文件需要 755 个权限（rwxr-xr-x）。

在 Linux 和 MacOS 中，使用 chmod 命令更改部署包中文件和目录的文件权限。例如，要为可执行文件提供正确的权限，请运行以下命令。

```
chmod 755 <filepath>
```

要在 Windows 中更改文件权限，请参阅 Microsoft Windows 文档中的 Set, View, Change, or Remove Permissions on an Object。

使用控制台通过 .zip 文件创建和更新函数

要创建新函数，必须先在控制台中创建该函数，然后上传您的 .zip 归档。要更新现有函数，请打开函数页面，然后按照相同的步骤添加更新的 .zip 文件。

如果您的 .zip 文件小于 50MB，则可以通过直接从本地计算机上传该文件来创建或更新函数。对于大于 50MB 的 .zip 文件，必须首先将您的程序包上传到 Amazon S3 存储桶。有关如何使用 AWS Management Console 将文件上传到 Amazon S3 存储桶的说明，请参阅 Amazon S3 入门。要使用 AWS CLI 上传文件，请参阅《AWS CLI 用户指南》中的移动对象。

> **Note**
>
> 您无法将现有容器映像函数转换为使用 .zip 归档。您必须创建新函数。

创建新函数（控制台）

1. 打开 Lambda 控制台的“函数”页面，然后选择创建函数。
2. 选择从头开始创作。
3. 在基本信息中，执行以下操作：
   a. 对于函数名称，输入函数的名称。
   b. 对于 Runtime (运行时)，请选择 `provided.al2023`。
4. （可选）在 Permissions（权限）下，展开 Change default execution role（更改默认执行角色）。您可以创建新的执行角色，也可以使用现有角色。
5. 选择 Create function (创建函数)。Lambda 使用您选择的运行时系统创建基本“Hello world”函数。

从本地计算机上传 .zip 归档（控制台）

1. 在 Lambda 控制台的“函数”页面中，选择要为其上传 .zip 文件的函数。
2. 选择代码选项卡。

3. 在代码源窗格中，选择上传自。
4. 选择 .zip 文件。
5. 要上传 .zip 文件，请执行以下操作：
   a. 选择上传，然后在文件选择器中选择您的 .zip 文件。
   b. 选择打开。
   c. 选择保存。

从 Amazon S3 存储桶上传 .zip 归档（控制台）

1. 在 Lambda 控制台的“函数”页面中，选择要为其上传新 .zip 文件的函数。
2. 选择代码选项卡。
3. 在代码源窗格中，选择上传自。
4. 选择 Amazon S3 位置。
5. 粘贴 .zip 文件的 Amazon S3 链接 URL，然后选择保存。

使用 AWS CLI 通过 .zip 文件创建和更新函数

您可以使用 AWS CLI 创建新函数或使用 .zip 文件更新现有函数。使用 create-function 和 update-function-code 命令部署 .zip 程序包。如果您的 .zip 文件小于 50MB，则可以从本地生成计算机上的文件位置上传 .zip 程序包。对于较大的文件，必须从 Amazon S3 存储桶上传 .zip 程序包。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

> **Note**
>
> 如果您使用 AWS CLI 从 Amazon S3 存储桶上传 .zip 文件，则该存储桶必须与您的函数位于同一个 AWS 区域 中。

要通过 AWS CLI 使用 .zip 文件创建新函数，则必须指定以下内容：

- 函数的名称 (`--function-name`)
- 函数的运行时系统 (`--runtime`)
- 函数的执行角色 (`--role`) 的 Amazon 资源名称（ARN）
- 函数代码 (`--handler`) 中处理程序方法的名称

还必须指定 .zip 文件的位置。如果 .zip 文件位于本地生成计算机上的文件夹中，请使用 `--zip-file` 选项指定文件路径，如以下示例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

要指定 .zip 文件在 Amazon S3 存储桶中的位置，请使用 `--code` 选项，如以下示例命令所示。您只需对版本控制对象使用 `S3ObjectVersion` 参数。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

要使用 CLI 更新现有函数，请使用 `--function-name` 参数指定函数的名称。您还必须指定要用于更新函数代码的 .zip 文件的位置。如果 .zip 文件位于本地生成计算机上的文件夹中，请使用 `--zip-file` 选项指定文件路径，如以下示例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

要指定 .zip 文件在 Amazon S3 存储桶中的位置，请使用 `--s3-bucket` 和 `--s3-key` 选项，如以下示例命令所示。您只需对版本控制对象使用 `--s3-object-version` 参数。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

使用 Lambda API 通过 .zip 文件创建和更新函数

要使用 .zip 文件归档创建和更新函数，请使用以下 API 操作：

- CreateFunction
- UpdateFunctionCode

使用 AWS SAM 通过 .zip 文件创建和更新函数

AWS Serverless Application Model（AWS SAM）是一个工具包，可帮助简化在 AWS 上构建和运行无服务器应用程序的过程。您可以在 YAML 或 JSON 模板中为应用程序定义资源，并使用 AWS SAM 命令行界面（AWS SAM CLI）构建、打包和部署应用程序。当您通过 AWS SAM 模板构建 Lambda 函数时，AWS SAM 会使用您的函数代码和您指定的任何依赖项自动创建 .zip 部署包或容器映像。要了解有关使用 AWS SAM 构建和部署 Lambda 函数的更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 AWS SAM 入门。

您可以使用 AWS SAM 创建使用现有 .zip 文件归档的 Lambda 函数。要使用 AWS SAM 创建 Lambda 函数，您可以将 .zip 文件保存在 Amazon S3 存储桶或生成计算机上的本地文件夹中。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

在 AWS SAM 模板中，`AWS::Serverless::Function` 资源将指定 Lambda 函数。在此资源中，设置以下属性以创建使用 .zip 文件归档的函数：

- `PackageType` – 设置为 `Zip`
- `CodeUri` – 设置为函数代码的 Amazon S3 URI、本地文件夹的路径或 FunctionCode 对象
- `Runtime` – 设置为您选择的运行时系统

使用 AWS SAM，如果 .zip 文件大于 50MB，则不需要先将其上传到 Amazon S3 存储桶。AWS SAM 可以从本地生成计算机上的某个位置上传最大允许大小为 250MB（已解压缩）的 .zip 程序包。

要了解有关在 AWS SAM 中使用 .zip 文件部署函数的更多信息，请参阅《AWS SAM 开发人员指南》中的 AWS::Serverless::Function。

示例：使用 AWS SAM，通过 provided.al2023 构建 Go 函数

1. 创建具有以下属性的 AWS SAM 模板：

   - BuildMethod：为应用程序指定编译器。使用 `go1.x`。
   - Runtime：使用 `provided.al2023`。
   - CodeUri：输入代码路径。
   - Architectures：对于 arm64 架构，使用 `[arm64]`。对于 x86_64 指令集架构，使用 `[amd64]` 或删除 `Architectures` 属性。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Metadata:
      BuildMethod: go1.x
    Properties:
      CodeUri: hello-world/ # folder where your main program resides
      Handler: bootstrap
      Runtime: provided.al2023
      Architectures: [arm64]
```

2. 使用 sam build 命令编译可执行文件。

```
sam build
```

3. 使用 sam deploy 命令将函数部署到 Lambda。

```
sam deploy --guided
```

使用 AWS CloudFormation 通过 .zip 文件创建和更新函数

您可以使用 AWS CloudFormation 创建使用 .zip 文件归档的 Lambda 函数。要从 .zip 文件创建 Lambda 函数，必须先将您的文件上传到 Amazon S3 存储桶。有关如何使用 AWS CLI 将文件上传到 Amazon S3 存储桶的说明，请参阅《AWS CLI 用户指南》中的移动对象。

在 AWS CloudFormation 模板中，`AWS::Lambda::Function` 资源将指定 Lambda 函数。在此资源中，设置以下属性以创建使用 .zip 文件归档的函数：

- `PackageType` – 设置为 `Zip`
- `Code` – 在 `S3Bucket` 和 `S3Key` 字段中输入 Amazon S3 存储桶名称和 .zip 文件名。
- `Runtime` – 设置为您选择的运行时系统

AWS CloudFormation 生成的 .zip 文件不能超过 4MB。要了解有关在 AWS CloudFormation 中使用 .zip 文件部署函数的更多信息，请参阅《AWS CloudFormation 用户指南》中的 AWS::Lambda::Function。

使用容器镜像部署 Go Lambda 函数

有两种方法可以为 Go Lambda 函数构建容器映像：

- 使用 AWS 仅限操作系统的基础镜像

  Go 的实施方式与其他托管式运行时系统不同。由于 Go 本机编译为可执行的二进制文件，因此它不需要专用的语言运行时。使用仅限操作系统的基础映像为 Lambda 构建 Go 映像。要使映像与 Lambda 兼容，您必须在映像中包含 `aws-lambda-go/lambda` 软件包。
- 使用非 AWS 基本映像

  您还可以使用其他容器注册表的备用基本映像，例如 Alpine Linux 或 Debian。您还可以使用您的组织创建的自定义映像。要使映像与 Lambda 兼容，您必须在映像中包含 `aws-lambda-go/lambda` 软件包。

> **Tip**
>
> 要缩短 Lambda 容器函数激活所需的时间，请参阅 Docker 文档中的使用多阶段构建。要构建高效的容器映像，请遵循编写 Dockerfiles 的最佳实践。

此页面介绍了如何为 Lambda 构建、测试和部署容器映像。

用于部署 Go 函数的 AWS 基础映像

Go 的实施方式与其他托管式运行时系统不同。由于 Go 本机编译为可执行的二进制文件，因此它不需要专用的语言运行时。使用仅限操作系统的基础映像将 Go 函数部署到 Lambda。

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
仅限操作系统的运行时系统	`provided.al2023`	Amazon Linux 2023	未计划	未计划	未计划
仅限操作系统的运行时系统	`provided.al2`	Amazon Linux 2	未计划	未计划	未计划

Amazon Elastic Container Registry Public Gallery：gallery.ecr.aws/lambda/provided

Go 运行时系统接口客户端

`aws-lambda-go/lambda` 程序包包括运行时接口的实施。有关如何在映像中使用 `aws-lambda-go/lambda` 的示例，请参阅 使用 AWS 仅限操作系统的基础镜像 或 使用非 AWS 基本映像。

使用 AWS 仅限操作系统的基础镜像

Go 的实施方式与其他托管式运行时系统不同。由于 Go 本机编译为可执行的二进制文件，因此它不需要专用的语言运行时。使用仅限操作系统的基础映像为 Go 函数构建容器映像。

标签	运行时	操作系统	Dockerfile	淘汰
al2023	仅限操作系统的运行时系统	Amazon Linux 2023	GitHub 上用于仅限操作系统的运行时系统的 Dockerfile	未计划
al2	仅限操作系统的运行时系统	Amazon Linux 2	GitHub 上用于仅限操作系统的运行时系统的 Dockerfile	未计划

有关这些基本映像的更多信息，请参阅 Amazon ECR Public Gallery 中的 provided。

您必须在 Go 处理程序中包含 aws-lambda-go/lambda 程序包。此程序包实施 Go 的编程模型，包括运行时系统接口。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Go
- Docker
- AWS CLI 版本 2

从 provided.al2023 基本映像创建映像

使用 **`provided.al2023`** 基本映像构建和部署 Go 函数

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir hello
cd hello
```

2. 初始化一个新的 Go 模块。

```
go mod init example.com/hello-world
```

3. 添加 lambda 库作为新模块的依赖项。

```
go get github.com/aws/aws-lambda-go/lambda
```

4. 创建一个名为 main.go 的文件，然后在文本编辑器中打开它。这是适用于 Lambda 函数的代码。您可以使用以下示例代码进行测试，也可以将其替换为您自己的代码。

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. 使用文本编辑器在项目目录中创建一个 Dockerfile。

   - 以下示例 Dockerfile 将使用多阶段构建。这样便可在每个步骤中使用不同的基本映像。您可以使用一个映像（例如 Go 基本映像）来编译代码并构建可执行二进制文件。然后，您可以在最后的 FROM 语句中使用不同的映像（例如 provided.al2023）来定义部署到 Lambda 的映像。构建过程与最终部署映像分开，因此最终映像仅包含运行应用程序所需的文件。

- 您可以使用可选的 `lambda.norpc` 标签排除 lambda 库的远程过程调用（RPC）组件。只有在使用已弃用的 Go 1.x 运行时系统时才需要 RPC 组件。排除 RPC 会减小部署包的大小。
- 请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 `root` 用户。

Example – 多阶段构建 Dockerfile

> **Note**
>
> 确保您在 Dockerfile 中指定的 Go 版本（例如 `golang:1.20`）与用于创建应用程序的 Go 版本相同。

```
FROM golang:1.20 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build with optional lambda.norpc tag
COPY main.go .
RUN go build -tags lambda.norpc -o main main.go
# Copy artifacts to a clean image
FROM public.ecr.aws/lambda/provided:al2023
COPY --from=build /helloworld/main ./main
ENTRYPOINT [ "./main" ]
```

6. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 `docker-image` 并为其提供 `test` 标签。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 该命令指定了 `--platform linux/amd64` 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 `--platform linux/arm64` 选项。

（可选）在本地测试映像

使用运行时系统接口仿真器在本地测试映像。provided.al2023 基本映像还包括运行时系统接口仿真器。

在本地计算机上运行运行时系统接口仿真器

1. 使用 docker run 命令启动 Docker 映像。请注意以下几点：
   - docker-image 是映像名称，test 是标签。
   - ./main 是您的 Dockerfile 中的 ENTRYPOINT。

   ```
   docker run -d -p 9000:8080 \
   --entrypoint /usr/local/bin/aws-lambda-rie \
   docker-image:test ./main
   ```

   此命令会将映像作为容器运行，并在 localhost:9000/2015-03-31/functions/function/invocations 创建本地端点。
2. 在新的终端窗口中，使用 curl 命令将事件发布到以下端点：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令使用空事件调用函数并返回响应。某些函数可能需要 JSON 负载。例如：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```
3. 获取容器 ID。

   ```
   docker ps
   ```
4. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

   ```
   docker kill 3766c4ab331c
   ```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 `111122223333` 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
```

```
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。
   - 将 `<ECRrepositoryUri>` 替换为复制的 `repositoryUri`。确保 URI 末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 `ImageUri`，指定之前的存储库 URI。确保 URI 末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，`--publish` 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

使用非 AWS 基本映像

您可以从非 AWS 基本映像为 Go 构建容器映像。以下步骤中的示例 Dockerfile 使用 Alpine 基本映像。

您必须在 Go 处理程序中包含 aws-lambda-go/lambda 程序包。此程序包实施 Go 的编程模型，包括运行时系统接口。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- Go

- Docker
- AWS CLI 版本 2

从备用基本映像创建映像

使用 Alpine 基本映像构建和部署 Go 函数

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir hello
cd hello
```

2. 初始化一个新的 Go 模块。

```
go mod init example.com/hello-world
```

3. 添加 lambda 库作为新模块的依赖项。

```
go get github.com/aws/aws-lambda-go/lambda
```

4. 创建一个名为 `main.go` 的文件，然后在文本编辑器中打开它。这是适用于 Lambda 函数的代码。您可以使用以下示例代码进行测试，也可以将其替换为您自己的代码。

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
```

```
 lambda.Start(handler)
}
```

5. 使用文本编辑器在项目目录中创建一个 Dockerfile。以下示例 Dockerfile 使用 Alpine 基本映像。请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 `root` 用户。

   Example Dockerfile

   > **Note**
   >
   > 确保您在 Dockerfile 中指定的 Go 版本（例如 `golang:1.20`）与用于创建应用程序的 Go 版本相同。

   ```
   FROM golang:1.20.2-alpine3.16 as build
   WORKDIR /helloworld
   # Copy dependencies list
   COPY go.mod go.sum ./
   # Build
   COPY main.go .
   RUN go build -o main main.go
   # Copy artifacts to a clean image
   FROM alpine:3.16
   COPY --from=build /helloworld/main /main
   ENTRYPOINT [ "/main" ]
   ```

6. 使用 docker build 命令构建 Docker 映像。以下示例将映像命名为 `docker-image` 并为其提供 `test` 标签。

   ```
   docker build --platform linux/amd64 -t docker-image:test .
   ```

   > **Note**
   >
   > 该命令指定了 `--platform linux/amd64` 选项，可确保无论生成计算机的架构如何，容器始终与 Lambda 执行环境兼容。如果打算使用 ARM64 指令集架构创建 Lambda 函数，请务必将命令更改为使用 `--platform linux/arm64` 选项。

（可选）在本地测试映像

使用运行时系统接口仿真器在本地测试映像。您可以将仿真器构建到映像中，也可以使用以下程序将其安装在本地计算机上。

在本地计算机上安装并运行运行时系统接口仿真器

1. 从项目目录中，运行以下命令以从 GitHub 下载运行时系统接口仿真器（x86-64 架构）并将其安装在本地计算机上。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   要安装 arm64 仿真器，请将上一条命令中的 GitHub 存储库 URL 替换为以下内容：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   要安装 arm64 模拟器，请将 `$downloadLink` 替换为以下内容：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

2. 使用 docker run 命令启动 Docker 映像。请注意以下几点：

- docker-image 是映像名称，test 是标签。
- /main 是您的 Dockerfile 中的 ENTRYPOINT。

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /main
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /main
```

此命令会将映像作为容器运行，并在 localhost:9000/2015-03-31/functions/function/invocations 创建本地端点。

> **Note**
>
> 如果为 ARM64 指令集架构创建 Docker 映像，请务必使用 --platform linux/*arm64* 选项，而不是 --platform linux/*amd64* 选项。

3. 将事件发布到本地端点。

Linux/macOS

在 Linux 和 macOS 中，运行以下 curl 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，运行以下 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令使用空事件调用函数并返回响应。如果您使用自己的函数代码而不是示例函数代码，则可能需要使用 JSON 负载调用函数。例如：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 获取容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，将 3766c4ab331c 替换为上一步中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

将映像上传到 Amazon ECR 并创建 Lambda 函数

1. 运行 get-login-password 命令，以针对 Amazon ECR 注册表进行 Docker CLI 身份验证。
   - 将 `--region` 值设置为要在其中创建 Amazon ECR 存储库的 AWS 区域。
   - 将 111122223333 替换为您的 AWS 账户 ID。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中创建存储库。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 存储库必须与 Lambda 函数位于同一 AWS 区域 内。

如果成功，您将会看到如下响应：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 从上一步的输出中复制 `repositoryUri`。
4. 运行 docker tag 命令，将本地映像作为最新版本标记到 Amazon ECR 存储库中。在此命令中：
   - `docker-image:test` 是 Docker 映像的名称和标签。这是您在 `docker build` 命令中指定的映像名称和标签。
   - 将 `<ECRrepositoryUri>` 替换为复制的 `repositoryUri`。确保 URI 末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例如：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 运行 docker push 命令，以将本地映像部署到 Amazon ECR 存储库。确保存储库 URI 末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 如果您还没有函数的执行角色，请创建执行角色。在下一步中，您需要提供角色的 Amazon 资源名称（ARN）。
7. 创建 Lambda 函数。对于 `ImageUri`，指定之前的存储库 URI。确保 URI 末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像与 Lambda 函数位于同一区域内，您就可以使用其他 AWS 账户中的映像创建函数。有关更多信息，请参阅 Amazon ECR 跨账户权限。

8. 调用函数。

```
aws lambda invoke --function-name hello-world response.json
```

应出现如下响应：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 要查看函数的输出，请检查 `response.json` 文件。

要更新函数代码，您必须再次构建映像，将新映像上传到 Amazon ECR 存储库，然后使用 update-function-code 命令将映像部署到 Lambda 函数。

Lambda 会将映像标签解析为特定的映像摘要。这意味着，如果您将用于部署函数的映像标签指向 Amazon ECR 中的新映像，则 Lambda 不会自动更新该函数以使用新映像。

要将新映像部署到相同的 Lambda 函数，即使 Amazon ECR 中的映像标签保持不变，也必须使用 update-function-code 命令。在以下示例中，`--publish` 选项使用更新的容器映像创建函数的新版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

使用 Go Lambda 函数的层

Lambda 层是包含补充代码或数据的 .zip 文件存档。层通常包含库依赖项、自定义运行时系统或配置文件。创建层涉及三个常见步骤：

1. 打包层内容。此步骤需要创建 .zip 文件存档，其中包含要在函数中使用的依赖项。
2. 在 Lambda 中创建层。
3. 将层添加到函数。

我们不建议使用层来管理用 Go 编写的 Lambda 函数的依赖项。这是因为 Go 中的 Lambda 函数编译成单个可执行文件，您在部署函数时将其提供给 Lambda。这个可执行文件包含您编译的函数代码及其所有依赖项。使用层不仅会使此过程复杂化，还会导致冷启动时间增加，因为函数需要在初始化阶段将额外的程序集手动加载到内存中。

要在 Go 处理程序中使用外部依赖项，请直接将其包含在部署包中。这样就可以简化部署过程，还可以利用内置的 Go 编译器优化。有关如何在函数中导入和使用依赖项（如适用于 Go 的 AWS SDK）的示例，请参阅 the section called “处理程序”。

Go Lambda 函数日志记录和监控

AWS Lambda 将代表您自动监控 Lambda 函数并将日志记录发送至 Amazon CloudWatch。您的 Lambda 函数带有一个 CloudWatch Logs 日志组以及函数的每个实例的日志流。Lambda 运行时环境会将每个调用的详细信息发送到日志流，然后中继函数代码的日志和其他输出。有关更多信息，请参阅将 CloudWatch Logs 日志与 Lambda 结合使用。

本页旨在介绍如何从 Lambda 函数的代码生成日志输出，并使用 AWS Command Line Interface、Lambda 控制台或 CloudWatch 控制台访问日志。

Sections

创建返回日志的函数

您可以使用 fmt 程序包中的方法或写入到 `stdout` 或 `stderr` 的任何日志记录库，从函数代码输出日志。以下示例使用日志包。

Example main.go – 日志记录

```
func handleRequest(ctx context.Context, event events.SQSEvent) (string, error) {
  // event
  eventJson, _ := json.MarshalIndent(event, "", "  ")
  log.Printf("EVENT: %s", eventJson)
  // environment variables
  log.Printf("REGION: %s", os.Getenv("AWS_REGION"))
  log.Println("ALL ENV VARS:")
  for _, element := range os.Environ() {
    log.Println(element)
  }
```

Example 日志格式

```
START RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Version: $LATEST
```

```
2020/03/27 03:40:05 EVENT: {
  "Records": [
    {
      "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
      "md5OfBody": "7b27xmplb47ff90a553787216d55d91d",
      "md5OfMessageAttributes": "",
      "attributes": {
        "ApproximateFirstReceiveTimestamp": "1523232000001",
        "ApproximateReceiveCount": "1",
        "SenderId": "123456789012",
        "SentTimestamp": "1523232000000"
      },
      ...
2020/03/27 03:40:05 AWS_LAMBDA_LOG_STREAM_NAME=2020/03/27/
[$LATEST]569cxmplc3c34c7489e6a97ad08b4419
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_NAME=blank-go-function-9DV3XMPL6XBC
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_MEMORY_SIZE=128
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_VERSION=$LATEST
2020/03/27 03:40:05 AWS_EXECUTION_ENV=AWS_Lambda_go1.x
END RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71
REPORT RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Duration: 38.66 ms Billed
 Duration: 39 ms Memory Size: 128 MB Max Memory Used: 54 MB Init Duration: 203.69 ms
XRAY TraceId: 1-5e7d7595-212fxmpl9ee07c4884191322 SegmentId: 42ffxmpl0645f474 Sampled:
 true
```

Go 运行时记录每次调用的 START、END 和 REPORT 行。报告行提供了以下详细信息：

REPORT 行数据字段

- RequestId – 调用的唯一请求 ID。
- Duration（持续时间）– 函数的处理程序方法处理事件所花费的时间。
- Billed Duration（计费持续时间）– 针对调用计费的时间量。
- Memory Size（内存大小）– 分配给函数的内存量。
- Max Memory Used（最大内存使用量）– 函数使用的内存量。如果调用共享执行环境，Lambda 会报告所有调用使用的最大内存。此行为可能会导致报告值高于预期。
- Init Duration（初始持续时间）– 对于提供的第一个请求，为运行时在处理程序方法外部加载函数和运行代码所花费的时间。
- XRAY TraceId – 对于追踪的请求，为 AWS X-Ray 追踪 ID。

- SegmentId – 对于追踪的请求，为 X-Ray 分段 ID。
- Sampled（采样）– 对于追踪的请求，为采样结果。

在 Lambda 控制台中查看日志

调用 Lambda 函数后，您可以使用 Lambda 控制台查看日志输出。

如果可以在嵌入式代码编辑器中测试代码，则可以在执行结果中找到日志。使用控制台测试功能调用函数时，可以在详细信息部分找到日志输出。

在 CloudWatch 控制台中查看日志

您可以使用 Amazon CloudWatch 控制台查看所有 Lambda 函数调用的日志。

使用 CloudWatch 控制台查看日志

1. 打开 CloudWatch 控制台的 Log groups（日志组页面）。
2. 选择您的函数 (/aws/lambda/***your-function-name***) 的日志组。
3. 创建日志流。

每个日志流对应一个函数实例。日志流会在您更新 Lambda 函数以及创建更多实例来处理多个并发调用时显示。要查找特定调用的日志，建议您使用 AWS X-Ray 检测函数。X-Ray 会在追踪中记录有关请求和日志流的详细信息。

使用 AWS Command Line Interface（AWS CLI）查看日志

AWS CLI 是一种开源工具，让您能够在命令行 Shell 中使用命令与 AWS 服务进行交互。要完成本节中的步骤，您必须拥有 AWS CLI 版本 2。

您可以通过 AWS CLI，使用 `--log-type` 命令选项检索调用的日志。响应包含一个 `LogResult` 字段，其中包含多达 4KB 来自调用的 base64 编码日志。

Example 检索日志 ID

以下示例说明如何从 `LogResult` 字段中检索名为 `my-function` 的函数的日志 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解码日志

在同一命令提示符下，使用 base64 实用程序解码日志。以下示例说明如何为 my-function 检索 base64 编码的日志。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

您应看到以下输出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64 实用程序在 Linux、macOS 和 Ubuntu on Windows 上可用。macOS 用户可能需要使用 base64 -D。

Example get-logs.sh 脚本

在同一命令提示符下，使用以下脚本下载最后五个日志事件。此脚本使用 sed 从输出文件中删除引号，并休眠 15 秒以等待日志可用。输出包括来自 Lambda 的响应，以及来自 get-log-events 命令的输出。

复制以下代码示例的内容并将其作为 get-logs.sh 保存在 Lambda 项目目录中。

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux（仅限）

在同一命令提示符下，macOS 和 Linux 用户可能需要运行以下命令以确保脚本可执行。

```
chmod -R 755 get-logs.sh
```

Example 检索最后五个日志事件

在同一命令提示符下，运行以下脚本以获取最后五个日志事件。

```
./get-logs.sh
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
```

```
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


删除日志

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期之后，日志将自动删除。

在 AWS Lambda 中检测 Go 代码

Lambda 与 AWS X-Ray 集成，以帮助您跟踪、调试和优化 Lambda 应用程序。您可以在某个请求遍历应用程序中的资源（其中可能包括 Lambda 函数和其他 AWS 服务）时，使用 X-Ray 跟踪该请求。

要将跟踪数据发送到 X-Ray，您可以使用以下两个软件开发工具包 (SDK) 库之一：

- 适用于 OpenTelemetry 的 AWS 发行版 (ADOT) – 一种安全、可供生产、支持 AWS 的 OpenTelemetry (OTel) SDK 的分发版本。
- 适用于 Go 的 AWS X-Ray 软件开发工具包 – 用于生成跟踪数据并将其发送到 X-Ray 的 SDK。

每个开发工具包均提供了将遥测数据发送到 X-Ray 服务的方法。然后，您可以使用 X-Ray 查看、筛选和获得对应用程序性能指标的洞察，从而发现问题和优化机会。

> **Important**
>
> X-Ray 和 Powertools for AWS Lambda SDK 是 AWS 提供的紧密集成的分析解决方案的一部分。ADOT Lambda Layers 是全行业通用的跟踪分析标准的一部分，该标准通常会收集更多数据，但可能不适用于所有使用案例。您可以使用任一解决方案在 X-Ray 中实现端到端跟踪。要了解有关如何在两者之间进行选择的更多信息，请参阅在 AWS Distro for Open Telemetry 和 X-Ray 开发工具包之间进行选择。

Sections

使用 ADOT 分析您的 Go 函数

ADOT 提供完全托管式 Lambda 层，这些层使用 OTel SDK，将收集遥测数据所需的一切内容打包起来。通过使用此层，您可以在不必修改任何函数代码的情况下，对您的 Lambda 函数进行分析。您

还可以将您的层配置为对 OTel 进行自定义初始化。有关更多信息，请参阅 ADOT 文档中的适用于 Lambda 上的 ADOT 收集器的自定义配置。

对于 Go 运行时，可以添加 适用于 ADOT Go 的 AWS 托管 Lambda 层以自动分析您的函数。有关如何添加此层的详细说明，请参阅 ADOT 文档中的 AWS Distro for OpenTelemetry Lambda 对 Go 的支持。

使用 X-Ray SDK 分析您的 Go 函数

要记录有关 Lambda 函数对应用程序中的其他资源进行的调用的详细信息，您还可以使用适用于 Go 的 AWS X-Ray 开发工具包。要获取开发工具包，请使用 `go get` 从 GitHub 存储库下载开发工具包：

```
go get github.com/aws/aws-xray-sdk-go
```

要检测AWS开发工具包客户端，请将客户端传递给 `xray.AWS()` 方法。然后，您可以使用该方法的 `WithContext` 版本来跟踪调用。

```
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

在添加正确的依赖项并进行必要的代码更改后，请通过 Lambda 控制台或 API 激活函数配置中的跟踪。

使用 Lambda 控制台激活跟踪

要使用控制台切换 Lambda 函数的活动跟踪，请按照以下步骤操作：

打开活跃跟踪

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration（配置），然后选择 Monitoring and operations tools（监控和操作工具）。
4. 选择编辑。
5. 在 X-Ray 下方，开启 Active tracing（活动跟踪）。

6. 选择保存。

使用 Lambda API 激活跟踪

借助 AWS CLI 或 AWS SDK 在 Lambda 函数上配置跟踪，请使用以下 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下示例 AWS CLI 命令对名为 my-function 的函数启用活跃跟踪。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

跟踪模式是发布函数版本时版本特定配置的一部分。您无法更改已发布版本上的跟踪模式。

使用 AWS CloudFormation 激活跟踪

要对 AWS CloudFormation 模板中的 `AWS::Lambda::Function` 资源激活跟踪，请使用 `TracingConfig` 属性。

Example function-inline.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

对于 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 资源，请使用 `Tracing` 属性。

Example template.yml – 跟踪配置

```
Resources:
```

```
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

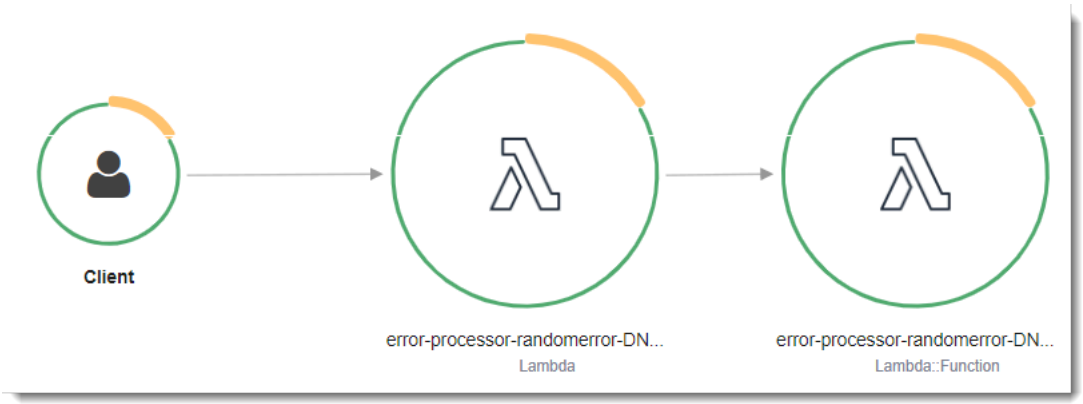
解释 X-Ray 跟踪

您的函数需要权限才能将跟踪数据上载到 X-Ray。在 Lambda 控制台中激活跟踪后，Lambda 会将所需权限添加到函数的执行角色。如果没有，请将 AWSXRayDaemonWriteAccess 策略添加到执行角色。

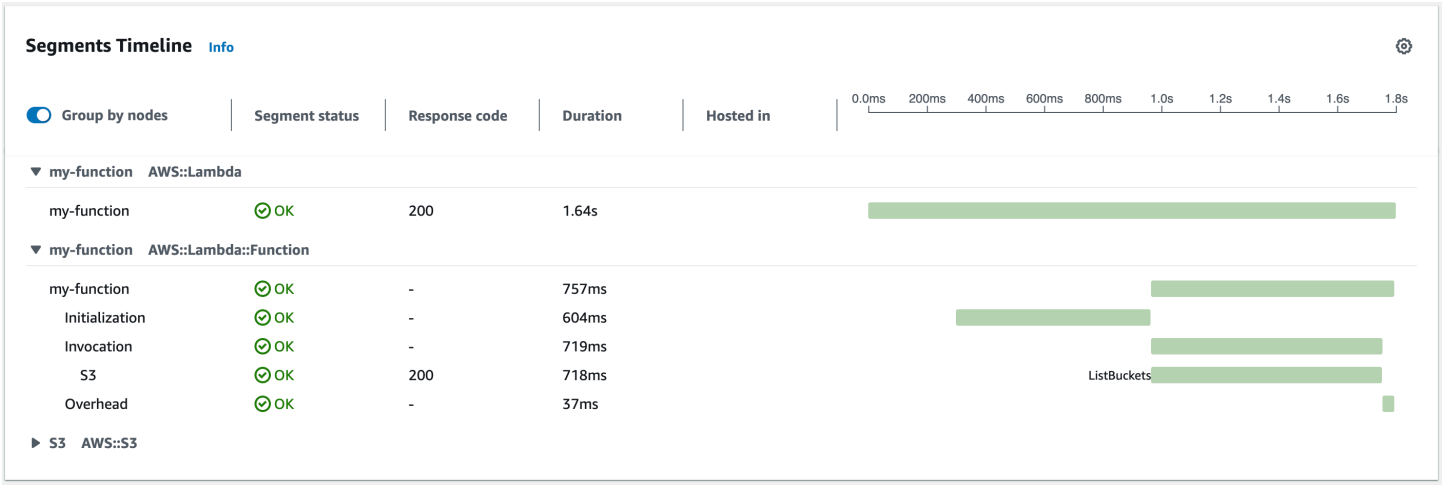
在配置活跃跟踪后，您可以通过应用程序观察特定请求。X-Ray 服务图将显示有关应用程序及其所有组件的信息。以下示例显示了具有两个函数的应用程序。主函数处理事件，有时会返回错误。位于顶部的第二个函数将处理第一个函数的日志组中显示的错误，并使用 AWS SDK 调用 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

X-Ray 无法跟踪对应用程序的所有请求。X-Ray 将应用采样算法确保跟踪有效，同时仍会提供所有请求的一个代表性样本。采样率是每秒 1 个请求和 5% 的其他请求。您无法为函数配置此 X-Ray 采样率。

在 X-Ray 中，跟踪记录有关由一个或多个服务处理的请求的信息。Lambda 会每个跟踪记录 2 个分段，这些分段将在服务图上创建两个节点。下图突出显示了这两个节点：



位于左侧的第一个节点表示接收调用请求的 Lambda 服务。第二个节点表示特定的 Lambda 函数。以下示例显示了一个包含这 2 个分段的跟踪。两者都命名为 my-function，但其中一个函数具有 AWS::Lambda 源，另一个则具有 AWS::Lambda::Function 源。如果 AWS::Lambda 分段显示错误，则表示 Lambda 服务存在问题。如果 AWS::Lambda::Function 分段显示错误，则说明函数存在问题。



此示例将展开 AWS::Lambda::Function 分段，以显示其三个子分段。

> **Note**
>
> AWS 目前正在实施对 Lambda 服务的更改。由于这些更改，您可能会看到 AWS 账户 中不同 Lambda 函数发出的系统日志消息和跟踪分段的结构和内容之间存在细微差异。
> 此处显示的示例跟踪说明了旧样式函数分段。以下段落介绍了新旧样式分段之间的差异。
> 这些更改将在未来几周内实施，除中国和 GovCloud 区域外，所有 AWS 区域 的函数都将过渡到使用新格式的日志消息和跟踪分段。

旧样式函数分段包含以下子分段：

- 初始化 – 表示加载函数和运行初始化代码所花费的时间。此子分段仅对由您的函数的每个实例处理的第一个事件显示。
- 调用 – 表示执行处理程序代码花费的时间。
- 开销 – 表示 Lambda 运行时为准备处理下一个事件而花费的时间。

新样式函数分段不包含 `Invocation` 子分段。而是将客户子分段直接附加到函数分段。有关新旧样式函数分段结构的更多信息，请参阅 the section called “了解 X-Ray 跟踪”。

您还可以分析 HTTP 客户端、记录 SQL 查询以及使用注释和元数据创建自定义子段。有关更多信息，请参阅《AWS X-Ray 开发人员指南》中的适用于 Go 的 AWS X-Ray 开发工具包。

> **定价**
>
> 作为 AWS 免费套餐的组成部分，您可以每月免费使用 X-Ray 跟踪，但不能超过一定限制。超出该阈值后，X-Ray 会对跟踪存储和检索进行收费。有关更多信息，请参阅 AWS X-Ray 定价。

使用 C# 构建 Lambda 函数

您可以使用托管的 .NET 6 或 .NET 8 运行时系统、自定义运行时系统或容器映像在 Lambda 中运行您的 .NET 应用程序。编译应用程序代码后，您可以将其作为 .zip 文件或容器映像部署到 Lambda。Lambda 为.NET 语言提供以下运行时系统：

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
.NET 8	`dotnet8`	Amazon Linux 2023	未计划	未计划	未计划
.NET 6	`dotnet6`	Amazon Linux 2	2024 年 12 月 20 日	2025 年 2 月 28 日	2025 年 3 月 31 日

设置 .NET 开发环境

要开发和构建 Lambda 函数，您可以使用任何常用的 .NET 集成式开发环境（IDE），包括 Microsoft Visual Studio、Visual Studio Code 和 JetBrains Rider。为了简化您的开发体验，AWS 提供了一组 .NET 项目模板以及 `Amazon.Lambda.Tools` 命令行界面（CLI）。

运行以下 .NET CLI 命令来安装这些项目模板和命令行工具。

安装 .NET 项目模板

安装项目模板（.NET 8）：

```
dotnet new install Amazon.Lambda.Templates
```

要安装项目模板（.NET 6）：

```
dotnet new --install Amazon.Lambda.Templates
```

> **Note**
>
> 如果您使用的是.NET 6 托管 Lambda 运行时系统，我们建议您升级为使用 .NET 8。要了解更多信息，请参阅 AWS 计算博客上的管理 AWS Lambda 运行时系统升级和适用于 AWS Lambda 的 .NET 8 运行时系统简介。

安装和更新 CLI 工具

运行以下命令来安装、更新和卸载 Amazon.Lambda.Tools CLI。

要安装命令行工具：

```
dotnet tool install -g Amazon.Lambda.Tools
```

要更新命令行工具：

```
dotnet tool update -g Amazon.Lambda.Tools
```

要卸载命令行工具：

```
dotnet tool uninstall -g Amazon.Lambda.Tools
```

定义采用 C# 的 Lambda 函数处理程序

Lambda 函数处理程序是函数代码中处理事件的方法。当调用函数时，Lambda 运行处理程序方法。您的函数会一直运行，直到处理程序返回响应、退出或超时。

当您的函数被调用且 Lambda 运行了函数的处理程序方法时，它会向您的函数传递两个参数。第一个参数是 `event` 对象。当另一个 AWS 服务 调用您的函数时，`event` 对象包含有关导致您的函数被调用的事件的数据。例如，来自 API Gateway 的 `event` 对象包含有关路径、HTTP 方法和 HTTP 标头的信息。确切的事件结构因调用函数的 AWS 服务 不同而有所不同。有关各个服务的事件格式的更多信息，请参阅 与其他服务集成。

Lambda 还会将 `context` 对象传递给函数。此对象包含有关调用、函数和执行环境的信息。有关更多信息，请参阅 the section called “上下文”。

所有 Lambda 事件的本机格式都是表示 JSON 格式的事件的字节流。除非函数输入和输出参数的类型为 `System.IO.Stream`，否则您需要对这些参数执行序列化。通过设置程序 `LambdaSerializer` 集属性来指定要使用的序列化器。有关更多信息，请参阅 the section called “Lambda 函数中的序列化”。

主题

- Lambda 的 .NET 执行模型
- 类库处理程序
- 可执行程序集处理程序
- Lambda 函数中的序列化
- 使用 Lambda 注释框架简化函数代码
- Lambda 函数处理程序限制
- C# Lambda 函数的代码最佳实践

Lambda 的 .NET 执行模型

在 .NET 中运行 Lambda 函数有两种不同的执行模型：类库方法和可执行程序集方法。

在类库方法中，您可以向 Lambda 提供一个字符串，以指示要调用的函数的 `AssemblyName`、`ClassName` 和 `Method`。有关此字符串的格式的更多信息，请参阅 the section called “类库处理程序”。在函数的初始化阶段，会初始化函数的类，构造函数中的所有代码都会运行。

在可执行程序集方法中，使用 C# 9 的顶级语句功能。这种方法会生成一个可执行程序集，每当 Lambda 收到函数的调用命令时，它就会运行该程序集。您仅向 Lambda 提供要运行的可执行程序集的名称。

以下各节给出了这两种方法的示例函数代码。

类库处理程序

以下 Lambda 函数代码显示了使用类库方法的 Lambda 函数的处理方法（`FunctionHandler`）的示例。在此示例函数中，Lambda 从 API Gateway 接收了一个调用该函数的事件。该函数从数据库读取了一条记录，并将该记录作为 API Gateway 响应的一部分返回。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

创建 Lambda 函数时，您需要以处理程序字符串的形式向 Lambda 提供有关您的函数处理程序的信息。这会告知 Lambda 在函数被调用时运行代码中的哪个方法。在 C# 中，使用类库方法时，处理程序字符串的格式如下所示：

`ASSEMBLY::TYPE::METHOD`，其中：

- `ASSEMBLY` 是您的应用程序的 .NET 程序集文件的名称。如果您使用 `Amazon.Lambda.Tools` CLI 来构建应用程序，但未使用 .csproj 文件中的 `AssemblyName` 属性设置程序集名称，则 `ASSEMBLY` 只是 .csproj 文件的名称。
- `TYPE` 是包含 `Namespace` 和 `ClassName` 的处理程序类型的全名。
- `METHOD` 中您的代码中的函数处理程序方法的名称。

对于所示的示例代码，如果程序集被命名为 `GetProductHandler`，则处理程序字符串将为 `GetProductHandler::GetProductHandler.Function::FunctionHandler`。

可执行程序集处理程序

在以下示例中，Lambda 函数被定义为了可执行程序集。此代码中的处理程序方法被命名为 `Handler`。使用可执行程序集时，必须引导 Lambda 运行时系统。为此，请使用 `LambdaBootstrapBuilder.Create` 方法。此方法将您的函数用作处理程序的方法和要使用的 Lambda 序列化程序作为输入。

有关使用顶级语句的更多信息，请参阅 AWS 计算博客上的推出适用于 AWS Lambda 的 .NET 6 运行时系统。

```
namespace GetProductHandler;

IDatabaseRepository repo = new DatabaseRepository();

await LambdaBootstrapBuilder.Create<APIGatewayProxyRequest>(Handler, new
 DefaultLambdaJsonSerializer())
    .Build()
    .RunAsync();

async Task<APIGatewayProxyResponse> Handler(APIGatewayProxyRequest apigProxyEvent,
 ILambdaContext context)
{
    var id = apigProxyEvent.PathParameters["id"];
```

```
    var databaseRecord = await this.repo.GetById(id);

    return new APIGatewayProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = JsonSerializer.Serialize(databaseRecord)
    };
};
```

使用可执行程序集时，告知 Lambda 如何运行代码的处理程序字符串是程序集的名称。在本示例中，这将为 `GetProductHandler`。

Lambda 函数中的序列化

如果您的 Lambda 函数使用 `Stream` 对象以外的输入或输出类型，您必须向应用程序中添加一个序列化库。您可以使用 `System.Text.Json` 和 `Newtonsoft.Json` 提供的基于标准反射的序列化来实现序列化，也可以使用源代码生成的序列化来实现序列化。

使用源代码生成的序列化

源代码生成的序列化是 .NET 版本 6 及更高版本的一项功能，它允许在编译时生成序列化代码。它消除了反射需求，可以提高函数性能。要在函数中使用源代码生成的序列化，请执行以下操作：

- 创建一个继承自 `JsonSerializerContext` 的新部分类，为所有需要序列化或反序列化的类型添加 `JsonSerializable` 属性。
- 配置 `LambdaSerializer` 以使用 `SourceGeneratorLambdaJsonSerializer<T>`。
- 更新应用程序代码中的任何手动序列化或反序列化，以使用新创建的类。

以下代码显示了使用源代码生成的序列化的示例函数。

```
[assembly:
 LambdaSerializer(typeof(SourceGeneratorLambdaJsonSerializer<CustomSerializer>))]

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
```

```
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord,
 CustomSerializer.Default.Product)
        };
    }
}

[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{

}
```

> **Note**
>
> 如果您想在 Lambda 中使用本机提前编译（AOT），则必须使用源代码生成的序列化。

使用基于反射的序列化

AWS 提供了预先构建的库，使您可以快速向应用程序添加序列化。您可以使用 `Amazon.Lambda.Serialization.SystemTextJson` 或 `Amazon.Lambda.Serialization.Json` NuGet 包对此进行配置。在后台，`Amazon.Lambda.Serialization.SystemTextJson` 使用 `System.Text.Json` 执行序列化任务，`Amazon.Lambda.Serialization.Json` 使用 `Newtonsoft.Json` 程序包。

您还可以通过实施 `ILambdaSerializer` 接口（作为 `Amazon.Lambda.Core` 库的一部分提供）创建您自己的序列化库。该接口定义了两种方法：

- `T Deserialize<T>(Stream requestStream);`

  通过实施此方法，您可以将请求负载从 Invoke API 反序列化至传递到 Lambda 函数处理程序的对象中。

- `T Serialize<T>(T response, Stream responseStream);`

  通过实施此方法，您可以将从 Lambda 函数处理程序中返回的结果序列化到 Invoke API 操作返回的响应负载中。

使用 Lambda 注释框架简化函数代码

Lambda 注释是一个适用于 .NET 6 和 .NET 8 的框架，它简化了使用 C# 编写 Lambda 函数的过程。使用注释框架，您可以替换使用常规编程模型所编写的 Lambda 函数中的大部分代码。使用该框架所编写的代码使用了更简单的表达式，使您可以专注于业务逻辑。

以下示例代码展示了使用注释框架可以如何简化 Lambda 函数的编写。第一个示例显示了使用常规 Lambda 程序模型编写的代码，第二个示例显示了使用注解框架编写的等效代码。

```
public APIGatewayHttpApiV2ProxyResponse LambdaMathAdd(APIGatewayHttpApiV2ProxyRequest
 request, ILambdaContext context)
{
    if (!request.PathParameters.TryGetValue("x", out var xs))
    {
        return new APIGatewayHttpApiV2ProxyResponse
        {
            StatusCode = (int)HttpStatusCode.BadRequest
        };
    }
    if (!request.PathParameters.TryGetValue("y", out var ys))
    {
        return new APIGatewayHttpApiV2ProxyResponse
        {
            StatusCode = (int)HttpStatusCode.BadRequest
        };
    }
    var x = int.Parse(xs);
    var y = int.Parse(ys);
    return new APIGatewayHttpApiV2ProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = (x + y).ToString(),
```

```
        Headers = new Dictionary#string, string> { { "Content-Type", "text/plain" } }
    };
}
```

```
[LambdaFunction]
[HttpApi(LambdaHttpMethod.Get, "/add/{x}/{y}")]
public int Add(int x, int y)
{
    return x + y;
}
```

有关使用 Lambda 注释如何简化代码的另一个示例，请参阅 awsdocs/aws-doc-sdk-examples GitHub 存储库中的此跨服务示例应用程序。文件夹 PamApiAnnotations 在主 function.cs 文件中使用 Lambda 注释。相比之下，PamApi 文件夹包含使用常规 Lambda 编程模型编写的等效文件。

注释框架使用源代码生成器来生成从 Lambda 编程模型转换为第二个示例中所显示代码的代码。

有关如何使用适用于 .NET 的 Lambda 注释的更多信息，请参阅以下资源：

- aws/aws-lambda-dotnet GitHub 存储库。
- AWS 开发人员工具博客中的介绍 .NET 注释 Lambda 框架（预览版）。
- Amazon.Lambda.Annotations NuGet 程序包。

使用 Lambda 注释框架进行依赖关系注入

您还可以使用 Lambda 注释框架，使用您熟悉的语法向 Lambda 函数添加依赖关系注入。当您向 Startup.cs 文件添加 [LambdaStartup] 属性时，Lambda 注释框架将在编译时生成所需的代码。

```
[LambdaStartup]
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IDatabaseRepository, DatabaseRepository>();
    }
}
```

您的 Lambda 函数可以使用构造函数注入或使用 [FromServices] 属性注入到各个方法中来注入服务。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(IDatabaseRepository repo)
    {
        this._repo = repo;
    }

    [LambdaFunction]
    [HttpApi(LambdaHttpMethod.Get, "/product/{id}")]
    public async Task<Product> FunctionHandler([FromServices] IDatabaseRepository
 repository, string id)
    {
        return await this._repo.GetById(id);
    }
}
```

Lambda 函数处理程序限制

请注意，处理程序签名存在一些限制。

- 它不能是 unsafe 并且不得在处理程序签名中使用指针类型，但您可以在处理程序方法及其依赖项中使用 unsafe 上下文。有关更多信息，请参阅 Microsoft 文档网站上的不安全（C# 参考）
- 处理程序签名不得使用 params 关键字传递可变数量的参数，也不得将用于支持可变数量参数的 ArgIterator 用作输入或返回参数。
- 处理程序不能是泛型方法，例如，IList<T> Sort<T>(IList<T> input)。
- 不支持使用签名 async void 的 Async 处理程序。

C# Lambda 函数的代码最佳实践

在构建 Lambda 函数时，请遵循以下列表中的指南，采用最佳编码实践：

- 从核心逻辑中分离 Lambda 处理程序。这样您可以创建更容易进行单元测试的函数。

- 控制函数部署程序包中的依赖关系。AWS Lambda 执行环境包含许多库。Lambda 会定期更新这些库，以支持最新的功能组合和安全更新。这些更新可能会使 Lambda 函数的行为发生细微变化。要完全控制您的函数所用的依赖项，请使用部署程序包来打包所有依赖项。
- 将依赖关系的复杂性降至最低。首选在执行环境启动时可以快速加载的更简单的框架。
- 将部署程序包大小精简为只包含运行时必要的部分。这样会减少调用前下载和解压缩部署程序包所需的时间。对于用 .NET 编写的函数，请不要将整个 AWS SDK 库作为部署程序包的一部分上传。而是要根据所需的模块有选择地挑选软件开发工具包中的组件（例如 DynamoDB、Simple Storage Service (Amazon S3) 软件开发工具包模块和 Lambda 核心库）。

- 利用执行环境重用来提高函数性能。连接软件开发工具包 (SDK) 客户端和函数处理程序之外的数据库，并在 `/tmp` 目录中本地缓存静态资产。由函数的同一实例处理的后续调用可重用这些资源。这样就可以通过缩短函数运行时间来节省成本。

  为了避免调用之间潜在的数据泄露，请不要使用执行环境来存储用户数据、事件或其他具有安全影响的信息。如果您的函数依赖于无法存储在处理程序的内存中的可变状态，请考虑为每个用户创建单独的函数或单独的函数版本。
- 使用 keep-alive 指令来维护持久连接。Lambda 会随着时间的推移清除空闲连接。在调用函数时尝试重用空闲连接会导致连接错误。要维护您的持久连接，请使用与运行时关联的 keep-alive 指令。有关示例，请参阅在 Node.js 中通过 Keep-Alive 重用连接。
- 使用环境变量将操作参数传递给函数。例如，您在写入 Amazon S3 存储桶时，不应对要写入的存储桶名称进行硬编码，而应将存储桶名称配置为环境变量。
- 避免在 Lambda 函数中使用递归调用，在这种情况下，函数会调用自己或启动可能再次调用该函数的进程。这可能会导致意想不到的函数调用量和升级成本。如果您看到意外的调用量，请立即将函数保留并发设置为 `0` 来限制对函数的所有调用，同时更新代码。
- Lambda 函数代码中不要使用非正式的非公有 API。对于 AWS Lambda 托管式运行时，Lambda 会定期为 Lambda 的内部 API 应用安全性和功能更新。这些内部 API 更新可能不能向后兼容，会导致意外后果，例如，假设您的函数依赖于这些非公有 API，则调用会失败。请参阅 API 参考以查看公开发布的 API 列表。
- 编写幂等代码。为您的函数编写幂等代码可确保以相同的方式处理重复事件。您的代码应该正确验证事件并优雅地处理重复事件。有关更多信息，请参阅如何使我的 Lambda 函数具有幂等性？。

使用 .zip 文件归档构建和部署 C# Lambda 函数

.NET 部署包（.zip 文件存档），包含您的函数的已编译程序集以及其所有程序集依赖项。该程序包还包含一个 *proj*`.deps.json` 文件。这将向 .NET 运行时系统告知您的所有函数的依赖项和 *proj*`.runtimeconfig.json` 文件，后者用于配置运行时系统。

要部署单个 Lambda 函数，您可以使用 `Amazon.Lambda.Tools` .NET Lambda Global CLI。使用 `dotnet lambda deploy-function` 命令会自动创建.zip 部署包并将其部署到 Lambda。但是，建议您使用类似 AWS Serverless Application Model（AWS SAM）或 AWS Cloud Development Kit (AWS CDK) 等框架来将 .NET 应用程序部署到 AWS。

无服务器应用程序通常由 Lambda 函数和其他托管 AWS 服务 组合而成，它们共同执行特定的业务任务。AWS SAM 和 AWS CDK 简化了使用其他 AWS 服务 大规模构建和部署 Lambda 函数的过程。AWS SAM 模板规范提供了一种简单而干净的语法，用于描述构成无服务器应用程序的 Lambda 函数、API、权限、配置和其他 AWS 资源。使用 AWS CDK，您可以将云基础设施定义为代码，以帮助您借助 .NET 等现代编程语言和框架，在云中构建可靠、可扩展且成本高效的应用程序。AWS CDK 和 AWS SAM 都使用 .NET Lambda Global CLI 来打包您的函数。

尽管可以使用 .NET Core CLI 将 Lambda 层与 C# 中的函数结合使用，但我们不建议这样做。C# 中使用层的函数会在 Init 阶段 期间将共享程序集手动加载到内存中，而这可能会增加冷启动时间。您可以改为在编译时包含所有共享代码，以利用 .NET 编译器的内置优化。

您可以在以下各节中找到有关使用 AWS SAM、AWS CDK 和 .NET Lambda Global CLI 构建和部署 .NET Lambda 函数的说明。

主题

- 使用 .NET Lambda Global CLI
- 使用 AWS SAM 部署 C# Lambda 函数
- 使用 AWS CDK 部署 C# Lambda 函数
- 部署 ASP.NET 应用程序

使用 .NET Lambda Global CLI

.NET CLI 和.NET Lambda 全球工具扩展（`Amazon.Lambda.Tools`）提供了一种跨平台的方式来创建基于 .NET 的 Lambda 应用程序、将其打包并部署到 Lambda。在本节中，您将学习如何使用 .NET CLI 和 Amazon Lambda 模板创建新的 Lambda .NET 项目，以及如何使用 `Amazon.Lambda.Tools` 对其进行打包和部署

主题

先决条件

.NET 8 SDK

如果您尚未安装 .NET 8 SDK 和运行时系统，则请安装。

AWS Amazon.Lambda.Templates .NET 项目模板

要生成您的 Lambda 函数代码，请使用 Amazon.Lambda.Templates NuGet 包。要安装此模板包，请运行以下命令：

```
dotnet new install Amazon.Lambda.Templates
```

AWS Amazon.Lambda.Tools .NET Global CLI 工具

要创建您的 Lambda 函数，请使用 Amazon.Lambda.Tools .NET 全球工具扩展。要安装 Amazon.Lambda.Tools，请运行以下命令：

```
dotnet tool install -g Amazon.Lambda.Tools
```

有关 Amazon.Lambda.Tools .NET CLI 扩展的更多信息，请参阅 GitHub 上的适用于 .NET CLI 的 AWS 扩展存储库。

使用 .NET CLI 创建 .NET 项目

在 .NET CLI 中，您可以使用 `dotnet new` 命令从命令行创建 .NET 项目。Lambda 使用 `Amazon.Lambda.Templates` NuGet 软件包提供了其他模板。

安装此软件包后，运行以下命令，以查看可用模板的列表。

```
dotnet new list
```

要检查有关模板的详细信息，请使用 `help` 选项。例如，要查看有关 `lambda.EmptyFunction` 模板的详细信息，请运行以下命令。

```
dotnet new lambda.EmptyFunction --help
```

要为 .NET Lambda 函数创建基本模板，请使用 `lambda.EmptyFunction` 模板。这将创建一个简单的函数，该函数将字符串作为输入，并使用 `ToUpper` 方法将其转换为大写。此模板支持以下选项：

- `--name` – 函数的名称。
- `--region` – 要在其中创建函数的 AWS 区域。
- `--profile` – AWS SDK for .NET 凭证文件中配置文件的名称。要详细了解 .NET 中的凭证配置文件，请参阅《适用于 .NET 的 AWS SDK 开发人员指南》中的配置 AWS 凭证。

在此示例中，我们使用默认配置文件和 AWS 区域 设置创建了一个名为 `myDotnetFunction` 的新空函数：

```
dotnet new lambda.EmptyFunction --name myDotnetFunction
```

此命令在您的项目目录中创建了以下文件和目录。

```
### myDotnetFunction
    ### src
    #   ### myDotnetFunction
    #       ### Function.cs
    #       ### Readme.md
    #       ### aws-lambda-tools-defaults.json
    #       ### myDotnetFunction.csproj
    ### test
        ### myDotnetFunction.Tests
            ### FunctionTest.cs
            ### myDotnetFunction.Tests.csproj
```

在 `src/myDotnetFunction` 目录下，检查以下文件：

- aws-lambda-tools-defaults.json：这是您部署 Lambda 函数时指定命令行选项的位置。例如：

```
  "profile" : "default",
  "region" : "us-east-2",
  "configuration" : "Release",
```

```
  "function-architecture": "x86_64",
  "function-runtime":"dotnet8",
  "function-memory-size" : 256,
  "function-timeout" : 30,
  "function-handler" : "myDotnetFunction::myDotnetFunction.Function::FunctionHandler"
```

- Function.cs：您的 Lambda 处理程序函数代码。它是一个 C# 模板，该模板包含默认 `Amazon.Lambda.Core` 库和默认 `LambdaSerializer` 属性。有关序列化要求和选项的更多信息，请参阅Lambda 函数中的序列化。它还包含一个示例函数，您可以编辑该函数以应用您的 Lambda 函数代码。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted into
 a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializ

namespace myDotnetFunction;

public class Function
{

    /// <summary>
    /// A simple function that takes a string and does a ToUpper
    /// </summary#
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}
```

- myDotnetFunction.csproj：列出构成您的应用程序的文件和程序集的 MSBuild 文件。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
```

```
    <AWSProjectType>Lambda</AWSProjectType>
    <!-- This property makes the build directory similar to a publish directory and
 helps the AWS .NET Lambda Mock Test Tool find project dependencies. -->
    <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
    <!-- Generate ready to run images during publishing to improve cold start time.
 -->
    <PublishReadyToRun>true</PublishReadyToRun>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson"
 Version="2.4.0" />
  </ItemGroup>
</Project>
```

- Readme：使用此文件记录您的 Lambda 函数。

在 `myfunction/test` 目录下，检查以下文件：

- myDotnetFunction.Tests.csproj：如前所述，这是一个 MSBuild 文件，其中列出了构成您的测试项目的文件和程序集。另请注意，它包含 `Amazon.Lambda.Core` 库，因此您可以无缝集成测试函数所需的任何 Lambda 模板。

```
<Project Sdk="Microsoft.NET.Sdk">
   ...

    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0 " />
   ...
```

- FunctionTest.cs：`src` 目录中包含的相同 C# 代码模板文件。编辑此文件，以镜像您函数的生产代码并对其进行测试，然后将您的 Lambda 函数上载到生产环境。

```
using Xunit;
using Amazon.Lambda.Core;
using Amazon.Lambda.TestUtilities;

using MyFunction;

namespace MyFunction.Tests
{
    public class FunctionTest
    {
```

```
        [Fact]
        public void TestToUpperFunction()
        {

            // Invoke the lambda function and confirm the string was upper cased.
            var function = new Function();
            var context = new TestLambdaContext();
            var upperCase = function.FunctionHandler("hello world", context);

            Assert.Equal("HELLO WORLD", upperCase);
        }
    }
}
```

使用 .NET CLI 部署 .NET 项目

要构建您的部署包并将其部署到 Lambda，您可以使用 `Amazon.Lambda.Tools` CLI 工具。要使用您在前面步骤中创建的文件部署函数，请先导航到包含函数的 `.csproj` 文件的文件夹。

```
cd myDotnetFunction/src/myDotnetFunction
```

要将您的代码作为 .zip 部署包部署到 Lambda，请运行以下命令。选择您自己的函数名称。

```
dotnet lambda deploy-function myDotnetFunction
```

在部署过程中，向导会要求您选择 the section called “执行角色（函数访问其他资源的权限）”。在本示例中，选择 `lambda_basic_role`。

部署函数后，您可以在云中使用 `dotnet lambda invoke-function` 命令对其进行测试。对于 `lambda.EmptyFunction` 模板中的示例代码，您可以通过使用 `--payload` 选项传递字符串来测试您的函数。

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
```

如果您的函数已成功部署，则应看到与以下内容相似的输出。

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
Amazon Lambda Tools for .NET Core applications (5.8.0)
```

```
Project Home: https://github.com/aws/aws-extensions-for-dotnet-cli, https://github.com/
aws/aws-lambda-dotnet

Payload:
"JUST CHECKING IF EVERYTHING IS OK"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

将 Lambda 层与 .NET CLI 结合使用

> **Note**
>
> 在 C# 等编译语言中将层与函数结合使用，不一定会产生与使用 Python 等解释性语言的相同效果。由于 C# 是一种编译语言，因此函数仍然需要在初始化阶段将所有共享程序集手动加载到内存中，而这可能会增加冷启动时间。我们建议您改为在编译时包含任何共享代码，以充分利用内置编译器的优化。

.NET CLI 支持帮助您发布层并部署使用层的 C# 函数的命令。要将层发布到指定的 Amazon S3 存储桶，请在 `.csproj` 文件所在的同一目录中运行以下命令：

```
dotnet lambda publish-layer <layer_name> --layer-type runtime-package-store --s3-
bucket <s3_bucket_name>
```

然后，当您使用 .NET CLI 部署函数时，请在以下命令中指定要使用的层 ARN：

```
dotnet lambda deploy-function <function_name> --function-layers arn:aws:lambda:us-
east-1:123456789012:layer:layer-name:1
```

有关 Hello World 函数的完整示例，请参阅 blank-csharp-with-layer 示例。

使用 AWS SAM 部署 C# Lambda 函数

AWS Serverless Application Model（AWS SAM）是一个工具包，可帮助简化在 AWS 上构建和运行无服务器应用程序的过程。您可以在 YAML 或 JSON 模板中为应用程序定义资源，并使用 AWS SAM 命令行界面（AWS SAM CLI）构建、打包和部署应用程序。当您通过 AWS SAM 模板构建 Lambda

函数时，AWS SAM 会使用您的函数代码和您指定的任何依赖项自动创建 .zip 部署包或容器映像。然后，AWS SAM 使用 AWS CloudFormation 堆栈部署您的函数。要了解有关使用 AWS SAM 构建和部署 Lambda 函数的更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 AWS SAM 入门。

以下示例向您显示如何使用 AWS SAM 下载、构建和部署示例 Hellow World .NET 应用程序。此示例应用程序使用 Lambda 函数和 Amazon API Gateway 端点来实现基本的 API 后端。当您向 API Gateway 端点发送 HTTP GET 请求时，API Gateway 会调用您的 Lambda 函数。该函数返回了一条“hello world”消息，以及处理您请求的 Lambda 函数实例的 IP 地址。

当您使用 AWS SAM 构建和部署应用程序时，AWS SAM CLI 会在后台使用 `dotnet lambda package` 命令来打包各个 Lambda 函数代码包。

先决条件

.NET 8 SDK

安装 .NET 8 SDK 和运行时系统。

AWS SAM CLI 版本 1.39 或更高版本

要了解如何安装 AWS SAM CLI 的最新版本，请参阅安装 AWS SAM CLI。

部署示例 AWS SAM 应用程序

1. 使用以下命令，通过 Hello world .NET 模板初始化应用程序。

```
sam init --app-template hello-world --name sam-app \
--package-type Zip --runtime dotnet8
```

此命令在您的项目目录中创建了以下文件和目录。

```
### sam-app
    ### README.md
    ### events
    #   ### event.json
    ### omnisharp.json
    ### samconfig.toml
    ### src
    #   ### HelloWorld
    #       ### Function.cs
```

```
#       ### HelloWorld.csproj
#       ### aws-lambda-tools-defaults.json
### template.yaml
### test
    ### HelloWorld.Test
        ### FunctionTest.cs
        ### HelloWorld.Tests.csproj
```

2. 导航到包含 `template.yaml file` 的目录。此文件是一个模板，用于定义应用程序的 AWS 资源，包括您的 Lambda 函数和 API Gateway API。

```
cd sam-app
```

3. 要生成应用程序的来源，请运行以下命令。

```
sam build
```

4. 要将应用程序部署到 AWS，请运行以下命令。

```
sam deploy --guided
```

此命令使用以下一系列提示打包您的应用程序，并进行部署。要接受默认选项，请按 Enter 键。

> **Note**
>
> 对于 HelloWorldFunction 可能没有定义授权，确定执行此操作吗？，确保输入 y。

- 堆栈名称：要部署到 AWS CloudFormation 的堆栈的名称。该名称必须是您的 AWS 账户 和 AWS 区域 的唯一名称。
- AWS 区域：您要将应用程序部署到的 AWS 区域。
- 部署前确认更改：选择“是”可在 AWS SAM 部署应用程序更改之前手动查看所有更改集。如果选择“否”，AWS SAM CLI 会自动部署应用程序更改。
- 允许创建 SAM CLI IAM 角色：许多 AWS SAM 模板，包括本示例中的 Hello world 模板，都会创建 AWS Identity and Access Management（IAM）角色来授予您的 Lambda 函数访问其他 AWS 服务 的权限。选择“是”以提供部署用于创建或修改 IAM 角色的 AWS CloudFormation 堆栈的权限。
- 禁用回滚：默认情况下，如果 AWS SAM 在创建或部署堆栈的过程中遇到错误，它会将堆栈回滚到以前的版本。选择“否”接受此默认值。

- HelloWorldFunction 可能没有定义授权，确定执行此操作吗，输入 y。
- 将参数保存到 samconfig.toml：选择“是”以保存您的配置选择。将来，您可以在没有参数的情况下重新运行 `sam deploy`，以将更改部署到您的应用程序。

5. 应用程序部署完成后，CLI 会返回 Hello World Lambda 函数的 Amazon 资源名称（ARN）以及为其创建的 IAM 角色。它还会显示您的 API Gateway API 的端点。要测试应用程序，请在浏览器中打开端点。您可以看到类似以下内容的响应。

```
{"message":"hello world","location":"34.244.135.203"}
```

6. 要删除您的资源，请运行以下命令。请注意，您创建的 API 端点是可通过互联网访问的公共端点。我们建议您在测试后删除该端点。

```
sam delete
```

后续步骤

要了解有关使用 AWS SAM 借助 .NET 构建和部署 Lambda 函数的更多信息，请参阅以下资源：

- AWS Serverless Application Model（AWS SAM）开发人员指南
- 使用 AWS Lambda 和 SAM CLI 构建无服务器 .NET 应用程序

使用 AWS CDK 部署 C# Lambda 函数

AWS Cloud Development Kit (AWS CDK) 是一个开源软件开发框架，用于将云基础设施定义为使用现代编程语言和框架（如 .NET）的代码。执行 AWS CDK 项目是为了生成 AWS CloudFormation 模板，然后使用这些模板来部署您的代码。

要使用 AWS CDK 构建和部署示例 Hello world .NET 应用程序，请按照以下各节中的说明进行操作。示例应用程序实现了一个基本 API 后端，该后端由 API Gateway 端点和 Lambda 函数组成。在向端点发送 HTTP GET 请求时，API Gateway 会调用 Lambda 函数。该函数返回一条“Hello world”消息，以及处理您的请求的 Lambda 实例的 IP 地址。

先决条件

.NET 8 SDK

安装 .NET 8 SDK 和运行时系统。

AWS CDK 版本 2

要了解如何安装最新版本的 AWS CDK，请参阅 AWS Cloud Development Kit (AWS CDK) v2 开发人员指南中的 AWS CDK 入门。

部署示例 AWS CDK 应用程序

1. 为示例应用程序创建项目目录，并导航到该项目目录。

```
mkdir hello-world
cd hello-world
```

2. 通过运行以下命令来初始化新的 AWS CDK 应用程序。

```
cdk init app --language csharp
```

此命令在您的项目目录中创建以下文件和目录

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
```

3. 打开 `src` 目录并使用 .NET CLI. 创建新的 Lambda 函数。这是您将使用 AWS CDK 部署的函数。在此示例中，您将使用 `lambda.EmptyFunction` 模板创建名为 `HelloWorldLambda` 的 Hello world 函数。

```
cd src
dotnet new lambda.EmptyFunction -n HelloWorldLambda
```

完成此步骤后，您的项目目录内的目录结构应如下所示。

```
### README.md
### cdk.json
### src
```

```
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
    ### HelloWorldLambda
        ### src
        #   ### HelloWorldLambda
        #       ### Function.cs
        #       ### HelloWorldLambda.csproj
        #       ### Readme.md
        #       ### aws-lambda-tools-defaults.json
        ### test
            ### HelloWorldLambda.Tests
                ### FunctionTest.cs
                ### HelloWorldLambda.Tests.csproj
```

4. 打开 src/HelloWorld 目录中的 HelloWorldStack.cs 文件。将文件的内容替换为以下代码。

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.Logs;
using Constructs;

namespace CdkTest
{
    public class HelloWorldStack : Stack
    {
        internal HelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var buildOption = new BundlingOptions()
            {
                Image = Runtime.DOTNET_8.BundlingImage,
                User = "root",
                OutputType = BundlingOutput.ARCHIVED,
                Command = new string[]{
            "/bin/sh",
                "-c",
                " dotnet tool install -g Amazon.Lambda.Tools"+
                " && dotnet build"+
```

```
                " && dotnet lambda package --output-package /asset-output/
function.zip"
                }
            };

             var helloWorldLambdaFunction = new Function(this,
 "HelloWorldFunction", new FunctionProps
            {
                Runtime = Runtime.DOTNET_8,
                MemorySize = 1024,
                LogRetention = RetentionDays.ONE_DAY,
                Handler =
 "HelloWorldLambda::HelloWorldLambda.Function::FunctionHandler",
                Code = Code.FromAsset("./src/HelloWorldLambda/src/
HelloWorldLambda", new Amazon.CDK.AWS.S3.Assets.AssetOptions
                {
                    Bundling = buildOption
                }),
            });
        }
    }
}
```

这是用于编译和捆绑应用程序代码的代码，也是 Lambda 函数本身的定义。BundlingOptions 对象允许创建 zip 格式文件以及一组用于生成 zip 格式文件内容的命令。在这种情况下，dotnet lambda package 命令用于编译和生成 zip 格式文件。

5. 要部署应用程序，请运行以下命令。

```
cdk deploy
```

6. 使用 .NET Lambda CLI 调用已部署的 Lambda 函数。

```
dotnet lambda invoke-function HelloWorldFunction -p "hello world"
```

7. 除非您想要保留您创建的资源，否则在您完成测试后，可立即将其删除。请运行以下命令以删除您的资源。

```
cdk destroy
```

后续步骤

要了解有关使用 AWS CDK 借助 .NET 构建和部署 Lambda 函数的更多信息，请参阅以下资源：

- 在 C# 中使用 AWS CDK
- 使用 AWS CDK 构建、打包和发布 .NET C# Lambda 函数

部署 ASP.NET 应用程序

除了托管事件驱动型函数外，您还可以将 .NET 与 Lambda 结合使用来托管轻量级 ASP.NET 应用程序。您可以使用 `Amazon.Lambda.AspNetCoreServer` NuGet 包构建和部署 ASP.NET 应用程序。在本节中，您将学习如何使用 .NET Lambda CLI 工具将 ASP.NET Web API 部署到 Lambda。

主题

- 先决条件
- 将 ASP.NET Web API 部署到 Lambda
- 将 ASP.NET 最小 API 部署到 Lambda

先决条件

.NET 8 SDK

安装 .NET 8 SDK 和 ASP.NET Core 运行时系统。

Amazon.Lambda.Tools

要创建您的 Lambda 函数，请使用 Amazon.Lambda.Tools .NET 全球工具扩展。要安装 Amazon.Lambda.Tools，请运行以下命令：

```
dotnet tool install -g Amazon.Lambda.Tools
```

有关 Amazon.Lambda.Tools .NET CLI 扩展的更多信息，请参阅 GitHub 上的适用于 .NET CLI 的 AWS 扩展存储库。

Amazon.Lambda.Templates

要生成您的 Lambda 函数代码，请使用 Amazon.Lambda.Templates NuGet 包。要安装此模板包，请运行以下命令：

```
dotnet new --install Amazon.Lambda.Templates
```

将 ASP.NET Web API 部署到 Lambda

要使用 ASP.NET 部署 Web API，您可以使用 .NET Lambda 模板创建新的 Web API 项目。使用以下命令初始化新的 ASP.NET Web API 项目。在示例命令中，我们将项目命名为 `AspNetOnLambda`。

```
dotnet new serverless.AspNetCoreWebAPI -n AspNetOnLambda
```

此命令在您的项目目录中创建了以下文件和目录。

```
.
### AspNetOnLambda
    ### src
    #   ### AspNetOnLambda
    #       ### AspNetOnLambda.csproj
    #       ### Controllers
    #       #   ### ValuesController.cs
    #       ### LambdaEntryPoint.cs
    #       ### LocalEntryPoint.cs
    #       ### Readme.md
    #       ### Startup.cs
    #       ### appsettings.Development.json
    #       ### appsettings.json
    #       ### aws-lambda-tools-defaults.json
    #       ### serverless.template
    ### test
        ### AspNetOnLambda.Tests
            ### AspNetOnLambda.Tests.csproj
            ### SampleRequests
            #   ### ValuesController-Get.json
            ### ValuesControllerTests.cs
            ### appsettings.json
```

当 Lambda 调用您的函数时，它使用的入口点是 `LambdaEntryPoint.cs` 文件。由 .NET Lambda 模板创建的文件包含以下代码。

```
namespace AspNetOnLambda;

public class LambdaEntryPoint : Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction
```

```
{
    protected override void Init(IWebHostBuilder builder)
    {
        builder
            .UseStartup#Startup#();
    }

    protected override void Init(IHostBuilder builder)
    {
    }
}
```

Lambda 使用的入口点必须继承自 Amazon.Lambda.AspNetCoreServer 包中的三个基类之一。这三个基类为：

- APIGatewayProxyFunction
- APIGatewayHttpApiV2ProxyFunction
- ApplicationLoadBalancerFunction

使用提供的 .NET Lambda 模板创建 LambdaEntryPoint.cs 文件时使用的默认类是 APIGatewayProxyFunction。您在函数中所使用的基类取决于您的 Lambda 函数前面是哪个 API 层。

三个基类中的每一个都包含一个名为 FunctionHandlerAsync 的公共方法。此方法的名称将构成 Lambda 用于调用函数的处理程序字符串的一部分。FunctionHandlerAsync 方法将入站事件有效负载转换为正确的 ASP.NET 格式，将 ASP.NET 响应转换回 Lambda 响应有效负载。对于所示的示例 AspNetOnLambda 项目，处理程序字符串将如下所示。

```
AspNetOnLambda::AspNetOnLambda.LambdaEntryPoint::FunctionHandlerAsync
```

要将 API 部署到 Lambda，请运行以下命令导航到包含源代码文件的目录并使用 AWS CloudFormation 部署您的函数。

```
cd AspNetOnLambda/src/AspNetOnLambda
dotnet lambda deploy-serverless
```

> **Tip**
>
> 使用 **`dotnet lambda deploy-serverless`** 命令部署 API 时，AWS CloudFormation 会根据您在部署期间指定的堆栈名称为您的 Lambda 函数命名。要为您的 Lambda 函数指定自定义名称，请编辑 `serverless.template` 文件以向 `AWS::Serverless::Function` 资源添加 `FunctionName` 属性。要了解更多信息，请参阅《AWS CloudFormation 用户指南》中的名称类型。

将 ASP.NET 最小 API 部署到 Lambda

要将 ASP.NET 最小 API 部署到 Lambda，您可以使用 .NET Lambda 模板创建新的 API 项目。使用以下命令初始化新的最小 API 项目。在此示例中，我们将项目命名为 `MinimalApiOnLambda`。

```
dotnet new serverless.AspNetCoreMinimalAPI -n MinimalApiOnLambda
```

此命令在您的项目目录中创建以下文件和目录。

```
### MinimalApiOnLambda
    ### src
        ### MinimalApiOnLambda
            ### Controllers
            #   ### CalculatorController.cs
            ### MinimalApiOnLambda.csproj
            ### Program.cs
            ### Readme.md
            ### appsettings.Development.json
            ### appsettings.json
            ### aws-lambda-tools-defaults.json
            ### serverless.template
```

`Program.cs` 文件包含以下代码。

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();

// Add AWS Lambda support. When application is run in Lambda Kestrel is swapped out as
 the web server with Amazon.Lambda.AspNetCoreServer. This
```

```
// package will act as the webserver translating request and responses between the
 Lambda event source and ASP.NET Core.
builder.Services.AddAWSLambdaHosting(LambdaEventSource.RestApi);

var app = builder.Build();


app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.MapGet("/", () => "Welcome to running ASP.NET Core Minimal API on AWS Lambda");

app.Run();
```

要将您的最小 API 配置为在 Lambda 上运行，您可能需要对此代码进行编辑，以便正确翻译 Lambda 和 ASP.NET Core 之间的请求和响应。默认情况下，该函数是针对 REST API 事件源进行配置的。对于 HTTP API 或应用程序负载均衡器，请将 (LambdaEventSource.RestApi) 替换为以下选项之一：

- (LambdaEventSource.HttpAPi)
- (LambdaEventSource.ApplicationLoadBalancer)

要将您的最小 API 部署到 Lambda，请运行以下命令导航到包含源代码文件的目录并使用 AWS CloudFormation 部署您的函数。

```
cd MinimalApiOnLambda/src/MinimalApiOnLambda
dotnet lambda deploy-serverless
```

使用容器映像部署 .NET Lambda 函数

有三种方法可以为 .NET Lambda 函数构建容器映像：

- 使用 .NET 的 AWS 基本映像

  AWS 基本映像会预加载一个语言运行时系统、一个用于管理 Lambda 和函数代码之间交互的运行时系统接口客户端，以及一个用于本地测试的运行时系统接口仿真器。

- 使用 AWS 仅限操作系统的基础镜像

  AWS 仅限操作系统的运行时系统包含 Amazon Linux 发行版和运行时系统接口模拟器。这些镜像通常用于为编译语言（例如 Go 和 Rust）以及 Lambda 未提供基础映像的语言或语言版本（例如 Node.js 19）创建容器镜像。您也可以使用仅限操作系统的基础映像来实施自定义运行时系统。要使映像与 Lambda 兼容，您必须在映像中包含 .NET 的运行时系统接口客户端。

- 使用非 AWS 基本映像

  您还可以使用其他容器注册表的备用基本映像，例如 Alpine Linux 或 Debian。您还可以使用您的组织创建的自定义映像。要使映像与 Lambda 兼容，您必须在映像中包含 .NET 的运行时系统接口客户端。

> **Tip**
>
> 要缩短 Lambda 容器函数激活所需的时间，请参阅 Docker 文档中的使用多阶段构建。要构建高效的容器映像，请遵循编写 Dockerfiles 的最佳实践。

此页面介绍了如何为 Lambda 构建、测试和部署容器映像。

主题

AWS.NET 的基本映像

AWS 为 .NET 提供了以下基本映像：

标签	运行时	操作系统	Dockerfile	淘汰
8	.NET 8	Amazon Linux 2023	GitHub 上适用于 .NET 8 的 Dockerfile	未计划
6	.NET 6	Amazon Linux 2	GitHub 上适用于 .NET 6 的 Dockerfile	2024 年 12 月 20 日

Amazon ECR 存储库：gallery.ecr.aws/lambda/dotnet

使用 .NET 的 AWS 基本映像

先决条件

要完成本节中的步骤，您必须满足以下条件：

- .NET 开发工具包 – 以下步骤使用 .NET 8 基本映像。确保 .NET 版本与您在 Dockerfile 中指定的基本映像版本相符。
- Docker

使用基本映像创建和部署映像

在以下步骤中，您将使用 Amazon.Lambda.Templates 和 Amazon.Lambda.Tools 创建 .NET 项目。然后，构建 Docker 映像，将该映像上传到 Amazon ECR，并将其部署到 Lambda 函数。

1. 安装 Amazon.Lambda.Templates NuGet 程序包。

```
dotnet new install Amazon.Lambda.Templates
```

2. 使用 `lambda.image.EmptyFunction` 模板创建 .NET 项目。

```
dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
```

3. 导航到 *MyFunction*/src/*MyFunction* 目录。这是存储项目文件的位置。检查以下文件：

   - aws-lambda-tools-defaults.json – 此文件是您部署 Lambda 函数时指定命令行选项的位置。
   - Function.cs – 您的 Lambda 处理程序函数代码。这是一个 C# 模板，该模板包含默认 `Amazon.Lambda.Core` 库和默认 `LambdaSerializer` 属性。有关序列化要求和选项的更多

信息，请参阅 Lambda 函数中的序列化。您可以使用提供的代码进行测试，也可以将其替换为您自己的代码。

- MyFunction.csproj – 列出构成您应用程序的文件和程序集的 .NET 项目文件。
- Readme.md – 此文件包含有关示例 Lambda 函数的更多信息。

4. 检查 src/*MyFunction* 目录中的 Dockerfile。您可以使用提供的 Dockerfile 进行测试，也可以将其替换为您自己的 Dockerfile。如果您使用自己的 Dockerfile，请确保：

   - 将 FROM 属性设置为基本映像的 URI。.NET 版本必须与基本映像版本相符。
   - 将 CMD 参数设置为 Lambda 函数处理程序。这应与 aws-lambda-tools-defaults.json 中的 image-command 相符。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 root 用户。

   Example Dockerfile

   ```
   # You can also pull these images from DockerHub amazon/aws-lambda-dotnet:8
   FROM public.ecr.aws/lambda/dotnet:8

   # Copy function code to Lambda-defined environment variable
   COPY publish/* ${LAMBDA_TASK_ROOT}

   # Set the CMD to your handler (could also be done as a parameter override outside
    of the Dockerfile)
   CMD [ "MyFunction::MyFunction.Function::FunctionHandler" ]
   ```

5. 安装 Amazon.Lambda.Tools .NET Global Tool.

   ```
   dotnet tool install -g Amazon.Lambda.Tools
   ```

   如果已安装 Amazon.Lambda.Tools，请确保您使用的是最新版本。

   ```
   dotnet tool update -g Amazon.Lambda.Tools
   ```

6. 如果尚未安装，请将目录更改为 *MyFunction*/src/*MyFunction*。

   ```
   cd src/MyFunction
   ```

7. 使用 Amazon.Lambda.Tools 构建 Docker 映像，将其推送到新的 Amazon ECR 存储库，然后部署 Lambda 函数。

   对于 `--function-role`，指定函数执行角色的角色名称，而不是 Amazon 资源名称（ARN）。例如，`lambda-role`。

   ```
   dotnet lambda deploy-function MyFunction --function-role lambda-role
   ```

   有关 Amazon.Lambda.Tools .NET Global Tool 的更多信息，请参阅 GitHub 上的AWS适用于 .NET CLI 的扩展程序存储库。

8. 调用函数。

   ```
   dotnet lambda invoke-function MyFunction --payload "Testing the function"
   ```

   如果所有操作成功，您将看到以下内容：

   ```
   Payload:
   "TESTING THE FUNCTION"

   Log Tail:
   START RequestId: id Version: $LATEST
   END RequestId: id
   REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
    Size: 256 MB     Max Memory Used: 12 MB
   ```

9. 删除 Lambda 函数。

   ```
   dotnet lambda delete-function MyFunction
   ```

将备用基本映像与运行时系统接口客户端配合使用

如果使用仅限操作系统的基础映像或者备用基础映像，则必须在映像中包括运行时系统接口客户端。运行时系统接口客户端可扩展 将 Lambda 运行时 API 用于自定义运行时，用于管理 Lambda 和函数代码之间的交互。

以下示例演示如何使用非 AWS 基础映像构建 .NET 的容器映像，以及如何添加 Amazon.Lambda.RuntimeSupport 程序包，该程序包是 .NET 的 Lambda 运行时系统接口客户端。示例 Dockerfile 使用 Microsoft .NET 8 基本映像。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- .NET SDK – 以下步骤使用 .NET 8 基本映像。确保 .NET 版本与您在 Dockerfile 中指定的基本映像版本相符。
- Docker

使用备用基本映像创建和部署映像

1. 安装 Amazon.Lambda.Templates NuGet 程序包。

```
dotnet new install Amazon.Lambda.Templates
```

2. 使用 `lambda.CustomRuntimeFunction` 模板创建 .NET 项目。此模板包括 Amazon.Lambda.RuntimeSupport 程序包。

```
dotnet new lambda.CustomRuntimeFunction --name MyFunction --region us-east-1
```

3. 导航到 *MyFunction*/src/*MyFunction* 目录。这是存储项目文件的位置。检查以下文件：
   - aws-lambda-tools-defaults.json – 此文件是您部署 Lambda 函数时指定命令行选项的位置。
   - Function.cs – 该代码包含一个类，其 `Main` 方法将 `Amazon.Lambda.RuntimeSupport` 库初始化为引导。`Main` 方法是函数进程的入口点。`Main` 方法将函数处理程序封装在引导可以使用的包装器中。有关更多信息，请参阅 GitHub 存储库中的使用 Amazon.Lambda.RuntimeSupport 作为类库。
   - MyFunction.csproj – 列出构成您应用程序的文件和程序集的 .NET 项目文件。
   - Readme.md – 此文件包含有关示例 Lambda 函数的更多信息。
4. 打开 `aws-lambda-tools-defaults.json` 文件，然后添加以下各行：

```
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
```

   - package-type：将部署包定义为容器映像。
   - docker-host-build-output-dir：设置构建过程的输出目录。

Example aws-lambda-tools-defaults.json

```
{
  "Information": [
    "This file provides default values for the deployment wizard inside Visual
 Studio and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "",
  "region": "us-east-1",
  "configuration": "Release",
  "function-runtime": "provided.al2023",
  "function-memory-size": 256,
  "function-timeout": 30,
  "function-handler": "bootstrap",
  "msbuild-parameters": "--self-contained true",
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
}
```

5. 在 *MyFunction*/src/*MyFunction* 目录中创建一个 Dockerfile。以下示例 Dockerfile 使用 Microsoft .NET 基本映像而不是 AWS 基本映像。

   - 将 FROM 属性设置为基本映像标识符。.NET 版本必须与基本映像版本相符。
   - 使用 COPY 命令将函数复制到 /var/task 目录中。
   - 将 ENTRYPOINT 设置为您希望 Docker 容器在启动时运行的模块。在这种情况下，该模块是引导，它会初始化 Amazon.Lambda.RuntimeSupport 库。

   请注意，示例 Dockerfile 不包含 USER 指令。当您将容器映像部署到 Lambda 时，Lambda 会自动定义具有最低权限的默认 Linux 用户。这与标准 Docker 行为不同，标准 Docker 在未提供 USER 指令时默认为 root 用户。

   Example Dockerfile

```
# You can also pull these images from DockerHub amazon/aws-lambda-dotnet:8
```

```
FROM mcr.microsoft.com/dotnet/runtime:8.0

# Set the image's internal work directory
WORKDIR /var/task

# Copy function code to Lambda-defined environment variable
COPY "bin/Release/net8.0/linux-x64"  .

# Set the entrypoint to the bootstrap
ENTRYPOINT ["/usr/bin/dotnet", "exec", "/var/task/bootstrap.dll"]
```

6. 安装 Amazon.Lambda.Tools .NET Global Tools 扩展。

```
dotnet tool install -g Amazon.Lambda.Tools
```

如果已安装 Amazon.Lambda.Tools，请确保您使用的是最新版本。

```
dotnet tool update -g Amazon.Lambda.Tools
```

7. 使用 Amazon.Lambda.Tools 构建 Docker 映像，将其推送到新的 Amazon ECR 存储库，然后部署 Lambda 函数。

对于 `--function-role`，指定函数执行角色的角色名称，而不是 Amazon 资源名称（ARN）。例如，`lambda-role`。

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

有关 Amazon.Lambda.Tools .NET CLI 扩展程序的更多信息，请参阅 GitHub 上的适用于 .NET CLI 的 AWS 扩展程序存储库。

8. 调用函数。

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

如果所有操作成功，您将看到以下内容：

```
Payload:
"TESTING THE FUNCTION"

Log Tail:
START RequestId: id Version: $LATEST
```

```
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

9. 删除 Lambda 函数。

```
dotnet lambda delete-function MyFunction
```

将 .NET Lambda 函数代码编译为本地运行时格式

.NET 8 支持本机的提前（AOT）编译。通过本机 AOT，您可以将 Lambda 函数代码编译为本机运行时格式，从而无需在运行时编译 .NET 代码。本机 AOT 编译可以减少您在.NET 中编写的 Lambda 函数的冷启动时间。有关更多信息，请参阅 AWS 计算博客上的适用于 AWS Lambda 的 .NET 8 运行时系统简介。

Sections

Lambda 运行时

要部署使用本机 AOT 编译构建的 Lambda 函数，请使用托管的 .NET 8 Lambda 运行时系统。该运行时系统支持使用 x86_64 和 arm64 结构。

当您在不使用 AOT 的情况下部署 .NET Lambda 函数时，您的应用程序首先将编译为中间语言（IL）代码。在运行时，Lambda 运行时系统中的即时（JIT）编译器获取 IL 代码并根据需要将其编译为机器代码。借助使用原生 AOT 提前编译的 Lambda 函数，您可以在部署函数时将代码编译成机器代码，这样您就不必依赖 .NET 运行时系统或 Lambda 运行时系统中的 SDK 来在代码运行之前对其进行编译。

AOT 的一个限制是，您的应用程序代码必须在 Amazon Linux 2023（AL2023）操作系统与 .NET 8 运行时系统所用操作系统相同的环境中编译。.NET Lambda CLI 提供了使用 AL2023 映像在 Docker 容器中编译应用程序的功能。

为避免跨架构兼容性方面的潜在问题，我们强烈建议您在与您为功能配置的处理器架构相同的环境中编译代码。要详细了解跨架构编译的局限性，请参阅 Microsoft .NET 文档中的交叉编译。

先决条件

Docker

要使用本机 AOT，必须在与 .NET 8 运行时系统具有相同 AL2023 操作系统的环境中进行编译。以下各节中的 .NET CLI 命令使用 Docker 在 AL2023 环境中开发和构建 Lambda 函数。

.NET 8 SDK

本机 AOT 编译是 .NET 8 的一项功能。您必须在生成计算机上安装 .NET 8 SDK，而不仅仅是在运行时系统上。

Amazon.Lambda.Tools

要创建您的 Lambda 函数，请使用 Amazon.Lambda.Tools .NET 全球工具扩展。要安装 Amazon.Lambda.Tools，请运行以下命令：

```
dotnet tool install -g Amazon.Lambda.Tools
```

有关 Amazon.Lambda.Tools .NET CLI 扩展的更多信息，请参阅 GitHub 上的适用于 .NET CLI 的 AWS 扩展存储库。

Amazon.Lambda.Templates

要生成您的 Lambda 函数代码，请使用 Amazon.Lambda.Templates NuGet 包。要安装此模板包，请运行以下命令：

```
dotnet new install Amazon.Lambda.Templates
```

开始使用

.NET 全局 CLI 和 AWS Serverless Application Model（AWS SAM）都提供了使用本机 AOT 构建应用程序的入门模板。要构建您的第一个本机 AOT Lambda 函数，请执行以下说明中的步骤。

要初始化和部署本机 AOT 编译的 Lambda 函数

1. 使用本机 AOT 模板初始化新项目，然后导航到包含已创建的 `.cs` 和 `.csproj` 文件的目录。在此示例中，我们将函数命名为 `NativeAotSample`。

   ```
   dotnet new lambda.NativeAOT -n NativeAotSample
   cd ./NativeAotSample/src/NativeAotSample
   ```

   由本机 AOT 模板创建的 `Function.cs` 文件包含以下函数代码。

   ```
   using Amazon.Lambda.Core;
   using Amazon.Lambda.RuntimeSupport;
   using Amazon.Lambda.Serialization.SystemTextJson;
   using System.Text.Json.Serialization;
   ```

```
namespace NativeAotSample;

public class Function
{
    /// <summary>
    /// The main entry point for the Lambda function. The main function is called
 once during the Lambda init phase. It
    /// initializes the .NET Lambda runtime client passing in the function handler
 to invoke for each Lambda event and
    /// the JSON serializer to use for converting Lambda JSON format to the .NET
 types.
    /// </summary>
    private static async Task Main()
    {
        Func<string, ILambdaContext, string> handler = FunctionHandler;
        await LambdaBootstrapBuilder.Create(handler, new
 SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>())
            .Build()
            .RunAsync();
    }

    /// <summary>
    /// A simple function that takes a string and does a ToUpper.
    ///
    /// To use this handler to respond to an AWS event, reference the appropriate
 package from
    /// https://github.com/aws/aws-lambda-dotnet#events
    /// and change the string input parameter to the desired event type. When the
 event type
    /// is changed, the handler type registered in the main method needs to be
 updated and the LambdaFunctionJsonSerializerContext
    /// defined below will need the JsonSerializable updated. If the return type
 and event type are different then the
    /// LambdaFunctionJsonSerializerContext must have two JsonSerializable
 attributes, one for each type.
    ///
    // When using Native AOT extra testing with the deployed Lambda functions is
 required to ensure
    // the libraries used in the Lambda function work correctly with Native AOT. If
 a runtime
    // error occurs about missing types or methods the most likely solution will be
 to remove references to trim-unsafe
```

```
    // code or configure trimming options. This sample defaults to partial TrimMode
 because currently the AWS
    // SDK for .NET does not support trimming. This will result in a larger
 executable size, and still does not
    // guarantee runtime trimming errors won't be hit.
    /// </summary>
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public static string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}

/// <summary>
/// This class is used to register the input event and return type for the
 FunctionHandler method with the System.Text.Json source generator.
/// There must be a JsonSerializable attribute for each type used as the input and
 return type or a runtime error will occur
/// from the JSON serializer unable to find the serialization information for
 unknown types.
/// </summary>
[JsonSerializable(typeof(string))]
public partial class LambdaFunctionJsonSerializerContext : JsonSerializerContext
{
    // By using this partial class derived from JsonSerializerContext, we can
 generate reflection free JSON Serializer code at compile time
    // which can deserialize our class and properties. However, we must attribute
 this class to tell it what types to generate serialization code for.
    // See https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-
text-json-source-generation
```

本机 AOT 将您的应用程序编译成单个本机二进制文件。该二进制文件的入口点是 `static Main` 方法。在 `static Main` 内，引导 Lambda 运行时系统并设置 `FunctionHandler` 方法。作为运行时系统引导程序的一部分，使用 `new SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()` 配置源代码生成的序列化器

2. 要将您的应用程序部署到 Lambda，请确保 Docker 在您的本地环境中运行，并运行以下命令。

```
dotnet lambda deploy-function
```

在后台，.NET 全局 CLI 会下载 AL2023 Docker 映像，并在正在运行的容器中编译您的应用程序代码。在部署到 Lambda 之前，编译后的二进制文件会输出回您的本地文件系统。

3. 通过运行以下命令来测试您的函数。将 <FUNCTION_NAME> 替换为您在部署向导中为函数选择的名称。

```
dotnet lambda invoke-function <FUNCTION_NAME> --payload "hello world"
```

来自 CLI 的响应包括冷启动的性能详细信息（初始化持续时间）以及函数调用的总运行时间。

4. 要删除按照上述步骤创建的 AWS 资源，请运行以下命令。将 <FUNCTION_NAME> 替换为您在部署向导中为函数选择的名称。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

```
dotnet lambda delete-function <FUNCTION_NAME>
```

序列化

要使用本机 AOT 将函数部署到 Lambda，您的函数代码必须使用源代码生成的序列化。源代码生成器不是使用运行时系统反射来收集访问对象属性以进行序列化所需的元数据，而是生成在构建应用程序时编译的 C# 源文件。要正确配置源生成的序列化器，请确保包含了函数使用的任何输入和输出对象，以及任何自定义类型。例如，从 API Gateway REST API 接收事件并返回自定义 `Product` 类型的 Lambda 函数将包括一个定义如下的序列化器。

```
[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}
```

修剪

本机 AOT 会在编译过程中对应用程序代码进行修剪，以确保二进制文件尽可能小。与以前的 .NET 版本相比，适用于 Lambda 的 .NET 8 提供了改进的修剪支持。已增加对 Lambda 运行时系统库、AWS .NET SDK、.NET Lambda 注释和 .NET 8 本身的支持。

这些改进有可能消除生成时修剪警告，但是 .NET 永远不会完全安全。这意味着在编译步骤中，您的函数所依赖的部分库可能会被剪掉。您可以通过将 `TrimmerRootAssemblies` 定义为 `.csproj` 文件的一部分来进行管理，如以下示例所示。

```
<ItemGroup>
    <TrimmerRootAssembly Include="AWSSDK.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Handlers.AwsSdk" />
    <TrimmerRootAssembly Include="Amazon.Lambda.APIGatewayEvents" />
    <TrimmerRootAssembly Include="bootstrap" />
    <TrimmerRootAssembly Include="Shared" />
</ItemGroup>
```

请注意，当您收到修剪警告时，添加生成警告的类到 `TrimmerRootAssembly` 中可能无法解决问题。修剪警告表示，该类正在尝试访问一些直到运行时才能确定的其他类。为避免运行时系统错误，请将第二个类添加到 `TrimmerRootAssembly`。

要了解有关管理修剪警告的更多信息，请参阅 Microsoft .NET 文档中的修剪警告简介。

故障排除

错误：不支持跨操作系统本机编译。

您的 Amazon.Lambda.Tools .NET Core Global Tool 版本已过期。更新到最新版本，并重新尝试。

Docker：映像操作系统“linux”不能在此平台上使用。

系统上的 Docker 配置为使用 Windows 容器。切换到 Linux 容器以运行本机 AOT 构建环境。

有关常见错误的更多信息，请参阅 GitHub 上的 AWS NativeAOT for .NET 存储库。

使用 Lambda 上下文对象检索 C# 函数信息

Lambda 运行您的函数时，会将 context 对象传递到处理程序。此对象提供的属性包含有关调用、函数和执行环境的信息。

上下文属性

- `FunctionName` – Lambda 函数的名称。
- `FunctionVersion` – 函数的版本
- `InvokedFunctionArn` – 用于调用函数的 Amazon Resource Name (ARN)。表明调用者是否指定了版本号或别名。
- `MemoryLimitInMB` – 为函数分配的内存量。
- `AwsRequestId` – 调用请求的标识符。
- `LogGroupName` – 函数的日志组。
- `LogStreamName` – 函数实例的日志流。
- `RemainingTime` (`TimeSpan`) – 执行超时前剩余的毫秒数。
- `Identity` – （移动应用程序）授权请求的 Amazon Cognito 身份的相关信息。
- `ClientContext` – （移动应用程序）客户端应用程序提供给 Lambda 的客户端上下文。
- `Logger` 函数的记录器对象。

您可以使用 `ILambdaContext` 对象中的信息输出有关函数调用的信息，以便进行监控。以下代码提供了如何向结构化日志记录框架添加上下文信息的示例。在此示例中，该函数将 `AwsRequestId` 添加到日志输出中。如果即将到达 Lambda 函数超时，该函数还使用 `RemainingTime` 属性取消正在进行的任务。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
```

```
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        Logger.AppendKey("AwsRequestId", context.AwsRequestId);

        var id = request.PathParameters["id"];

        using var cts = new CancellationTokenSource();

        try
        {
            cts.CancelAfter(context.RemainingTime.Add(TimeSpan.FromSeconds(-1)));

            var databaseRecord = await this._repo.GetById(id, cts.Token);

            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.OK,
                Body = JsonSerializer.Serialize(databaseRecord)
            };
        }
        finally
        {
            cts.Cancel();

            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.InternalServerError,
                Body = JsonSerializer.Serialize(databaseRecord)
            };
        }
    }
}
```

C# Lambda 函数日志记录和监控

AWS Lambda 将自动监控 Lambda 函数并将日志条目发送到 Amazon CloudWatch。您的 Lambda 函数带有一个 CloudWatch Logs 日志组以及函数的每个实例的日志流。Lambda 运行时系统环境会将每次调用的详细信息以及函数代码的其他输出发送到该日志流。有关 CloudWatch Logs 的更多信息，请参阅将 CloudWatch Logs 日志与 Lambda 结合使用。

Sections

创建返回日志的函数

要从函数代码输出日志，您可以使用控制台类的方法或使用写入到 `stdout` 或 `stderr` 的任何日志记录库。

.NET 运行时记录每次调用的 `START`、`END` 和 `REPORT` 行。报告行提供了以下详细信息：

REPORT 行数据字段

- RequestId – 调用的唯一请求 ID。
- Duration（持续时间）– 函数的处理程序方法处理事件所花费的时间。
- Billed Duration（计费持续时间）– 针对调用计费的时间量。
- Memory Size（内存大小）– 分配给函数的内存量。
- Max Memory Used（最大内存使用量）– 函数使用的内存量。如果调用共享执行环境，Lambda 会报告所有调用使用的最大内存。此行为可能会导致报告值高于预期。
- Init Duration（初始持续时间）– 对于提供的第一个请求，为运行时在处理程序方法外部加载函数和运行代码所花费的时间。
- XRAY TraceId – 对于追踪的请求，为 AWS X-Ray 追踪 ID。

- SegmentId – 对于追踪的请求，为 X-Ray 分段 ID。
- Sampled（采样）– 对于追踪的请求，为采样结果。

使用日志记录工具和库

Powertools for AWS Lambda（.NET）是一个开发人员工具包，用于实施无服务器最佳实践并提高开发人员速度。日志记录实用程序提供经优化的 Lambda 日志记录程序，其中包含有关所有函数的函数上下文的附加信息，输出结构为 JSON。请使用该实用程序执行以下操作：

- 从 Lambda 上下文中捕获关键字段，冷启动并将日志记录输出结构化为 JSON
- 根据指示记录 Lambda 调用事件（默认情况下禁用）
- 通过日志采样仅针对一定百分比的调用输出所有日志（默认情况下禁用）
- 在任何时间点将其他键附加到结构化日志
- 使用自定义日志格式设置程序（自带格式设置程序），从而在与组织的日志记录 RFC 兼容的结构中输出日志

将 Powertools for AWS Lambda（.NET）和 AWS SAM 用于结构化日志记录

请按照以下步骤使用 AWS SAM，通过集成的 Powertools for AWS Lambda（.NET）模块来下载、构建和部署示例 Hello World C# 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 `hello world` 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- .NET 6 或 .NET 8
- AWS CLI 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS SAM 应用程序

1. 使用 Hello World TypeScript 模板初始化该应用程序。

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. 构建应用程序。

```
cd sam-app && sam build
```

3. 部署应用程序。

```
sam deploy --guided
```

4. 按照屏幕上的提示操作。要在交互式体验中接受提供的默认选项，请按 Enter。

> **Note**
>
> 对于 HelloWorldFunction 可能没有定义授权，确定执行此操作吗？，确保输入 y。

5. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 调用 API 端点：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

7. 要获取该函数的日志，请运行 sam logs。有关更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 使用日志。

```
sam logs --stack-name sam-app
```

该日志输出类似于以下示例：

```
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8
 2023-02-20T14:15:27.988000 INIT_START Runtime Version:
```

```
 dotnet:6.v13        Runtime Version ARN: arn:aws:lambda:ap-
southeast-2::runtime:699f346a05dae24c58c45790bc4089f252bf17dae3997e79b17d939a288aa1ec
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:28.229000
 START RequestId: bed25b38-d012-42e7-ba28-f272535fb80e Version: $LATEST
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:29.259000
 2023-02-20T14:15:29.201Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528962,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ColdStart","Unit":"Count"}],"Dimensions":
[["FunctionName"],["Service"]]}]},"FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","Service":"PowertoolsHelloWorld","ColdStart":1}
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.479000
 2023-02-20T14:15:30.479Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
 {"ColdStart":true,"XrayTraceId":"1-63f3807f-5dbcb9910c96f50742707542","CorrelationId":"d3d
a549-4d67b2fdc015","FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionVersion":"$LATEST","FunctionMemorySize":256,"FunctionArn":"arn:aws:l
southeast-2:123456789012:function:sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionRequestId":"bed25b38-d012-42e7-ba28-
f272535fb80e","Timestamp":"2023-02-20T14:15:30.4602970Z","Level":"Information","Service":"P
 world API - HTTP 200"}
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.599000
 2023-02-20T14:15:30.599Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528922,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ApiRequestCount","Unit":"Count"}],"Dimensions":
[["Service"]]}]},"Service":"PowertoolsHelloWorld","ApiRequestCount":1}
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.680000 END
 RequestId: bed25b38-d012-42e7-ba28-f272535fb80e
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.680000
 REPORT RequestId: bed25b38-d012-42e7-ba28-f272535fb80e  Duration: 2450.99 ms
  Billed Duration: 2451 ms Memory Size: 256 MB     Max Memory Used: 74 MB  Init
 Duration: 240.05 ms
XRAY TraceId: 1-63f3807f-5dbcb9910c96f50742707542       SegmentId: 16b362cd5f52cba0
```

8. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
sam delete
```

管理日志保留日期

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期结束后，日志将自动删除。要设置日志保留日期，请将以下内容添加到您的 AWS SAM 模板中：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

在 Lambda 控制台中查看日志

调用 Lambda 函数后，您可以使用 Lambda 控制台查看日志输出。

如果可以在嵌入式代码编辑器中测试代码，则可以在执行结果中找到日志。使用控制台测试功能调用函数时，可以在详细信息部分找到日志输出。

在 CloudWatch 控制台中查看日志

您可以使用 Amazon CloudWatch 控制台查看所有 Lambda 函数调用的日志。

使用 CloudWatch 控制台查看日志

1. 打开 CloudWatch 控制台的 Log groups（日志组页面）。
2. 选择您的函数 (/aws/lambda/***your-function-name***) 的日志组。
3. 创建日志流。

每个日志流对应一个函数实例。日志流会在您更新 Lambda 函数以及创建更多实例来处理多个并发调用时显示。要查找特定调用的日志，建议您使用 AWS X-Ray 检测函数。X-Ray 会在追踪中记录有关请求和日志流的详细信息。

使用 AWS Command Line Interface（AWS CLI）查看日志

AWS CLI 是一种开源工具，让您能够在命令行 Shell 中使用命令与 AWS 服务进行交互。要完成本节中的步骤，您必须拥有 AWS CLI 版本 2。

您可以通过 AWS CLI，使用 `--log-type` 命令选项检索调用的日志。响应包含一个 `LogResult` 字段，其中包含多达 4KB 来自调用的 base64 编码日志。

Example 检索日志 ID

以下示例说明如何从 LogResult 字段中检索名为 my-function 的函数的日志 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解码日志

在同一命令提示符下，使用 base64 实用程序解码日志。以下示例说明如何为 my-function 检索 base64 编码的日志。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

您应看到以下输出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64 实用程序在 Linux、macOS 和 Ubuntu on Windows 上可用。macOS 用户可能需要使用 base64 -D。

Example get-logs.sh 脚本

在同一命令提示符下，使用以下脚本下载最后五个日志事件。此脚本使用 sed 从输出文件中删除引号，并休眠 15 秒以等待日志可用。输出包括来自 Lambda 的响应，以及来自 get-log-events 命令的输出。

复制以下代码示例的内容并将其作为 get-logs.sh 保存在 Lambda 项目目录中。

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux（仅限）

在同一命令提示符下，macOS 和 Linux 用户可能需要运行以下命令以确保脚本可执行。

```
chmod -R 755 get-logs.sh
```

Example 检索最后五个日志事件

在同一命令提示符下，运行以下脚本以获取最后五个日志事件。

```
./get-logs.sh
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
```

```
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

删除日志

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期之后，日志将自动删除。

在 AWS Lambda 中检测 C# 代码

Lambda 与 AWS X-Ray 集成，以帮助您跟踪、调试和优化 Lambda 应用程序。您可以在某个请求遍历应用程序中的资源（其中可能包括 Lambda 函数和其他 AWS 服务）时，使用 X-Ray 跟踪该请求。

要将跟踪数据发送到 X-Ray，您可以使用以下三个开发工具包库之一：

- 适用于 OpenTelemetry 的 AWS 发行版 (ADOT) – 一种安全、可供生产、支持 AWS 的 OpenTelemetry (OTel) SDK 的分发版本。
- AWS X-Ray SDK for .NET – 用于生成跟踪数据并将其发送到 X-Ray 的 SDK
- Powertools for AWS Lambda（.NET） – 一个开发人员工具包，用于实施无服务器最佳实践并提高开发人员速度。

每个开发工具包均提供了将遥测数据发送到 X-Ray 服务的方法。然后，您可以使用 X-Ray 查看、筛选和获得对应用程序性能指标的洞察，从而发现问题和优化机会。

> **Important**
>
> X-Ray 和 Powertools for AWS Lambda SDK 是 AWS 提供的紧密集成的分析解决方案的一部分。ADOT Lambda Layers 是全行业通用的跟踪分析标准的一部分，该标准通常会收集更多数据，但可能不适用于所有使用案例。您可以使用任一解决方案在 X-Ray 中实现端到端跟踪。要了解有关如何在两者之间进行选择的更多信息，请参阅在 AWS Distro for Open Telemetry 和 X-Ray 开发工具包之间进行选择。

Sections

将 Powertools for AWS Lambda（.NET）和 AWS SAM 用于跟踪

请按照以下步骤使用 AWS SAM，通过集成的 Powertools for AWS Lambda（.NET）模块来下载、构建和部署示例 Hello World C# 应用程序。此应用程序实现了基本的 API 后端，并使用 Powertools 发送日志、指标和跟踪。它由 Amazon API Gateway 端点和 Lambda 函数组成。在向 API Gateway 端点发送 GET 请求时，Lambda 函数会使用嵌入式指标格式向 CloudWatch 调用、发送日志和指标，并向 AWS X-Ray 发送跟踪。该函数将返回一条 hello world 消息。

先决条件

要完成本节中的步骤，您必须满足以下条件：

- .NET 6 或 .NET 8
- AWS CLI 版本 2
- AWS SAM CLI 版本 1.75 或更高版本。如果您使用的是旧版本的 AWS SAM CLI，请参阅升级 AWS SAM CLI。

部署示例 AWS SAM 应用程序

1. 使用 Hello World TypeScript 模板初始化该应用程序。

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. 构建应用程序。

```
cd sam-app && sam build
```

3. 部署应用程序。

```
sam deploy --guided
```

4. 按照屏幕上的提示操作。要在交互式体验中接受提供的默认选项，请按 Enter。

> **Note**
>
> 对于 HelloWorldFunction 可能没有定义授权，确定执行此操作吗？，确保输入 y。

5. 获取已部署应用程序的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 调用 API 端点：

```
curl <URL_FROM_PREVIOUS_STEP>
```

如果成功，您将会看到如下响应：

```
{"message":"hello world"}
```

7. 要获取该函数的跟踪信息，请运行 sam traces。

```
sam traces
```

该跟踪输出类似于以下示例：

```
New XRay Service Graph
  Start time: 2023-02-20 23:05:16+08:00
  End time: 2023-02-20 23:05:16+08:00
  Reference Id: 0 - AWS::Lambda - sam-app-HelloWorldFunction-pNjujb7mEoew - Edges:
 [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.814
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-pNjujb7mEoew
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.429
  Reference Id: 2 - (Root) AWS::ApiGateway::Stage - sam-app/Prod - Edges: [0]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
```

```
      - fault count(5XX): 0
      - total response time: 2.839
  Reference Id: 3 - client - sam-app/Prod - Edges: [2]
   Summary_statistics:
      - total requests: 0
      - ok count(2XX): 0
      - error count(4XX): 0
      - fault count(5XX): 0
      - total response time: 0

XRay Event [revision 3] at (2023-02-20T23:05:16.521000) with id
 (1-63f38c2c-270200bf1d292a442c8e8a00) and duration (2.877s)
 - 2.839s - sam-app/Prod [HTTP: 200]
   - 2.836s - Lambda [HTTP: 200]
 - 2.814s - sam-app-HelloWorldFunction-pNjujb7mEoew [HTTP: 200]
 - 2.429s - sam-app-HelloWorldFunction-pNjujb7mEoew
   - 0.230s - Initialization
   - 2.389s - Invocation
     - 0.600s - ## FunctionHandler
       - 0.517s - Get Calling IP
   - 0.039s - Overhead
```

8. 这是一个可以通过互联网访问的公有 API 端点。我们建议您在测试后删除该端点。

```
sam delete
```

X-Ray 无法跟踪对应用程序的所有请求。X-Ray 将应用采样算法确保跟踪有效，同时仍会提供所有请求的一个代表性样本。采样率是每秒 1 个请求和 5% 的其他请求。您无法为函数配置此 X-Ray 采样率。

使用 X-Ray SDK 分析 .NET 函数

您可以使用函数代码来记录元数据并跟踪下游调用。要记录有关您的函数对其他资源和服务进行调用的详细信息，请使用 AWS X-Ray SDK for .NET。要获取开发工具包，请将 `AWSXRayRecorder` 程序包添加到您的项目文件中。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
```

```
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.SQSEvents" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="2.1.0" />
    <PackageReference Include="AWSSDK.Core" Version="3.7.103.24" />
    <PackageReference Include="AWSSDK.Lambda" Version="3.7.104.3" />
    <PackageReference Include="AWSXRayRecorder.Core" Version="2.13.0" />
    <PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.11.0" />
  </ItemGroup>
</Project>
```

有一系列 Nuget 软件包可以为 AWS SDK、实体框架和 HTTP 请求提供自动分析。要查看完整的配置选项集，请参阅《AWS X-Ray 开发人员指南》中的适用于 .NET 的 AWS X-Ray SDK。

添加所需的 Nuget 软件包后，请配置自动分析。最佳实践是在函数的处理函数之外执行此配置。这样一来，您可以利用执行环境重用来提高函数性能。在以下代码示例中，在函数构造函数中调用 `RegisterXRayForAllServices` 方法，为所有 AWS SDK 调用添加分析。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        // Add auto instrumentation for all AWS SDK calls
        // It is important to call this method before initializing any SDK clients
        AWSSDKHandler.RegisterXRayForAllServices();
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);
```

```
        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

使用 Lambda 控制台激活跟踪

要使用控制台切换 Lambda 函数的活动跟踪，请按照以下步骤操作：

打开活跃跟踪

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration（配置），然后选择 Monitoring and operations tools（监控和操作工具）。
4. 选择编辑。
5. 在 X-Ray 下方，开启 Active tracing（活动跟踪）。
6. 选择保存。

使用 Lambda API 激活跟踪

借助 AWS CLI 或 AWS SDK 在 Lambda 函数上配置跟踪，请使用以下 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下示例 AWS CLI 命令对名为 my-function 的函数启用活跃跟踪。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

跟踪模式是发布函数版本时版本特定配置的一部分。您无法更改已发布版本上的跟踪模式。

使用 AWS CloudFormation 激活跟踪

要对 AWS CloudFormation 模板中的 `AWS::Lambda::Function` 资源激活跟踪，请使用 `TracingConfig` 属性。

Example function-inline.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

对于 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 资源，请使用 `Tracing` 属性。

Example template.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

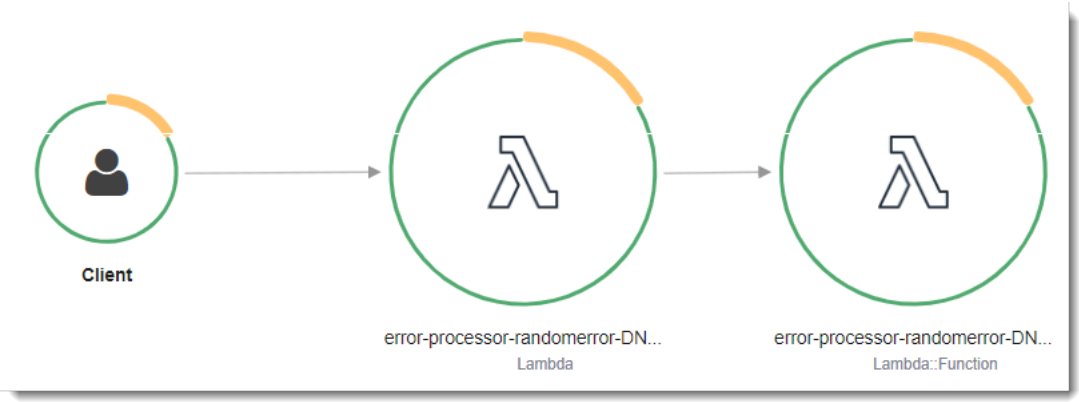
解释 X-Ray 跟踪

您的函数需要权限才能将跟踪数据上载到 X-Ray。在 Lambda 控制台中激活跟踪后，Lambda 会将所需权限添加到函数的执行角色。如果没有，请将 AWSXRayDaemonWriteAccess 策略添加到执行角色。

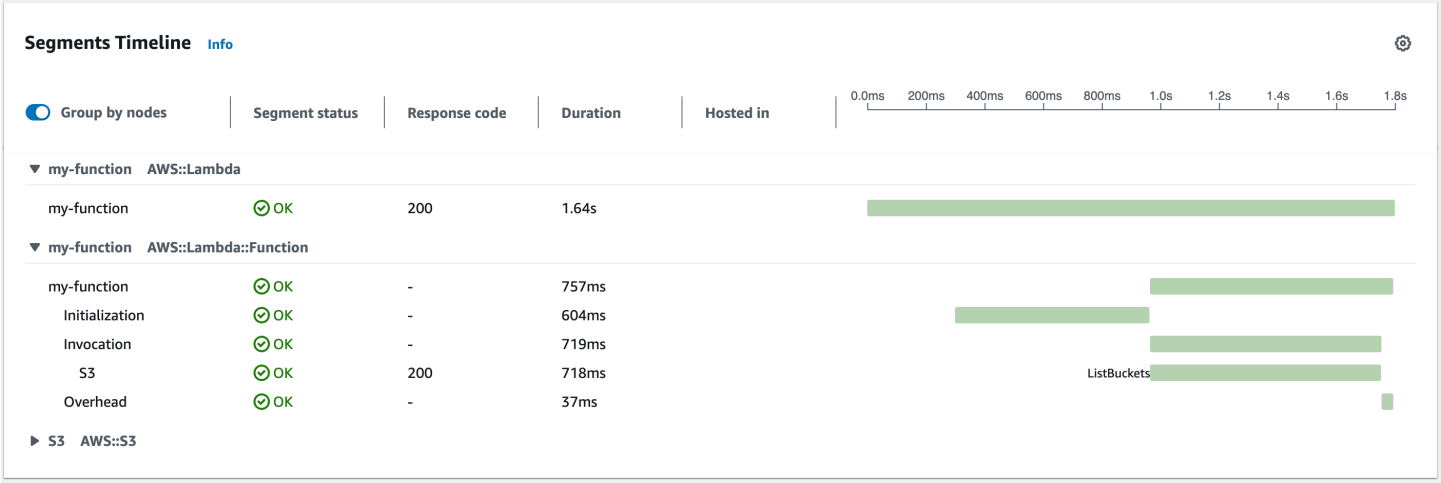
在配置活跃跟踪后，您可以通过应用程序观察特定请求。X-Ray 服务图将显示有关应用程序及其所有组件的信息。以下示例显示了具有两个函数的应用程序。主函数处理事件，有时会返回错误。位于顶部的第二个函数将处理第一个函数的日志组中显示的错误，并使用 AWS SDK 调用 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

X-Ray 无法跟踪对应用程序的所有请求。X-Ray 将应用采样算法确保跟踪有效，同时仍会提供所有请求的一个代表性样本。采样率是每秒 1 个请求和 5% 的其他请求。您无法为函数配置此 X-Ray 采样率。

在 X-Ray 中，跟踪记录有关由一个或多个服务处理的请求的信息。Lambda 会每个跟踪记录 2 个分段，这些分段将在服务图上创建两个节点。下图突出显示了这两个节点：



位于左侧的第一个节点表示接收调用请求的 Lambda 服务。第二个节点表示特定的 Lambda 函数。以下示例显示了一个包含这 2 个分段的跟踪。两者都命名为 my-function，但其中一个函数具有 `AWS::Lambda` 源，另一个则具有 `AWS::Lambda::Function` 源。如果 `AWS::Lambda` 分段显示错误，则表示 Lambda 服务存在问题。如果 `AWS::Lambda::Function` 分段显示错误，则说明函数存在问题。



此示例将展开 `AWS::Lambda::Function` 分段，以显示其三个子分段。

> **Note**
>
> AWS 目前正在实施对 Lambda 服务的更改。由于这些更改，您可能会看到 AWS 账户 中不同 Lambda 函数发出的系统日志消息和跟踪分段的结构和内容之间存在细微差异。
> 此处显示的示例跟踪说明了旧样式函数分段。以下段落介绍了新旧样式分段之间的差异。
> 这些更改将在未来几周内实施，除中国和 GovCloud 区域外，所有 AWS 区域 的函数都将过渡到使用新格式的日志消息和跟踪分段。

旧样式函数分段包含以下子分段：

- 初始化 – 表示加载函数和运行初始化代码所花费的时间。此子分段仅对由您的函数的每个实例处理的第一个事件显示。
- 调用 – 表示执行处理程序代码花费的时间。
- 开销 – 表示 Lambda 运行时为准备处理下一个事件而花费的时间。

新样式函数分段不包含 `Invocation` 子分段。而是将客户子分段直接附加到函数分段。有关新旧样式函数分段结构的更多信息，请参阅 the section called “了解 X-Ray 跟踪”。

您还可以分析 HTTP 客户端、记录 SQL 查询以及使用注释和元数据创建自定义子段。有关更多信息，请参阅 AWS X-Ray 开发人员指南中的 AWS X-Ray SDK for .NET。

定价

作为 AWS 免费套餐的组成部分，您可以每月免费使用 X-Ray 跟踪，但不能超过一定限制。超出该阈值后，X-Ray 会对跟踪存储和检索进行收费。有关更多信息，请参阅 AWS X-Ray 定价。

C# 中的 AWS Lambda 函数测试

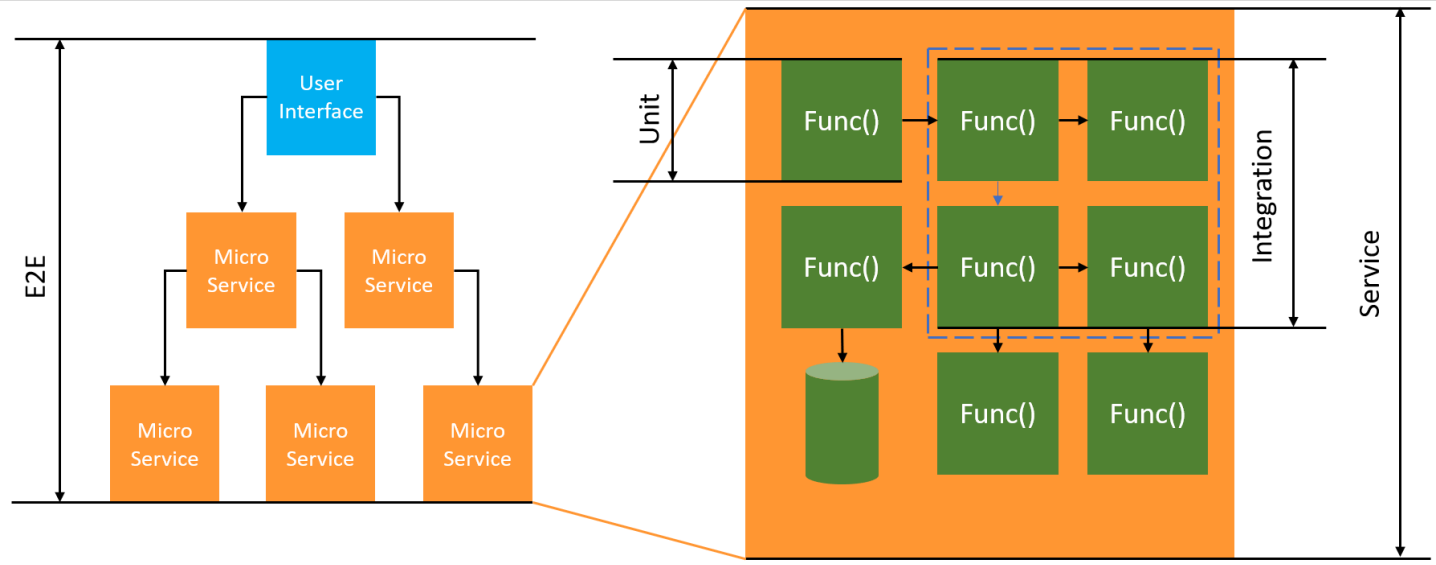
> **Note**
>
> 有关测试无服务器解决方案的技术和最佳实践的完整介绍，请参阅测试函数一章。

测试无服务器函数使用传统的测试类型和技术，但您还必须考虑对无服务器应用程序进行整体测试。基于云的测试将为您的函数和无服务器应用程序的质量提供最准确的衡量。

无服务器应用程序架构包括可通过 API 调用提供关键应用程序功能的托管服务。因此，您的开发周期应包括在函数与服务交互时验证功能的自动化测试。

如果您不创建基于云的测试，则可能会由于本地环境和已部署环境之间的差异而遇到问题。在将代码提升到下一个部署环境（例如 QA、暂存或生产）之前，您的持续集成过程应针对在云端预置的一套资源运行测试。

继续阅读本简短指南，了解无服务器应用程序的测试策略，或者访问无服务器测试示例存储库，深入了解特定于您所选语言和运行时系统的实用示例。



对于无服务器测试，您仍需要编写单元、集成和端到端测试。

- 单元测试 – 针对隔离代码块运行的测试。例如，验证业务逻辑以计算给定特定项目和目的地的配送费用。
- 集成测试 – 涉及通常在云环境中交互的两个或更多组件或服务的测试。例如，验证函数是否会处理队列中的事件。

- 端到端测试 – 验证整个应用程序行为的测试。例如，确保正确设置基础设施，并确保事件在服务之间按预期流动，以记录客户的订单。

测试无服务器应用程序

您通常会使用多种方法来测试无服务器应用程序代码，包括在云端进行测试、使用 Mock 进行测试，以及偶尔使用仿真器进行测试。

在云端进行测试

在云端进行测试对于各个阶段的测试（包括单元测试、集成测试和端到端测试）而言都很有价值。您可以针对部署在云端并与基于云的服务进行交互的代码运行测试。这种方法可以准确衡量代码的质量。

在云端调试 Lambda 函数的一种便捷方法是在控制台中使用测试事件。测试事件是函数的一个 JSON 输入。如果函数不需要输入，则事件可以是空 JSON 文档（{}）。控制台为各种服务集成提供示例事件。在控制台中创建事件后，您可以将其与团队共享，以简化测试并保持一致性。

> **Note**
>
> 在控制台中测试函数是一种快速开始的方法，但自动化测试周期可确保应用程序质量和开发速度。

测试工具

为了加速开发周期，您可以在测试函数时使用多种工具和技术。例如，AWS SAM Accelerate 和 AWS CDK 监视模式都减少了更新云环境所需的时间。

您定义 Lambda 函数代码的方式使添加单元测试变得简单。Lambda 需要公共、无参数的构造函数来初始化您的类。引入第二个内部构造函数可以让您控制应用程序使用的依赖关系。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;
```

```
    public Function(): this(null)
    {
    }

    internal Function(IDatabaseRepository repo)
    {
        this._repo = repo ?? new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

要为此函数编写测试，您可以初始化 Function 类的新实例，然后传递 IDatabaseRepository 的模拟实现。以下示例使用 XUnit、Moq 和 FluentAssertions 来编写简单的测试，以确保 FunctionHandler 返回 200 状态代码。

```
using Xunit;
using Moq;
using FluentAssertions;

public class FunctionTests
{
    [Fact]
    public async Task TestLambdaHandler_WhenInputIsValid_ShouldReturn200StatusCode()
    {
        // Arrange
        var mockDatabaseRepository = new Mock<IDatabaseRepository>();

        var functionUnderTest = new Function(mockDatabaseRepository.Object);

        // Act
```

```
        var response = await functionUnderTest.FunctionHandler(new
 APIGatewayProxyRequest());

        // Assert
        response.StatusCode.Should().Be(200);
    }
}
```

有关更详细的示例，包括异步测试的示例，请参阅 GitHub 上的 .NET 测试示例存储库。

使用 PowerShell 构建 Lambda 函数

以下部分说明在使用 PowerShell 编写 Lambda 函数代码时如何应用常见的编程模式和核心概念。

Lambda 为 PowerShell 提供了以下示例应用程序：

- blank-powershell – 此 PowerShell 函数可显示日志记录、环境变量和AWS开发工具包的使用情况。

在您开始之前，您必须先设置 PowerShell 开发环境。有关如何执行此操作的说明，请参阅 设置 PowerShell 开发环境。

要了解如何使用 AWSLambdaPSCore 模块从模板中下载示例 PowerShell 项目、创建 PowerShell 部署程序包，以及在AWS云中部署 PowerShell 函数，请参阅 使用 .zip 文件归档部署 PowerShell Lambda 函数。

Lambda 为.NET 语言提供以下运行时系统：

名称	标识符	操作系统	弃用日期	阻止函数创建	阻止函数更新
.NET 8	`dotnet8`	Amazon Linux 2023	未计划	未计划	未计划
.NET 6	`dotnet6`	Amazon Linux 2	2024 年 12 月 20 日	2025 年 2 月 28 日	2025 年 3 月 31 日

主题

设置 PowerShell 开发环境

Lambda 为 PowerShell 运行时提供一组工具和库。有关安装说明，请参阅 GitHub 上的 Lambda tools for Powershell。

AWSLambdaPSCore 模块包括以下 cmdlet，可帮助创作和发布 PowerShell Lambda 函数。

- Get-AWSPowerShellLambdaTemplate – 返回入门模板列表。
- New-AWSPowerShellLambda – 根据模板创建初始 PowerShell 脚本。
- Publish-AWSPowerShellLambda – 将给定 PowerShell 脚本发布到 Lambda。
- New-AWSPowerShellLambdaPackage – 创建 Lambda 部署程序包，可以在 CI/CD 系统中用于部署。

使用 .zip 文件归档部署 PowerShell Lambda 函数

PowerShell 运行时部署程序包包含您的 PowerShell 脚本、PowerShell 脚本所需的 PowerShell 模块，以及托管 PowerShell Core 所需的程序集。

创建 Lambda 函数

要使用 Lambda 开始编写并调用 PowerShell 脚本，您可以使用 `New-AWSPowerShellLambda` cmdlet 基于模板创建一个入门脚本。您可以使用 `Publish-AWSPowerShellLambda` cmdlet 将脚本部署到 Lambda。然后，您可以通过命令行或 Lambda 控制台测试您的脚本。

要创建、上传和测试新的 PowerShell 脚本，请执行以下操作：

1. 要查看可用模板列表，请运行以下命令：

```
PS C:\> Get-AWSPowerShellLambdaTemplate

Template               Description
--------               -----------
Basic                  Bare bones script
CodeCommitTrigger      Script to process AWS CodeCommit Triggers
...
```

2. 要基于 `Basic` 模板创建示例脚本，请运行以下命令：

```
New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
```

当前目录的新子目录中创建一个名为 `MyFirstPSScript.ps1` 的新文件。目录名称基于 `-ScriptName` 参数。您可以使用 `-Directory` 参数来选择其他目录。

您可以看到新文件包含以下内容：

```
# PowerShell script file to run as a Lambda function
#
# When executing in Lambda the following variables are predefined.
#   $LambdaInput - A PSObject that contains the Lambda function input data.
#   $LambdaContext - An Amazon.Lambda.Core.ILambdaContext object that contains
 information about the currently running Lambda environment.
#
# The last item in the PowerShell pipeline is returned as the result of the Lambda
 function.
```

```
#
# To include PowerShell modules with your Lambda function, like the
 AWSPowerShell.NetCore module, add a "#Requires" statement
# indicating the module and version.

#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}

# Uncomment to send the input to CloudWatch Logs
# Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
```

3. 要查看日志消息如何从您的 PowerShell 脚本发送到 Amazon CloudWatch Logs，请取消示例脚本的 `Write-Host` 行的注释。

   要演示如何从 Lambda 函数返回数据，请使用 `$PSVersionTable` 在脚本末尾添加新的一行。这会将 `$PSVersionTable` 添加到 PowerShell 管道。PowerShell 脚本完成后，PowerShell 管道中的最后一个对象是 Lambda 函数的返回数据。`$PSVersionTable` 是 PowerShell 全局变量，还提供有关正在运行的环境的信息。

   做出这些更改之后，示例脚本的最后两行如下所示：

   ```
   Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
   $PSVersionTable
   ```

4. 编辑 `MyFirstPSScript.ps1` 文件后，将目录更改为脚本的位置。然后运行以下命令，将脚本发布到 Lambda：

   ```
   Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name
    MyFirstPSScript -Region us-east-2
   ```

   注意，`-Name` 参数指定 Lambda 函数名称，该名称将显示在 Lambda 控制台中。您可以使用此函数手动调用脚本。

5. 使用 AWS Command Line Interface (AWS CLI) `invoke` 命令调用您的函数。

   ```
   > aws lambda invoke --function-name MyFirstPSScript out
   ```

定义采用 PowerShell 的 Lambda 函数处理程序

当调用 Lambda 函数时，Lambda 处理程序会调用 PowerShell 脚本。

调用 PowerShell 脚本时，以下变量已预定义：

- *$LambdaInput* – 包含处理程序输入的 PSObject。该输入可以是事件数据（由事件源发布）或您提供的自定义输入（如字符串或任意自定义数据对象）。
- *$LambdaContext* – 一个 Amazon.Lambda.Core.ILambdaContext 对象，用于访问有关当前调用的信息（例如当前函数的名称、内存限制、剩余执行时间和日志记录）。

例如，请考虑以下 PowerShell 示例代码。

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

此脚本返回从 $LambdaContext 变量获得的 FunctionName 属性。

> **Note**
>
> 您需要使用 PowerShell 脚本中的 `#Requires` 语句指示您的脚本所依赖的模块。此语句可执行两个重要任务。1) 它可告知其他开发人员该脚本使用的模块，2) 它可标识 AWS PowerShell 工具在部署过程中使用脚本打包所需要的从属模块。有关 PowerShell 中的 `#Requires` 语句的更多信息，请参阅关于 Requires。有关 PowerShell 部署程序包的更多信息，请参阅使用 .zip 文件归档部署 PowerShell Lambda 函数。
>
> 当您的 PowerShell Lambda 函数使用 AWS PowerShell cmdlet 时，请务必设置一个 `#Requires` 语句，使其引用 `AWSPowerShell.NetCore` 模块，该模块支持 PowerShell Core，而不是 `AWSPowerShell` 模块，该模块仅支持 Windows PowerShell。此外，请确保使用 3.3.270.0 版或更新版本的 `AWSPowerShell.NetCore`，其优化了 cmdlet 导入过程。如果使用较旧版本，冷启动时间较长。有关更多信息，请参阅 AWS Tools for PowerShell。

返回数据

有些 Lambda 调用旨在为调用方返回数据。例如，如果某个调用是为了响应来自 API Gateway 的 Web 请求，则我们的 Lambda 函数需要返回响应。对于 PowerShell Lambda，添加到 PowerShell 管道的最后一个对象是 Lambda 调用返回的数据。如果对象是字符串，数据将按原样返回。否则，对象将使用 `ConvertTo-Json` cmdlet 转换为 JSON。

例如，请考虑以下 PowerShell 语句，该语句在 PowerShell 管道中添加 `$PSVersionTable`：

```
$PSVersionTable
```

PowerShell 脚本完成后，PowerShell 管道中的最后一个对象是 Lambda 函数的返回数据。`$PSVersionTable` 是 PowerShell 全局变量，还提供有关正在运行的环境的信息。

使用 Lambda 上下文对象检索 PowerShell 函数信息

当 Lambda 运行您的函数时，它通过使 $LambdaContext 变量可用于处理程序来传递上下文信息。此变量提供的方法和属性包含有关调用、函数和执行环境的信息。

上下文属性

- FunctionName – Lambda 函数的名称。
- FunctionVersion – 函数的版本
- InvokedFunctionArn – 用于调用函数的 Amazon Resource Name (ARN)。表明调用者是否指定了版本号或别名。
- MemoryLimitInMB – 为函数分配的内存量。
- AwsRequestId – 调用请求的标识符。
- LogGroupName – 函数的日志组。
- LogStreamName – 函数实例的日志流。
- RemainingTime – 执行超时前剩余的毫秒数。
- Identity – (移动应用程序) 有关授权请求的 Amazon Cognito 身份的信息。
- ClientContext – (移动应用程序) 客户端应用程序提供给 Lambda 的客户端上下文。
- Logger – 函数的记录器对象。

下面的 PowerShell 代码片段显示了一个打印部分上下文信息的简便处理程序函数。

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function name:' $LambdaContext.FunctionName
Write-Host 'Remaining milliseconds:' $LambdaContext.RemainingTime.TotalMilliseconds
Write-Host 'Log group name:' $LambdaContext.LogGroupName
Write-Host 'Log stream name:' $LambdaContext.LogStreamName
```

Powershell Lambda 函数日志记录和监控

AWS Lambda 将代表您自动监控 Lambda 函数并将日志记录发送至 Amazon CloudWatch。您的 Lambda 函数带有一个 CloudWatch Logs 日志组以及函数的每个实例的日志流。Lambda 运行时环境会将每个调用的详细信息发送到日志流，然后中继函数代码的日志和其他输出。有关更多信息，请参阅将 CloudWatch Logs 日志与 Lambda 结合使用。

本页旨在介绍如何从 Lambda 函数的代码生成日志输出，并使用 AWS Command Line Interface、Lambda 控制台或 CloudWatch 控制台访问日志。

Sections

创建返回日志的函数

要从您的函数代码输出日志，您可以在 Microsoft.PowerShell.Utility 上使用 cmdlets，或者使用任何写入到 `stdout` 或 `stderr` 的日志记录模块。下面的示例使用了 `Write-Host`。

Example function/Handler.ps1 – 日志记录

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host `## Environment variables
Write-Host AWS_LAMBDA_FUNCTION_VERSION=$Env:AWS_LAMBDA_FUNCTION_VERSION
Write-Host AWS_LAMBDA_LOG_GROUP_NAME=$Env:AWS_LAMBDA_LOG_GROUP_NAME
Write-Host AWS_LAMBDA_LOG_STREAM_NAME=$Env:AWS_LAMBDA_LOG_STREAM_NAME
Write-Host AWS_EXECUTION_ENV=$Env:AWS_EXECUTION_ENV
Write-Host AWS_LAMBDA_FUNCTION_NAME=$Env:AWS_LAMBDA_FUNCTION_NAME
Write-Host PATH=$Env:PATH
Write-Host `## Event
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 3)
```

Example 日志格式

```
START RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Version: $LATEST
```

```
Importing module ./Modules/AWSPowerShell.NetCore/3.3.618.0/AWSPowerShell.NetCore.psd1
[Information] - ## Environment variables
[Information] - AWS_LAMBDA_FUNCTION_VERSION=$LATEST
[Information] - AWS_LAMBDA_LOG_GROUP_NAME=/aws/lambda/blank-powershell-
function-18CIXMPLHFAJJ
[Information] - AWS_LAMBDA_LOG_STREAM_NAME=2020/04/01/
[$LATEST]53c5xmpl52d64ed3a744724d9c201089
[Information] - AWS_EXECUTION_ENV=AWS_Lambda_dotnet6_powershell_1.0.0
[Information] - AWS_LAMBDA_FUNCTION_NAME=blank-powershell-function-18CIXMPLHFAJJ
[Information] - PATH=/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin
[Information] - ## Event
[Information] -
{
    "Records": [
        {
            "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
            "receiptHandle": "MessageReceiptHandle",
            "body": "Hello from SQS!",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1523232000000",
                "SenderId": "123456789012",
                "ApproximateFirstReceiveTimestamp": "1523232000001"
            },
            ...
END RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed
REPORT RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Duration: 3906.38 ms Billed
 Duration: 4000 ms Memory Size: 512 MB Max Memory Used: 367 MB Init Duration: 5960.19
 ms
XRAY TraceId: 1-5e843da6-733cxmple7d0c3c020510040 SegmentId: 3913xmpl20999446 Sampled:
 true
```

.NET 运行时记录每次调用的 START、END 和 REPORT 行。报告行提供了以下详细信息：

REPORT 行数据字段

- RequestId – 调用的唯一请求 ID。
- Duration（持续时间）– 函数的处理程序方法处理事件所花费的时间。
- Billed Duration（计费持续时间）– 针对调用计费的时间量。
- Memory Size（内存大小）– 分配给函数的内存量。
- Max Memory Used（最大内存使用量）– 函数使用的内存量。如果调用共享执行环境，Lambda 会报告所有调用使用的最大内存。此行为可能会导致报告值高于预期。

- Init Duration（初始持续时间）– 对于提供的第一个请求，为运行时在处理程序方法外部加载函数和运行代码所花费的时间。
- XRAY TraceId – 对于追踪的请求，为 AWS X-Ray 追踪 ID。
- SegmentId – 对于追踪的请求，为 X-Ray 分段 ID。
- Sampled（采样）– 对于追踪的请求，为采样结果。

在 Lambda 控制台中查看日志

调用 Lambda 函数后，您可以使用 Lambda 控制台查看日志输出。

如果可以在嵌入式代码编辑器中测试代码，则可以在执行结果中找到日志。使用控制台测试功能调用函数时，可以在详细信息部分找到日志输出。

在 CloudWatch 控制台中查看日志

您可以使用 Amazon CloudWatch 控制台查看所有 Lambda 函数调用的日志。

使用 CloudWatch 控制台查看日志

1. 打开 CloudWatch 控制台的 Log groups（日志组页面）。
2. 选择您的函数 (/aws/lambda/***your-function-name***) 的日志组。
3. 创建日志流。

每个日志流对应一个函数实例。日志流会在您更新 Lambda 函数以及创建更多实例来处理多个并发调用时显示。要查找特定调用的日志，建议您使用 AWS X-Ray 检测函数。X-Ray 会在追踪中记录有关请求和日志流的详细信息。

使用 AWS Command Line Interface（AWS CLI）查看日志

AWS CLI 是一种开源工具，让您能够在命令行 Shell 中使用命令与 AWS 服务进行交互。要完成本节中的步骤，您必须拥有 AWS CLI 版本 2。

您可以通过 AWS CLI，使用 `--log-type` 命令选项检索调用的日志。响应包含一个 `LogResult` 字段，其中包含多达 4KB 来自调用的 base64 编码日志。

Example 检索日志 ID

以下示例说明如何从 `LogResult` 字段中检索名为 `my-function` 的函数的日志 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解码日志

在同一命令提示符下，使用 base64 实用程序解码日志。以下示例说明如何为 my-function 检索 base64 编码的日志。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

您应看到以下输出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64 实用程序在 Linux、macOS 和 Ubuntu on Windows 上可用。macOS 用户可能需要使用 base64 -D。

Example get-logs.sh 脚本

在同一命令提示符下，使用以下脚本下载最后五个日志事件。此脚本使用 sed 从输出文件中删除引号，并休眠 15 秒以等待日志可用。输出包括来自 Lambda 的响应，以及来自 get-log-events 命令的输出。

复制以下代码示例的内容并将其作为 `get-logs.sh` 保存在 Lambda 项目目录中。

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux（仅限）

在同一命令提示符下，macOS 和 Linux 用户可能需要运行以下命令以确保脚本可执行。

```
chmod -R 755 get-logs.sh
```

Example 检索最后五个日志事件

在同一命令提示符下，运行以下脚本以获取最后五个日志事件。

```
./get-logs.sh
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
```

```
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


删除日志

删除函数时，日志组不会自动删除。要避免无限期存储日志，请删除日志组，或配置一个保留期，在该保留期之后，日志将自动删除。

使用 Rust 构建 Lambda 函数

由于 Rust 可编译为原生代码，因而无需专用运行时系统即可在 Lambda 上运行 Rust 代码。因此，请使用 Rust 运行时系统客户端在本地生成项目，然后通过 `provided.al2023` 或 `provided.al2` 运行时系统将其部署到 Lambda。使用 `provided.al2023` 或 `provided.al2` 时，Lambda 会自动使用最新补丁，确保操作系统保持更新状态。

> **Note**
>
> Rust 运行时系统客户端是实验性程序包。它随时可能更改，并且仅用于评估目的。

适用于 Rust 的工具和库

- AWS SDK for Rust：适用于 Rust 的 AWS SDK 提供用于与 Amazon Web Services 基础设施服务进行交互的 Rust API。
- 适用于 Lambda 的 Rust 运行时系统客户端：Rust 运行时系统客户端是实验性程序包。它随时可能发生重大更改，不建议用于生产。
- Cargo Lambda：此库提供命令行应用程序，以处理使用 Rust 构建的 Lambda 函数。
- Lambda HTTP：此库提供包装程序，以处理 HTTP 事件。
- Lambda 扩展：此库支持使用 Rust 写入 Lambda 扩展。
- AWS Lambda 事件：此库提供常见事件源集成的类型定义。

适用于 Rust 的 Lambda 应用程序示例

- 基本 Lambda 函数：此 Rust 函数可演示如何处理基本事件。
- 具有错误处理功能的 Lambda 函数：此 Rust 函数可演示如何在 Lambda 中处理自定义 Rust 错误。
- 具有共享资源的 Lambda 函数：此 Rust 项目用于初始化共享资源然后再创建 Lambda 函数。
- Lambda HTTP 事件：此 Rust 函数可处理 HTTP 事件。
- 包含 CORS 标头的 Lambda HTTP 事件：此 Rust 函数使用 Tower 注入 CORS 标头。
- Lambda REST API：此 REST API 可通过 Axum 和 Diesel 连接到 PostgreSQL 数据库。
- 无服务器 Rust 演示：此 Rust 项目可演示如何使用 Lambda 的 Rust 库、日志记录、环境变量和 AWS SDK。
- 基本 Lambda 扩展：此 Rust 扩展可演示如何处理基本扩展事件。

- Lambda Logs Amazon Data Firehose 扩展：此 Rust 扩展可演示如何将 Lambda 日志发送到 Firehose。

主题

- 定义采用 Rust 的 Lambda 函数处理程序
- 使用 Lambda 上下文对象检索 Rust 函数信息
- 使用 Rust 处理 HTTP 事件
- 使用 .zip 文件存档部署 Rust Lambda 函数
- Rust Lambda 函数日志记录和监控

定义采用 Rust 的 Lambda 函数处理程序

> **Note**
>
> Rust 运行时系统客户端是实验性程序包。它随时可能更改，并且仅用于评估目的。

Lambda 函数处理程序是函数代码中处理事件的方法。当调用函数时，Lambda 运行处理程序方法。您的函数会一直运行，直到处理程序返回响应、退出或超时。

主题

- Rust 处理程序基础知识
- 使用共享状态
- Rust Lambda 函数的代码最佳实践

Rust 处理程序基础知识

将 Lambda 函数代码编写为 Rust 可执行文件。实施处理程序函数代码和主函数，并包括以下内容：

- 来自 crates.io 的 lambda_runtime crate，可实现适用于 Rust 的 Lambda 编程模型。
- 将 Tokio 包含在依赖项中。适用于 Lambda 的 Rust 运行时系统客户端使用 Tokio 来处理异步调用。

Example – 处理 JSON 事件的 Rust 处理程序

以下示例使用 serde_json crate 来处理基本 JSON 事件：

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
 let payload = event.payload;
 let first_name = payload["firstName"].as_str().unwrap_or("world");
 Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
```

```
    lambda_runtime::run(service_fn(handler)).await
}
```

请注意以下几点：

- use：导入 Lambda 函数所需的库。
- async fn main：运行 Lambda 函数代码的入口点。Rust 运行时系统客户端使用 Tokio 作为异步运行时系统，因此您必须使用 #[tokio::main] 为主函数添加注释。
- async fn handler(event: LambdaEvent<Value>) -> Result<Value,Error>：这是 Lambda 处理程序签名。包括调用函数时运行的代码。
  - LambdaEvent<Value>：这是一种泛型类型，用于描述 Lambda 运行时系统以及 Lambda 函数上下文接收到的事件。
  - Result<Value, Error>：函数返回 Result 类型。如果函数成功，则结果为 JSON 值。如果函数不成功，则结果为错误。

使用共享状态

您可以声明独立于 Lambda 函数的处理程序代码的共享变量。这些变量可以帮助您在函数接收任何事件之前的 Init 阶段 期间加载状态信息。

Example – 跨函数实例共享 Amazon S3 客户端

请注意以下几点：

- use aws_sdk_s3::Client：此示例要求您将 aws-sdk-s3 = "0.26.0" 添加到 Cargo.toml 文件中的依赖项列表。
- aws_config::from_env：此示例要求您将 aws-config = "0.55.1" 添加到 Cargo.toml 文件中的依赖项列表。

```
use aws_sdk_s3::Client;
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde::{Deserialize, Serialize};

#[derive(Deserialize)]
struct Request {
    bucket: String,
}
```

```
#[derive(Serialize)]
struct Response {
    keys: Vec<String>,
}

async fn handler(client: &Client, event: LambdaEvent<Request>) -> Result<Response,
 Error> {
    let bucket = event.payload.bucket;
    let objects = client.list_objects_v2().bucket(bucket).send().await?;
    let keys = objects
        .contents()
        .map(|s| s.iter().flat_map(|o| o.key().map(String::from)).collect())
        .unwrap_or_default();
    Ok(Response { keys })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    let shared_config = aws_config::from_env().load().await;
    let client = Client::new(&shared_config);
    let shared_client = &client;
    lambda_runtime::run(service_fn(move |event: LambdaEvent<Request>| async move {
        handler(&shared_client, event).await
    }))
    .await
}
```

Rust Lambda 函数的代码最佳实践

在构建 Lambda 函数时，请遵循以下列表中的指南，采用最佳编码实践：

- 从核心逻辑中分离 Lambda 处理程序。这样您可以创建更容易进行单元测试的函数。
- 将依赖关系的复杂性降至最低。首选在执行环境启动时可以快速加载的更简单的框架。
- 将部署程序包大小精简为只包含运行时必要的部分。这样会减少调用前下载和解压缩部署程序包所需的时间。

- 利用执行环境重用来提高函数性能。连接软件开发工具包 (SDK) 客户端和函数处理程序之外的数据库，并在 `/tmp` 目录中本地缓存静态资产。由函数的同一实例处理的后续调用可重用这些资源。这样就可以通过缩短函数运行时间来节省成本。

为了避免调用之间潜在的数据泄露，请不要使用执行环境来存储用户数据、事件或其他具有安全影响的信息。如果您的函数依赖于无法存储在处理程序的内存中的可变状态，请考虑为每个用户创建单独的函数或单独的函数版本。

- 使用 keep-alive 指令来维护持久连接。Lambda 会随着时间的推移清除空闲连接。在调用函数时尝试重用空闲连接会导致连接错误。要维护您的持久连接，请使用与运行时关联的 keep-alive 指令。有关示例，请参阅在 Node.js 中通过 Keep-Alive 重用连接。
- 使用环境变量将操作参数传递给函数。例如，您在写入 Amazon S3 存储桶时，不应对要写入的存储桶名称进行硬编码，而应将存储桶名称配置为环境变量。
- 避免在 Lambda 函数中使用递归调用，在这种情况下，函数会调用自己或启动可能再次调用该函数的进程。这可能会导致意想不到的函数调用量和升级成本。如果您看到意外的调用量，请立即将函数保留并发设置为 0 来限制对函数的所有调用，同时更新代码。
- Lambda 函数代码中不要使用非正式的非公有 API。对于 AWS Lambda 托管式运行时，Lambda 会定期为 Lambda 的内部 API 应用安全性和功能更新。这些内部 API 更新可能不能向后兼容，会导致意外后果，例如，假设您的函数依赖于这些非公有 API，则调用会失败。请参阅 API 参考以查看公开发布的 API 列表。
- 编写幂等代码。为您的函数编写幂等代码可确保以相同的方式处理重复事件。您的代码应该正确验证事件并优雅地处理重复事件。有关更多信息，请参阅如何使我的 Lambda 函数具有幂等性？。

使用 Lambda 上下文对象检索 Rust 函数信息

> **Note**
>
> Rust 运行时系统客户端是实验性程序包。它随时可能更改，并且仅用于评估目的。

当 Lambda 运行函数时，会将上下文对象添加到处理程序接收的 LambdaEvent 中。此对象提供的属性包含有关调用、函数和执行环境的信息。

上下文属性

- `request_id`：由 Lambda 服务生成的 AWS 请求 ID。
- `deadline`：当前调用的执行截止时间（以毫秒为单位）。
- `invoked_function_arn`：正在调用的 Lambda 函数的 Amazon 资源名称（ARN）。
- `xray_trace_id`：当前调用的 AWS X-Ray 跟踪 ID。
- `client_content`：AWS Mobile SDK 发送的客户端上下文对象。除非使用 AWS Mobile SDK 调用函数，否则此字段为空。
- `identity`：已调用函数的 Amazon Cognito 身份。除非使用由 Amazon Cognito 身份池颁发的 AWS 凭证向 Lambda API 发出调用请求，否则此字段为空。
- `env_config`：来自本地环境变量的 Lambda 函数配置。此属性包括函数名称、内存分配、版本和日志流等信息。

访问调用上下文信息

Lambda 函数可以访问有关其环境和调用请求的元数据。函数处理程序接收的 `LambaEvent` 对象包含 `context` 元数据：

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    let invoked_function_arn = event.context.invoked_function_arn;
    Ok(json!({ "message": format!("Hello, this is function
 {invoked_function_arn}!") }))
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

使用 Rust 处理 HTTP 事件

> **Note**
>
> Rust 运行时系统客户端是实验性程序包。它随时可能更改，并且仅用于评估目的。

Amazon API Gateway API、应用程序负载均衡器和 Lambda 函数 URL 都可将 HTTP 事件发送到 Lambda。您可以使用 crates.io 中的 aws_lambda_events crate 处理来自这些来源的事件。

Example – 处理 API Gateway 代理请求

请注意以下几点：

- `use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}`：aws_lambda_events crate 包含多个 Lambda 事件。要缩短编译时间，请使用功能标志激活所需事件。示例：`aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }`。
- `use http::HeaderMap`：此导入要求您将 http crate 添加到依赖项中。

```
use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse};
use http::HeaderMap;
use lambda_runtime::{service_fn, Error, LambdaEvent};

async fn handler(
    _event: LambdaEvent<ApiGatewayProxyRequest>,
) -> Result<ApiGatewayProxyResponse, Error> {
    let mut headers = HeaderMap::new();
    headers.insert("content-type", "text/html".parse().unwrap());
    let resp = ApiGatewayProxyResponse {
        status_code: 200,
        multi_value_headers: headers.clone(),
        is_base64_encoded: false,
        body: Some("Hello AWS Lambda HTTP request".into()),
        headers,
    };
    Ok(resp)
}

#[tokio::main]
```

```
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

适用于 Lambda 的 Rust 运行时系统客户端还提供对这些事件类型的抽象，允许您使用本机 HTTP 类型，无论哪个服务发送事件。以下代码等同于前面的示例，可以结合 Lambda 函数 URL、应用程序负载均衡器和 API Gateway 直接使用。

> **Note**
>
> lambda_http crate 将使用下方的 lambda_runtime crate。无需单独导入 `lambda_runtime`。

Example – 处理 HTTP 请求

```
use lambda_http::{service_fn, Error, IntoResponse, Request, RequestExt, Response};

async fn handler(event: Request) -> Result<impl IntoResponse, Error> {
    let resp = Response::builder()
        .status(200)
        .header("content-type", "text/html")
        .body("Hello AWS Lambda HTTP request")
        .map_err(Box::new)?;
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_http::run(service_fn(handler)).await
}
```

关于如何使用 `lambda_http` 的另一个示例，请参阅 AWS Labs GitHub 存储库上的 http-axum 代码示例。

适用于 Rust 的示例 HTTP Lambda 事件

- Lambda HTTP 事件：此 Rust 函数可处理 HTTP 事件。
- 包含 CORS 标头的 Lambda HTTP 事件：此 Rust 函数使用 Tower 注入 CORS 标头。
- 具有共享资源的 Lambda HTTP 事件：此 Rust 函数使用在创建函数处理程序之前初始化的共享资源。

使用 .zip 文件存档部署 Rust Lambda 函数

> **Note**
>
> Rust 运行时系统客户端是实验性程序包。它随时可能更改，并且仅用于评估目的。

本页将介绍如何编译 Rust 函数，然后使用 Cargo Lambda 将已编译的二进制文件部署到 AWS Lambda。此外，还会显示如何使用 AWS Command Line Interface 和 AWS Serverless Application Model CLI 部署已编译的二进制文件。

Sections

先决条件

- Rust
- AWS CLI 版本 2

在 macOS、Windows 或 Linux 上构建 Rust 函数

以下步骤将演示如何使用 Rust 为第一个 Lambda 函数创建项目，并使用 Cargo Lambda 对其进行编译。

1. 安装 Cargo Lambda（一个 Cargo 子命令），它会为 macOS、Windows 和 Linux 上的 Lambda 编译 Rust 函数。

   要在任何安装有 Python 3 的系统上安装 Cargo Lambda，请使用 pip：

   ```
   pip3 install cargo-lambda
   ```

   要在 macOS 或 Linux 上安装 Cargo Lambda，请使用 Homebrew：

```
brew tap cargo-lambda/cargo-lambda
brew install cargo-lambda
```

要在 Windows 上安装 Cargo Lambda，请使用 Scoop：

```
scoop bucket add cargo-lambda
scoop install cargo-lambda/cargo-lambda
```

有关其他选项，请参阅 Cargo Lambda 文档中的安装。

2. 创建程序包结构。此命令可在 `src/main.rs` 中创建一些基本的函数代码。您可以使用此代码进行测试，也可以将其替换为您自己的代码。

```
cargo lambda new my-function
```

3. 在程序包的根目录中，运行 build 子命令以编译函数中的代码。

```
cargo lambda build --release
```

（可选）如果要在 Lambda 上使用 AWS Graviton2，请添加 `--arm64` 标志以针对 ARM CPU 编译代码。

```
cargo lambda build --release --arm64
```

4. 在部署 Rust 函数之前，请在计算机上配置 AWS 凭证。

```
aws configure
```

使用 Cargo Lambda 部署 Rust 函数二进制文件

使用 deploy 子命令将已编译的二进制文件部署到 Lambda。此命令可创建执行角色，然后创建 Lambda 函数。要指定现有的执行角色，请使用 --iam-role 标志。

```
cargo lambda deploy my-function
```

使用 AWS CLI 部署 Rust 函数二进制文件

您也可以使用 AWS CLI 部署二进制文件。

1. 使用 build 子命令构建 .zip 部署包。

```
cargo lambda build --release --output-format zip
```

2. 要将 .zip 包部署到 Lambda，请运行 create-function 命令。

   - 对于 `--runtime`，请指定 `provided.al2023`。这是仅限操作系统的运行时。仅限操作系统的运行时用于将编译的二进制文件和自定义运行时部署到 Lambda。
   - 对于 `--role`，指定执行角色的 ARN。

```
aws lambda create-function \
     --function-name my-function \
     --runtime provided.al2023 \
     --role arn:aws:iam::111122223333:role/lambda-role \
     --handler rust.handler \
     --zip-file fileb://target/lambda/my-function/bootstrap.zip
```

使用 AWS SAM CLI 部署 Rust 函数二进制文件

您也可以使用 AWS SAM CLI 部署二进制文件。

1. 使用资源和属性定义创建 AWS SAM 模板。对于 `Runtime`，请指定 `provided.al2023`。这是仅限操作系统的运行时。仅限操作系统的运行时用于将编译的二进制文件和自定义运行时部署到 Lambda。

   有关使用 AWS SAM 部署 Lambda 函数的更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 AWS::Serverless::Function。

   Example Rust 二进制文件的 SAM 资源和属性定义

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM template for Rust binaries
Resources:
  RustFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: target/lambda/my-function/
      Handler: rust.handler
```

```
      Runtime: provided.al2023
Outputs:
  RustFunction:
    Description: "Lambda Function ARN"
    Value: !GetAtt RustFunction.Arn
```

2. 使用 build 子命令编译函数。

```
cargo lambda build --release
```

3. 使用 sam deploy 命令将函数部署到 Lambda。

```
sam deploy --guided
```

有关使用 AWS SAM CLI 构建 Rust 函数的更多信息，请参阅《AWS Serverless Application Model 开发人员指南》中的使用 Cargo Lambda 构建 Rust Lambda 函数。

使用 Cargo Lambda 调用 Rust 函数

使用 invoke 子命令，通过负载测试函数。

```
cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function
```

使用 AWS CLI 调用 Rust 函数

您也可以使用 AWS CLI 调用函数。

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"command": "Hello world"}' /tmp/out.txt
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

Rust Lambda 函数日志记录和监控

> **Note**
>
> Rust 运行时系统客户端是实验性程序包。它随时可能更改，并且仅用于评估目的。

AWS Lambda 将代表您自动监控 Lambda 函数并将日志记录发送至 Amazon CloudWatch。您的 Lambda 函数带有一个 CloudWatch Logs 日志组以及函数的每个实例的日志流。Lambda 运行时环境会将每个调用的详细信息发送到日志流，然后中继函数代码的日志和其他输出。有关更多信息，请参阅 将 CloudWatch Logs 日志与 Lambda 结合使用。本页将介绍如何从 Lambda 函数的代码生成日志输出。

创建写入日志的函数

要从函数代码输出日志，您可以使用写入到 `stdout` 或 `stderr` 的任何日志记录函数，例如 `println!` 宏。以下示例使用 `println!` 在函数处理程序启动时和完成之前输出消息。

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    println!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    println!("Rust function responds to {}", &first_name);
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

使用 Tracing crate 实现高级日志记录

跟踪是一个框架，用于分析 Rust 程序以收集基于事件的结构化诊断信息。该框架提供了用于自定义日志记录输出级别和格式的实用程序，例如创建结构化 JSON 日志消息。要使用此框架，您必须在实现函数处理程序之前将其初始化为 `subscriber`。然后，您可以使用 `debug`、`info` 和 `error` 等跟踪宏来指定每个场景所需的日志记录级别。

Example – 使用 Tracing crate

请注意以下几点：

- `tracing_subscriber::fmt().json()`：如果包含此选项，则日志将采用 JSON 格式。要使用此选项，您必须将 `json` 功能包含在 `tracing-subscriber` 依赖项中（例如 `tracing-subscriber = { version = "0.3.11", features = ["json"] }`）。
- `#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]`：每次调用处理程序时，此注释都会生成一个跨度。该跨度会将请求 ID 添加到每个日志行。
- `{ %first_name }`：此构造会将 `first_name` 字段添加到使用该字段的日志行。此字段的值与同名变量对应。

```
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    tracing::info!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    tracing::info!({ %first_name }, "Rust function responds to event");
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt().json()
        .with_max_level(tracing::Level::INFO)
        // this needs to be set to remove duplicated information in the log.
        .with_current_span(false)
        // this needs to be set to false, otherwise ANSI color codes will
        // show up in a confusing manner in CloudWatch logs.
        .with_ansi(false)
        // disabling time is handy because CloudWatch will add the ingestion time.
        .without_time()
        // remove the name of the function from every log entry
        .with_target(false)
        .init();
    lambda_runtime::run(service_fn(handler)).await
}
```

调用此 Rust 函数时，它会输出类似于以下内容的两个日志行：

```
{"level":"INFO","fields":{"message":"Rust function invoked"},"spans":
[{"req_id":"45daaaa7-1a72-470c-9a62-e79860044bb5","name":"handler"}]}
{"level":"INFO","fields":{"message":"Rust function responds to
 event","first_name":"David"},"spans":[{"req_id":"45daaaa7-1a72-470c-9a62-
e79860044bb5","name":"handler"}]}
```

使用 AWS Lambda 函数的最佳实践

以下是推荐使用 AWS Lambda 的最佳实践：

主题

函数代码

- 利用执行环境重用来提高函数性能。连接软件开发工具包 (SDK) 客户端和函数处理程序之外的数据库，并在 `/tmp` 目录中本地缓存静态资产。由函数的同一实例处理的后续调用可重用这些资源。这样就可以通过缩短函数运行时间来节省成本。

  为了避免调用之间潜在的数据泄露，请不要使用执行环境来存储用户数据、事件或其他具有安全影响的信息。如果您的函数依赖于无法存储在处理程序的内存中的可变状态，请考虑为每个用户创建单独的函数或单独的函数版本。
- 使用 keep-alive 指令来维护持久连接。Lambda 会随着时间的推移清除空闲连接。在调用函数时尝试重用空闲连接会导致连接错误。要维护您的持久连接，请使用与运行时关联的 keep-alive 指令。有关示例，请参阅在 Node.js 中通过 Keep-Alive 重用连接。
- 使用环境变量将操作参数传递给函数。例如，您在写入 Amazon S3 存储桶时，不应对要写入的存储桶名称进行硬编码，而应将存储桶名称配置为环境变量。
- 避免在 Lambda 函数中使用递归调用，在这种情况下，函数会调用自己或启动可能再次调用该函数的进程。这可能会导致意想不到的函数调用量和升级成本。如果您看到意外的调用量，请立即将函数保留并发设置为 `0` 来限制对函数的所有调用，同时更新代码。
- Lambda 函数代码中不要使用非正式的非公有 API。对于 AWS Lambda 托管式运行时，Lambda 会定期为 Lambda 的内部 API 应用安全性和功能更新。这些内部 API 更新可能不能向后兼容，会导致意外后果，例如，假设您的函数依赖于这些非公有 API，则调用会失败。请参阅 API 参考以查看公开发布的 API 列表。

- 编写幂等代码。为您的函数编写幂等代码可确保以相同的方式处理重复事件。您的代码应该正确验证事件并优雅地处理重复事件。有关更多信息，请参阅如何使我的 Lambda 函数具有幂等性？。

有关特定语言的代码最佳实践，请参阅以下章节：

- the section called “Node.js Lambda 函数的代码最佳实践”
- the section called “Typescript Lambda 函数的代码最佳实践”
- the section called “Python Lambda 函数的代码最佳实践”
- the section called “Ruby Lambda 函数的代码最佳实践”
- the section called “Java Lambda 函数的代码最佳实践”
- the section called “Go Lambda 函数的代码最佳实践”
- the section called “C# Lambda 函数的代码最佳实践”
- the section called “Rust Lambda 函数的代码最佳实践”

函数配置

- 对您的 Lambda 函数进行性能测试是确保选择最佳内存大小配置的关键环节。增加内存大小会触发函数可用 CPU 的同等水平的增加。函数的内存使用率是根据调用情况确定的，可以在 Amazon CloudWatch 中查看。每次调用都将生成一个 `REPORT:` 条目，如下所示：

  ```
  REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration: 12.34 ms Billed
   Duration: 100 ms Memory Size: 128 MB Max Memory Used: 18 MB
  ```

  分析 `Max Memory Used:` 字段能够确定函数是否需要更多内存，或函数的内存大小是否过度配置。

  要为您的函数查找适合的内存配置，我们建议使用开源 AWS Lambda 功率调谐项目。有关更多信息，请参阅 GitHub 上的 AWS Lambda 功率调谐。

  为了优化函数性能，我们还建议部署可以利用高级矢量扩展 2 (AVX2) 的库。这样一来，您就可以处理艰巨的工作负载，如机器学习推理、媒体处理、高性能计算（HPC）、科学模拟和财务建模。有关更多信息，请参阅使用 AVX2 创建更快的 AWS Lambda 函数。
- 对您的 Lambda 函数进行加载测试，确定最佳超时值。分析函数的运行时间很重要，这样更容易确定依赖关系服务是否有问题，这些问题可能导致并发函数以超出您的预期的速度增加。如果您的

Lambda 函数进行网络调用的资源无法处理 Lambda 扩缩，这就更加重要。有关对应用程序进行负载测试的更多信息，请参阅 AWS 上的分布式负载测试。

- 设置 IAM policy 时使用最严格的权限。了解您的 Lambda 函数所需的资源和操作，并限制这些权限的执行角色。有关更多信息，请参阅在 AWS Lambda 中管理权限：
- 熟悉Lambda 配额。在确定运行时资源限制时，负载大小、文件描述符和 /tmp 空间通常会被忽略。
- 删除不再使用的 Lambda 函数。这样，未使用的函数就不会不必要地占用有限的部署程序包空间。
- 如果您使用 Amazon Simple Queue Service 作为事件源，请确保该函数的预计调用时间值不超过队列上的可见性超时值。这同样适用于 CreateFunction 和 UpdateFunctionConfiguration。
  - 对于 CreateFunction，AWS Lambda 会使函数创建流程失败。
  - 对于 UpdateFunctionConfiguration，它可能会导致该函数的重复调用。

功能可扩展性

- 熟悉您的上游和下游吞吐量限制。虽然 Lambda 函数可随负载无缝扩展，但上游和下游依赖项可能不具有相同的吞吐量。如果您需要限制函数可以扩展的幅度，可以为函数配置预留并发。
- 内置节流容错能力：如果同步函数因流量超过 Lambda 的扩展速率而遭遇节流，则可使用以下策略来提高节流容错能力：
  - 使用超时、重试和抖动回退：实施这些策略可以平滑重试的调用，有助于确保 Lambda 可以在几秒钟内纵向扩展，尽量减少最终用户节流。
  - 使用预置并发：预置并发是 Lambda 分配给函数的预初始化执行环境的数量。Lambda 在可用时使用预置并发处理传入请求。如有必要，Lambda 还可以将函数扩展到预置并发设置之上。配置预置并发会让 AWS 账户产生额外费用。

指标和警报

- 使用 查看 Lambda 函数的指标 和 CloudWatch Alarms，而不是在您的 Lambda 函数代码中创建和更新指标。追踪 Lambda 函数的运行状况是更加有效的方式，这样您就可以在早期开发过程中发现问题。例如，您可以根据 Lambda 函数调用的预计持续时间配置警报，以解决函数代码引起的瓶颈或延迟。
- 利用您的日志记录库和 AWS Lambda 指标和维度捕捉应用程序错误（例如，ERR、ERROR、WARNING 等）
- 使用 AWS Cost Anomaly Detection 来检测您账户中的异常活动。Cost Anomaly Detection 使用机器学习技术来持续监控您的成本和使用情况，并尽力减少误报。Cost Anomaly Detection 使用来自

AWS Cost Explorer 的数据，该数据最长可能会延迟 24 小时。因此，发生使用后最长可能需要 24 小时才会检测到异常。要开始使用 Cost Anomaly Detection，您必须首先注册 Cost Explorer。然后访问 Cost Anomaly Detection。

处理流

- 测试不同批处理和记录的大小，这样每个事件源的轮询频率都会根据函数完成任务的速度进行调整。CreateEventSourceMapping BatchSize 参数控制每次调用可向您的函数发送记录的最大数量。批处理大小如果较大，通常可以更有效地吸收大量记录的调用开销，从而增加吞吐量。

  默认情况下，Lambda 会在记录可用时尽快调用您的函数。如果 Lambda 从事件源中读取的批处理只有一条记录，则 Lambda 将会只向该函数发送一条记录。为避免在记录数量较少的情况下调用该函数，您可以配置 batching window（批处理时段），让事件源缓冲最多五分钟的记录。调用函数前，Lambda 会持续从事件源中读取记录，直到收集完整批处理、批处理时段到期或批处理达到 6MB 的有效负载时为止。有关更多信息，请参阅 批处理行为。

  > **Warning**
  >
  > Lambda 事件源映射至少处理每个事件一次，有可能出现重复处理记录的情况。为避免与重复事件相关的潜在问题，我们强烈建议您将函数代码设为幂等性。要了解更多信息，请参阅 AWS 知识中心的如何使我的 Lambda 函数具有幂等性。

- 通过增加分区提高 Kinesis 流处理吞吐量。一个 Kinesis 流由一个或多个分区组成。Lambda 从 Kinesis 读取数据的速率随着分片数量的增加而线性扩展。增加分区数量会直接增加 Lambda 函数并发调用的最大数量，还可增加 Kinesis 流处理的吞吐量。有关分片和函数调用之间的更多信息，请参阅 the section called “ 轮询和批处理流”。如果您增加 Kinesis 流中的分区数量，请确保您已为数据选择了合适的分区键（请参阅分区键），这样相关记录将会位于同一分区中，而且也可合理分配您的数据。
- 在 IteratorAge 上使用 Amazon CloudWatch，确定是否正在处理您的 Kinesis 流。例如，将 CloudWatch 警报的最大值设置配置为 30000（30 秒）。

安全最佳实操

- 监控 AWS Lambda 的使用情况，因为它与使用 AWS Security Hub 的安全最佳实践有关。Security Hub 使用安全控件来评估资源配置和安全标准，以帮助您遵守各种合规框架。有关使用 Security

Hub 评估 Lambda 资源的更多信息，请参阅《AWS Security Hub 用户指南》中的 AWS Lambda 控件。

- 使用 Amazon GuardDuty Lambda Protection 监控 Lambda 网络活动日志：在 AWS 账户 中调用 Lambda 函数时，GuardDuty Lambda Protection 可帮助识别潜在安全威胁。以某个函数查询与加密货币相关活动关联的 IP 地址为例。GuardDuty 会监控在调用 Lambda 函数时生成的网络活动日志。要了解更多信息，请参阅《Amazon GuardDuty User Guide》中的 Lambda Protection。

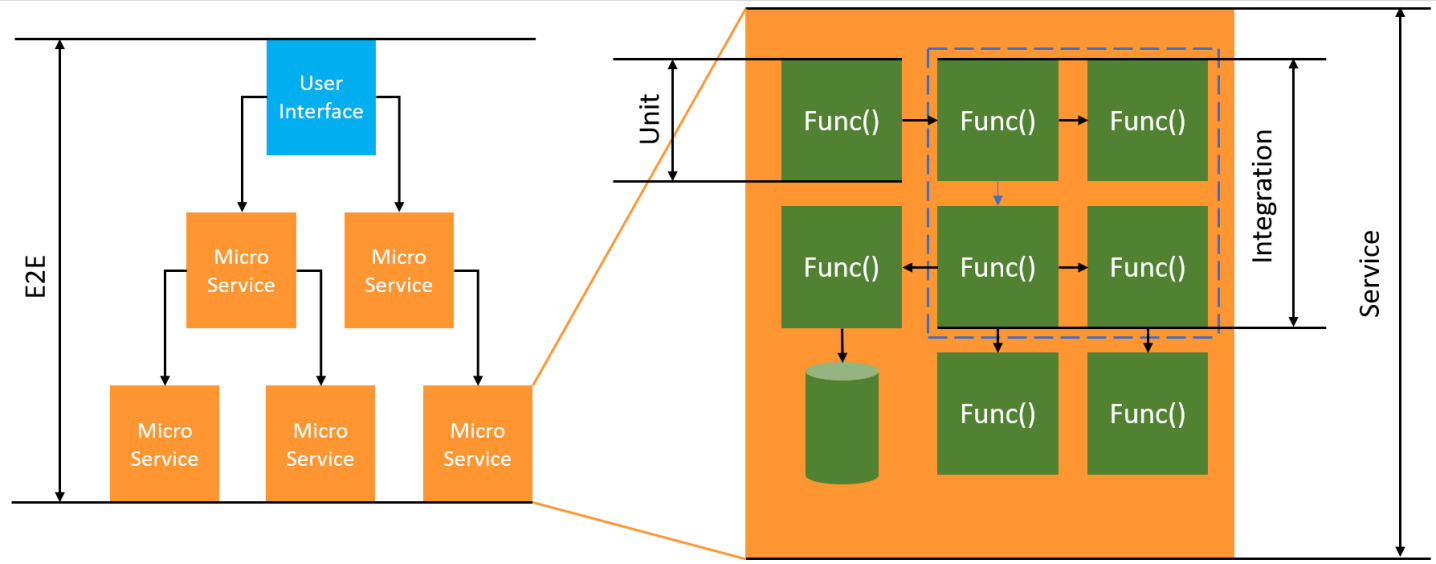
如何测试无服务器函数和应用程序

测试无服务器函数使用传统的测试类型和技术，但您还必须考虑对无服务器应用程序进行整体测试。基于云的测试将为您的函数和无服务器应用程序的质量提供最准确的衡量。

无服务器应用程序架构包括可通过 API 调用提供关键应用程序功能的托管服务。因此，您的开发周期应包括在函数与服务交互时验证功能的自动化测试。

如果您不创建基于云的测试，则可能会由于本地环境和已部署环境之间的差异而遇到问题。在将代码提升到下一个部署环境（例如 QA、暂存或生产）之前，您的持续集成过程应针对在云端预置的一套资源运行测试。

继续阅读本简短指南，了解无服务器应用程序的测试策略，或者访问无服务器测试示例存储库，深入了解特定于您所选语言和运行时系统的实用示例。



对于无服务器测试，您仍需要编写单元、集成和端到端测试。

- 单元测试 – 针对隔离代码块运行的测试。例如，验证业务逻辑以计算给定特定项目和目的地的配送费用。
- 集成测试 – 涉及通常在云环境中交互的两个或更多组件或服务的测试。例如，验证函数是否会处理队列中的事件。
- 端到端测试 – 验证整个应用程序行为的测试。例如，确保正确设置基础设施，并确保事件在服务之间按预期流动，以记录客户的订单。

目标业务成果

测试无服务器解决方案可能需要稍多时间来设置测试，以验证服务之间的事件驱动型交互。阅读本指南时，请牢记以下实际商业原因：

- 提高应用程序的质量
- 缩短构建功能和修复错误的时间

应用程序的质量取决于测试各种场景以验证功能。仔细考虑业务场景，以及自动执行这些测试以针对云服务运行，将提高应用程序的质量。

在迭代开发周期中发现软件错误和配置问题时，对成本和计划的影响最小。如果在开发过程中始终未检测到问题，那么在生产环境中发现和修复问题时将需要更多人付出更多努力。

精心规划的无服务器测试策略将通过验证您的 Lambda 函数和应用程序是否在云环境中按预期执行来提高软件质量并缩短迭代时间。

测试内容

我们建议采用测试策略来测试托管服务行为、云配置、安全策略以及与您的代码的集成，以提高软件质量。行为测试，也称为黑盒测试，用于验证系统在不了解所有内部信息的情况下是否按预期运行。

- 运行单元测试以检查 Lambda 函数内部的业务逻辑。
- 验证是否已实际调用集成服务，输入参数是否正确。
- 检查事件是否在工作流中端到端遍历所有预期服务。

在基于服务器的传统架构中，团队通常将测试范围定义为仅包括在应用程序服务器上运行的代码。其他组件、服务或依赖项通常被认为是外部的，并且超出测试范围。

无服务器应用程序通常由小型工作单元组成，例如从数据库检索产品、处理队列中的项目或调整存储中图像大小的 Lambda 函数。每个组件均在自己的环境中运行。团队很可能会负责单个应用程序中的许多此类小型单元。

可以将某些应用程序功能完全委派给 Amazon S3 等托管服务，也可以在不使用任何内部开发代码的情况下创建。无需测试这些托管服务，但您确实需要测试与这些服务的集成。

如何测试无服务器

您可能熟悉如何测试本地部署的应用程序：编写针对完全在桌面操作系统上或容器内运行的代码运行的测试。例如，您可以使用请求调用本地 Web 服务组件，然后对响应做出断言。

无服务器解决方案通过函数代码和基于云的托管服务（如队列、数据库、事件总线和消息传送系统）而构建。这些组件均通过事件驱动型架构，其中消息（称为事件）从一个资源流向另一个资源。这些交互可以是同步操作（例如 Web 服务立即返回结果），也可以是稍后完成的异步操作（例如将项目放入队列或启动工作流步骤）。您的测试策略必须包括这两个场景并测试服务之间的交互。对于异步交互，可能需要检测下游组件中可能无法立即观察到的副作用。

复制整个云环境（包括队列、数据库表、事件总线、安全策略等）并不可行。由于本地环境与云端已部署环境之间的差异，您难免会遇到问题。不同环境之间的差异将会增加重现和修复错误的时间。

在无服务器应用程序中，架构组件通常完全存在于云端，因此必须对云端的代码和服务进行测试以开发功能和修复错误。

测试技术

实际上，您的测试策略可能包括多种可提高解决方案质量的技术。您将使用快速交互式测试来调试控制台中的函数，使用自动化单元测试来检查隔离的业务逻辑，通过 Mock 验证对外部服务的调用，以及偶尔对模仿服务的仿真器进行测试。

- 在云端进行测试 – 您可以部署基础设施和代码，使用实际服务、安全策略、配置和基础设施特定参数进行测试。基于云的测试可以最准确地衡量代码质量。

  在控制台中调试函数是在云端进行测试的一种快速方法。您可以从示例测试事件库中进行选择，也可以创建自定义事件来在隔离环境中测试函数。您还可以通过控制台与您的团队共享测试事件。

  要在开发和构建生命周期中自动执行测试，您需要在控制台之外进行测试。有关自动化策略和资源的更多信息，请参阅本指南中特定于语言的测试部分。
- 使用 Mock 进行测试（也称为伪件）– Mock 是代码中的对象，用于模拟和替代外部服务。Mock 提供预定义的行为来验证服务调用和参数。伪件是一种 Mock 实施，它采用捷径来简化或提高性能。例如，虚设数据访问对象可能会从内存中数据存储返回数据。Mock 可以模仿和简化复杂的依赖项，但也可以导致生成更多的 Mock，以替换嵌套依赖项。
- 使用仿真器进行测试 – 您可以设置应用程序（有时来自第三方）以在本地环境中模仿云服务。速度是其优势，但设置复杂和与生产服务的对等性较差是其缺点。请谨慎使用仿真器。

在云端进行测试

在云端进行测试对于各个阶段的测试（包括单元测试、集成测试和端到端测试）而言都很有价值。当您针对基于云的且与基于云的服务进行交互的代码运行测试时，可以最准确地衡量代码质量。

在云端运行 Lambda 函数的一种便捷方法是在 AWS Management Console 中使用测试事件。测试事件是函数的一个 JSON 输入。如果函数不需要输入，则事件可以是空 JSON 文档（{}）。控制台为各种服务集成提供示例事件。在控制台中创建事件后，您还可以将其与团队共享，以简化测试并保持一致性。

了解如何在控制台中调试示例函数。

> **Note**
>
> 尽管在控制台中运行函数是一种快速的调试方法，但自动执行测试周期对于提高应用程序质量和开发速度而言具有至关重要的意义。

无服务器测试示例存储库中提供了测试自动化示例。以下命令行会运行自动化 Python 集成测试示例：

```
python -m pytest -s tests/integration -v
```

尽管测试在本地运行，但它会与基于云的资源进行交互。这些资源使用 AWS Serverless Application Model 和 AWS SAM 命令行工具部署。测试代码首先检索已部署的堆栈输出，包括 API 端点、函数 ARN 和安全角色。接下来，测试会向 API 端点发送请求，该端点使用 Amazon S3 存储桶列表进行响应。此测试完全针对基于云的资源运行，以验证这些资源是否已部署、受到保护并按预期工作。

```
========================= test session starts ==========================
      platform darwin -- Python 3.10.10, pytest-7.3.1, pluggy-1.0.0
      -- /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-lambda/
venv/bin/python
      cachedir: .pytest_cache
      rootdir: /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-
lambda
      plugins: mock-3.10.0
      collected 1 item


      tests/integration/test_api_gateway.py::TestApiGateway::test_api_gateway
```

```
      --> Stack outputs:

        HelloWorldApi
        = https://p7teqs3162.execute-api.us-east-2.amazonaws.com/Prod/hello/
        > API Gateway endpoint URL for Prod stage for Hello World function

        PythonTestDemo
        = arn:aws:lambda:us-east-2:123456789012:function:testing-apigw-lambda-
PythonTestDemo-iSij8evaTdxl
        > Hello World Lambda Function ARN

        PythonTestDemoIamRole
        = arn:aws:iam::123456789012:role/testing-apigw-lambda-PythonTestDemoRole-
IZELQQ9MG4HQ
        > Implicit IAM Role created for Hello World function

      --> Found API endpoint for "testing-apigw-lambda" stack...
      --> https://p7teqs3162.execute-api.us-east-2.amazonaws.com/Prod/hello/
      API Gateway response:
      amplify-dev-123456789-deployment|myapp-prod-p-loggingbucket-123456|s3-java-
bucket-123456789
      PASSED

      ========================= 1 passed in 1.53s =========================
```

对于云原生应用程序开发，在云端进行测试具有以下优势：

- 您可以测试每个可用服务。
- 您始终使用最新的服务 API 和返回值。
- 云测试环境与您的生产环境非常相似。
- 测试可以涵盖安全策略、服务限额、配置和基础设施特定参数。
- 每个开发人员都可以在云端快速创建一个或多个测试环境。
- 云测试可增强您的信心，确保代码在生产环境中正确运行。

在云端进行测试确实有一些缺点。在云端进行测试最明显的缺点是，部署到云环境通常会比部署到本地桌面环境花费更长的时间。

所幸，如 AWS Serverless Application Model（AWS SAM）Accelerate、AWS 云开发工具包（AWS CDK）监视模式和 SST（第三方）等工具可减少云部署迭代所涉及的延迟。这些工具可以监控基础设施和代码，并将增量更新自动部署到您的云环境中。

> **Note**
>
> 请参阅《无服务器开发人员指南》中的如何创建基础设施即代码，以了解有关 AWS Serverless Application Model、AWS CloudFormation 和 AWS Cloud Development Kit (AWS CDK) 的更多信息。

与本地测试不同，在云端进行测试需要额外的资源，这可能会产生服务成本。创建隔离的测试环境可能会增加 DevOps 团队的负担，尤其是在对账户和基础设施有严格控制的组织中更是如此。即便如此，在处理复杂的基础设施场景时，开发人员设置和维护错综复杂的本地环境所花费的时间成本可能与使用一次性测试环境（通过基础设施即代码自动化工具创建）的时间成本相似（或更昂贵）。

即使考虑到这些因素，在云端进行测试仍然是保证无服务器解决方案质量的最佳方式。

使用 Mock 进行测试

使用 Mock 进行测试是在代码中创建替换对象以模拟云服务行为的一种技术。

例如，您可以编写一个测试，该测试使用 Amazon S3 服务的 Mock，每当调用 CreateObject 方法时，其返回特定的响应。测试运行时，Mock 会返回该编程响应，而无需调用 Amazon S3 或任何其他服务端点。

Mock 对象通常由 Mock 框架生成，以减少开发工作量。一些 Mock 框架是通用的，而其他框架则是专门为 AWS SDK 设计的，例如 Moto，这是一个用于模拟 AWS 服务和资源的 Python 库。

请注意，Mock 对象与仿真器的不同之处在于，Mock 对象通常由开发人员作为测试代码的一部分创建或配置，而仿真器是独立的应用程序，以与其模拟的系统相同的方式公开功能。

使用 Mock 的优势包括：

- Mock 可以模拟应用程序无法控制的第三方服务，例如 API 和软件即服务（SaaS）提供程序，而无需直接访问这些服务。
- Mock 对于测试失败条件很有用，尤其是当此类情况难以模拟时（例如服务中断）更是如此。
- 配置完成后，Mock 可以提供快速本地测试。
- Mock 可以为几乎任何类型的对象提供替换行为，因此与仿真器相比，Mock 策略可以针对更多种类的服务创建覆盖面。
- 当新功能或行为可用时，Mock 测试可以更快地做出反应。通过使用通用 Mock 框架，您可以在更新的 AWS SDK 可用后立即模拟新功能。

Mock 测试有以下缺点：

- Mock 通常需要极大量的设置和配置工作，特别是在尝试确定来自不同服务的返回值以正确模拟响应时更是如此。
- Mock 由开发人员编写、配置且必须由开发人员维护，这会加重开发人员的职责。
- 您可能需要访问云才能了解服务的 API 和返回值。
- Mock 可能很难维护。当 Mock 模拟的云 API 签名更改或返回值架构发生变化时，您需要更新 Mock。如果您扩展应用程序逻辑以调用新的 API，则还需要更新 Mock。
- 使用 Mock 的测试可能在桌面环境中通过，但在云端失败。结果可能与当前 API 不匹配。无法测试服务配置和限额。
- Mock 框架仅限于测试或检测 AWS Identity and Access Management（IAM）policy 或限额限制。尽管 Mock 更擅长在授权失败或超过限额时进行模拟，但测试无法确定生产环境中实际会出现哪种结果。

使用仿真进行测试

仿真器通常是在本地运行的应用程序，它会模仿生产 AWS 服务。

仿真器的 API 与云对应项类似，可提供相似的返回值。它们还可以模拟由 API 调用启动的状态更改。例如，您可以借助 AWS SAM Local 使用 AWS SAM 运行函数，以仿真 Lambda 服务，从而可以快速调用函数。有关详细信息，请参阅《AWS Serverless Application Model 开发人员指南》中的 AWS SAM Local。

使用仿真器进行测试的优点包括以下几点：

- 仿真器可以促进快速本地开发迭代和测试。
- 仿真器为习惯于在本地环境中开发代码的开发人员提供熟悉的环境。例如，如果您熟悉 n 层应用程序的开发，则您可能拥有在本地计算机上运行的数据库引擎和 Web 服务器（类似于在生产环境中运行的数据库引擎和 Web 服务器），以提供快速的本地隔离测试功能。
- 仿真器不需要对云基础架构（例如开发人员云账户）进行任何更改，因此很容易使用现有的测试模式进行实施。

使用仿真器进行测试具有以下缺点：

- 仿真器可能很难设置和复制，尤其是在 CI/CD 管道中使用时更是如此。这样可能会增加管理自有软件的 IT 人员或开发人员的工作量。
- 仿真功能和 API 通常落后于服务更新。这样可能会导致发生错误，因为测试的代码与实际的 API 不匹配，并会阻碍新功能的采用。

- 仿真器需要支持、更新、错误修复和同等功能增强。这些是仿真器作者的职责，仿真器作者可以是某个第三方公司。
- 依赖于仿真器的测试可能会在本地提供成功的结果，但由于生产安全策略、服务间配置或超出 Lambda 限额，可能会在云端失败。
- 许多 AWS 服务没有可用的仿真器。如果您依赖于仿真，则部分应用程序可能无法提供令人满意的测试选项。

最佳实践

以下各节为成功测试无服务器应用程序提供了建议。

您可以在无服务器测试示例存储库中找到测试和测试自动化的实用示例。

优先考虑在云端进行测试

在云端进行测试可提供最可靠、最准确和最完整的测试覆盖范围。在云环境中进行测试不仅可以全面测试业务逻辑，还可以全面测试安全策略、服务配置、限额以及最新的 API 签名和返回值。

构建代码以提高测试可行性

通过将特定于 Lambda 的代码与您的核心业务逻辑分开，从而简化您的测试和 Lambda 函数。

您的 Lambda 函数处理程序应该是一个纤薄的适配器，它接收事件数据并仅将重要的详细信息传递给您的业务逻辑方法。通过此策略，您可以围绕业务逻辑进行全面测试，而不必担心特定于 Lambda 的详细信息。您的 AWS Lambda 函数应该不需要设置复杂的环境或大量依赖项来创建并初始化所测试组件。

一般来说，您应该编写一个处理程序，用于从传入的事件和上下文对象中提取和验证数据，然后将该输入发送到执行您的业务逻辑的方法。

加速开发反馈循环

有一些工具和技术可以加速开发反馈循环。例如，AWS SAM Accelerate 和 AWS CDK 监视模式都减少了更新云环境所需的时间。

GitHub 无服务器测试示例存储库中的示例探讨了其中一些技术。

我们还建议您在开发期间尽早创建和测试云资源，而不仅仅是在签入到源代码控制之后执行。这种做法有助于在开发解决方案时更快地进行探索和实验。此外，从开发计算机自动执行部署可以帮助您更快地发现云配置问题，并减少在更新和代码审查流程上浪费的工作量。

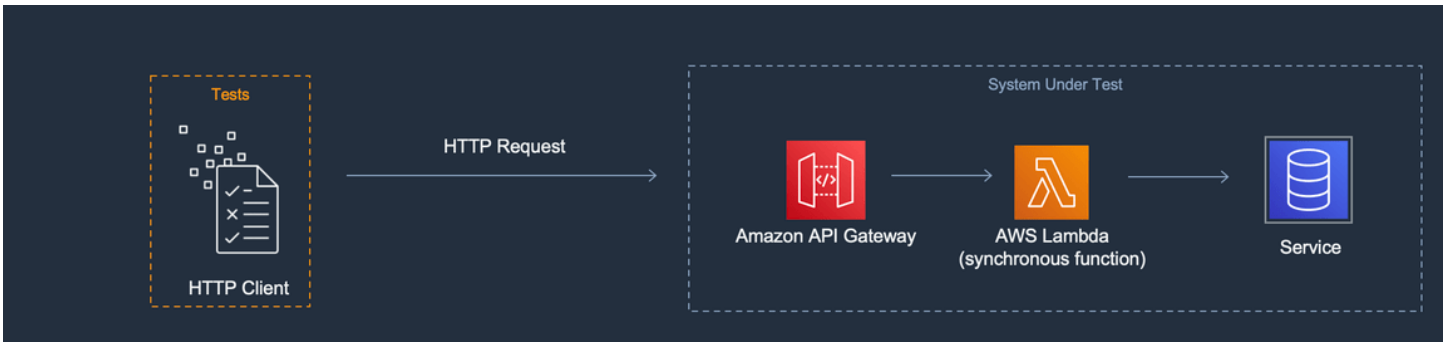
专注于集成测试

使用 Lambda 构建应用程序时，将组件一起测试是一项最佳实践。

针对两个或更多架构组件运行的测试称为集成测试。集成测试的目标不仅是了解您的代码将如何跨组件执行，而且要了解托管代码的环境将如何运行。端到端测试是特殊类型的集成测试，用于验证整个应用程序的行为。

要构建集成测试，请将您的应用程序部署到云环境。此操作可以在本地环境中完成，也可以通过 CI/CD 管道完成。然后，编写测试以执行被测系统（SUT）并验证预期行为。

例如，被测系统可能是使用 API Gateway、Lambda 和 DynamoDB 的应用程序。测试可以对 API Gateway 端点进行合成 HTTP 调用，并验证响应是否包含预期的负载。此测试可验证 AWS Lambda 代码是否正确，以及每项服务是否均已正确配置以处理请求，包括它们之间的 IAM 权限。此外，您可以将测试设计为写入各种大小的记录，以验证您的服务限额（例如 DynamoDB 中的最大记录大小）是否设置正确。



创建隔离的测试环境

在云端进行测试通常需要隔离的开发人员环境，这样测试、数据和事件就不会重叠。

一种方法是为每位开发人员提供一个专用 AWS 账户。这样将避免与资源命名发生冲突，而当多个开发人员在共享代码库中工作、尝试部署资源或调用 API 时，可能会发生此类冲突。

自动测试过程应为每个堆栈创建唯一命名资源。例如，您可以设置脚本或 TOML 配置文件，以便 AWS SAM CLI sam deploy 或 sam sync 命令将自动指定具有唯一前缀的堆栈。

在某些情况下，多名开发人员会共享一个 AWS 账户。这可能是由于堆栈中的资源在操作或预置和配置方面成本都很高昂。例如，可以共享数据库，以便更容易正确设置数据和为数据设定种子

如果开发人员共享账户，则应设置边界，以确定所有权并消除重叠。执行此操作的一种方法是使用开发人员用户 ID 作为堆栈名称的前缀。另一种常用方法是基于代码分支设置堆栈。借助分支边界，可实现

即隔离环境，开发人员又可以共享资源，例如关系数据库。当开发人员一次处理多个分支时，这种方法就是最佳实践。

在云端进行测试对于各个阶段的测试（包括单元测试、集成测试和端到端测试）而言都很有价值。保持适当的隔离至关重要；但您的 QA 环境最好与您的生产环境尽可能相似。出于此原因，团队为 QA 环境添加了更改控制流程。

对于预生产和生产环境，通常在账户级别绘制边界，以将工作负载与嘈杂的相邻问题隔离开来，并实施最低权限安全控制来保护敏感数据。工作负载有限额。您不希望测试占用为生产分配的限额（嘈杂的相邻）或访问客户数据。负载测试是您应该与生产堆栈隔离的另一项活动。

在任何情况下，都应为环境配置警报和控件，以避免产生不必要的支出。例如，您可以限制可创建的资源的类型、层级或大小，并在估计成本超出给定阈值时设置电子邮件警报。

使用 Mock 来实现隔离的业务逻辑

Mock 框架是编写快速单元测试的重要工具。当测试涵盖复杂的内部业务逻辑（例如数学或财务计算或模拟）时，此框架尤其有用。寻找具有大量测试用例或输入变体的单元测试，其中，这些输入不会改变调用其他云服务的模式或内容。

通过 Mock 进行的单元测试所涵盖的代码也应包含在云端测试中。之所以建议这样做，是因为开发人员笔记本电脑或生成计算机环境的配置可能与云端的生产环境不同。例如，当使用某些输入参数运行时，您的 Lambda 函数使用的内存或时间可能会比分配的内存或时间多。或者您的代码可能包含没有以相同方式配置的环境变量（或者根本未配置），这些差异可能导致代码的行为不同或失败。

对于集成测试而言，Mock 的优势较少，因为实施必要 Mock 的工作量会随着连接点数量的增加而加重。端到端测试不应使用 Mock，因为这些测试通常涉及无法使用 Mock 框架轻松模拟的状态和复杂逻辑。

最后，避免使用 Mock 模拟的云服务来验证服务调用是否正确实施。反之，应在云端进行云服务调用以验证行为、配置和功能实现。

请谨慎使用仿真器

仿真器在某些使用案例中可能很方便，例如，对于互联网访问受限、不可靠或速度慢的开发团队而言，确实如此。但是，在大多数情况下，请选择谨慎使用仿真器。

通过避免使用仿真器，您将能够使用最新的服务功能和最新的 API 进行构建和创新。您不会一直等待供应商发布以实现同等功能。您将减少在多个开发系统和生成计算机上购买和配置的前期和持续费用。此外，您也不会遇到许多云服务根本没有可用仿真器的问题。依赖于仿真的测试策略会使您无法使用这些服务（这会导致需要采用潜在成本更高昂的解决方法）或生成未经全面测试的代码和配置。

当您确实使用仿真进行测试时，仍必须在云端进行测试以验证配置，并测试与只能在仿真环境中模拟或 Mock 模拟的云服务的交互。

在本地测试所面临的挑战

当您使用仿真器和 Mock 模拟的调用在本地桌面上进行测试时，随着代码在 CI/CD 管道中从一个环境进行到另一个环境，您可能会遇到测试不一致的情况。在桌面上验证应用程序业务逻辑的单元测试可能无法准确测试云服务的关键方面。

以下示例提供了当使用 Mock 和仿真器进行本地测试时需要注意的情况：

示例：Lambda 函数创建一个 S3 存储桶

如果 Lambda 函数的逻辑依赖于创建 S3 存储桶，则完整的测试应确认已调用 Amazon S3 并已成功创建相应存储桶。

- 在 Mock 测试设置中，您可以 Mock 模拟成功响应，并可能会添加一个测试用例来处理失败响应。
- 在仿真测试场景中，可能会调用 CreateBucket API，但您需要注意，进行本地调用的身份不会源自 Lambda 服务。调用身份不会像在云端那样担任安全角色，因此将改用占位符身份验证，可能使用权限更宽松的角色或用户身份，而在云端运行时会有所不同。

Mock 和仿真设置将测试 Lambda 函数在调用 Amazon S3 时会做什么；但是，这些测试不会验证所配置的 Lambda 函数是否能够成功创建 Amazon S3 存储桶。您必须确保分配给该函数的角色具有允许该函数执行 `s3:CreateBucket` 操作的附加安全策略。否则，该函数在部署到云环境时可能会失败。

示例：Lambda 函数处理来自某个 Amazon SQS 队列的消息

如果 Amazon SQS 队列是 Lambda 函数的来源，则完整的测试应验证在将消息放入队列时是否已成功调用 Lambda 函数。

仿真测试和 Mock 测试通常设置为直接运行 Lambda 函数代码，并通过传递 JSON 事件负载（或反序列化对象）作为函数处理程序的输入来模拟 Amazon SQS 集成。

模拟 Amazon SQS 集成的本地测试将测试 Amazon SQS 使用给定负载调用 Lambda 函数时，该函数将执行什么操作，但测试不会验证 Amazon SQS 在部署到云环境时能否成功调用 Lambda 函数。

您在使用 Amazon SQS 和 Lambda 时可能会遇到的一些配置问题示例包括：

- Amazon SQS 可见性超时设置过低，从而导致在仅打算调用一次的情况下进行多次调用。

- Lambda 函数的执行角色不允许从队列读取消息（通过 `sqs:ReceiveMessage`、`sqs:DeleteMessage` 或 `sqs:GetQueueAttributes`）。
- 传递给 Lambda 函数的示例事件超出了 Amazon SQS 消息大小限额。因此，该测试无效，因为 Amazon SQS 永远无法发送这一大小的消息。

如这些示例所示，涵盖业务逻辑但不涵盖不同云服务之间配置的测试可能会提供不可靠的结果。

常见问题解答

我有一个 Lambda 函数，该函数可以在不调用任何其他服务的情况下执行计算并返回结果。我还需要在云端对其进行测试吗？

是。Lambda 函数的配置参数可能会改变测试结果。所有 Lambda 函数代码均依赖于超时和内存设置，如果这些设置不正确，这可能会导致函数失败。Lambda 策略还允许将标准输出记录到 Amazon CloudWatch 中。即使您的代码不直接调用 CloudWatch，也需要权限才能启用日志记录。这种必需的权限无法准确地 Mock 模拟或仿真。

在云端进行测试如何帮助单元测试？ 如果测试在云端并连接到其他资源，那不属于集成测试吗？

我们将单元测试定义为在架构组件上单独运行的测试，但这样并不能阻止测试包括可能调用其他服务或使用某些网络通信的组件。

许多无服务器应用程序都有可以单独测试的架构组件，即使在云端也是如此。一个示例为 Lambda 函数，该函数获取输入、处理数据并将消息发送到 Amazon SQS 队列。此函数的单元测试可能会测试输入值是否导致某些值出现在队列消息中。

考虑使用以“安排、执行、断言”模式编写的测试：

- 安排：分配资源（接收消息的队列和所测试的函数）。
- 执行：调用所测试的函数。
- 断言：检索函数发送的消息，并验证输出。

Mock 测试方法包括使用进行中的 Mock 对象模拟队列，以及创建包含 Lambda 函数代码的类或模块的进行中的实例。在断言阶段，将从 Mock 模拟对象中检索队列消息。

在基于云的方法中，测试将面向测试目的创建 Amazon SQS 队列，并将部署带有配置为使用隔离的 Amazon SQS 队列作为输出目标的环境变量的 Lambda 函数。运行 Lambda 函数后，测试将从 Amazon SQS 队列中检索消息。

基于云的测试将运行相同的代码，断言相同的行为，并验证应用程序的功能正确性。但是，它还有一个额外的优势，那就是能够验证 Lambda 函数的设置：IAM 角色、IAM policy 以及函数的超时和内存设置。

后续步骤和资源

使用以下资源可了解更多信息，并探索测试的实际示例。

示例实施

GitHub 上的无服务器测试示例存储库包含遵循本指南中所描述模式和最佳实践的具体测试示例。存储库包含前几节中描述的示例代码和 Mock、仿真和云端测试流程的引导式演练。使用此存储库可以快速了解 AWS 发布的最新无服务器测试指南。

延伸阅读

访问 Serverless Land，以访问有关 AWS 无服务器技术的最新博客、视频和培训。

还建议您阅读以下 AWS 博客文章：

- 使用 AWS SAM Accelerate 加速无服务器开发（AWS 博客文章）
- 使用 CDK Watch 提高开发速度（AWS 博客文章）
- 使用 AWS Step Functions Local Mock 模拟服务集成（AWS 博客文章）
- 测试无服务器应用程序入门（AWS 博客文章）

工具

- AWS SAM — 测试和调试无服务器应用程序
- AWS SAM – 与自动测试集成
- Lambda – 在 Lambda 控制台中测试 Lambda 函数

使用来自其 AWS 他服务的事件调用 Lambda

某些 AWS 服务可以使用触发器直接调用 Lambda 函数。这些服务将事件推送到 Lambda，并在指定事件发生时立即调用该函数。触发器适用于离散事件和实时处理。当您使用 Lambda 控制台创建触发器时，控制台会与相应的 AWS 服务交互以配置该服务的事件通知。触发器实际上由生成事件的服务而不是 Lambda 存储和管理。

事件使用 JSON 格式的数据结构。JSON 结构因生成它的服务和事件类型而异，但它们都包含函数处理事件所需的数据。

一个函数可具有多个触发器。每个触发器都可以作为一个客户端独立调用您的函数，Lambda 传递到您的函数的每个事件仅具有一个触发器的数据。Lambda 将事件文档转换为一个对象，并将该对象传递给函数处理程序。

取决于服务，事件驱动调用可以是同步的，也可以是异步的。

- 对于同步调用，生成事件的服务服务等待来自您的函数的响应。该服务定义函数需要在响应中返回的数据。该服务控制错误策略，例如是否重试错误。
- 对于异步调用，Lambda 先将事件排队，然后再将事件传递给您的函数。当 Lambda 将事件排队时，它会立即向生成事件的服务发送成功响应。函数处理事件后，Lambda 不会返回事件生成服务的响应。

创建触发器

创建触发器最简单的方法是使用 Lambda 控制台。使用控制台创建触发器时，Lambda 会自动将所需的权限添加到函数基于资源的策略。

要使用 Lambda 控制台创建触发器

1. 打开 Lamba 控制台的函数页面。
2. 选择要为其创建触发器的函数。
3. 在函数概述窗格中，选择添加触发器。
4. 选择要调用函数的 AWS 服务。
5. 填写触发器配置窗格中的选项，然后选择添加。根据您选择调用函数的 AWS 服务，触发器配置选项会有所不同。

可以调用 Lambda 函数的服务

下表列出了可以调用 Lambda 函数的服务。

服务	调用方法
Amazon Managed Streaming for Apache Kafka	事件源映射
自行管理的 Apache Kafka	事件源映射
Amazon API Gateway	事件驱动；同步调用
AWS CloudFormation	事件驱动；异步调用
Amazon CloudWatch Logs	事件驱动；异步调用
AWS CodeCommit	事件驱动；异步调用
AWS CodePipeline	事件驱动；异步调用
Amazon Cognito	事件驱动；同步调用
AWS Config	事件驱动；异步调用
Amazon Connect	事件驱动；同步调用
Amazon DynamoDB	事件源映射
Amazon Elastic File System	特殊集成
Elastic Load Balancing（应用程序负载均衡器）	事件驱动；同步调用
Amazon EventBridge (CloudWatch Events)	事件驱动；异步调用（事件总线）、同步或异步调用（管道和计划）
AWS IoT	事件驱动；异步调用
Amazon Kinesis	事件源映射

服务	调用方法
Amazon Data Firehose	事件驱动；同步调用
Amazon Lex	事件驱动；同步调用
Amazon MQ	事件源映射
Amazon Simple Email Service	事件驱动；异步调用
Amazon Simple Notification Service	事件驱动；异步调用
Amazon Simple Queue Service	事件源映射
Amazon Simple Storage Service (Amazon S3)	事件驱动；异步调用
Amazon Simple Storage Service 批处理	事件驱动；同步调用
Secrets Manager	特殊集成
AWS Step Functions	事件驱动；同步或异步调用
Amazon VPC Lattice	事件驱动；同步调用
AWS X-Ray	特殊集成

将 Lambda 与自行管理的 Apache Kafka 结合使用

> **Note**
>
> 如果想要将数据发送到 Lambda 函数以外的目标，或要在发送数据之前丰富数据，请参阅 Amazon EventBridge Pipes（Amazon EventBridge 管道）。

Lambda 支持将 Apache Kafka 作为事件源。Apache Kafka 是一个开源事件流平台，支持数据管道和流分析等工作负载。

您可以使用AWS托管的 Kafka 服务 Amazon Managed Streaming for Apache Kafka（Amazon MSK），简称自行管理的 Kafka 集群。有关结合 Amazon MSK 使用 Lambda 的详细信息，请查阅 结合 Amazon MSK 使用 Lambda。

本主题介绍了如何将 Lambda 与自行管理的 Kafka 集群结合使用。在AWS术语中，自行管理的群集包括非AWS托管 Kafka 集群。例如，可以通过 Confluent Cloud 等云提供程序来托管 Kafka 集群。

Apache Kafka 作为事件源，运行方式与使用 Amazon Simple Queue Service (Amazon SQS) 或 Amazon Kinesis 相似。Lambda 在内部轮询来自事件源的新消息，然后同步调用目标 Lambda 函数。Lambda 批量读取消息，并将这些消息作为事件有效负载提供给您的函数。最大批处理大小可配置。（默认值为 100 个消息。）

> **Warning**
>
> Lambda 事件源映射至少处理每个事件一次，有可能出现重复处理记录的情况。为避免与重复事件相关的潜在问题，我们强烈建议您将函数代码设为幂等性。要了解更多信息，请参阅 AWS 知识中心的如何使我的 Lambda 函数具有幂等性。

对于基于 Kafka 的事件源，Lambda 支持处理控制参数，例如批处理时段和批处理大小。有关更多信息，请参阅 批处理行为。

有关如何使用自行管理的 Kafka 作为事件源的示例，请参阅AWS计算博客上的使用自行托管的 Apache Kafka 作为 AWS Lambda 事件源。

主题

- 示例事件

- 为 Lambda 配置自托管式 Apache Kafka 事件源
- 使用 Lambda 处理自托管式 Apache Kafka 消息
- 对自行管理的 Apache Kafka 事件源使用事件筛选
- 捕获自托管式 Apache Kafka 事件源的丢弃批次
- 纠正自托管式 Apache Kafka 事件源映射错误

示例事件

当 Lambda 调用 Lambda 函数时，它会在事件参数中发送一批消息。事件负载包含一个消息数组。每个数组项目都包含 Kafka 主题和 Kafka 分区标识符的详细信息，以及时间戳和 base64 编码的消息。

```
{
   "eventSource": "SelfManagedKafka",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
```

```
                    ]
                }
            ]
        }
    ]
  }
}
```

为 Lambda 配置自托管式 Apache Kafka 事件源

在为自托管式 Apache Kafka 集群创建事件源映射之前，必须确保集群及其所在的 VPC 配置正确。您还要确保 Lambda 函数的执行角色具有必要的 IAM 权限。

按照以下各节中的说明配置自托管式 Apache Kafka 集群和 Lambda 函数。要了解如何创建事件源映射，请参阅the section called “将 Kafka 集群添加为事件源”。

主题

- Kafka 集群身份验证
- API 访问和 Lambda 函数权限
- 配置网络安全

Kafka 集群身份验证

Lambda 支持多种方法来使用自行管理的 Apache Kafka 集群进行身份验证。请确保将 Kafka 集群配置为使用支持的下列身份验证方法之一：有关 Kafka 安全的更多信息，请参阅 Kafka 文档的安全部分。

SASL/SCRAM 身份验证

Lambda 支持使用传输层安全性协议（TLS）加密 (SASL_SSL) 进行简单身份验证和安全层/加盐质疑应答身份验证机制（SASL/SCRAM）身份验证。Lambda 发送已加密凭据以使用集群进行身份验证。Lambda 不支持使用明文 (SASL_PLAINTEXT) 的 SASL/SCRAM。有关 SASL/SCRAM 身份验证的更多信息，请参阅 RFC 5802。

Lambda 还支持 SASL/PLAIN 身份验证。由于此机制使用明文凭证，因此与服务器的连接必须使用 TLS 加密以确保凭证受到保护。

为了 SASL 身份验证，需要将登录凭证作为密钥存储在 AWS Secrets Manager 中。有关使用 Secrets Manager 的更多信息，请参阅 AWS Secrets Manager 用户指南中的教程：创建和检索密钥。

> **Important**
>
> 要使用 Secrets Manager 进行身份验证，密钥必须存储在 Lambda 函数所在的同一 AWS 区域中。

双向 TLS 身份验证

双向 TLS（mTLS）在客户端和服务器之间提供双向身份验证。客户端向服务器发送证书以便服务器验证客户端，而服务器又向客户端发送证书以便客户端验证服务器。

在自行托管的 Apache Kafka 中，Lambda 充当客户端。您可以配置客户端证书（作为 Secrets Manager 中的密钥），以使用 Kafka 代理对 Lambda 进行身份验证。客户端证书必须由服务器信任存储中的 CA 签名。

Kafka 集群向 Lambda 发送服务器证书，以便使用 Lambda 对 Kafka 代理进行身份验证。服务器证书可以是公有 CA 证书。也可以是私有 CA/自签名证书。公有 CA 证书必须由 Lambda 信任存储中的证书颁发机构（CA）签名。对于私有 CA /自签名证书，您可以配置服务器根 CA 证书（作为 Secrets Manager 中的密钥）。Lambda 使用根证书来验证 Kafka 代理。

有关 mTLS 的更多信息，请参阅为作为事件源的 Amazon MSK 引入双向 TLS 身份验证。

配置客户端证书密钥

CLIENT_CERTIFICATE_TLS_AUTH 密钥需要证书字段和私有密钥字段。对于加密的私有密钥，密钥需要私有密钥密码。证书和私有密钥必须采用 PEM 格式。

> **Note**
>
> Lambda 支持 PBES1（而不是 PBES2）私有密钥加密算法。

证书字段必须包含证书列表，首先是客户端证书，然后是任何中间证书，最后是根证书。每个证书都必须按照以下结构在新行中启动：

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager 支持最多包含 65536 字节的密钥，这为长证书链提供了充足的空间。

私有密钥必须采用 PKCS #8 格式，并具有以下结构：

```
-----BEGIN PRIVATE KEY-----
        <private key contents>
-----END PRIVATE KEY-----
```

对于加密的私有密钥，请使用以下结构：

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
         <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

以下示例显示使用加密私有密钥进行 mTLS 身份验证的密钥内容。对于加密的私有密钥，可以在密钥中包含私有密钥密码。

```
{"privateKeyPassword":"testpassword",
"certificate":"-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
"privateKey":"-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

配置服务器根 CA 证书密钥

如果您的 Kafka 代理使用 TLS 加密（具有由私有 CA 签名的证书），则创建此密钥。您可以将 TLS 加密用于 VPC、SASL/SCRAM、SASL/PLAIN 或 mTLS 身份验证。

服务器根 CA 证书密钥需要一个字段，其中包含 PEM 格式的 Kafka 代理的根 CA 证书。以下示例显示密钥的结构。

```
{"certificate":"-----BEGIN CERTIFICATE-----
MIID7zCCAtegAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
-----END CERTIFICATE-----"
}
```

API 访问和 Lambda 函数权限

除了访问自行托管的 Kafka 集群外，您的 Lambda 函数还需要执行各种 API 操作的权限。您可以为函数的执行角色添加这些权限。如果您的用户需要访问任何 API 操作，请将所需权限添加到 AWS Identity and Access Management（IAM）用户或角色的身份策略中。

所需的 Lambda 函数权限

要在 Amazon CloudWatch Logs 中创建日志并将日志存储到日志组，Lambda 函数必须在它的执行角色中具有以下权限：

- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

可选的 Lambda 函数权限

您的 Lambda 函数还可能需要权限来：

- 描述您的 Secrets Manager 密钥。
- 访问 AWS Key Management Service（AWS KMS）客户管理的密钥。
- 访问 Amazon VPC。
- 将失败调用的记录发送到目标。

Secrets Manager 和 AWS KMS 权限

根据您为 Kafka 代理配置的访问控制类型，Lambda 函数可能需要访问您的 Secrets Manager 密钥或解密 AWS KMS 客户管理的密钥的权限。要连接到这些资源，函数的执行角色必须具有以下权限：

- secretsmanager:GetSecretValue
- kms:Decrypt

VPC 权限

如果只有 VPC 内的用户才能访问您自行托管的 Apache Kafka 集群，则 Lambda 函数必须具有访问 Amazon VPC 资源的权限。这些资源包括您的 VPC、子网、安全组和网络接口。要连接到这些资源，函数的执行角色必须具有以下权限：

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups

向执行角色添加权限

要访问自行管理的 Apache Kafka 集群使用的其他AWS服务，Lambda 需使用您在 Lambda 函数执行角色中定义的权限策略。

默认情况下，Lambda 无权为自行管理的 Apache Kafka 集群执行必需或可选操作。您必须在 IAM 信任策略中为您的执行角色创建和定义这些操作。此示例演示了如何创建允许 Lambda 访问您的 Amazon VPC 资源的策略。

```
{
        "Version":"2012-10-17",
        "Statement":[
           {
              "Effect":"Allow",
              "Action":[
                 "ec2:CreateNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeVpcs",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeSubnets",
                 "ec2:DescribeSecurityGroups"
              ],
              "Resource":"*"
```

```
            }
        ]
    }
```

使用 IAM policy 授予用户访问权限

默认情况下，用户和角色无权执行事件源 API 操作。要向组织或账户中的用户授予访问权限，您可以创建或更新基于身份的策略。有关更多信息，请参阅 IAM 用户指南中的使用策略控制对AWS资源的访问权限。

配置网络安全

要通过事件源映射向 Lambda 提供对自主管理的 Apache Kafka 的完全访问权限，集群必须使用公有端点（公有 IP 地址），或者您必须提供对您在其中创建了集群的 Amazon VPC 的访问权限。

将自主管理的 Apache Kafka 与 Lambda 配合使用时，我们建议您创建 AWS PrivateLinkVPC 端点，并向函数提供对 Amazon VPC 中资源的访问权限。

创建端点以提供对以下资源的访问权限：

- Lambda：为 Lambda 服务主体创建端点。
- AWS STS：为 AWS STS 创建端点，以便服务主体代您代入角色。
- Secrets Manager：如果集群使用 Secrets Manager 来存储凭证，请为 Secrets Manager 创建端点。

也可以在 Amazon VPC 中的每个公有子网上配置 NAT 网关。有关更多信息，请参阅 the section called “VPC 函数的互联网访问权限”。

为自主管理的 Apache Kafka 创建事件源映射时，Lambda 会检查为 Amazon VPC 配置的子网和安全组是否已经存在弹性网络接口（ENI）。如果 Lambda 发现现有 ENI，则会尝试重用这些 ENI。否则，Lambda 会创建新的 ENI 来连接到事件源并调用函数。

> **Note**
>
> Lambda 函数始终在 Lambda 服务拥有的 Amazon VPC 中运行。函数的 VPC 配置不会影响事件源映射。只有事件源的网络配置才能决定 Lambda 连接到事件源的方式。

为包含集群的 Amazon VPC 配置安全组。默认情况下，自主管理的 Apache Kafka 使用以下端口：9092。

- 入站规则：允许与事件源关联安全组的默认集群端口的所有流量。
- 出站规则：允许所有目标的端口 443 上的所有流量。允许与事件源关联安全组的默认集群的所有流量。
- Amazon VPC 端点入站规则：如果您正在使用 Amazon VPC 端点，则与 Amazon VPC 端点关联的安全组，必须允许来自集群安全组的端口 443 上的入站流量。

如果集群使用身份验证，您还可以限制 Secrets Manager 端点的端点策略。要调用 Secrets Manager API，Lambda 会使用函数角色而非 Lambda 服务主体。

Example VPC 端点策略：Secrets Manager 端点

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
                  ]
              },
              "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
          }
      ]
  }
```

当您使用 Amazon VPC 端点时，AWS 会使用端点的弹性网络接口（ENI）路由 API 调用来调用函数。Lambda 服务主体需要针对使用这些 ENI 的任何函数调用 `lambda:InvokeFunction`。默认情况下，Amazon VPC 端点具有开放的 IAM 策略，允许对资源进行广泛访问。要在生产环境中将自主管理的 Apache Kafka 与 Lambda 配合使用，您可以将这些策略限制为仅允许特定的主体访问特定的角色和函数。

Example 端点策略：Lambda 端点

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
```

```
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "arn:aws::lambda:us-west-2:123456789012:function:my-function"
        }
    ]
}
```

此外，对于事件源映射，如果您希望与 Lambda 集成的资源在 AWS 账户中部署，则 Lambda 服务主体必须执行 `sts:AssumeRole` 才能代入使用弹性网络接口（ENI）的角色。

Example 端点策略：AWS STS 端点

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "arn:aws::iam::123456789012:role/my-role"
        }
    ]
}
```

Warning

将端点策略限制为仅允许来自组织内部的 API 调用，这将阻止事件源映射正常运行。

使用 Lambda 处理自托管式 Apache Kafka 消息

> **Note**
>
> 如果想要将数据发送到 Lambda 函数以外的目标，或要在发送数据之前丰富数据，请参阅 Amazon EventBridge Pipes（Amazon EventBridge 管道）。

主题

将 Kafka 集群添加为事件源

创建事件源映射，使用 Lambda 控制台、AWS开发工具包，或 AWS Command Line Interface (AWS CLI) 将您的 Kafka 集群添加为 Lambda 函数触发器。

本节介绍了如何使用 Lambda 控制台和 AWS CLI 创建事件源映射。

先决条件

- 自行管理的 Apache Kafka 集群。Lambda 支持 Apache Kafka 版本 0.10.1.0 及更高版本。
- 一个有权访问您自行管理的 Kafka 集群所用 AWS 资源的执行角色。

可自定义的使用者组 ID

将 Kafka 设置为事件源时，您可以指定使用者组 ID。此使用者组 ID 是您希望 Lambda 函数加入的 Kafka 使用者组的现有标识符。您可以使用此功能将任何正在进行的 Kafka 记录处理设置从其他使用者无缝迁移到 Lambda。

如果指定了使用者组 ID，并且该使用者组中还有其他活跃的轮询器，则 Kafka 会向所有使用者分发消息。换句话说，Lambda 不会收到 Kafka 主题的所有消息。如果希望 Lambda 处理主题中的所有消息，请关闭该使用者组中的任何其他轮询器。

此外，如果指定了使用者组 ID，而 Kafka 找到了具有相同 ID 的有效现有使用者组，则 Lambda 会忽略事件源映射的 `StartingPosition` 参数。相反，Lambda 开始根据使用者组的已提交偏移量处理记录。如果指定了使用者组 ID，而 Kafka 找不到现有使用者组，则 Lambda 会使用指定的 `StartingPosition` 配置事件源。

在所有 Kafka 事件源中，您指定的使用者组 ID 必须是唯一的。在使用指定的使用者组 ID 创建 Kafka 事件源映射后，无法更新此值。

添加自行管理的 Kafka 集群（控制台）

按照以下步骤将自行管理的 Apache Kafka 集群和 Kafka 主题添加为 Lambda 函数的触发器。

将 Apache Kafka 触发器添加到 Lambda 函数（控制台）

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择 Lambda 函数的名称。
3. 在 Function overview（函数概览）下，选择 Add trigger（添加触发器）。
4. 在 Trigger configuration（触发配置）下，执行以下操作：

   a. 选择 Apache Kafka 触发类型。

   b. 对于 Bootstrap servers（Bootstrap 引导服务器），输入集群中 Kafka 代理的主机和端口对地址，然后选择 Add（添加）。对集群中的每个 Kafka 代理重复此操作。

   c. 对于 Topic name（主题名称），输入用于在集群中存储记录的 Kafka 主题的名称。

   d. （可选）对于 Batch size（批处理大小），输入要在单个批次中接收的最大记录数。

   e. 对于 Batch window（批处理时段），输入 Lambda 在调用函数之前收集记录所花费的最大秒数。

   f. （可选）对于 Consumer group ID（使用者组 ID），输入要加入的 Kafka 使用者组的 ID。

   g. （可选）对于起始位置，选择最新即可从最新记录开始读取流，选择最早即可从最早的可用记录开始读取流，选择在时间戳处即可从指定的时间戳开始读取流。

   h. （可选）对于 VPC，请为您的 Kafka 集群选择 Amazon VPC。然后，选择 VPC subnets（VPC 子网）和 VPC security groups（VPC 安全组）。

      如果仅 VPC 内部的用户访问代理，则需要此设置。

   i. （可选）对于 Authentication（身份验证），请选择 Add（添加），然后执行以下操作：

      i. 选择集群中 Kafka 代理的访问权限或身份验证协议。

- 如果 Kafka 代理使用 SASL/PLAIN 身份验证，请选择 BASIC_AUTH。
- 如果代理使用 SASL/SCRAM 身份验证，请选择其中一个 SASL_SCRAM 协议。
- 如果要配置 mTLS 身份验证，请选择 CLIENT_CERTIFICATE_TLS_AUTH 协议。

ii. 对于 SASL/SCRAM 或 mTLS 身份验证，请选择包含 Kafka 集群凭据的 Secrets Manager 私有密钥。

j. （可选）对于 Encryption（加密），如果您的 Kafka 代理使用由私有 CA 签名的证书，请选择包含根 CA 证书的 Secrets Manager 密钥，Kafka 代理使用该证进行 TLS 加密。

此设置适用于 SASL/SCRAM 或 SASL/PLAIN 的 TLS 加密，以及 mTLS 身份验证。

k. 要在禁用状态下创建触发器以进行测试（推荐），请清除 Enable trigger（启用触发器）。或者，要立即启用该触发器，请选择 Enable trigger（启用触发器）。

5. 要创建触发器，请选择 Add（添加）。

添加自行管理的 Kafka 集群 (AWS CLI)

使用以下示例 AWS CLI 命令为 Lambda 函数创建和查看自行管理的 Apache Kafka 触发器。

使用 SASL/SCRAM

如果 Kafka 用户通过互联网访问您的 Kafka 代理，则需要指定为 SASL/SCRAM 身份验证创建的 Secrets Manager。以下示例使用 create-event-source-mapping AWS CLI 命令将名为 `my-kafka-function` 的 Lambda 函数映射至名为 `AWSKafkaTopic` 的 Kafka 主题。

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-east-1:111122223333:secret:MyBrokerSecretName \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

使用 VPC

如果仅 VPC 中的 Kafka 用户访问 Kafka 代理，则必须指定 VPC、子网和 VPC 安全组。以下示例使用 create-event-source-mapping AWS CLI 命令将名为 `my-kafka-function` 的 Lambda 函数映射至名为 `AWSKafkaTopic` 的 Kafka 主题。

```
aws lambda create-event-source-mapping \
```

```
  --topics AWSKafkaTopic \
  --source-access-configuration '[{"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0011001100"}, {"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0022002200"}, {"Type": "VPC_SECURITY_GROUP", "URI":
 "security_group:sg-0123456789"}]' \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

使用 AWS CLI 查看状态

以下示例使用 get-event-source-mapping AWS CLI 命令来描述您创建的事件源映射的状态。

```
aws lambda get-event-source-mapping
          --uuid dh38738e-992b-343a-1077-3478934hjkfd7
```

自行管理的 Apache Kafka 配置参数

所有 Lambda 事件源类型共享相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，只有一些参数适用于 Apache Kafka。

参数	必需	默认值	备注
BatchSize	否	100	最大值：10000
已启用	否	已启用	none
FunctionName	是	不适用	none
FilterCriteria	否	不适用	控制 Lambda 向您的函数发送的事件
MaximumBatchingWindowInSeconds	否	500 毫秒	批处理行为
SelfManagedEventSource	Y	不适用	Kafka 代理列表。只能在 Create（创建）设置

参数	必需	默认值	备注
SelfManagedKafkaEventSourceConfig	否	包含 ConsumerGroupId 字段，该字段默认为唯一值。	只能在 Create（创建）设置
SourceAccessConfigurations	否	无凭证	集群的 VPC 信息或身份验证凭据 对于 SASL_PLAIN，设置为 BASIC_AUTH
StartingPosition	Y	不适用	AT_TIMESTAMP、TRIM_HORIZON 或 LATEST 只能在 Create（创建）设置
StartingPositionTimestamp	否	不适用	当 StartingPosition 设置为 AT_TIMESTAMP 时，为必需项
主题	Y	不适用	主题名称 只能在 Create（创建）设置

将 Kafka 集群用作事件源

当您添加 Apache Kafka 或 Amazon MSK 集群作为 Lambda 函数的触发器时，该集群将用作事件源。

Lambda 根据您指定的 `StartingPosition`，从您在 CreateEventSourceMapping 请求中指定为 `Topics` 的 Kafka 主题读取事件数据。成功进行处理后，会将 Kafka 主题提交给 Kafka 集群。

如果您指定 `StartingPosition` 作为 `LATEST`，则 Lambda 开始读取主题下每个分区中的最新消息。由于在触发器配置后 Lambda 开始读取消息之前可能会有一些延迟，因此 Lambda 不会读取在此窗口中生成的任何消息。

Lambda 处理来自一个或多个指定 Kafka 主题分区的记录，并将 JSON 有效负载发送到您的函数。当有更多记录可用时，Lambda 根据您在 CreateEventSourceMapping 中指定的 `BatchSize` 值，继续对记录进行批处理，直到函数赶上主题的速度。

如果函数为批处理中的任何消息返回错误，Lambda 将重试整批消息，直到处理成功或消息过期为止。您可以将所有重试都失败的记录发送到失败时的目标，以供日后处理。

> **Note**
>
> 尽管 Lambda 函数的最大超时限制通常为 15 分钟，但 Amazon MSK、自行管理的 Apache Kafka、Amazon DocumentDB、Amazon MQ for ActiveMQ 和 RabbitMQ 的事件源映射，仅支持最大超时限制为 14 分钟的函数。此约束可确保事件源映射可以正确处理函数错误和重试。

轮询和流的起始位置

请注意，事件源映射创建和更新期间的流轮询最终是一致的。

- 在事件源映射创建期间，可能需要几分钟才能开始轮询来自流的事件。
- 在事件源映射更新期间，可能需要几分钟才能停止和重新开始轮询来自流的事件。

此行为意味着，如果你指定 `LATEST` 作为流的起始位置，事件源映射可能会在创建或更新期间错过事件。为确保不会错过任何事件，请将流的起始位置指定为 `TRIM_HORIZON` 或 `AT_TIMESTAMP`。

Kafka 事件源的自动伸缩

当您最初创建 Apache Kafka 事件源 时，Lambda 会分配一个使用者来处理 Kafka 主题中的所有分区。每个使用者都使用多个并行运行的处理器来处理增加的工作负载。此外，Lambda 会根据工作负载自动增加或缩减使用者的数量。要保留每个分区中的消息顺序，使用者的最大数量为主题中每个分区一个使用者。

Lambda 会按一分钟的间隔时间来评估主题中所有分区的使用者偏移滞后。如果延迟太高，则分区接收消息的速度比 Lambda 处理消息的速度更快。如有必要，Lambda 会在主题中添加或删除使用者。增加或移除使用者的扩缩过程会在评估完成后的三分钟内进行。

如果您的目标 Lambda 函数过载，Lambda 会减少使用者的数量。此操作通过减少使用者可以检索和发送到函数的消息数来减少函数的工作负载。

要监控 Kafka 主题的吞吐量，您可以查看 Apache Kafka 使用者指标，例如 `consumer_lag` 和 `consumer_offset`。要检查并行发生的函数调用次数，还可以监控函数的并发指标。

Amazon CloudWatch 指标

Lambda 会在您的函数处理记录时发出 `OffsetLag` 指标。此指标的值是写入 Kafka 事件源主题的最后一条记录与函数的使用者组处理的最后一条记录之间的偏移量差值。您可以使用 `OffsetLag` 来估计添加记录和使用者组处理记录之间的延迟。

如果 `OffsetLag` 呈上升趋势，则可能表明函数的使用者组中的轮询器存在问题。有关更多信息，请参阅 查看 Lambda 函数的指标。

对自行管理的 Apache Kafka 事件源使用事件筛选

您可以使用事件筛选，控制 Lambda 将流或队列中的哪些记录发送给函数。有关事件筛选工作原理的一般信息，请参阅 the section called “事件筛选”。

本节重点介绍自行管理的 Apache Kafka 事件源的事件筛选。

主题

- 自行管理的 Apache Kafka 事件筛选基础知识

自行管理的 Apache Kafka 事件筛选基础知识

假设创建者以有效的 JSON 格式或纯字符串的形式将消息写入自行管理的 Apache Kafka 集群中的主题。示例记录将如下所示，`value` 字段中的消息会转换为 Base64 编码字符串。

```
{
    "mytopic-0":[
        {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

假设 Apache Kafka 创建器以如下 JSON 格式将消息写入主题。

```
{
```

```
    "device_ID": "AB1234",
    "session":{
        "start_time": "yyyy-mm-ddThh:mm:ss",
        "duration": 162
    }
}
```

您可以使用 value 键筛选记录。假设您只想筛选 device_ID 以字母 AB 开头的记录。FilterCriteria 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\": \"AB\" } ] } }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 Pattern 的值。

```
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
      }
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
```

```
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }'
```

通过自行管理的 Apache Kafka，您还可以筛选消息为纯字符串的记录。假设您想忽略字符串为“错误”的消息。`FilterCriteria` 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 `Pattern` 的值。

```
{
    "value": [
        {
        "anything-but": [ "error" ]
        }
    ]
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

自行管理的 Apache Kafka 消息必须是 UTF-8 编码的字符串，可以是纯字符串或 JSON 格式。这是因为 Lambda 在应用筛选条件之前将 Kafka 字节数组解码为 UTF-8。如果您的消息使用另一种编码，例如 UTF-16 或 ASCII，或者消息格式与 `FilterCriteria` 格式不匹配，则 Lambda 仅处理元数据筛选条件。下表汇总了具体行为：

传入消息格式	消息属性的筛选条件模式格式	导致的操作
纯字符串	纯字符串	Lambda 根据您的筛选条件进行筛选。
纯字符串	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
纯字符串	有效 JSON	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	纯字符串	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	有效 JSON	Lambda 根据您的筛选条件进行筛选。
非 UTF-8 编码字符串	JSON、纯字符串或无模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。

捕获自托管式 Apache Kafka 事件源的丢弃批次

要保留失败的事件源映射调用的记录，请在函数的事件源映射中添加一个目标。发送到目标的每条记录都是一个 JSON 文档，其中包含有关失败调用的元数据。您可以将任何 Amazon SNS 主题、Amazon SQS 队列或 S3 存储桶配置为目标。您的执行角色必须具有目标的权限：

- 对于 SQS 目标：sqs:SendMessage
- 对于 SNS 目标：sns:Publish
- 对于 S3 存储桶目标：s3:PutObject 和 s3:ListBuckets

您必须在 Apache Kafka 集群 VPC 中为故障目标服务部署 VPC 端点。

此外，如果您在目标上配置了 KMS 密钥，则根据具体目标类型，Lambda 需要以下权限：

- 如果您已使用自己的 KMS 密钥为 S3 目标启用加密，则需要 kms:GenerateDataKey。如果 KMS 密钥和 S3 存储桶目标与您的 Lambda 函数和执行角色位于不同的账户中，请将 KMS 密钥配置为信任执行角色以允许 kms:GenerateDataKey。
- 如果您已使用自己的 KMS 密钥为 SQS 目标启用加密，则需要 kms:Decrypt 和 kms:GenerateDataKey。如果 KMS 密钥和 SQS 队列目标与您的 Lambda 函数和执行角色位于不同的账户中，请将 KMS 密钥配置为信任执行角色以允许 kms:Decrypt、kms:GenerateDataKey、kms:DescribeKey 和 kms:ReEncrypt。
- 如果您已使用自己的 KMS 密钥为 SNS 目标启用加密，则需要 kms:Decrypt 和 kms:GenerateDataKey。如果 KMS 密钥和 SNS 主题目标与您的 Lambda 函数和执行角色位于不同的账户中，请将 KMS 密钥配置为信任执行角色以允许 kms:Decrypt、kms:GenerateDataKey、kms:DescribeKey 和 kms:ReEncrypt。

为自托管式 Apache Kafka 事件源映射配置失败时的目标

要使用控制台配置失败时的目标，请执行以下步骤：

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 在 Function overview (函数概览) 下，选择 Add destination (添加目标)。
4. 对于源，请选择事件源映射调用。
5. 对于事件源映射，请选择为此函数配置的事件源。
6. 在条件中，选择失败时。对于事件源映射调用，这是唯一可接受的条件。
7. 对于目标类型，请选择 Lambda 要发送调用记录的目标类型。
8. 对于 Destination (目标)，请选择一个资源。
9. 选择保存。

您还可以使用 AWS CLI 配置失败时的目标。例如，以 create-event-source-mapping 命令将带有 SQS 失败时目标的事件源映射添加到 `MyFunction`：

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
```

```
--event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

以下 update-event-source-mapping 命令将 S3 失败时目标添加到与输入 uuid 关联的事件源：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

要移除目标，请提供一个空字符串作为 destination-config 参数的实际参数：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

SNS 和 SQS 示例调用记录

以下示例显示了 Lambda 在 Kafka 事件源调用失败时向 SNS 主题或 SQS 队列目标发送的内容。recordsInfo 下面的每个密钥都包含 Kafka 主题和分区，用连字符分隔。例如，对于密钥 "Topic-0"，Topic 是 Kafka 主题，0 是分区。对于每个主题和分区，可以使用偏移量和时间戳数据来查找原始调用记录。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
```

```
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    }
}
```

S3 目标示例调用记录

对于 S3 目标，Lambda 会将整个调用记录以及元数据发送到目标。以下示例显示了 Lambda 因调用 Kafka 事件源失败而向 S3 存储桶目标发送消息。除了针对 SQS 和 SNS 目标的上一示例中的所有字段外，payload 字段还包含作为转义 JSON 字符串的原始调用记录。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
```

```
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

> **Tip**
>
> 我们建议在目标存储桶上启用 S3 版本控制。

纠正自托管式 Apache Kafka 事件源映射错误

以下主题针对在使用自托管式 Apache Kafka 和 Lambda 时可能遇到的错误和问题提供了故障排除建议。如果您发现某个问题未在此处列出，可以使用此页上的 Feedback 按钮来报告。

有关故障排除的更多帮助，请访问 AWS 知识中心。

身份验证和授权错误

如果缺少使用来自 Kafka 集群的数据所需的任何权限，Lambda 会在 LastProcessingResult 下的事件源映射中显示以下错误消息。

错误消息

- 集群未能授权 Lambda
- SASL 身份验证失败
- 服务器未能通过 Lambda 的身份验证
- Lambda 未能对服务器进行身份验证
- 提供的证书或私有密钥无效

集群未能授权 Lambda

对于 SASL/SCRAM 或 mTLS，此错误表明提供的用户不具有以下所有必需的 Kafka 访问控制列表（ACL）权限：

- DescribeConfigs 集群
- 描述组
- 读取组
- 描述主题
- 读取主题

当您使用所需的 `kafka-cluster` 权限创建 Kafka ACL 时，请将主题和组指定为资源。主题名称必须与事件源映射中的主题一致。组名称必须与事件源映射的 UUID 一致。

向执行角色添加所需的权限后，更改可能需要几分钟才会生效。

SASL 身份验证失败

对于 SASL/SCRAM 或 SASL/PLAIN，此错误表明提供的登录凭证无效。

服务器未能通过 Lambda 的身份验证

此错误表明 Kafka 代理未能对 Lambda 进行身份验证。出现此错误的可能原因如下：

- 您没有为 mTLS 身份验证提供客户端证书。

- 您提供了客户端证书，但未将 Kafka 代理配置为使用 mTLS 身份验证。
- Kafka 代理不信任客户端证书。

Lambda 未能对服务器进行身份验证

此错误表明 Lambda 未能对 Kafka 代理进行身份验证。出现此错误的可能原因如下：

- Kafka 代理使用自签名证书或私有 CA，但未提供服务器根 CA 证书。
- 服务器根 CA 证书与签署代理证书的根 CA 不匹配。
- 主机名验证失败，因为代理的证书未将该代理的 DNS 名称或 IP 地址用作主题替代名称。

提供的证书或私有密钥无效

此错误表明 Kafka 使用者无法使用提供的证书或私有密钥。确保证书和密钥使用 PEM 格式，并且私有密钥加密使用 PBES1 算法。

事件源映射错误

将 Apache Kafka 集群作为 Lambda 函数的事件源添加时，如果您的函数遇到错误，Kafka 使用者将停止处理记录。主题分区的使用者是那些订阅、阅读和处理记录的使用者。您的其他 Kafka 使用者可以继续处理记录，只要他们没有遇到同样的错误即可。

要确定使用者停止的原因，请检查 `StateTransitionReason` 响应中的 `EventSourceMapping` 字段。下表列出了您可能收到的事件源错误：

`ESM_CONFIG_NOT_VALID`

事件源映射配置无效。

`EVENT_SOURCE_AUTHN_ERROR`

Lambda 无法对事件源进行身份验证。

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda 没有访问事件源所需的权限。

`FUNCTION_CONFIG_NOT_VALID`

函数配置无效。

> **Note**
>
> 如果您的 Lambda 事件记录超过允许的 6 MB 大小限制，那么它们可能处于未处理状态。

使用 Amazon API Gateway 端点调用 Lambda 函数

您可以使用 Amazon API Gateway 为 Lambda 函数创建带有 HTTP 端点的 Web API。API Gateway 提供工具，用于创建和记录向 Lambda 函数路由 HTTP 请求的 Web API。您可以使用身份验证和授权控制来保护对 API 的访问。您的 API 可以通过互联网传输流量，也可以仅允许在您的 VPC 内访问。

API 中的资源会定义一个或多个方法，如 GET 或 POST。方法具有将请求传送给 Lambda 函数或其他集成类型的集成。您可以单独定义每个资源和方法，也可以使用特定的资源和方法类型来匹配属于特定模式的所有请求。代理资源会捕获某个资源下的所有路径。ANY 方法会捕获所有 HTTP 方法。

Sections

选择 API 类型

API Gateway 支持三种可调用 Lambda 函数的 API 类型：

- HTTP API：一种轻型的低延迟 RESTful API。
- REST API：一种功能丰富的可定制 RESTful API。
- WebSocket API：一种 Web API，可与客户端保持持久连接并进行全双工通信。

HTTP API 和 REST API 都是用于处理 HTTP 请求并返回响应的 RESTful API。HTTP API 后推出，是使用 API Gateway 版本 2 API 构建的。以下是 HTTP API 的新功能：

HTTP API 功能

- 自动部署 – 当您修改路线或集成时，更改会自动部署到启用了自动部署的阶段。

- 默认阶段 – 您可以创建默认阶段 (`$default`)，以便在 API URL 的根路径处提供请求。对于具名阶段，必须在路径的开头包含阶段名称。
- CORS 配置 – 您可以配置 API，使其将 CORS 标头添加到传出响应中，而不是在函数代码中手动添加。

REST API 是 API Gateway 自发布起就支持的典型 RESTful API。REST API 现在具有更多的自定义、集成和管理功能。

REST API 功能

- 集成类型 – REST API 支持自定义 Lambda 集成。使用自定义集成，您可以直接将请求正文发送到函数，也可以对其应用转换模板，然后再发送到函数。
- 访问控制 – REST API 支持更多身份验证和授权选项。
- 监控和跟踪 – REST API 支持 AWS X-Ray 跟踪和其他日志记录选项。

有关详细比较，请参阅《API Gateway 开发人员指南》中的在 HTTP API 和 REST API 之间选择。

WebSocket API 也使用 API Gateway 版本 2 API 并支持类似的功能集。对于受益于客户端和 API 之间的持久连接的应用程序，请使用 WebSocket API。WebSocket API 提供全双工通信，这意味着客户端和 API 都可以持续发送消息，而无需等待响应。

HTTP API 支持简化的事件格式（2.0 版）。以下示例显示了来自 HTTP API 的事件。

Example API Gateway 代理事件（HTTP API）

```
{
    "version": "2.0",
    "routeKey": "ANY /nodejs-apig-function-1G3XMPLZXVXYI",
    "rawPath": "/default/nodejs-apig-function-1G3XMPLZXVXYI",
    "rawQueryString": "",
    "cookies": [
        "s_fid=7AABXMPL1AFD9BBF-0643XMPL09956DE2",
        "regStatus=pre-register"
    ],
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
        "accept-encoding": "gzip, deflate, br",
        ...
```

```
    },
    "requestContext": {
        "accountId": "123456789012",
        "apiId": "r3pmxmplak",
        "domainName": "r3pmxmplak.execute-api.us-east-2.amazonaws.com",
        "domainPrefix": "r3pmxmplak",
        "http": {
            "method": "GET",
            "path": "/default/nodejs-apig-function-1G3XMPLZXVXYI",
            "protocol": "HTTP/1.1",
            "sourceIp": "205.255.255.176",
            "userAgent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36"
        },
        "requestId": "JKJaXmPLvHcESHA=",
        "routeKey": "ANY /nodejs-apig-function-1G3XMPLZXVXYI",
        "stage": "default",
        "time": "10/Mar/2020:05:16:23 +0000",
        "timeEpoch": 1583817383220
    },
    "isBase64Encoded": true
}
```

有关更多信息，请参阅针对 API Gateway 中的 HTTP API 创建 AWS Lambda 代理集成。

向 Lambda 函数添加终端节点

向 Lambda 函数添加公有端点

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 在 Function overview（函数概览）下，选择 Add trigger（添加触发器）。
4. 选择 API Gateway (API 网关)。
5. 选择 Create an API（创建 API）或 Use an existing API（使用现有 API）。
   a. New API（新 API）：对于 API type（API 类型），请选择 HTTP API。有关更多信息，请参阅 选择 API 类型。
   b. 现有 API：从下拉列表中选择 API 或输入 API ID（例如，r3pmxmplak）。
6. 对于 Security (安全性)，请选择 Open (打开)。
7. 选择添加。

代理集成

API Gateway API 由阶段、资源、方法和集成组成。阶段和资源决定终端节点的路径：

API 路径格式

- `/prod/` – `prod` 阶段和根资源。
- `/prod/user` – `prod` 阶段和 `user` 资源。
- `/dev/{proxy+}` – `dev` 阶段中的路线。
- `/` – (HTTP API) 默认阶段和根资源。

Lambda 集成将路径和 HTTP 方法的组合映射到 Lambda 函数。您可以将 API Gateway 配置为按原样（自定义集成）传递 HTTP 请求体，或者将请求正文封装在包含所有请求信息（包括标头、资源、路径和方法）的文档中。

有关更多信息，请参阅 API Gateway 中的 Lambda 代理集成。

事件格式

Amazon API Gateway 会使用包含 HTTP 请求的 JSON 表示形式的事件同步调用函数。对于自定义集成，该事件为请求的正文。对于代理集成，该事件具有已确定的结构。以下示例显示了来自 API Gateway REST API 的代理事件。

Example API Gateway 代理事件（REST API）

```
{
      "resource": "/",
      "path": "/",
      "httpMethod": "GET",
      "requestContext": {
          "resourcePath": "/",
          "httpMethod": "GET",
          "path": "/Prod/",
          ...
      },
      "headers": {
          "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
          "accept-encoding": "gzip, deflate, br",
```

```
          "Host": "70ixmpl4fl.execute-api.us-east-2.amazonaws.com",
          "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36",
          "X-Amzn-Trace-Id": "Root=1-5e66d96f-7491f09xmpl79d18acf3d050",
          ...
      },
      "multiValueHeaders": {
          "accept": [
              "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9"
          ],
          "accept-encoding": [
              "gzip, deflate, br"
          ],
          ...
      },
      "queryStringParameters": null,
      "multiValueQueryStringParameters": null,
      "pathParameters": null,
      "stageVariables": null,
      "body": null,
      "isBase64Encoded": false
  }
```

响应格式

API Gateway 会等待函数响应并将结果转发给调用方。对于自定义集成，您可以定义集成响应和方法响应，以将函数的输出转换为 HTTP 响应。对于代理集成，函数必须以特定格式的响应形式来做出响应。

以下示例显示了来自 Node.js 函数的响应对象。该响应对象表示包含 JSON 文档的成功 HTTP 响应。

Example index.mjs：代理集成响应对象（Node.js）

```
var response = {
      "statusCode": 200,
      "headers": {
        "Content-Type": "application/json"
      },
      "isBase64Encoded": false,
      "multiValueHeaders": {
        "X-Custom-Header": ["My value", "My other value"],
      },
```

```
      "body": "{\n  \"TotalCodeSize\": 104330022,\n  \"FunctionCount\": 26\n}"
    }
```

Lambda 运行时会将响应对象序列化为 JSON 并将其发送给 API。此 API 会解析该响应并用它来创建 HTTP 响应，然后再将其发送到发出原始请求的客户端。

Example HTTP 响应

```
< HTTP/1.1 200 OK
  < Content-Type: application/json
  < Content-Length: 55
  < Connection: keep-alive
  < x-amzn-RequestId: 32998fea-xmpl-4268-8c72-16138d629356
  < X-Custom-Header: My value
  < X-Custom-Header: My other value
  < X-Amzn-Trace-Id: Root=1-5e6aa925-ccecxmplbae116148e52f036
  <
  {
    "TotalCodeSize": 104330022,
    "FunctionCount": 26
  }
```

权限

Amazon API Gateway 将从函数的基于资源的策略获取调用函数的权限。您可以授予对整个 API 的调用权限，也可以仅授予对某个阶段、资源或方法的有限访问权限。

当您使用 Lambda 控制台、API Gateway 控制台或 AWS SAM 模板向函数添加 API 时，会自动更新函数的基于资源的策略。以下是一个示例函数策略。

Example 函数策略

```
{
  "Version": "2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "nodejs-apig-functiongetEndpointPermissionProd-BWDBXMPLXE2F",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
```

```
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:111122223333:function:nodejs-apig-
function-1G3MXMPLXVXYI",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:execute-api:us-east-2:111122223333:ktyvxmpls1/*/
GET/"
        }
      }
    }
  ]
}
```

您可以通过以下 API 操作手动管理函数策略权限：

- AddPermission
- RemovePermission
- GetPolicy

使用 add-permission 命令，可授予对现有 API 的调用权限。例如：

```
aws lambda add-permission \
  --function-name my-function \
  --statement-id apigateway-get --action lambda:InvokeFunction \
  --principal apigateway.amazonaws.com \
  --source-arn "arn:aws:execute-api:us-east-2:123456789012:mnh1xmpli7/default/GET/"
```

您应看到以下输出：

```
{
    "Statement": "{\"Sid\":\"apigateway-test-2\",\"Effect\":\"Allow\",\"Principal
\":{\"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-east-2:123456789012:function:my-function
\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:execute-api:us-
east-2:123456789012:mnh1xmpli7/default/GET\"}}}"
}
```

> **Note**
>
> 如果您的函数和 API 位于不同的 AWS 区域，则源 ARN 中的区域标识符必须与函数的区域（而不是 API 的区域）匹配。当 API Gateway 调用函数时，它会使用基于 API ARN 的资源 ARN，但该资源 ARN 已被修改为与函数的区域相匹配。

此示例中的源 ARN 授予对 API 默认阶段根资源 GET 方法的集成的权限，其 ID为 `mnh1xmpli7`。您可以在源 ARN 中使用星号授予对多个阶段、方法或资源的权限。

资源模式

- `mnh1xmpli7/*/GET/*` – 对所有阶段中的所有资源调用 GET 方法。
- `mnh1xmpli7/prod/ANY/user` – 对 `prod` 阶段中的 `user` 资源调用 ANY 方法。
- `mnh1xmpli7/*/*/*` – 对所有阶段中的所有资源调用 ANY 方法。

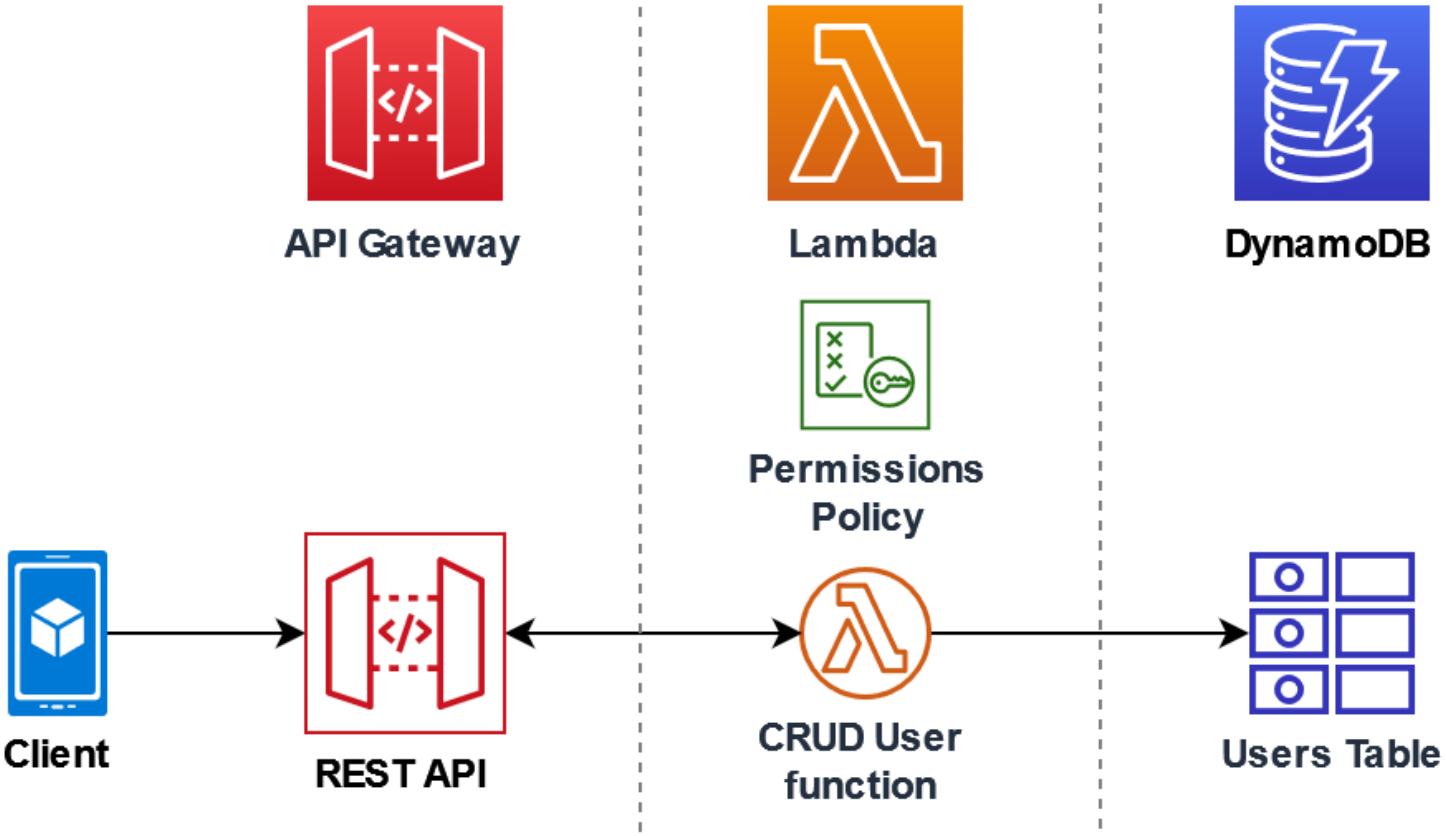
有关查看策略和删除语句的详细信息，请参阅在 Lambda 中使用基于资源的 IAM 策略。

示例应用程序

带有 Node.js 的 API Gateway 示例应用程序包含具有 AWS SAM 模板的函数，该模板可用于创建启用了 AWS X-Ray 跟踪的 REST API。它还包含用于部署和调用函数、测试 API 以及执行清理的脚本。

教程：利用 API Gateway 使用 Lambda

在本教程中，您将创建 REST API，您可以借助它使用 HTTP 请求调用 Lambda 函数。Lambda 函数将对 DynamoDB 表执行创建、读取、更新和删除（CRUD）操作。此处提供此函数用于演示，但您将学习配置可以调用任何 Lambda 函数的 API Gateway REST API。



使用 API Gateway 可为用户提供安全的 HTTP 端点来调用 Lambda 函数，并可以通过节流流量以及自动验证和授权 API 调用来帮助管理大量函数调用。API Gateway 还使用 AWS Identity and Access Management（IAM）和 Amazon Cognito 提供灵活的安全控制。对于需要预先授权才能调用应用程序的使用案例而言，这非常有用。

要完成本教程，您需要经历以下阶段：

1. 在 Python 或 Node.js 中创建和配置 Lambda 函数以对 DynamoDB 表执行操作。
2. 在 API Gateway 中创建 REST API 以连接 Lambda 函数。
3. 创建 DynamoDB 表，并在控制台中使用 Lambda 函数对其进行测试。
4. 部署 API 并在终端中使用 curl 测试完整设置。

完成这些阶段后，您将了解如何使用 API Gateway 创建 HTTP 端点，该端点可以安全地调用任何规模的 Lambda 函数。此外，您将了解如何部署 API、如何在控制台中对其进行测试以及如何使用终端发送 HTTP 请求。

Sections

- 先决条件

- 创建权限策略
- 创建执行角色
- 创建函数
- 使用 AWS CLI 调用函数
- 使用 API Gateway 创建 REST API
- 在 REST API 上创建资源
- 创建 HTTP POST 方法
- 创建 DynamoDB 表
- 测试 API Gateway、Lambda 和 DynamoDB 的集成
- 部署 API
- 使用 curl 通过 HTTP 请求调用函数
- 清除资源（可选）

先决条件

注册 AWS 账户

如果您还没有 AWS 账户，请完成以下步骤来创建一个。

注册 AWS 账户

1. 打开 https://portal.aws.amazon.com/billing/signup。
2. 按照屏幕上的说明进行操作。

   在注册时，将接到一通电话，要求使用电话键盘输入一个验证码。

   当您注册 AWS 账户时，系统将会创建一个 AWS 账户根用户。根用户有权访问该账户中的所有 AWS 服务 和资源。作为安全最佳实践，请为用户分配管理访问权限，并且只使用根用户来执行需要根用户访问权限的任务。

注册过程完成后，AWS 会向您发送一封确认电子邮件。在任何时候，您都可以通过转至 https://aws.amazon.com/ 并选择我的账户来查看当前的账户活动并管理您的账户。

创建具有管理访问权限的用户

注册 AWS 账户 后，请保护好您的 AWS 账户根用户，启用 AWS IAM Identity Center，并创建一个管理用户，以避免使用根用户执行日常任务。

保护您的 AWS 账户根用户

1. 选择根用户并输入您的 AWS 账户电子邮件地址，以账户拥有者身份登录 AWS Management Console。在下一页上，输入您的密码。

   要获取使用根用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的以根用户身份登录。
2. 为您的根用户启用多重身份验证 (MFA)。

   有关说明，请参阅《IAM 用户指南》中的为 AWS 账户 根用户启用虚拟 MFA 设备（控制台）。

创建具有管理访问权限的用户

1. 启用 IAM Identity Center。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的启用 AWS IAM Identity Center。
2. 在 IAM Identity Center 中，为用户授予管理访问权限。

   有关如何使用 IAM Identity Center 目录 作为身份源的教程，请参阅《AWS IAM Identity Center 用户指南》中的使用默认的 IAM Identity Center 目录 配置用户访问权限。

以具有管理访问权限的用户身份登录

- 要使用您的 IAM Identity Center 用户身份登录，请使用您在创建 IAM Identity Center 用户时发送到您的电子邮件地址的登录网址。

  要获取使用 IAM Identity Center 用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的登录 AWS 访问门户。

将访问权限分配给其他用户

1. 在 IAM Identity Center 中，创建一个权限集，该权限集遵循应用最低权限的最佳做法。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的创建权限集。
2. 将用户分配到一个组，然后为该组分配单点登录访问权限。

有关说明，请参阅《AWS IAM Identity Center 用户指南》中的添加组。

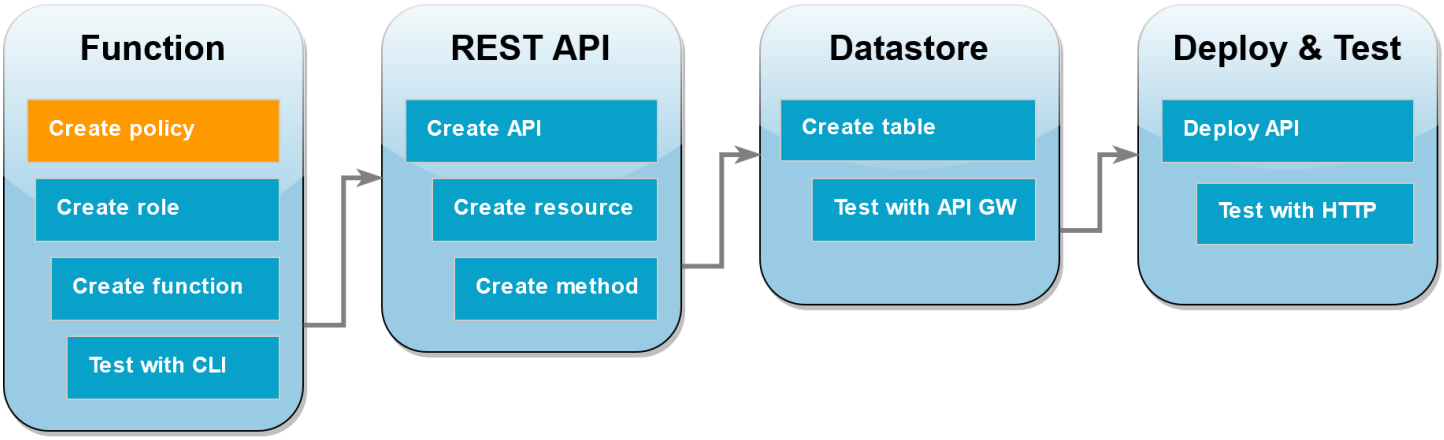
安装 AWS Command Line Interface

如果您尚未安装 AWS Command Line Interface，请按照安装或更新最新版本的 AWS CLI 中的步骤进行安装。

本教程需要命令行终端或 Shell 来运行命令。在 Linux 和 macOS 中，可使用您首选的 Shell 和程序包管理器。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常与 Lambda 一起使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。

创建权限策略



为 Lambda 函数创建执行角色之前，首先需要创建权限策略以授予函数访问所需 AWS 资源的权限。在本教程中，该策略允许 Lambda 对 DynamoDB 表执行 CRUD 操作并写入 Amazon CloudWatch Logs。

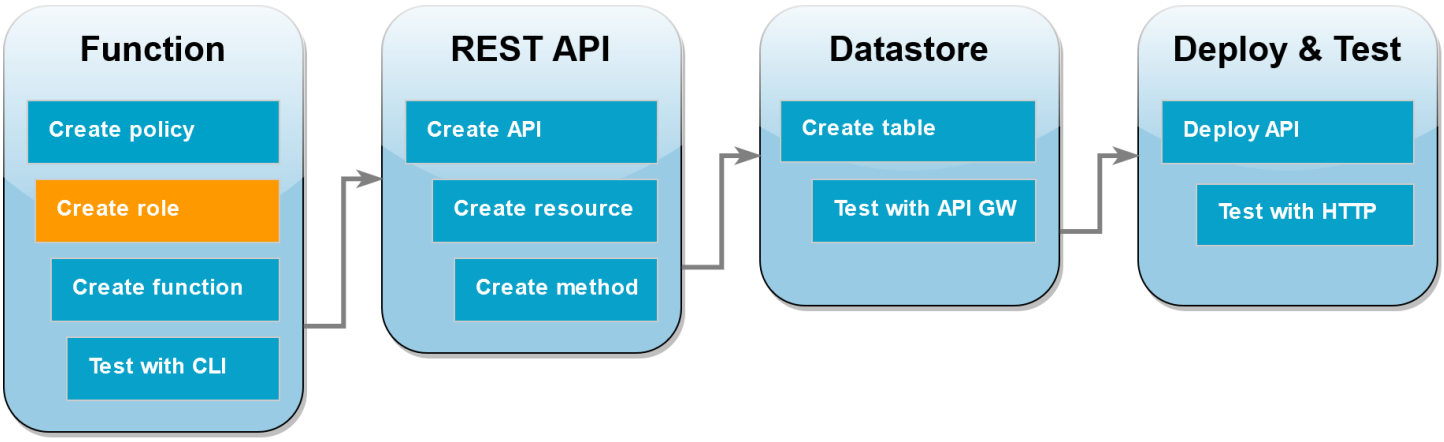
创建策略

1. 打开 IAM 控制台的 Policies（策略）页面。
2. 选择创建策略。
3. 选择 JSON 选项卡，然后将以下自定义策略粘贴到 JSON 编辑器中。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1428341300017",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "",
      "Resource": "*",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Effect": "Allow"
    }
  ]
}
```

4. 选择下一步：标签。
5. 选择下一步：审核。
6. 在 Review policy (查看策略) 下，为策略 Name (名称) 输入 **lambda-apigateway-policy**。
7. 选择创建策略。

创建执行角色



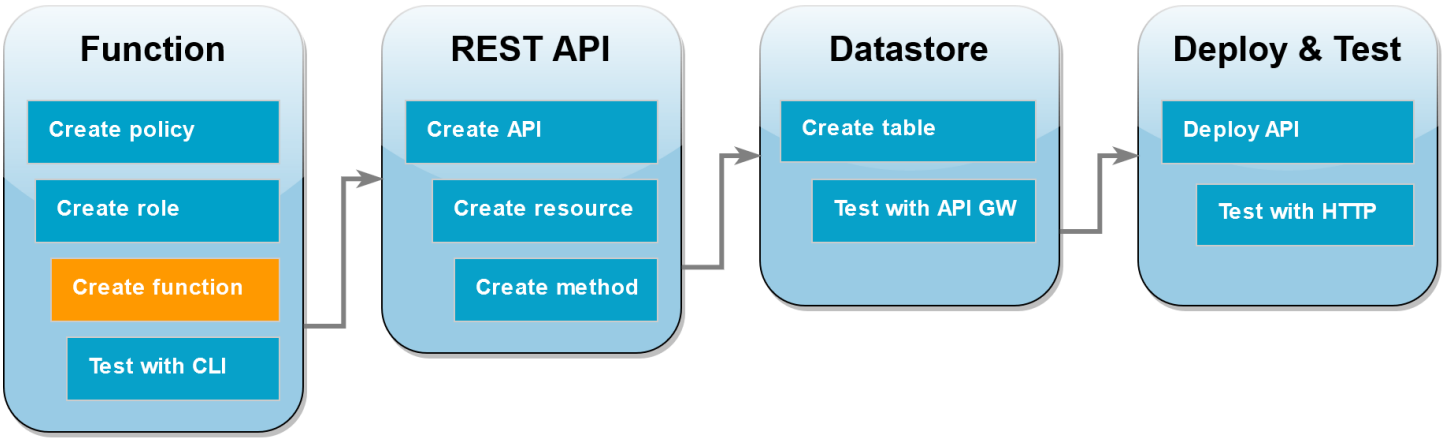
执行角色是一个 AWS Identity and Access Management（IAM）角色，用于向 Lambda 函数授予访问 AWS 服务和资源的权限。要使函数对 DynamoDB 表执行操作，您需要附加上一步中创建的权限策略。

创建执行角色并附加自定义权限策略

1. 打开 IAM 控制台的角色页面。
2. 选择 Create role（创建角色）。
3. 对于可信实体，选择 AWS 服务，对于使用案例，选择 Lambda。
4. 选择下一步。
5. 在策略搜索框中，输入 **`lambda-apigateway-policy`**。
6. 在搜索结果中，选择您创建的策略（`lambda-apigateway-policy`），然后选择 Next（下一步）。
7. 在 Role details（角色详细信息）下，为 Role name（角色名称）输入 **`lambda-apigateway-role`**，然后选择 Create role（创建角色）。

在本教程的后面部分，您需要提供刚刚创建的角色的 Amazon 资源名称（ARN）。在 IAM 控制台的 Roles（角色）页面上，选择角色名称（`lambda-apigateway-role`），然后复制 Summary（摘要）页面上显示的 Role ARN（角色 ARN）。

创建函数



以下代码示例接收来自 API Gateway 的事件输入，指定要在将创建的 DynamoDB 表上执行的操作和一些负载数据。如果函数接收的参数有效，则它会在表上执行所请求的操作。

Node.js

Example index.mjs

请注意 `region` 设置。该设置必须与部署 AWS 区域 函数和创建 DynamoDB 表的位置相匹配。

```
import { DynamoDBDocumentClient, PutCommand, GetCommand,
         UpdateCommand, DeleteCommand} from "@aws-sdk/lib-dynamodb";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

const ddbClient = new DynamoDBClient({ region: "us-east-2" });
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);

// Define the name of the DDB table to perform the CRUD operations on
const tablename = "lambda-apigateway";

/**
 * Provide an event that contains the following keys:
 *
 *   - operation: one of 'create,' 'read,' 'update,' 'delete,' or 'echo'
 *   - payload: a JSON object containing the parameters for the table item
 *     to perform the operation on
 */
export const handler = async (event, context) => {

     const operation = event.operation;
```

```
     if (operation == 'echo'){
          return(event.payload);
     }

    else {
        event.payload.TableName = tablename;
        let response;

        switch (operation) {
          case 'create':
               response = await ddbDocClient.send(new PutCommand(event.payload));
               break;
          case 'read':
               response = await ddbDocClient.send(new GetCommand(event.payload));
               break;
          case 'update':
               response = ddbDocClient.send(new UpdateCommand(event.payload));
               break;
          case 'delete':
               response = ddbDocClient.send(new DeleteCommand(event.payload));
               break;
          default:
            response = 'Unknown operation: ${operation}';
          }
        console.log(response);
        return response;
    }
};
```

> **Note**
>
> 在此示例中，DynamoDB 表的名称定义为函数代码中的变量。在实际应用程序中，最佳做法是将此参数作为环境变量传递，并避免对表名称进行硬编码。有关更多信息，请参阅使用 AWS Lambda 环境变量。

创建函数

1. 将代码示例另存为名为 `index.mjs` 的文件，如有必要，编辑代码中指定的 AWS 区域。代码中指定的区域必须与您稍后在本教程中创建 DynamoDB 表的区域相同。
2. 使用以下 `zip` 命令创建部署包。

```
zip function.zip index.mjs
```

3. 使用 `create-function` AWS CLI 命令创建 Lambda 函数。对于 `role` 参数，输入您先前复制的执行角色的 Amazon 资源名称（ARN）。

```
aws lambda create-function \
--function-name LambdaFunctionOverHttps \
--zip-file fileb://function.zip \
--handler index.handler \
--runtime nodejs20.x \
--role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
```

Python 3

Example LambdaFunctionOverHttps.py

```
import boto3

# Define the DynamoDB table that Lambda will connect to
table_name = "lambda-apigateway"

# Create the DynamoDB resource
dynamo = boto3.resource('dynamodb').Table(table_name)

# Define some functions to perform the CRUD operations
def create(payload):
    return dynamo.put_item(Item=payload['Item'])

def read(payload):
    return dynamo.get_item(Key=payload['Key'])

def update(payload):
    return dynamo.update_item(**{k: payload[k] for k in ['Key', 'UpdateExpression',
    'ExpressionAttributeNames', 'ExpressionAttributeValues'] if k in payload})

def delete(payload):
    return dynamo.delete_item(Key=payload['Key'])

def echo(payload):
    return payload
```

```
operations = {
    'create': create,
    'read': read,
    'update': update,
    'delete': delete,
    'echo': echo,
}

def lambda_handler(event, context):
    '''Provide an event that contains the following keys:
      - operation: one of the operations in the operations dict below
      - payload: a JSON object containing parameters to pass to the
        operation being performed
    '''

    operation = event['operation']
    payload = event['payload']

    if operation in operations:
        return operations[operation](payload)

    else:
        raise ValueError(f'Unrecognized operation "{operation}"')
```

> **Note**
>
> 在此示例中，DynamoDB 表的名称定义为函数代码中的变量。在实际应用程序中，最佳做法是将此参数作为环境变量传递，并避免对表名称进行硬编码。有关更多信息，请参阅使用 AWS Lambda 环境变量。

创建函数

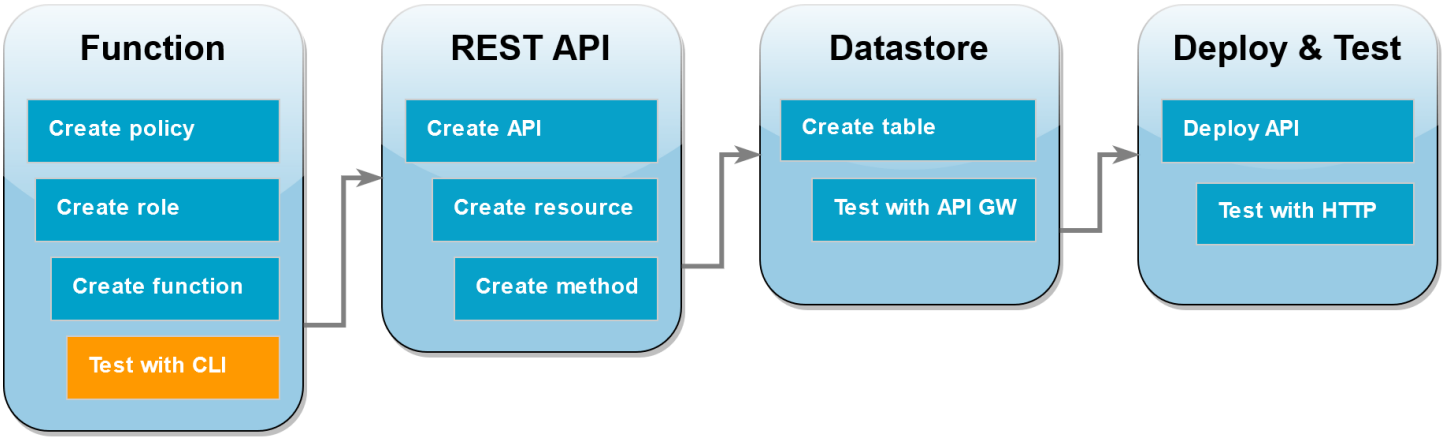
1. 将代码示例保存为名为 `LambdaFunctionOverHttps.py` 的文件。
2. 使用以下 `zip` 命令创建部署包。

   ```
   zip function.zip LambdaFunctionOverHttps.py
   ```

3. 使用 `create-function` AWS CLI 命令创建 Lambda 函数。对于 `role` 参数，输入您先前复制的执行角色的 Amazon 资源名称（ARN）。

```
aws lambda create-function \
--function-name LambdaFunctionOverHttps \
--zip-file fileb://function.zip \
--handler LambdaFunctionOverHttps.lambda_handler \
--runtime python3.12 \
--role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
```

使用 AWS CLI 调用函数



将函数与 API Gateway 集成之前，请确认您已成功部署该函数。创建一个测试事件，其中包含 API Gateway API 将发送给 Lambda 的参数，并使用 AWS CLI `invoke` 命令运行函数。

使用 AWS CLI 调用 Lambda 函数

1. 将下列 JSON 保存为名为 `input.txt` 的文件。

```
{
    "operation": "echo",
    "payload": {
        "somekey1": "somevalue1",
        "somekey2": "somevalue2"
    }
}
```

2. 运行以下 `invoke` AWS CLI 命令。

```
aws lambda invoke \
--function-name LambdaFunctionOverHttps \
--payload file://input.txt outputfile.txt \
```

```
--cli-binary-format raw-in-base64-out
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

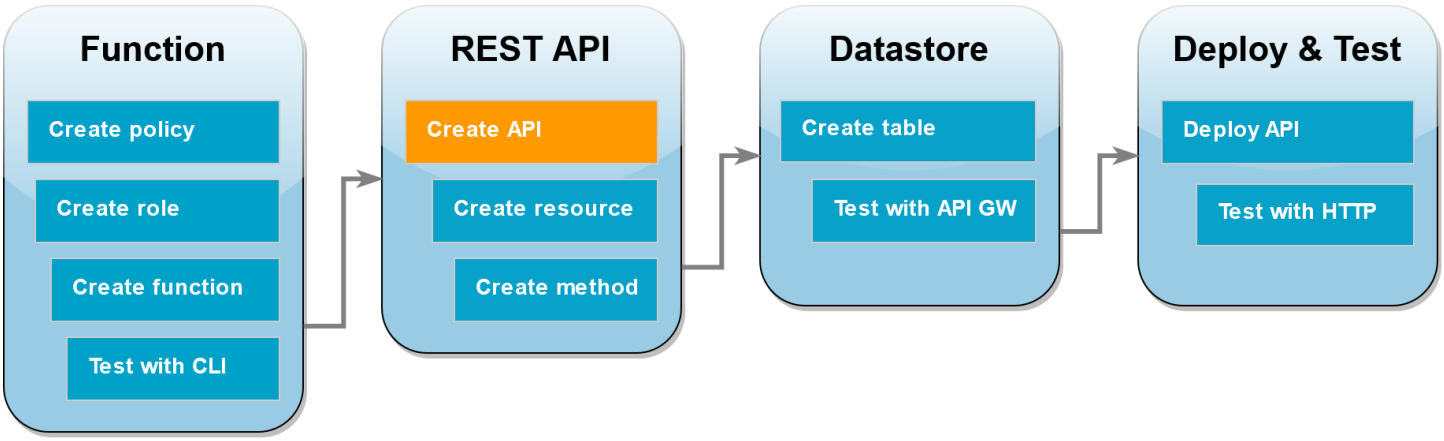
您应看到以下响应：

```
{
"StatusCode": 200,
"ExecutedVersion": "LATEST"
}
```

3. 请确认函数已执行您在 JSON 测试事件中指定的 `echo` 操作。检查 `outputfile.txt` 文件并验证其是否包含以下内容：

```
{"somekey1": "somevalue1", "somekey2": "somevalue2"}
```

使用 API Gateway 创建 REST API



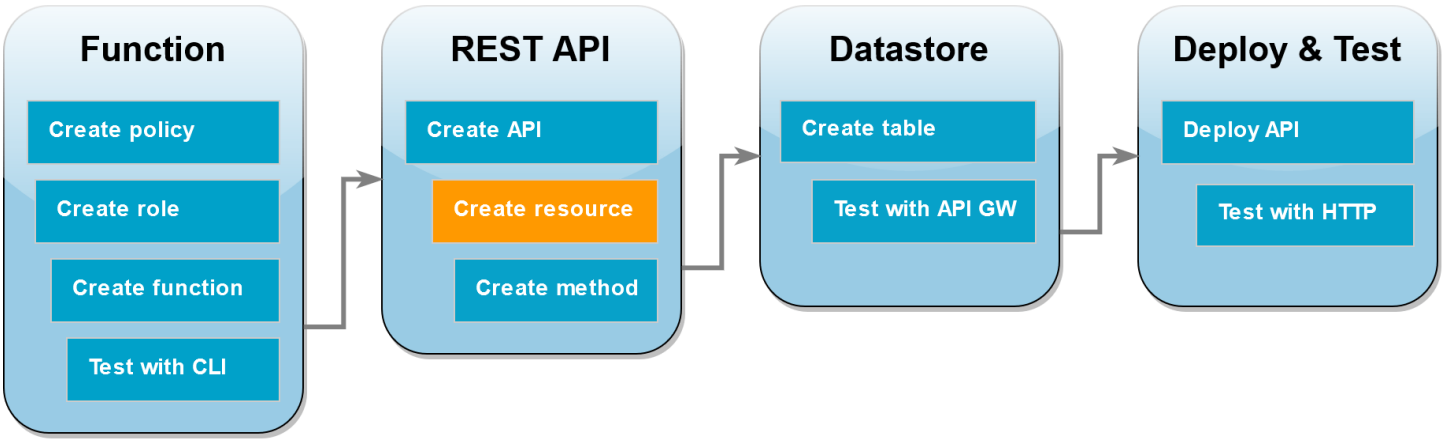
在此步骤中，您将创建用于调用 Lambda 函数的 API Gateway REST API。

创建 API

1. 打开 API Gateway 控制台。
2. 选择 Create API (创建 API)。
3. 在 REST API 框中，选择 Build (构建)。

4. 在 API 详细信息下，将新建 API 保留为选中状态，然后在 API 名称中输入 **`DynamoDBOperations`**。
5. 选择创建 API。

在 REST API 上创建资源

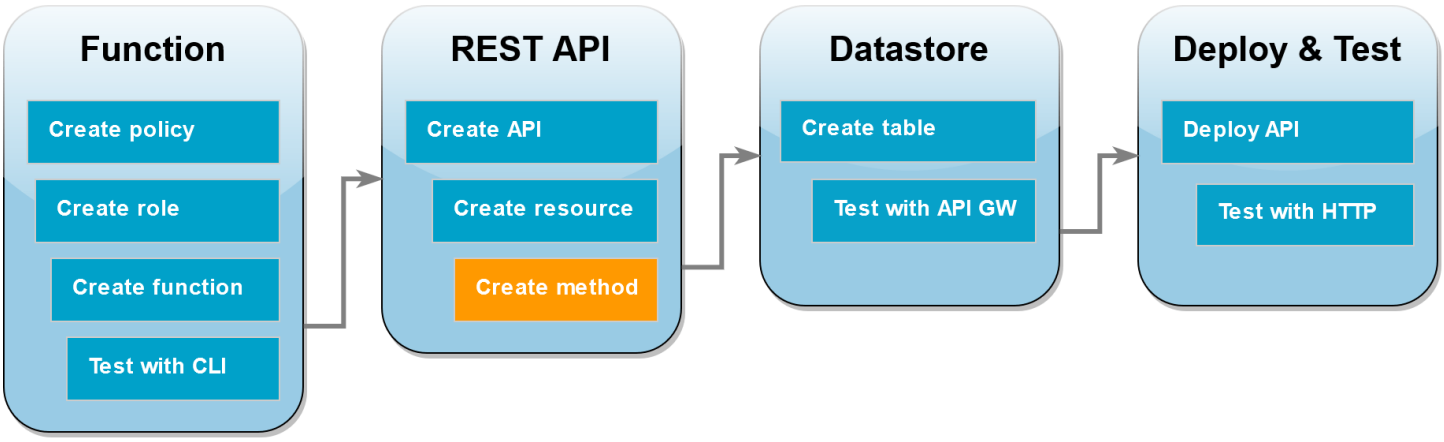


要将 HTTP 方法添加到 API，您首先需要创建资源供该方法运行。您可以在此处创建资源来管理 DynamoDB 表。

创建资源

1. 在 API Gateway 控制台中，在您的 API 的资源页面上，选择创建资源。
2. 在资源详细信息中，在资源名称中输入 **`DynamoDBManager`**。
3. 选择创建资源。

创建 HTTP POST 方法



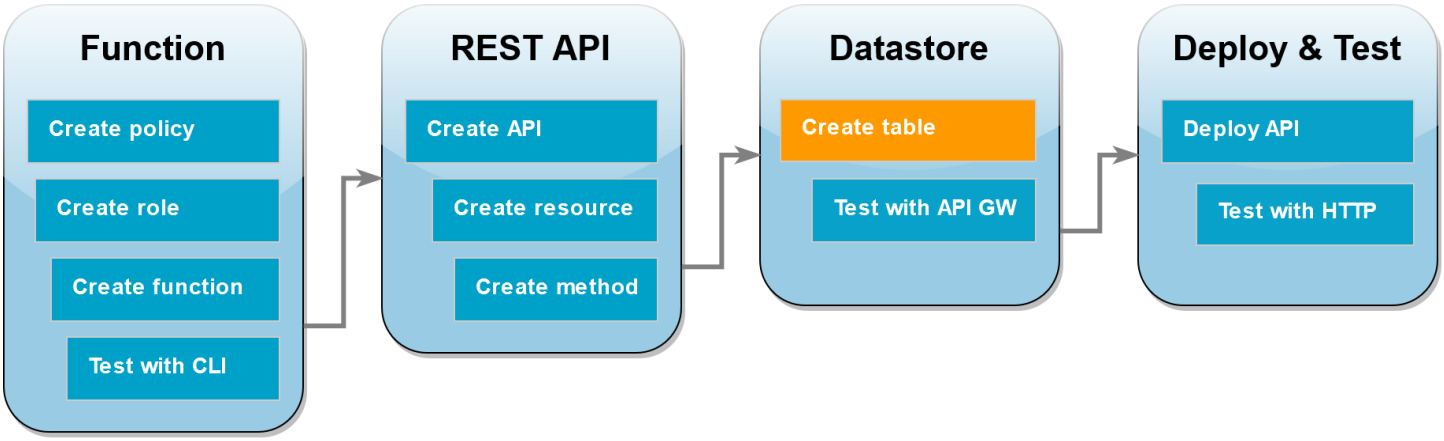
在此步骤中，您将为 DynamoDBManager 资源创建方法（POST）。您可以将此 POST 方法链接到 Lambda 函数，以便当该方法接收 HTTP 请求时，API Gateway 可以调用 Lambda 函数。

> **Note**
>
> 就本教程而言，一个 HTTP 方法（POST）可用于调用一个 Lambda 函数，该函数将对 DynamoDB 表执行全部操作。在实际应用程序中，最佳做法是针对每个操作使用不同的 Lambda 函数和 HTTP 方法。有关更多信息，请参阅 Serverless Land 中的 The Lambda monolith。

创建 POST 方法

1. 在您的 API 的资源页面上，确保突出显示 /DynamoDBManager 资源。然后，在方法窗格中，选择创建方法。
2. 对于方法类型，选择 POST。
3. 对于集成类型，将 Lambda 函数保留为选中状态。
4. 对于 Lambda 函数，选择函数（LambdaFunctionOverHttps）的 Amazon 资源名称（ARN）。
5. 选择创建方法。

创建 DynamoDB 表



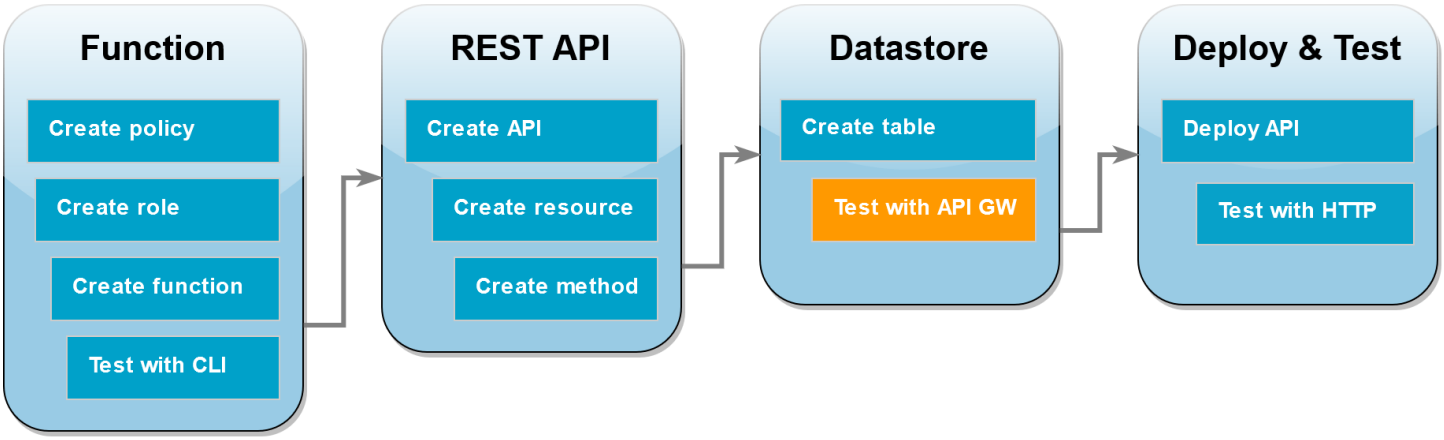
创建一个空的 DynamoDB 表，Lambda 函数将对其执行 CRUD 操作。

创建 DynamoDB 表

1. 打开 DynamoDB 控制台中 Tables page（表页面）。

2. 选择创建表。
3. 在 Table details (表详细信息) 下，执行以下操作：
   1. 对于表名称，输入 **`lambda-apigateway`**。
   2. 对于 Partition key (分区键)，输入 **`id`**，并将数据类型设置为 String (字符串)。
4. 在 Table settings（表设置）下，使用 Default settings（默认设置）。
5. 选择创建表。

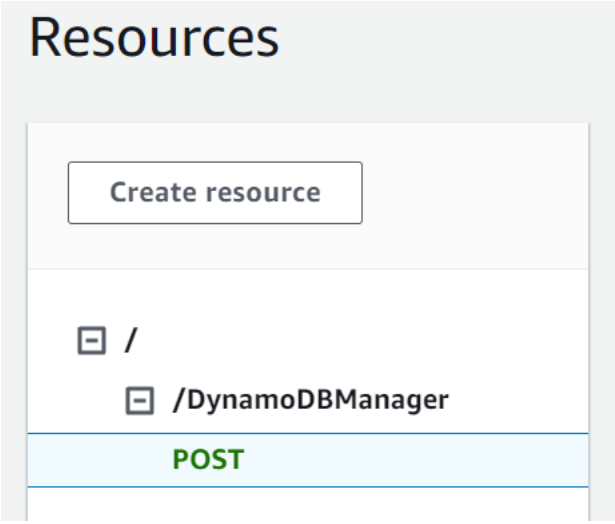
测试 API Gateway、Lambda 和 DynamoDB 的集成



现在，您已准备好测试 API Gateway API 方法与 Lambda 函数和 DynamoDB 表的集成。借助 API Gateway 控制台，您可以使用控制台的测试功能将请求直接发送到 POST 方法。在此步骤中，您首先使用 `create` 操作将新项目添加到 DynamoDB 表，然后使用 `update` 操作修改该项目。

测试 1：在 DynamoDB 表中创建新项目

1. 在 API Gateway console（API Gateway 控制台）中，选择 API（`DynamoDBOperations`）。
2. 在 `DynamoDBManager` 资源下，选择 POST 方法。



3. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
4. 在测试方法下，将查询字符串和标头留空。对于请求正文，粘贴以下 JSON：

```
{
  "operation": "create",
  "payload": {
    "Item": {
      "id": "1234ABCD",
      "number": 5
    }
  }
}
```

5. 选择测试。

   测试完成后，所显示的结果应显示状态 200。此状态代码表示 create 操作成功。

   要进行确认，请检查 DynamoDB 表现在是否包含新项目。

6. 打开 DynamoDB 控制台中的 Tables page（表页面），然后选择 lambda-apigateway 表。
7. 选择 Explore table items（浏览表项目）。在 Items returned（返回的项目）窗格中，您会看到一个带 id 1234ABCD 和 number（编号）5 的项目。

测试 2：更新 DynamoDB 表中的项目

1. 在 API Gateway 控制台中，返回到 POST 方法的测试选项卡。
2. 在测试方法下，将查询字符串和标头留空。对于请求正文，粘贴以下 JSON：

```
{
    "operation": "update",
    "payload": {
        "Key": {
            "id": "1234ABCD"
        },
        "UpdateExpression": "SET #num = :newNum",
        "ExpressionAttributeNames": {
            "#num": "number"
        },
        "ExpressionAttributeValues": {
            ":newNum": 10
        }
    }
}
```

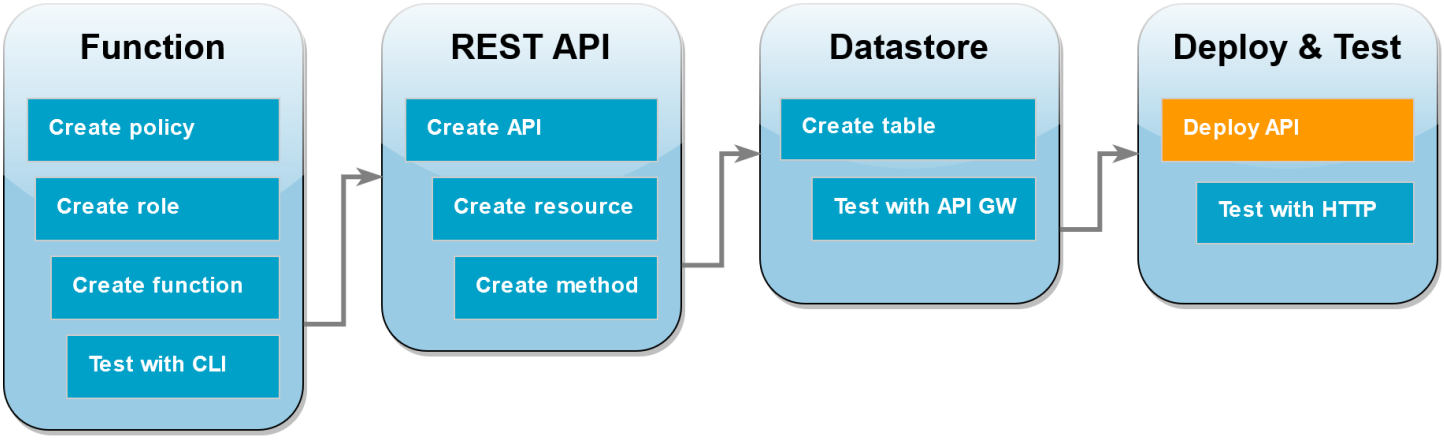
3. 选择测试。

   测试完成后，所显示的结果应显示状态 200。此状态代码表示 update 操作成功。

   要进行确认，请检查 DynamoDB 表中的项目是否进行过修改。

4. 打开 DynamoDB 控制台中的 Tables page（表页面），然后选择 lambda-apigateway 表。
5. 选择 Explore table items（浏览表项目）。在 Items returned（返回的项目）窗格中，您会看到一个带 id 1234ABCD 和 number（编号）10 的项目。

部署 API

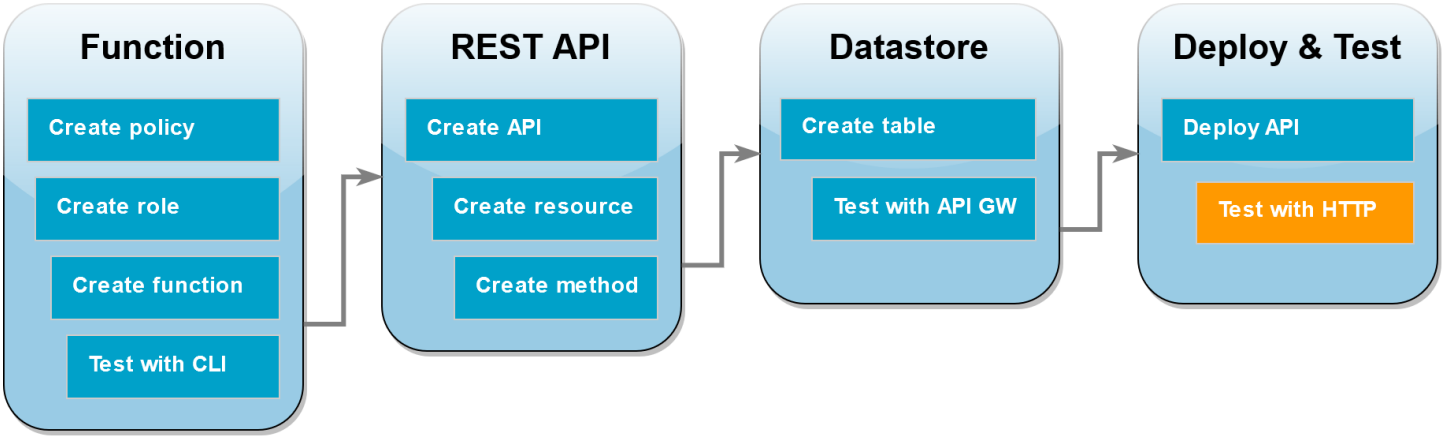


要让客户端调用 API，您必须创建部署和关联的阶段。阶段表示 API 的快照，包括其方法和集成。

部署 API

1. 打开 API Gateway console（API Gateway 控制台）的 API 页面，然后选择 `DynamoDBOperations` API。
2. 在您的 API 的资源页面上，选择部署 API。
3. 对于阶段，请选择*新建阶段*，然后在阶段名称中输入 **`test`**。
4. 选择部署。
5. 在阶段详细信息窗格中，复制调用 URL。您将在下一步中使用它，通过 HTTP 请求调用函数。

使用 curl 通过 HTTP 请求调用函数



现在，您可以通过向 API 发出 HTTP 请求来调用 Lambda 函数。在此步骤中，您将在 DynamoDB 表中创建一个新项目，然后对该项目执行读取、更新和删除操作。

使用 curl 在 DynamoDB 表中创建项目

1. 使用您在上一步中复制的调用 URL 运行以下 `curl` 命令。当您使用带 `-d`（数据）选项的 curl 时，它会自动使用 HTTP POST 方法。

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager \
   -d '{"operation": "create", "payload": {"Item": {"id": "5678EFGH", "number": 15}}}'
   ```

   如果操作成功，您应该会看到返回的响应，其中包含 HTTP 状态代码 200。
2. 您还可以使用 DynamoDB 控制台，通过执行以下操作来验证新项目是否在表中：

   1. 打开 DynamoDB 控制台中的 Tables page（表页面），然后选择 `lambda-apigateway` 表。

2. 选择 Explore table items（浏览表项目）。在 Items returned（返回的项目）窗格中，您会看到一个带 id 5678EFGH 和 number（编号）15 的项目。

使用 curl 读取 DynamoDB 表中的项目

- 运行以下 curl 命令读取您创建的项目的值。使用您自己的调用 URL。

```
curl https://avos4dr2rk.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager -d
 \
'{"operation": "read", "payload": {"Key": {"id": "5678EFGH"}}}'
```

根据您选择的是 Node.js 还是 Python 函数代码，您应该会看到如下输出：

Node.js

```
{"$metadata":
{"httpStatusCode":200,"requestId":"7BP3G5Q0C0O1E50FBQI9NS099JVV4KQNSO5AEMVJF66Q9ASUAAJG"
"attempts":1,"totalRetryDelay":0},"Item":{"id":"5678EFGH","number":15}}
```

Python

```
{"Item":{"id":"5678EFGH","number":15},"ResponseMetadata":
{"RequestId":"QNDJICE52E86B82VETR6RKBE5BVV4KQNSO5AEMVJF66Q9ASUAAJG",
"HTTPStatusCode":200,"HTTPHeaders":{"server":"Server","date":"Wed, 31 Jul 2024
 00:37:01 GMT","content-type":"application/x-amz-json-1.0",
"content-length":"52","connection":"keep-alive","x-amzn-
requestid":"QNDJICE52E86B82VETR6RKBE5BVV4KQNSO5AEMVJF66Q9ASUAAJG","x-amz-
crc32":"2589610852"},
"RetryAttempts":0}}
```

使用 curl 更新 DynamoDB 表中的项目

1. 运行以下 curl 命令，通过更改 number 值来更新您创建的项目。使用您自己的调用 URL。

```
curl https://avos4dr2rk.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager \
-d '{"operation": "update", "payload": {"Key": {"id": "5678EFGH"},
 "UpdateExpression": "SET #num = :new_value", "ExpressionAttributeNames": {"#num":
 "number"}, "ExpressionAttributeValues": {":new_value": 42}}}'
```

2. 要确认项目的 number 值已更新，再运行一条 read 命令：

```
curl https://avos4dr2rk.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager \
-d '{"operation": "read", "payload": {"Key": {"id": "5678EFGH"}}}'
```

使用 curl 删除 DynamoDB 表中的项目

1. 运行以下 curl 命令以删除刚刚创建的项目。使用您自己的调用 URL。

```
curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager \
-d '{"operation": "delete", "payload": {"Key": {"id": "5678EFGH"}}}'
```

2. 确认删除操作是否成功。在 DynamoDB 控制台 Explore items（浏览项目）页面的 Items returned（返回的项目）窗格中，验证表中是否不再包含带 id 5678EFGH 的项目。

清除资源（可选）

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择删除。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。
4. 在文本输入字段中输入角色名称，然后选择 Delete（删除）。

删除 API

1. 打开 API Gateway 控制台的 API 页面。
2. 选择您创建的 API。
3. 依次选择 Actions 和 Delete。
4. 选择 Delete（删除）。

删除 DynamoDB 表

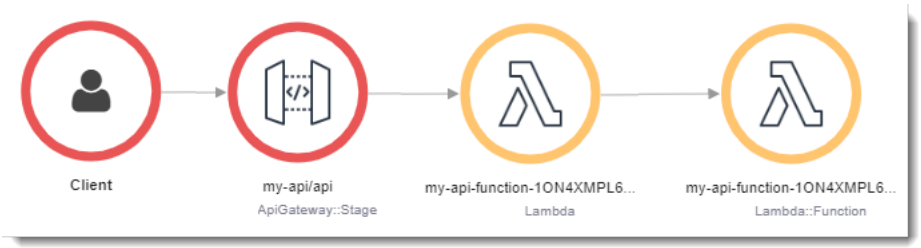
1. 打开 DynamoDB 控制台中 Tables page（表页面）。
2. 选择您创建的表。
3. 选择删除。
4. 在文本框中输入 **`delete`**。
5. 选择 删除表。

利用 API Gateway API 处理 Lambda 错误

API Gateway 将所有调用和函数错误视为内部错误。如果 Lambda API 拒绝调用请求，API Gateway 会返回 500 错误代码。如果函数运行但返回错误，或返回格式错误的响应，API Gateway 会返回 502。在这两种情况下，来自 API Gateway 的响应的正文都是 `{"message": "Internal server error"}`。

> **Note**
>
> API Gateway 不会重试 Lambda 调用。如果 Lambda 返回错误，API Gateway 会向客户端返回错误响应。

以下示例显示导致 API Gateway 中出现函数错误和 502 的请求的 X-Ray 跟踪映射。客户端收到通用错误消息。

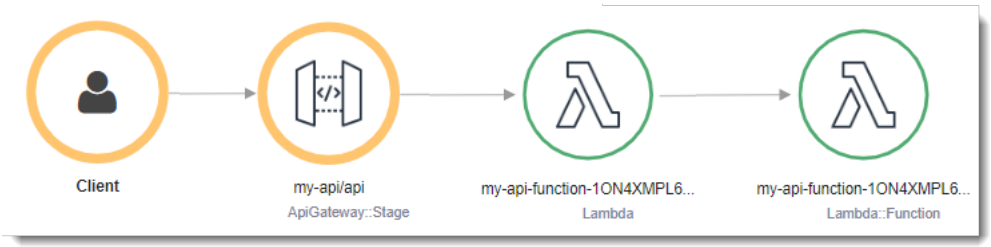


要自定义错误响应，您必须捕获代码中的错误并以所需格式设置响应的格式。

Example index.mjs：格式设置错误

```
var formatError = function(error){
  var response = {
    "statusCode": error.statusCode,
    "headers": {
      "Content-Type": "text/plain",
      "x-amzn-ErrorType": error.code
    },
    "isBase64Encoded": false,
    "body": error.code + ": " + error.message
  }
  return response
}
```

API Gateway 将此响应转换为带有自定义状态代码和正文的 HTTP 错误。在跟踪映射中，函数节点为绿色，因为它处理了错误。



配合使用 AWS Lambda和 AWS 基础设施编辑器

AWS 基础设施编辑器 是一款在 AWS 上设计现代应用程序的可视化生成器。您可以通过在可视画布中拖动、分组和连接 AWS 服务 来设计应用程序架构。基础设施编辑器根据您的设计创建基础设施即代码（IaC）模板，创建的模板可以使用 AWS SAM 或 AWS CloudFormation 进行部署。

将 Lambda 函数出到基础设施编辑器

开始使用基础设施编辑器时，您可以使用 Lambda 控制台根据现有 Lambda 函数的配置创建一个新项目。要将函数的配置和代码导出到基础设施编辑器来创建新项目，请执行以下操作：

1. 打开 Lamba 控制台的函数页面。
2. 选择您希望作为基础设施编辑器项目基础的函数。
3. 在函数概述窗格中，选择导出到基础设施编辑器。

为了将函数的配置和代码导出到基础设施编辑器，Lambda 在您的账户中创建了一个 Amazon S3 存储桶来临时存储此数据。

4. 在对话框中，选择确认并创建项目以接受此存储桶的默认名称，并将函数的配置和代码导出到基础设施编辑器。
5. （可选）要为 Lambda 创建的 Amazon S3 存储桶选择其他名称，请输入新名称并选择确认并创建项目。Amazon S3 存储桶的名称必须全局唯一，并遵守存储桶命名规则。
6. 要在基础设施编辑器中保存项目和函数文件，请激活本地同步模式。

> **Note**
>
> 如果您之前使用过导出到应用程序编辑器功能并使用默认名称创建了 Amazon S3 存储桶，Lambda 可以重复使用该存储桶（如果它仍然存在）。接受对话框中的默认存储桶名称以重复使用现有存储桶。

Amazon S3 传输存储桶配置

Lambda 为传输您的函数配置而创建的 Amazon S3 存储桶会使用 AES 256 加密标准自动加密对象。Lambda 还将存储桶配置为使用存储桶拥有者条件，以确保只有您的 AWS 账户 能向存储桶添加对象。

Lambda 将存储桶配置为在上传对象 10 天后自动将其删除。但是，Lambda 不会自动删除存储桶本身。要从您的 AWS 账户 删除存储桶，请按照删除存储桶中的说明进行操作。默认存储桶名称使用前缀 `lambdasam`、10 位字母数字字符串以及您创建函数的所在 AWS 区域：

```
lambdasam-06f22da95b-us-east-1
```

为避免给您的 AWS 账户 增加额外费用，建议在完成将函数导出到基础设施编辑器后立即删除 Amazon S3 存储桶。

标准 Amazon S3 定价适用。

所需的权限

要使用 Lambda 与基础设施编辑器集成功能，您需要一定的权限才能下载 AWS SAM 模板并将函数配置写入 Amazon S3。

要下载 AWS SAM 模板，您必须拥有使用以下 API 操作的权限：

- GetPolicy
- iam:GetPolicyVersion
- iam:GetRole
- iam:GetRolePolicy
- iam:ListAttachedRolePolicies
- iam:ListRolePolicies
- iam:ListRoles

通过将 `AWSLambda_ReadOnlyAccess` AWS 托管策略添加到 IAM 用户角色，可以授予使用所有这些操作的权限。

要让 Lambda 将您的函数配置写入 Amazon S3，您必须拥有使用以下 API 操作的权限：

- S3:PutObject
- S3:CreateBucket
- S3:PutBucketEncryption
- S3:PutBucketLifecycleConfiguration

如果无法将函数的配置导出到基础设施编辑器，请检查账户是否具有这些操作所需的权限。如果您拥有所需的权限但仍然无法导出函数配置，请检查基于资源的策略，因为这些策略可能会限制对 Amazon S3 的访问权限。

其他资源

有关如何在基础设施编辑器中基于现有 Lambda 函数设计无服务器应用程序的更详细教程，请参阅基础设施即代码（IaC）。

要使用基础设施编辑器以及 AWS SAM 通过 Lambda 设计和部署完整的无服务器应用程序，您也可以按照 AWS Serverless Patterns 讲习会中的 AWS 基础设施编辑器 教程进行操作。

配合使用 AWS Lambda 和 AWS CloudFormation

在 AWS CloudFormation 模板中，您可以指定 Lambda 函数作为自定义资源的对象。使用自定义资源来处理参数、检索配置值或者在堆栈生命周期事件期间调用其他 AWS 服务。

以下示例调用在模板中的其他位置定义的函数。

Example – 自定义资源定义

```
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

服务令牌是在您创建、更新或删除堆栈时，AWS CloudFormation 所调用函数的 Amazon 资源名称 (ARN)。您还可以按原样包含 FunctionName 传递到您函数的其他属性，例如 AWS CloudFormation。

AWS CloudFormation 通过包含回调 URL 的事件异步调用您的 Lambda 函数。

Example – AWS CloudFormation 消息事件

```
{
    "RequestType": "Create",
    "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
    "ResponseURL": "https://cloudformation-custom-resource-response-useast1.s3-us-
east-1.amazonaws.com/arn%3Aaws%3Acloudformation%3Aus-east-1%3A123456789012%3Astack/
lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456%7Cprimerinvoke
%7C5d478078-13e9-baf0-464a-7ef285ecc786?
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1555451971&Signature=28UijZePE5I4dvukKQqM
%2F9Rf1o4%3D",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke",
    "ResourceType": "AWS::CloudFormation::CustomResource",
    "ResourceProperties": {
```

```
        "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
        "FunctionName": "lambda-error-processor-randomerror-ZWUC391MQAJK"
    }
}
```

函数负责将指示成功还是失败的响应返回到回调 URL。有关完整响应语法，请参阅自定义资源响应对象。

Example – AWS CloudFormation 自定义资源响应

```
{
    "Status": "SUCCESS",
    "PhysicalResourceId": "2019/04/18/[$LATEST]b3d1bfc65f19ec610654e4d9b9de47a0",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke"
}
```

AWS CloudFormation 提供称为 `cfn-response` 的库来处理响应的发送。如果您在模板中定义函数，则可以按名称请求库。随后，AWS CloudFormation 将库添加到为函数创建的部署程序包。

如果自定义资源使用的函数附加了弹性网络接口，请将以下资源添加到 VPC 策略，其中 **`region`** 是函数所在的区域（不带破折号）。例如，`us-east-1` 为 `useast1`。这将允许自定义资源响应将信号发送回 AWS CloudFormation 堆栈的回调 URL。

```
arn:aws:s3:::cloudformation-custom-resource-response-region",
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

以下示例函数调用第二个函数。如果调用成功，则函数发送成功响应到 AWS CloudFormation，并且堆栈更新继续。该模板使用 AWS Serverless Application Model 提供的 AWS::Serverless::Function 资源类型。

Example – 自定义资源函数

```
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  primer:
    Type: AWS::Serverless::Function
    Properties:
```

```
      Handler: index.handler
      Runtime: nodejs16.x
      InlineCode: |
        var aws = require('aws-sdk');
        var response = require('cfn-response');
        exports.handler = function(event, context) {
            // For Delete requests, immediately send a SUCCESS response.
            if (event.RequestType == "Delete") {
                response.send(event, context, "SUCCESS");
                return;
            }
            var responseStatus = "FAILED";
            var responseData = {};
            var functionName = event.ResourceProperties.FunctionName
            var lambda = new aws.Lambda();
            lambda.invoke({ FunctionName: functionName }, function(err, invokeResult) {
                if (err) {
                    responseData = {Error: "Invoke call failed"};
                    console.log(responseData.Error + ":\n", err);
                }
                else responseStatus = "SUCCESS";
                response.send(event, context, responseStatus, responseData);
            });
        };
      Description: Invoke a function to create a log stream.
      MemorySize: 128
      Timeout: 8
      Role: !GetAtt role.Arn
      Tracing: Active
```

如果模板中未定义自定义资源调用的函数，您可以从 AWS CloudFormation 用户指南中的 cfn-response 模块获取 `cfn-response` 的源代码。

有关自定义资源的更多信息，请参阅 AWS CloudFormation 用户指南中的自定义资源。

使用 Lambda 处理 Amazon DocumentDB 事件

您可以通过将 Amazon DocumentDB 集群配置为事件源，从而使用 Lambda 函数来处理 Amazon DocumentDB（与 MongoDB 兼容）更改流中的事件。然后，您可以在 Amazon DocumentDB 集群每次发生数据更改时调用 Lambda 函数，从而实现事件驱动型工作负载的自动化。

> **Note**
>
> Lambda 仅支持 Amazon DocumentDB 4.0 和 5.0 版本。Lambda 不支持 3.6 版本。
> 此外，对于事件源映射，Lambda 仅支持基于实例的集群和区域性集群。Lambda 不支持弹性集群 或 全球集群。当使用 Lambda 作为客户端连接到 Amazon DocumentDB 时，此限制不适用。Lambda 可以连接到所有集群类型来执行 CRUD 操作。

Lambda 按照事件到达的顺序依次处理 Amazon DocumentDB 更改流中的事件。因此，您的函数一次只能处理一条来自 Amazon DocumentDB 的并发调用。要监控函数，您可以跟踪其并发指标。

> **Warning**
>
> Lambda 事件源映射至少处理每个事件一次，有可能出现重复处理记录的情况。为避免与重复事件相关的潜在问题，我们强烈建议您将函数代码设为幂等性。要了解更多信息，请参阅 AWS 知识中心的如何使我的 Lambda 函数具有幂等性。

主题

Amazon DocumentDB 事件示例

```
{
    "eventSourceArn": "arn:aws:rds:us-
east-1:123456789012:cluster:canaryclusterb2a659a2-qo5tcmqkcl03",
    "events": [
        {
            "event": {
                "_id": {
                    "_data": "0163eeb6e7000000090100000009000041e1"
                },
                "clusterTime": {
                    "$timestamp": {
                        "t": 1676588775,
                        "i": 9
                    }
                },
                "documentKey": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    }
                },
                "fullDocument": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    },
                    "anyField": "sampleValue"
                },
                "ns": {
                    "db": "test_database",
                    "coll": "test_collection"
                },
                "operationType": "insert"
            }
        }
    ],
    "eventSource": "aws:docdb"
}
```

有关此示例中的事件及其形状的更多信息，请参阅 MongoDB 文档网站上的更改事件。

先决条件和权限

将 Amazon DocumentDB 用作 Lambda 函数的事件源之前，请注意以下先决条件。您必须：

- 有一个现有的 Amazon DocumentDB 集群并且该集群必须与函数位于同一 AWS 账户和 AWS 区域中。如果您没有现有的集群，可以按照《Amazon DocumentDB 开发者指南》中的 Amazon DocumentDB 入门创建一个。此外，教程：将 AWS Lambda 与 Amazon DocumentDB 流结合使用中的第一组步骤将指导您创建满足所有必要先决条件的 Amazon DocumentDB 集群。
- 允许 Lambda 访问与 Amazon DocumentDB 集群关联的 Amazon Virtual Private Cloud（Amazon VPC）资源的权限。有关更多信息，请参阅 配置网络安全。
- 在 Amazon DocumentDB 集群上启用 TLS。这是默认设置。如果您禁用了 TLS，Lambda 将无法与您的集群通信。
- 在 Amazon DocumentDB 集群上激活更改流。有关更多信息，请参阅《Amazon DocumentDB 开发人员指南》中的将更改流与 Amazon DocumentDB 搭配使用。
- 为 Lambda 提供访问 Amazon DocumentDB 集群的凭证。在设置事件源时，请提供包含访问集群所需的身份验证详细信息（用户名和密码）的 AWS Secrets Manager 密钥。要在设置过程中提供此密钥，请执行以下任一操作：
  - 如果您使用 Lambda 控制台进行设置，请在 Secrets Manager 密钥字段中提供密钥。
  - 如果您使用 AWS Command Line Interface（AWS CLI）进行设置，请在 `source-access-configurations` 选项中提供此密钥。您可以将此选项包含在 `create-event-source-mapping` 命令或 `update-event-source-mapping` 命令中。例如：

    ```
    aws lambda create-event-source-mapping \
        ...
        --source-access-configurations
     '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
    west-2:123456789012:secret:DocDBSecret-AbC4E6"}]' \
        ...
    ```
- 向 Lambda 授予管理与 Amazon DocumentDB 流相关的资源的权限。将以下角色权限手动添加到您的函数的 执行角色：
  - rds:DescribeDBClusters
  - rds:DescribeDBClusterParameters
  - rds:DescribeDBSubnetGroups
  - ec2:CreateNetworkInterface
  - ec2:DescribeNetworkInterfaces

- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups
- kms:Decrypt
- secretsmanager:GetSecretValue

- 将您发送到 Lambda 的 Amazon DocumentDB 更改流事件的大小保持在 6MB 以下。Lambda 支持最大 6MB 的负载大小。如果更改流尝试向 Lambda 发送大于 6MB 的事件，Lambda 会丢弃该消息并发送 `OversizedRecordCount` 指标。Lambda 将尽力发送所有指标。

> **Note**
>
> 尽管 Lambda 函数的最大超时限制通常为 15 分钟，但 Amazon MSK、自行管理的 Apache Kafka、Amazon DocumentDB、Amazon MQ for ActiveMQ 和 RabbitMQ 的事件源映射，仅支持最大超时限制为 14 分钟的函数。此约束可确保事件源映射可以正确处理函数错误和重试。

配置网络安全

要通过事件源映射向 Lambda 提供对 Amazon DocumentDB 的完全访问权限，集群必须使用公有端点（公有 IP 地址），或者您必须提供对您在其中创建了集群的 Amazon VPC 的访问权限。

将 Amazon DocumentDB 与 Lambda 配合使用时，我们建议您创建 AWS PrivateLinkVPC 端点，并向函数提供对 Amazon VPC 中资源的访问权限。

创建端点以提供对以下资源的访问权限：

- Lambda：为 Lambda 服务主体创建端点。
- AWS STS：为 AWS STS 创建端点，以便服务主体代您代入角色。
- Secrets Manager：如果集群使用 Secrets Manager 来存储凭证，请为 Secrets Manager 创建端点。

也可以在 Amazon VPC 中的每个公有子网上配置 NAT 网关。有关更多信息，请参阅 the section called “VPC 函数的互联网访问权限”。

为 Amazon DocumentDB 创建事件源映射时，Lambda 会检查为 Amazon VPC 配置的子网和安全组是否已经存在弹性网络接口（ENI）。如果 Lambda 发现现有 ENI，则会尝试重用这些 ENI。否则，Lambda 会创建新的 ENI 来连接到事件源并调用函数。

> **Note**
>
> Lambda 函数始终在 Lambda 服务拥有的 Amazon VPC 中运行。函数的 VPC 配置不会影响事件源映射。只有事件源的网络配置才能决定 Lambda 连接到事件源的方式。

为包含集群的 Amazon VPC 配置安全组。默认情况下，Amazon DocumentDB 使用以下端口：27017。

- 入站规则：允许与事件源关联安全组的默认集群端口的所有流量。
- 出站规则：允许所有目标的端口 443 上的所有流量。允许与事件源关联安全组的默认集群的所有流量。
- Amazon VPC 端点入站规则：如果您正在使用 Amazon VPC 端点，则与 Amazon VPC 端点关联的安全组，必须允许来自集群安全组的端口 443 上的入站流量。

如果集群使用身份验证，您还可以限制 Secrets Manager 端点的端点策略。要调用 Secrets Manager API，Lambda 会使用函数角色而非 Lambda 服务主体。

Example VPC 端点策略：Secrets Manager 端点

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
                  ]
              },
              "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
          }
      ]
  }
```

当您使用 Amazon VPC 端点时，AWS 会使用端点的弹性网络接口（ENI）路由 API 调用来调用函数。Lambda 服务主体需要针对使用这些 ENI 的任何函数调用 `lambda:InvokeFunction`。默认情况下，Amazon VPC 端点具有开放的 IAM 策略，允许对资源进行广泛访问。要在生产环境中配合使用 Amazon DocumentDB 和 Lambda，您可以将这些策略限制为仅允许特定的主体访问特定的角色和函数。

Example 端点策略：Lambda 端点

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "arn:aws::lambda:us-west-2:123456789012:function:my-function"
          }
      ]
  }
```

此外，对于事件源映射，如果您希望与 Lambda 集成的资源在 AWS 账户中部署，则 Lambda 服务主体必须执行 `sts:AssumeRole` 才能代入使用弹性网络接口（ENI）的角色。

Example 端点策略：AWS STS 端点

```
{
      "Statement": [
          {
              "Action": "sts:AssumeRole",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "arn:aws::iam::123456789012:role/my-role"
          }
      ]
    }
```

> **Warning**
>
> 将端点策略限制为仅允许来自组织内部的 API 调用，这将阻止事件源映射正常运行。

创建 Amazon DocumentDB 事件源映射（控制台）

要配置 Lambda 函数以从 Amazon DocumentDB 集群更改流中读取数据，请创建一个 事件源映射。这一部分介绍了如何从 Lambda 控制台执行此操作。有关 AWS SDK 和 AWS CLI 说明，请参阅 the section called “创建 Amazon DocumentDB 事件源映射（SDK 或 CLI）”。

要创建 Amazon DocumentDB 事件源映射（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数的名称。
3. 在 Function overview（函数概览）下，选择 Add trigger（添加触发器）。
4. 在触发器配置下的下拉列表中，选择 DocumentDB。
5. 配置必填选项，然后选择 Add（添加）。

Lambda 支持 Amazon DocumentDB 事件源的以下选项：

- DocumentDB 集群 – 选择一个 Amazon DocumentDB 集群。
- 激活触发器 – 选择是否要立即激活触发器。如果选中此复选框，则在创建事件源映射后，您的函数会立即开始接收来自指定 Amazon DocumentDB 更改流的流量。我们建议取消选中此复选框，以便在禁用状态下创建事件源映射以进行测试。创建事件源映射后，您可以随时将其激活。
- 数据库名称 – 输入集群中要使用的数据库的名称。
- （可选）集合名称 – 输入数据库中要使用的集合的名称。如果您未指定集合，Lambda 将侦听来自数据库中每个集合的所有事件。
- 批处理大小 – 设置要在单个批处理中检索的最大消息数，最高为 10000。默认批处理大小为 100。
- 开始位置 – 选择将从流中开始读取记录的位置。
  - 最新 – 仅处理添加到流中的新记录。您的函数仅在 Lambda 完成创建事件源后才会开始处理记录。这意味着在成功创建事件源之前，某些记录可能会被丢弃。
  - 最早：处理流中的所有记录。Lambda 使用集群的日志保留期来确定从哪里开始读取事件。具体而言，Lambda 将从 `current_time - log_retention_duration` 开始读取事件。更改流必须在此时间戳之前已经处于活动状态，才能确保 Lambda 正确读取所有事件。

  - 在时间戳处：处理从特定时间开始的记录。更改流必须在指定的时间戳之前已经处于活动状态，才能确保 Lambda 正确读取所有事件。
- 身份验证 – 选择访问集群中的代理时将使用的身份验证方法。
  - BASIC_AUTH – 使用基本身份验证时，您必须提供包含访问集群凭据所需凭证的 Secrets Manager 密钥。
- Secrets Manager 密钥 – 选择包含访问 Amazon DocumentDB 集群所需身份验证详细信息（用户名和密码）的 Secrets Manager 密钥。
- （可选）批处理时长 – 在调用函数之前收集记录的最长时间（以秒为单位，最大为 300）。
- （可选）完整文档配置 – 对于文档更新操作，请选择要发送到流中的内容。默认值为 `Default`，这意味着对于每个更改流事件，Amazon DocumentDB 仅发送一个描述所发生更改的增量。有关此字段的更多信息，请参阅 MongoDB Javadocs API 文档中的 FullDocument。
  - 默认 – Lambda 将仅发送描述所发生更改的部分文档。
  - UpdateLookup – Lambda 将发送一个描述所发生更改的增量以及完整文档的副本。

创建 Amazon DocumentDB 事件源映射（SDK 或 CLI）

要使用 AWS SDK 来管理 Amazon DocumentDB 事件源映射，您可以使用以下 API 操作：

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

要使用 AWS CLI 创建事件源映射，请使用 `create-event-source-mapping` 命令。以下示例使用此命令将名为 `my-function` 的函数映射到一个 Amazon DocumentDB 更改流。事件源由 Amazon 资源名称（ARN）指定，批处理大小为 500，从以 Unix 时间表示的时间戳开始。该命令还指定了 Lambda 用来连接到 Amazon DocumentDB 的 Secrets Manager 密钥。此外，它还包含用于指定要从中读取数据的数据库和集合的 `document-db-event-source-config` 参数。

```
aws lambda create-event-source-mapping --function-name my-function \
    --event-source-arn arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy
    --batch-size 500 \
    --starting-position AT_TIMESTAMP \
```

```
    --starting-position-timestamp 1541139109 \
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocDBSecret-BAtjxi"}]' \
    --document-db-event-source-config '{"DatabaseName":"test_database",
 "CollectionName": "test_collection"}' \
```

应看到类似如下内容的输出：

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541348195.412,
    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "User action"
}
```

创建后，您可以使用 update-event-source-mapping 命令更新 Amazon DocumentDB 事件源的设置。以下示例会将批处理大小更新为 1000，并将批处理时长更新为 10 秒。对于此命令，您需要有事件源映射的 UUID，您可以通过 list-event-source-mapping 命令或 Lambda 控制台检索该值。

```
aws lambda update-event-source-mapping --function-name my-function \
    --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
    --batch-size 1000 \
    --batch-window 10
```

您应看到如下输出：

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
```

```
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Updating",
    "StateTransitionReason": "User action"
}
```

Lambda 会异步更新设置，因此在更新完成之前，您可能无法在输出中看到这些更改。要查看事件源映射的当前设置，请使用 get-event-source-mapping 命令。

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

您应看到如下输出：

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "BatchSize": 1000,
    "MaximumBatchingWindowInSeconds": 10,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Enabled",
    "StateTransitionReason": "User action"
}
```

要删除 Amazon DocumentDB 事件源映射，请使用 delete-event-source-mapping 命令：

```
aws lambda delete-event-source-mapping \
    --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

轮询和流的起始位置

请注意，事件源映射创建和更新期间的流轮询最终是一致的。

- 在事件源映射创建期间，可能需要几分钟才能开始轮询来自流的事件。
- 在事件源映射更新期间，可能需要几分钟才能停止和重新开始轮询来自流的事件。

此行为意味着，如果你指定 LATEST 作为流的起始位置，事件源映射可能会在创建或更新期间错过事件。为确保不会错过任何事件，请将流的起始位置指定为 TRIM_HORIZON 或 AT_TIMESTAMP。

监控 Amazon DocumentDB 事件源

为便于您监控 Amazon DocumentDB 事件源，在函数处理完一批记录后，Lambda 将发送 IteratorAge 指标。迭代器期限是最近事件的时间戳和当前时间戳之间的差值。基本上，IteratorAge 指标表示自处理该批处理中最后一条记录后已经过的时间。如果函数当前正在处理新事件，则可使用迭代器期限来估算新记录的添加时间与函数处理该记录的时间之间的延迟。如果 IteratorAge 呈上升趋势，则可能说明您的函数存在问题。有关更多信息，请参阅 查看 Lambda 函数的指标。

Amazon DocumentDB 更改流未经过优化，无法处理事件之间的较大时间间隔。如果 Amazon DocumentDB 事件源在很长一段时间内都没有收到任何事件，那么 Lambda 可能会禁用事件源映射。此时段的长度可能从几周到几个月不等，具体取决于集群规模和其他工作负载。

Lambda 支持最大 6MB 的负载。但是，Amazon DocumentDB 更改流事件的大小可达 16MB。如果更改流尝试向 Lambda 发送大于 6MB 的更改流事件，Lambda 会丢弃该消息并发送 OversizedRecordCount 指标。Lambda 将尽力发送所有指标。

教程：将 AWS Lambda 与 Amazon DocumentDB 流结合使用

在本教程中，您将创建一个基本的 Lambda 函数，该函数会消耗来自 Amazon DocumentDB（与 MongoDB 兼容）更改流的事件。要完成本教程，您需要经历以下阶段：

- 设置 Amazon DocumentDB 集群，连接到该集群，并在其上激活更改流。
- 创建 Lambda 函数，并将 Amazon DocumentDB 集群配置为函数的事件源。
- 将项目插入到 Amazon DocumentDB 数据库中，以测试端到端设置。

主题

先决条件

注册 AWS 账户

如果您还没有 AWS 账户，请完成以下步骤来创建一个。

注册 AWS 账户

1. 打开 https://portal.aws.amazon.com/billing/signup。
2. 按照屏幕上的说明进行操作。

   在注册时，将接到一通电话，要求使用电话键盘输入一个验证码。

   当您注册 AWS 账户时，系统将会创建一个 AWS 账户根用户。根用户有权访问该账户中的所有 AWS 服务 和资源。作为安全最佳实践，请为用户分配管理访问权限，并且只使用根用户来执行需要根用户访问权限的任务。

注册过程完成后，AWS 会向您发送一封确认电子邮件。在任何时候，您都可以通过转至 https://aws.amazon.com/ 并选择我的账户来查看当前的账户活动并管理您的账户。

创建具有管理访问权限的用户

注册 AWS 账户 后，请保护好您的 AWS 账户根用户，启用 AWS IAM Identity Center，并创建一个管理用户，以避免使用根用户执行日常任务。

保护您的 AWS 账户根用户

1. 选择根用户并输入您的 AWS 账户电子邮件地址，以账户拥有者身份登录 AWS Management Console。在下一页上，输入您的密码。

   要获取使用根用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的以根用户身份登录。
2. 为您的根用户启用多重身份验证 (MFA)。

   有关说明，请参阅《IAM 用户指南》中的为 AWS 账户 根用户启用虚拟 MFA 设备（控制台）。

创建具有管理访问权限的用户

1. 启用 IAM Identity Center。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的启用 AWS IAM Identity Center。
2. 在 IAM Identity Center 中，为用户授予管理访问权限。

   有关如何使用 IAM Identity Center 目录 作为身份源的教程，请参阅《AWS IAM Identity Center 用户指南》中的使用默认的 IAM Identity Center 目录 配置用户访问权限。

以具有管理访问权限的用户身份登录

- 要使用您的 IAM Identity Center 用户身份登录，请使用您在创建 IAM Identity Center 用户时发送到您的电子邮件地址的登录网址。

  要获取使用 IAM Identity Center 用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的登录 AWS 访问门户。

将访问权限分配给其他用户

1. 在 IAM Identity Center 中，创建一个权限集，该权限集遵循应用最低权限的最佳做法。

有关说明，请参阅《AWS IAM Identity Center 用户指南》中的创建权限集。

2. 将用户分配到一个组，然后为该组分配单点登录访问权限。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的添加组。

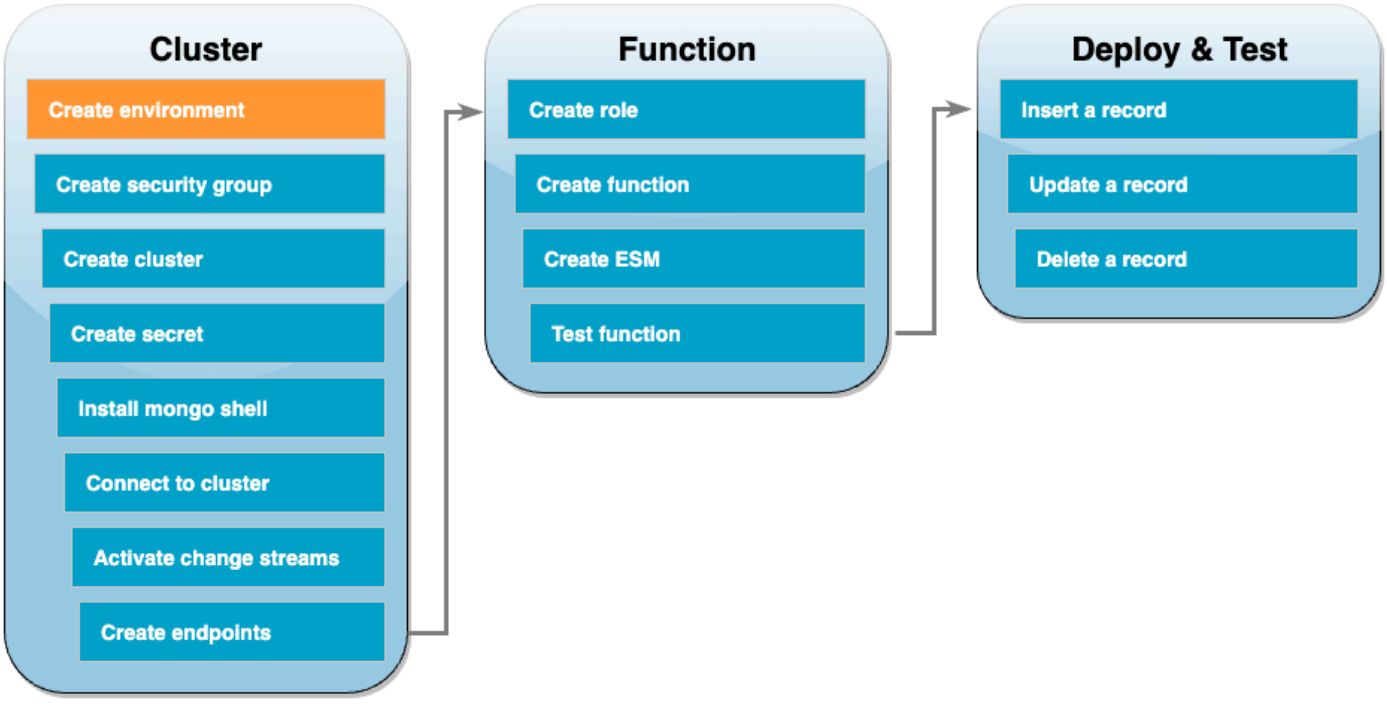
安装 AWS Command Line Interface

如果您尚未安装 AWS Command Line Interface，请按照安装或更新最新版本的 AWS CLI 中的步骤进行安装。

本教程需要命令行终端或 Shell 来运行命令。在 Linux 和 macOS 中，可使用您首选的 Shell 和程序包管理器。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常与 Lambda 一起使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。

创建 AWS Cloud9 环境



在创建 Lambda 函数之前，您需要创建和配置 Amazon DocumentDB 集群。本教程中的集群设置步骤基于 Amazon DocumentDB 入门中的过程。

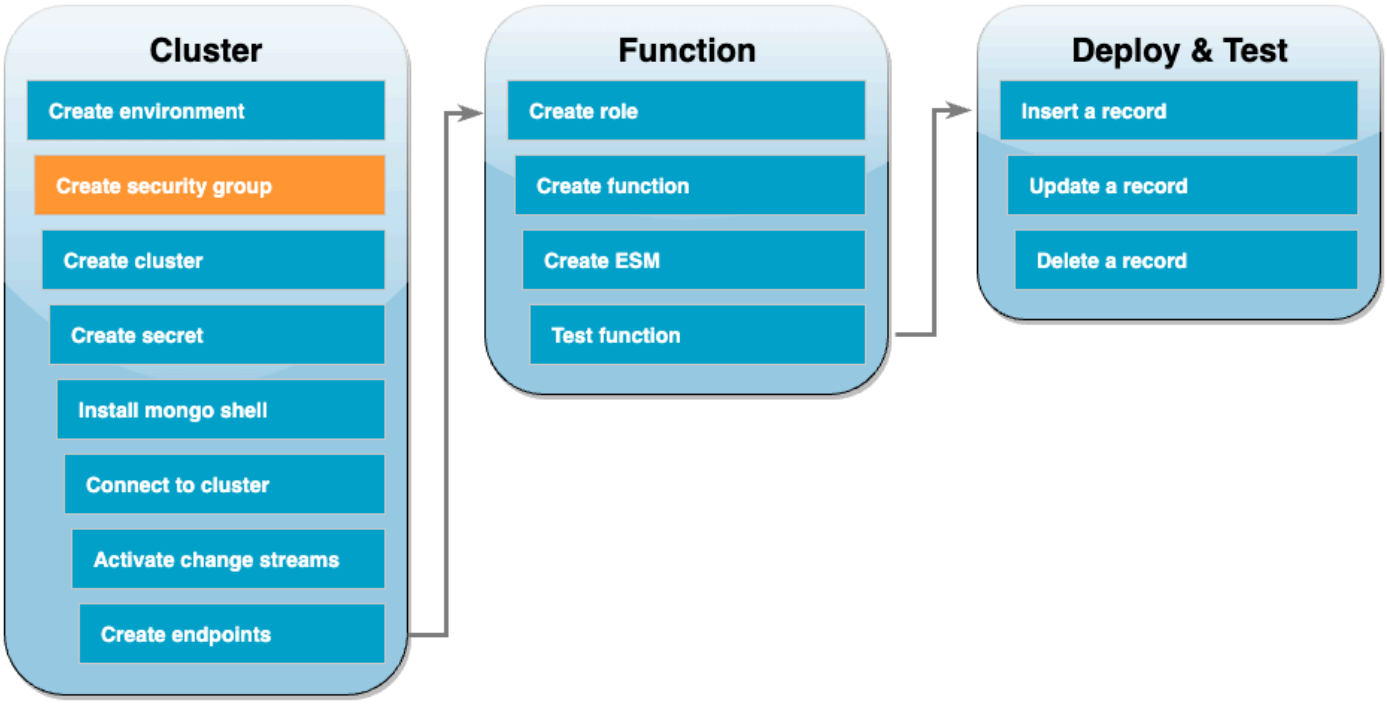
> **Note**
>
> 如果已设置 Amazon DocumentDB 集群，请确保激活更改流并创建必要的接口 VPC 端点。然后，您可以直接跳到函数创建步骤。

首先，创建一个 AWS Cloud9 环境。在本教程中，您将使用此环境来连接和查询 Amazon DocumentDB 集群。

创建 AWS Cloud9 环境

1. 打开 AWS Cloud9 控制台并选择创建环境。
2. 使用以下配置创建环境：
   - 在详细信息下：
     - 名称：`DocumentDBCloud9Environment`
     - 环境类型：新 EC2 实例
   - 在新 EC2 实例下：
     - 实例类型：`t2.micro`（1 GiB RAM + 1 个 vCPU）
     - 平台：Amazon Linux 2
     - 超时：30 分钟
   - 在网络设置下：
     - 连接：AWS Systems Manager（SSM）
     - 展开 VPC 设置下拉列表。
     - Amazon Virtual Private Cloud（VPC）：选择默认 VPC。
     - 子网：无首选项
   - 保留所有其他默认设置。
3. 选择创建。预配置新的 AWS Cloud9 环境可能需要花费几分钟的时间。

创建 Amazon EC2 安全组



接下来，使用允许 Amazon DocumentDB 集群和 AWS Cloud9 环境之间传输流量的规则创建 Amazon EC2 安全组。

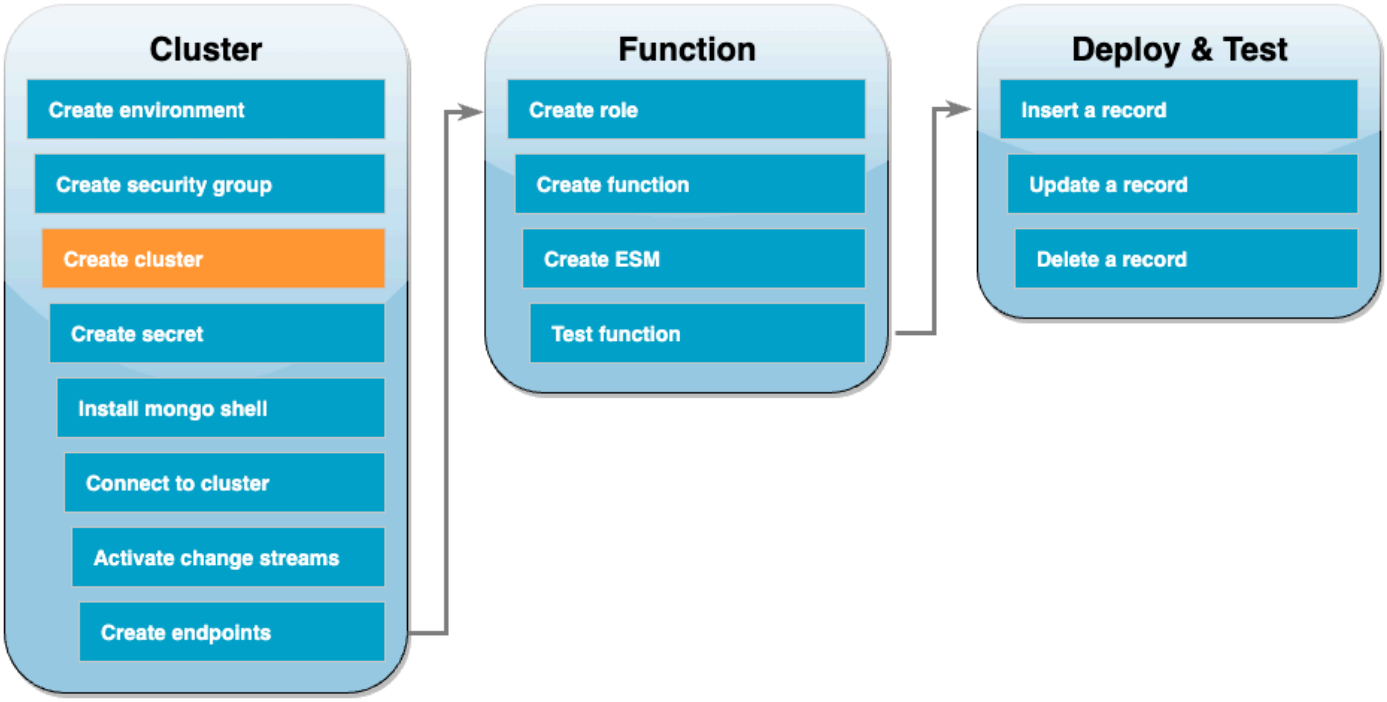
创建 EC2 安全组

1. 打开 EC2 控制台。在网络与安全性下，选择安全组。
2. 选择创建安全组。
3. 使用以下配置创建安全组：
   - 在基本详细信息下：
     - 安全组名称：`DocDBTutorial`
     - 描述：用于在 AWS Cloud9 与 Amazon DocumentDB 之间传输流量的安全组。
     - VPC：选择默认 VPC。
   - 在 Inbound rules (入站规则)下面，选择 Add rule (添加规则)。使用以下配置创建规则：
     - 类型：自定义 TCP
     - 端口范围：27017
     - Source：Custom

- 在源旁边的搜索框中，为您在上一步中创建的 AWS Cloud9 环境选择安全组。要查看可用安全组列表，请在搜索框中输入 `cloud9`。选择名为 `aws-cloud9-<environment_name>` 的安全组。
- 保留所有其他默认设置。

4. 选择创建安全组。

创建 Amazon DocumentDB 集群



在此步骤中，您将使用上一步中的安全组创建 Amazon DocumentDB 集群。

创建 Amazon DocumentDB 集群

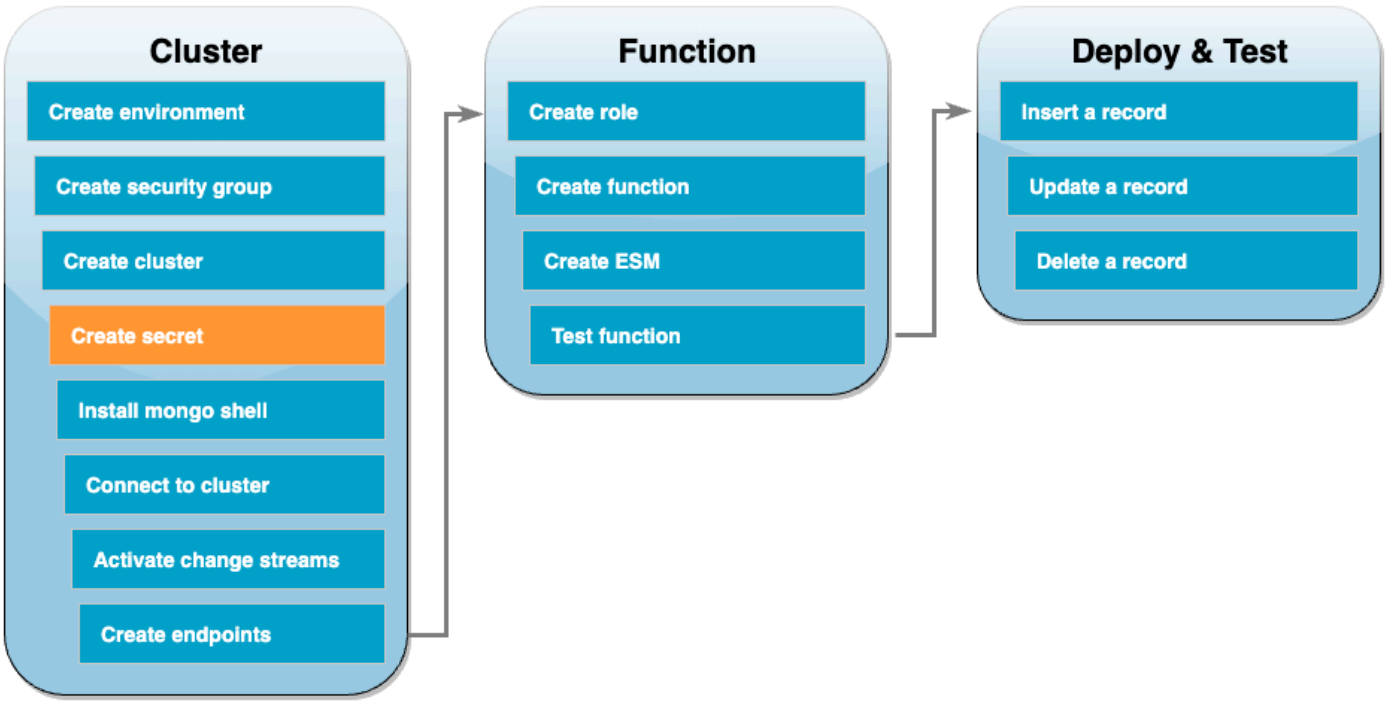
1. 打开 Amazon DocumentDB 控制台。在集群下，选择创建。
2. 使用以下配置创建集群：

   - 对于集群类型，选择“基于实例的集群”。
   - 在配置下：
     - 引擎版本：5.0.0
     - 实例类：db.t3.medium（可免费试用）
     - 实例数：1。

- 在身份验证下：
  - 输入连接到集群所需的用户名和密码（与上一步中用于创建密钥的凭证相同）。在确认密码中，确认您的密码。
- 开启显示高级设置。
- 在网络设置下：
  - Virtual Private Cloud（VPC）：选择默认 VPC。
  - 子网组：默认
  - VPC 安全组：除 `default (VPC)` 之外，选择您在上一步中创建的 `DocDBTutorial (VPC)` 安全组。
- 保留所有其他默认设置。

3. 选择创建集群。预置 Amazon DocumentDB 集群可能需要花费几分钟的时间。

在 Secrets Manager 中创建密钥



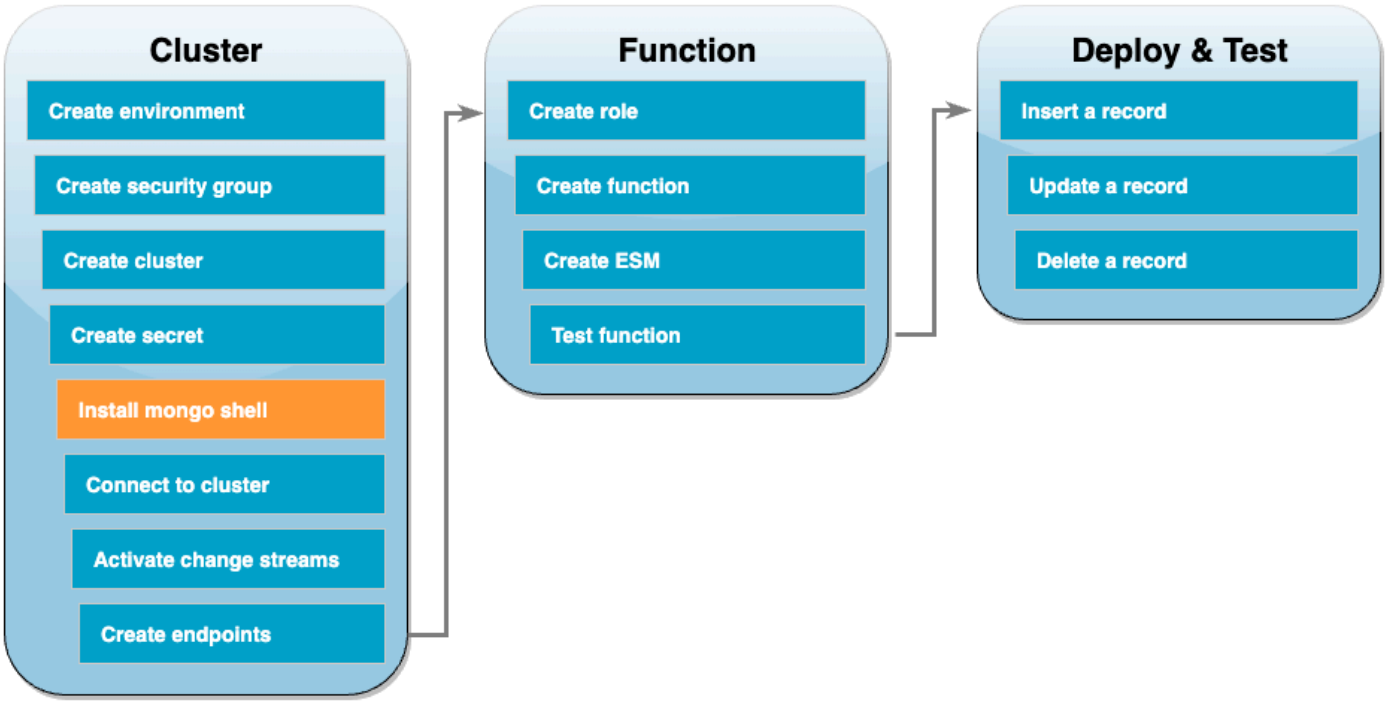
要手动访问 Amazon DocumentDB 集群，必须提供用户名和密码凭证。要允许 Lambda 访问集群，您必须在设置事件源映射时，提供包含这些相同访问凭证的 Secrets Manager 密钥。在此步骤中，您将创建此密钥。

在 Secrets Manager 中创建密钥

1. 打开 Secrets Manager 控制台并选择存储新密钥。
2. 对于选择密钥类型，请选择以下选项之一：
   - 在基本详细信息下：
     - 密钥类型：用于 Amazon DocumentDB 数据库的凭证
     - 在凭证下，输入用于访问 Amazon DocumentDB 集群的用户名和密码。
     - 数据库：选择 Amazon DocumentDB 集群。
     - 选择下一步。
3. 对于配置密钥，请选择以下选项之一：
   - 密钥名称 – `DocumentDBSecret`
   - 选择下一步。
4. 选择下一步。
5. 选择 Store (存储)。
6. 刷新控制台以验证 `DocumentDBSecret` 密钥是否成功存储。

记下密钥的密钥 ARN。您将在后面的步骤中用到它。

安装 mongo shell



在此步骤中，您将在 AWS Cloud9 环境中安装 Mongo Shell。Mongo Shell 是一个命令行实用程序，用于连接和查询 Amazon DocumentDB 集群。

在 AWS Cloud9 环境中安装 Mongo Shell

1. 打开AWS Cloud9控制台。在之前创建的 `DocumentDBCloud9Environment` 环境旁边，单击 AWS Cloud9 IDE 列下的打开链接。
2. 在终端窗口中，使用以下命令创建 MongoDB 存储库文件：

```
echo -e "[mongodb-org-5.0] \nname=MongoDB Repository\nbaseurl=https://
repo.mongodb.org/yum/amazon/2/mongodb-org/5.0/x86_64/\ngpgcheck=1 \nenabled=1
 \ngpgkey=https://www.mongodb.org/static/pgp/server-5.0.asc" | sudo tee /etc/
yum.repos.d/mongodb-org-5.0.repo
```

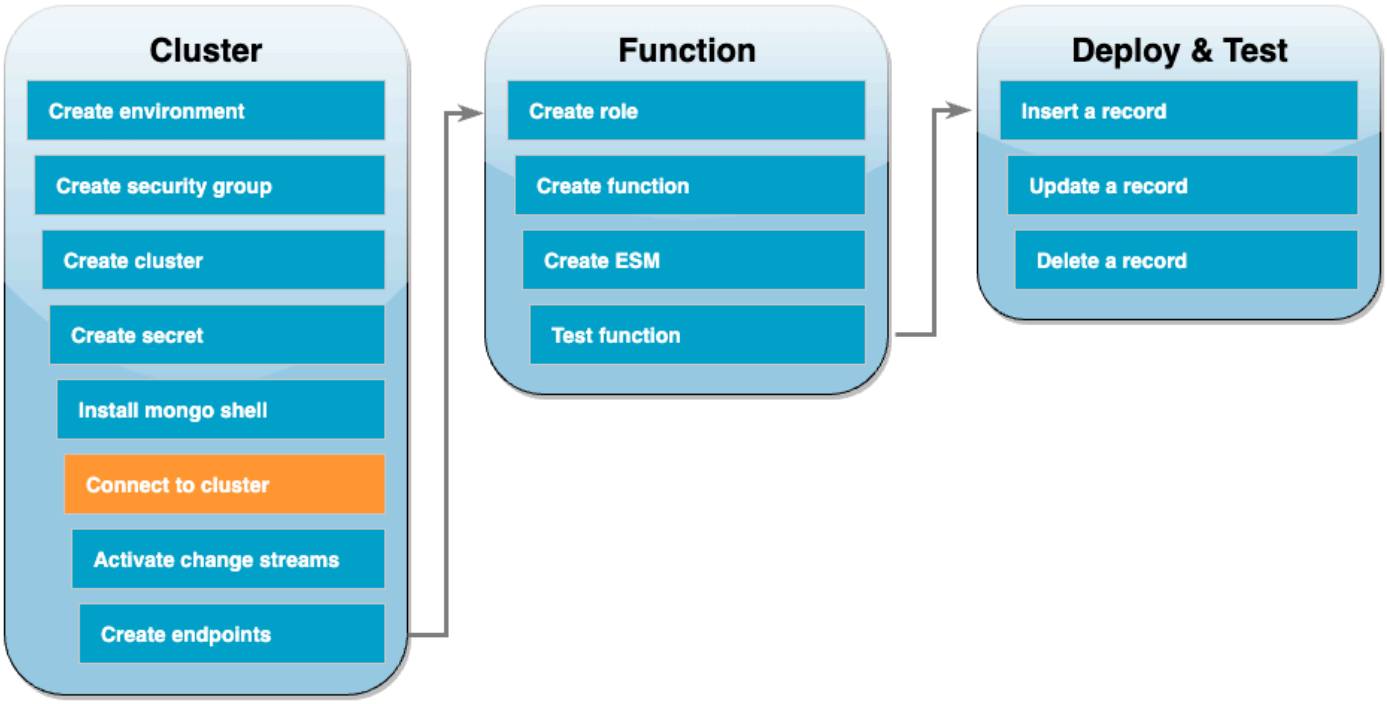
3. 然后，使用以下命令安装 Mongo Shell：

```
sudo yum install -y mongodb-org-shell
```

4. 要加密传输中数据，请下载 Amazon DocumentDB 的公有密钥。以下命令将下载名为 `global-bundle.pem` 的文件：

```
wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

连接到 Amazon DocumentDB 集群



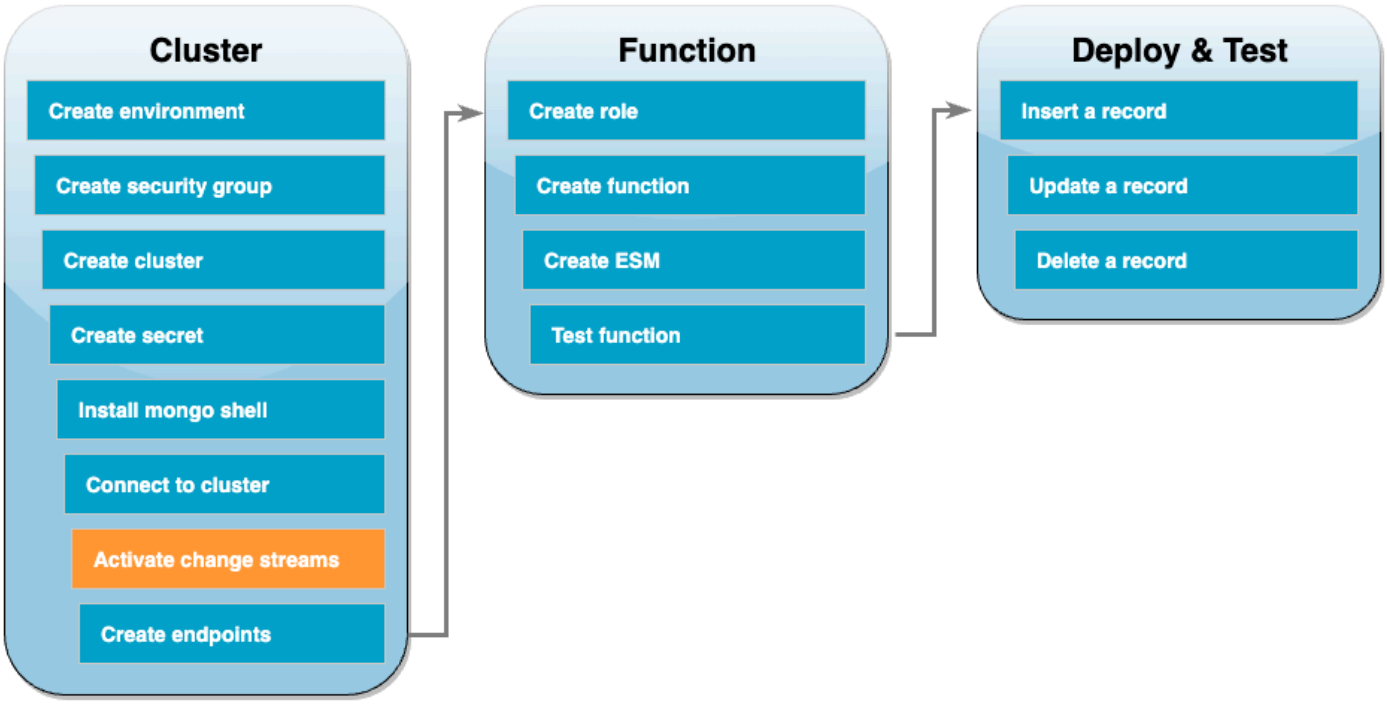
现在，您已经准备好使用 Mongo Shell 连接到 Amazon DocumentDB 集群。

连接到 Amazon DocumentDB 集群

1. 打开 Amazon DocumentDB 控制台。在集群下，通过选择集群标识符来选择集群。
2. 在连接和安全性选项卡中，在使用 Mongo Shell 连接到此集群下，选择复制。
3. 在 AWS Cloud9 环境中，将此命令粘贴到终端。将 `<insertYourPassword>` 替换为正确的密码。

输入此命令后，如果命令提示符变为 `rs0:PRIMARY>`，则表示您已连接到 Amazon DocumentDB 集群。

激活更改流



在本教程中，您将跟踪对 Amazon DocumentDB 集群中 docdbdemo 数据库 products 集合的更改。您可以通过激活更改流来完成此操作。首先，创建 docdbdemo 数据库并通过插入记录对其进行测试。

在集群内创建新数据库

1. 在 AWS Cloud9 环境中，确保仍然连接到 Amazon DocumentDB 集群。
2. 在终端窗口中，使用以下命令创建一个名为 docdbdemo 的新数据库：

```
use docdbdemo
```

3. 然后，使用以下命令将记录插入到 docdbdemo 中：

```
db.products.insert({"hello":"world"})
```

应看到类似如下内容的输出：

```
WriteResult({ "nInserted" : 1 })
```

4. 使用以下命令列出所有数据库：

```
show dbs
```

确保输出包含 docdbdemo 数据库：

```
docdbdemo  0.000GB
```

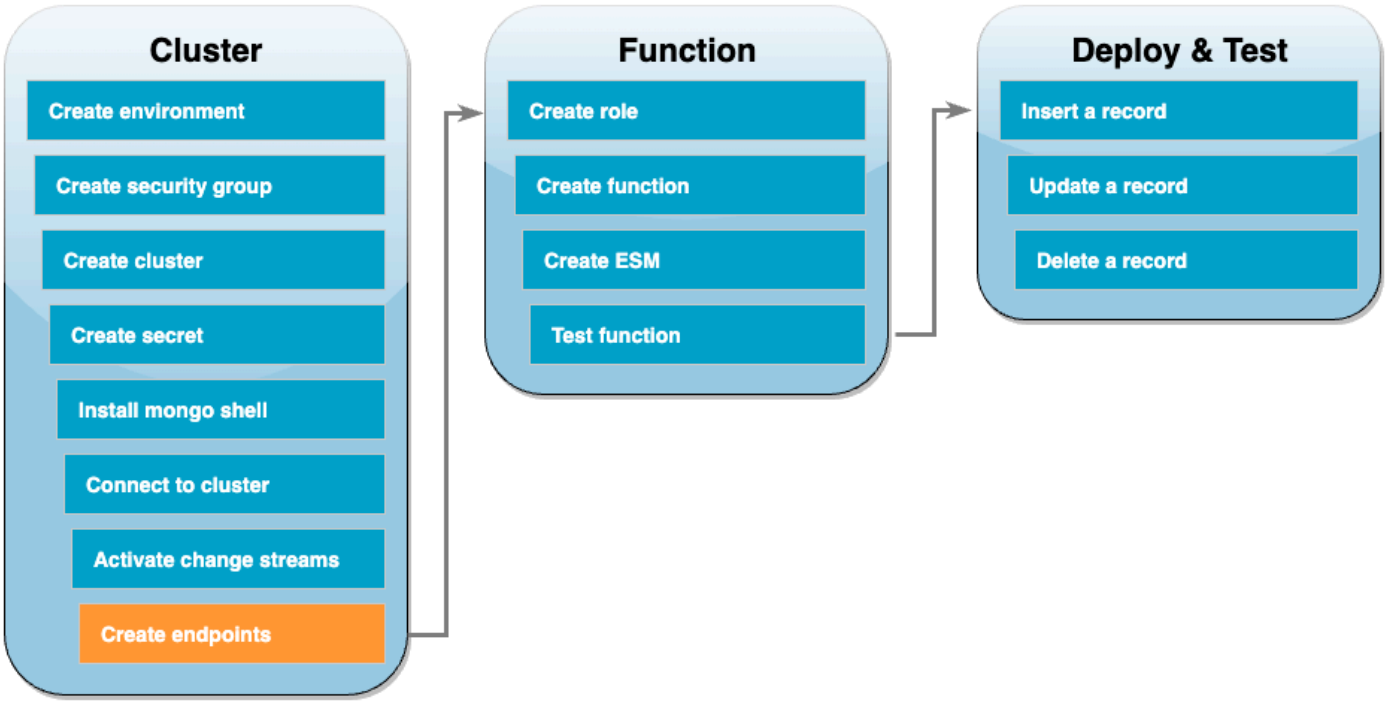
接下来，使用以下命令激活 docdbdemo 数据库 products 集合上的更改流：

```
db.adminCommand({modifyChangeStreams: 1,
    database: "docdbdemo",
    collection: "products",
    enable: true});
```

应看到类似如下内容的输出：

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```

创建接口 VPC 端点



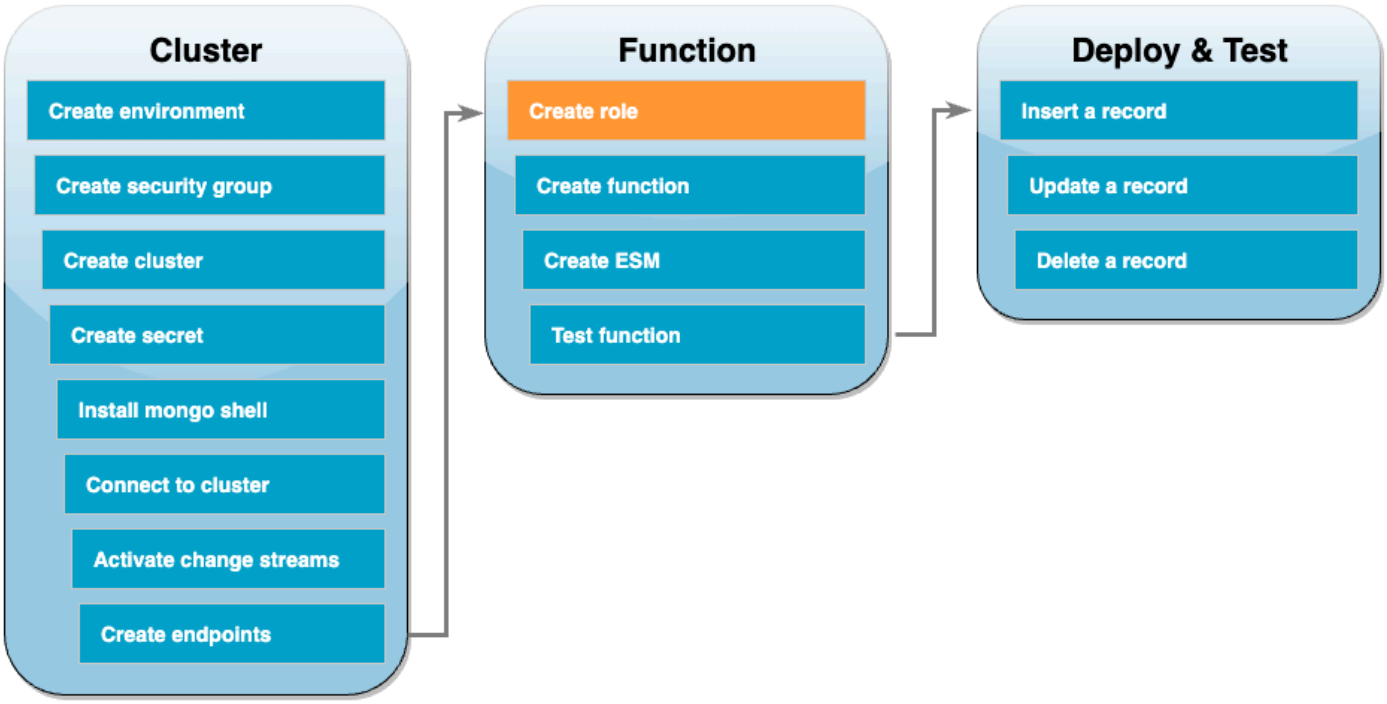
接下来，创建接口 VPC 端点，以确保 Lambda 和 Secrets Manager（稍后用于存储集群访问凭证）能够连接到默认 VPC。

创建接口 VPC 端点

1. 打开 VPC 控制台。在左侧菜单的虚拟私有云下，选择端点。
2. 选择创建端点。使用以下配置创建端点：
   - 对于名称标签，输入 `lambda-default-vpc`。
   - 对于服务类别，选择 AWS 服务。
   - 对于服务，在搜索框中输入 `lambda`。选择格式为 `com.amazonaws.<region>.lambda` 的服务。
   - 对于 VPC，选择默认 VPC。
   - 对于子网，选中每个可用区旁边的复选框。为每个可用区选择正确的子网 ID。
   - 对于 IP 地址类型，选择 IPv4。
   - 对于安全组，选择默认的 VPC 安全组（组名称为 `default`），以及之前创建的安全组（组名称为 `DocDBTutorial`）。
   - 保留所有其他默认设置。
   - 选择创建端点。
3. 再次选择创建端点。使用以下配置创建端点：
   - 对于名称标签，输入 `secretsmanager-default-vpc`。
   - 对于服务类别，选择 AWS 服务。
   - 对于服务，在搜索框中输入 `secretsmanager`。选择格式为 `com.amazonaws.<region>.secretsmanager` 的服务。
   - 对于 VPC，选择默认 VPC。
   - 对于子网，选中每个可用区旁边的复选框。为每个可用区选择正确的子网 ID。
   - 对于 IP 地址类型，选择 IPv4。
   - 对于安全组，选择默认的 VPC 安全组（组名称为 `default`），以及之前创建的安全组（组名称为 `DocDBTutorial`）。
   - 保留所有其他默认设置。
   - 选择创建端点。

本教程的集群设置部分到此完成。

创建执行角色



在接下来的一组步骤中，您将创建 Lambda 函数。首先，您需要创建执行角色，以向函数授予访问集群的权限。为此，您可以先创建 IAM policy，然后将此策略附加到 IAM 角色。

创建 IAM policy

1. 在 IAM 控制台中打开策略页面，然后选择创建策略。
2. 选择 JSON 选项卡。在以下策略中，将语句最后一行中的 Secrets Manager 资源 ARN 替换为之前的密钥 ARN，然后将策略复制到编辑器中。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LambdaESMNetworkingAccess",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeVpcs",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
```

```
                "ec2:DescribeSecurityGroups",
                "kms:Decrypt"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMAccess",
            "Effect": "Allow",
            "Action": [
                "rds:DescribeDBClusters",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBSubnetGroups"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMGetSecretValueAccess",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocumentDBSecret"
        }
    ]
}
```

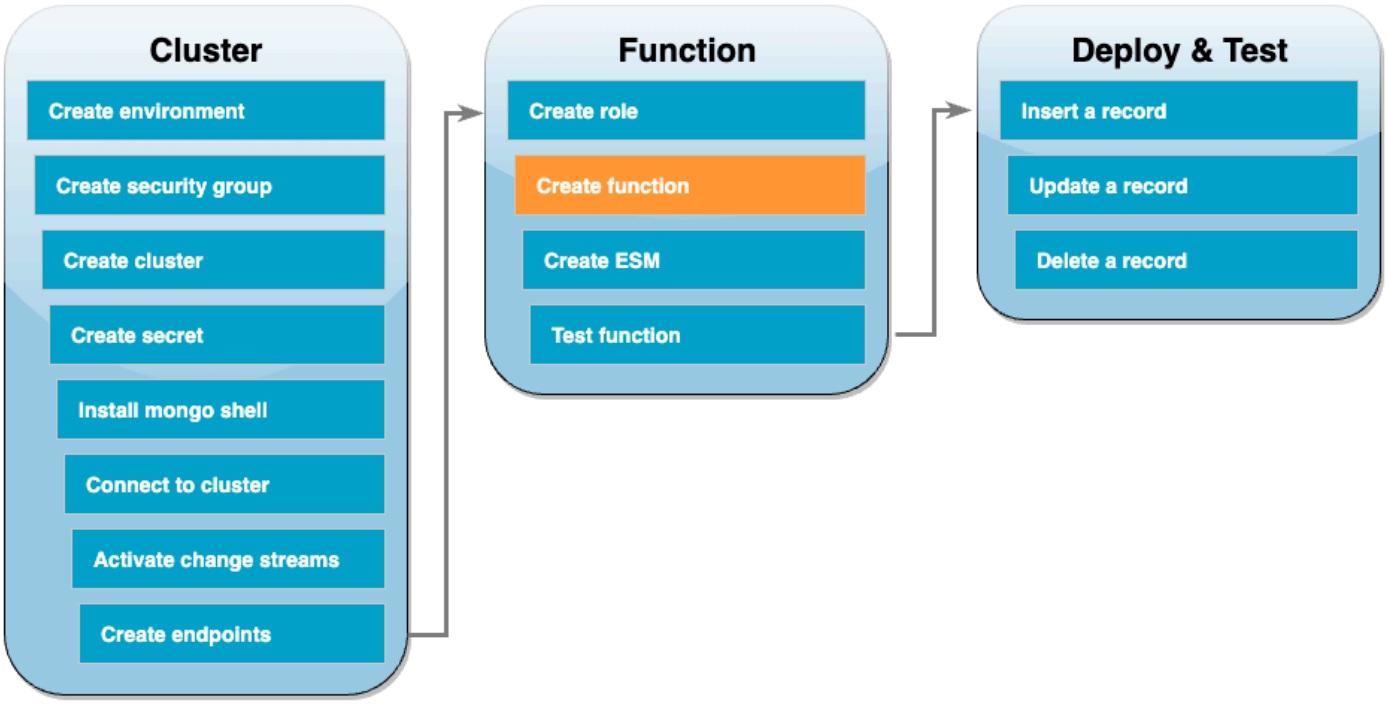
3. 选择下一步：标签，然后选择下一步：审核。
4. 对于 Name (名称)，请输入 AWSDocumentDBLambdaPolicy。
5. 选择创建策略。

创建 IAM 角色

1. 在 IAM 控制台中打开角色页面，然后选择创建角色。
2. 对于选择可信实体，请选择以下选项之一：
   - 可信实体类型 – AWS 服务
   - 应用场景 – Lambda
   - 选择下一步。

3. 对于添加权限，选择刚刚创建的 `AWSDocumentDBLambdaPolicy` 策略以及 `AWSLambdaBasicExecutionRole`，以向函数授予写入 Amazon CloudWatch Logs 的权限。
4. 选择下一步。
5. 对于 Role name（角色名称），输入 `AWSDocumentDBLambdaExecutionRole`。
6. 请选择 Create role（创建角色）。

创建 Lambda 函数



以下示例代码接收 Amazon DocumentDB 事件输入并对其所包含的消息进行处理。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 .NET 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
using Amazon.Lambda.Core;
using System.Text.Json;
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;
//Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaDocDb;

public class Function
{

     /// <summary>
    /// Lambda function entry point to process Amazon DocumentDB events.
    /// </summary>
    /// <param name="event">The Amazon DocumentDB event.</param>
    /// <param name="context">The Lambda context object.</param>
    /// <returns>A string to indicate successful processing.</returns>
    public string FunctionHandler(Event evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Events)
        {
            ProcessDocumentDBEvent(record, context);
        }

        return "OK";
    }

     private void ProcessDocumentDBEvent(DocumentDBEventRecord record,
 ILambdaContext context)
    {

        var eventData = record.Event;
        var operationType = eventData.OperationType;
        var databaseName = eventData.Ns.Db;
        var collectionName = eventData.Ns.Coll;
        var fullDocument = JsonSerializer.Serialize(eventData.FullDocument, new
 JsonSerializerOptions { WriteIndented = true });
```

```
        context.Logger.LogLine($"Operation type: {operationType}");
        context.Logger.LogLine($"Database: {databaseName}");
        context.Logger.LogLine($"Collection: {collectionName}");
        context.Logger.LogLine($"Full document:\n{fullDocument}");
    }



    public class Event
    {
        [JsonPropertyName("eventSourceArn")]
        public string EventSourceArn { get; set; }

        [JsonPropertyName("events")]
        public List<DocumentDBEventRecord> Events { get; set; }

        [JsonPropertyName("eventSource")]
        public string EventSource { get; set; }
    }

    public class DocumentDBEventRecord
    {
        [JsonPropertyName("event")]
        public EventData Event { get; set; }
    }

    public class EventData
    {
        [JsonPropertyName("_id")]
        public IdData Id { get; set; }

        [JsonPropertyName("clusterTime")]
        public ClusterTime ClusterTime { get; set; }

        [JsonPropertyName("documentKey")]
        public DocumentKey DocumentKey { get; set; }

        [JsonPropertyName("fullDocument")]
        public Dictionary<string, object> FullDocument { get; set; }

        [JsonPropertyName("ns")]
        public Namespace Ns { get; set; }

        [JsonPropertyName("operationType")]
```

```
        public string OperationType { get; set; }
    }

    public class IdData
    {
        [JsonPropertyName("_data")]
        public string Data { get; set; }
    }

    public class ClusterTime
    {
        [JsonPropertyName("$timestamp")]
        public Timestamp Timestamp { get; set; }
    }

    public class Timestamp
    {
        [JsonPropertyName("t")]
        public long T { get; set; }

        [JsonPropertyName("i")]
        public int I { get; set; }
    }

    public class DocumentKey
    {
        [JsonPropertyName("_id")]
        public Id Id { get; set; }
    }

    public class Id
    {
        [JsonPropertyName("$oid")]
        public string Oid { get; set; }
    }

    public class Namespace
    {
        [JsonPropertyName("db")]
        public string Db { get; set; }

        [JsonPropertyName("coll")]
        public string Coll { get; set; }
    }
```

```
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
```

```
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 JavaScript 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
```

```
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

使用 TypeScript 将 Amazon DocumentDB 事件与 Lambda 结合使用

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from 'aws-
lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 PHP 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
<?php

require __DIR__.'/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Handler;

class DocumentDBEventHandler implements Handler
{
    public function handle($event, Context $context): string
    {

        $events = $event['events'] ?? [];
        foreach ($events as $record) {
            $this->logDocumentDBEvent($record['event']);
        }
        return 'OK';
    }

    private function logDocumentDBEvent($event): void
    {
        // Extract information from the event record

        $operationType = $event['operationType'] ?? 'Unknown';
        $db = $event['ns']['db'] ?? 'Unknown';
        $collection = $event['ns']['coll'] ?? 'Unknown';
        $fullDocument = $event['fullDocument'] ?? [];

        // Log the event details

        echo "Operation type: $operationType\n";
```

```
        echo "Database: $db\n";
        echo "Collection: $collection\n";
        echo "Full document: " . json_encode($fullDocument, JSON_PRETTY_PRINT) .
 "\n";
    }
}
return new DocumentDBEventHandler();
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Ruby 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Rust 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::documentdb::{DocumentDbEvent, DocumentDbInnerEvent},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<DocumentDbEvent>) ->Result<(),
 Error> {

    tracing::info!("Event Source ARN: {:?}", event.payload.event_source_arn);
    tracing::info!("Event Source: {:?}", event.payload.event_source);

    let records = &event.payload.events;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_document_db_event(record);
```

```
    }

    tracing::info!("Document db records processed");

    // Prepare the response
    Ok(())

}

fn log_document_db_event(record: &DocumentDbInnerEvent)-> Result<(), Error>{
    tracing::info!("Change Event: {:?}", record.event);

    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

创建 Lambda 函数

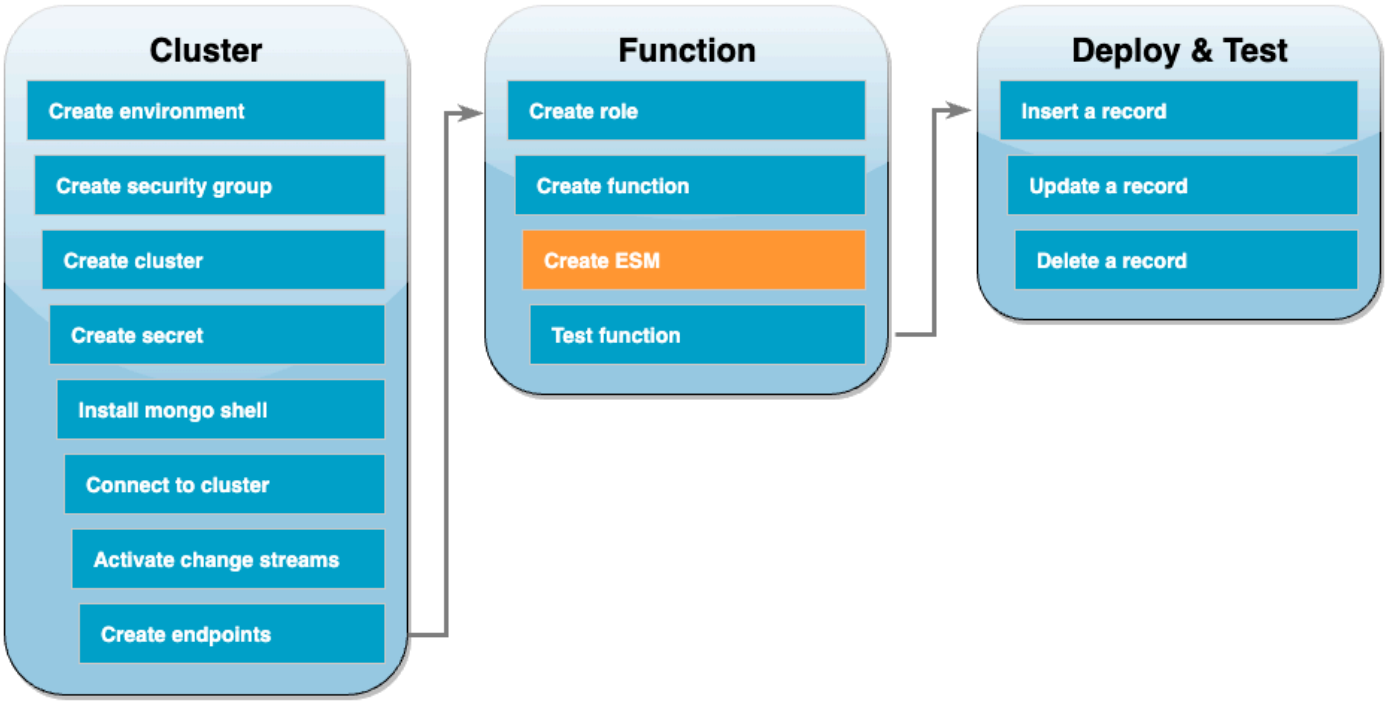
1. 将示例代码复制到名为 `index.js` 的文件中。
2. 使用以下命令创建部署包。

```
zip function.zip index.js
```

3. 使用以下 CLI 命令创建函数。将 `us-east-1` 替换为 AWS 区域，并将 `123456789012` 替换为账户 ID。

```
aws lambda create-function \
  --function-name ProcessDocumentDBRecords \
  --zip-file fileb://function.zip --handler index.handler --runtime nodejs20.x \
  --region us-east-1 \
  --role arn:aws:iam::123456789012:role/AWSDocumentDBLambdaExecutionRole
```

创建 Lambda 事件源映射



创建事件源映射，将 Amazon DocumentDB 更改流与 Lambda 函数相关联。创建此事件源映射后，AWS Lambda 即开始轮询该流。

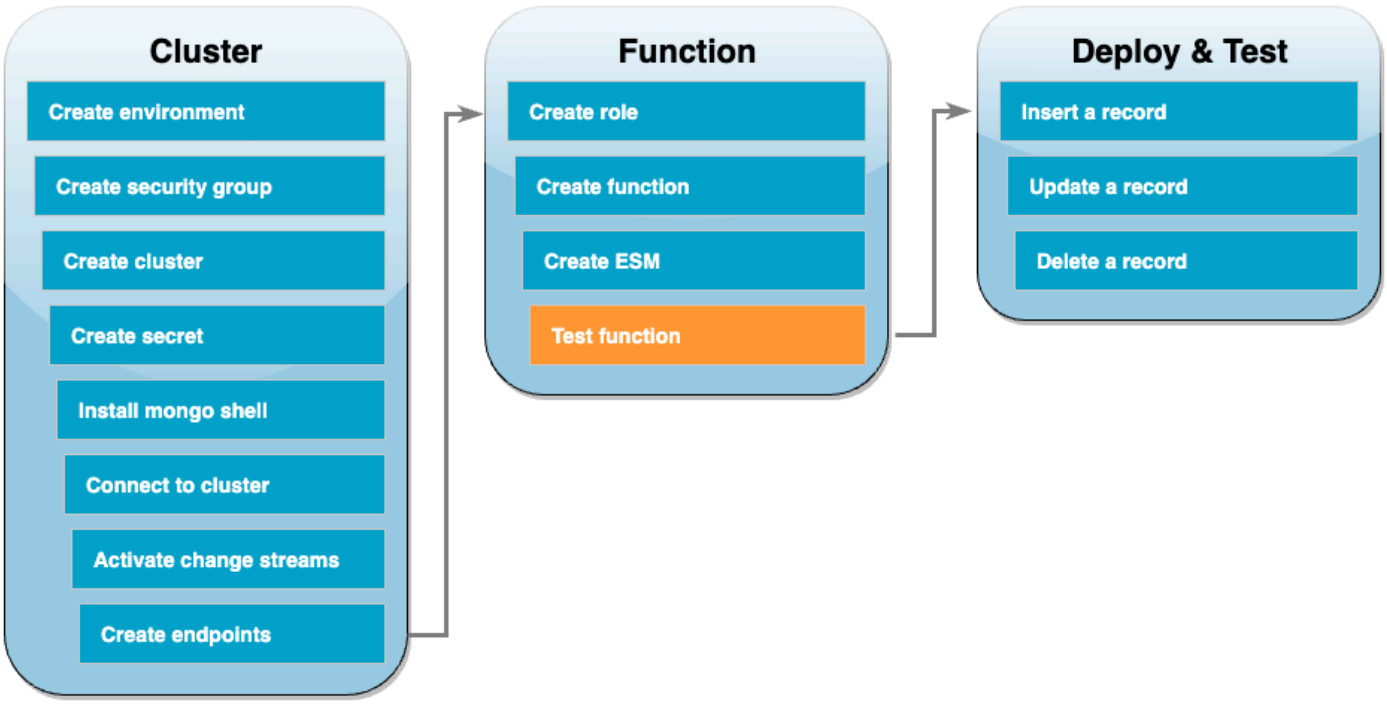
创建事件源映射

1. 在 Lambda 控制台中打开函数页面。
2. 选择您之前创建的 `ProcessDocumentDBRecords` 函数。
3. 选择配置选项卡，然后从左侧菜单中选择触发器。
4. 选择添加触发器。
5. 在触发器配置下，为源选择 Amazon DocumentDB。
6. 使用以下配置创建事件源映射：

- Amazon DocumentDB 集群 – 选择之前创建的集群。
- 数据库名称 – `docdbdemo`
- 集合名称 – 产品
- 批处理大小 – 1
- 起始位置 – 最新
- 身份验证 – BASIC_AUTH
- Secrets Manager 密钥 – 选择刚刚创建的 `DocumentDBSecret`。
- 批处理时段 – 1
- 完整文档配置 – UpdateLookup

7. 选择添加。创建事件源映射可能需要花费几分钟的时间。

测试函数 – 手动调用



要测试您是否正确创建函数和事件源映射，请使用 `invoke` 命令调用函数。为此，请先将以下事件 JSON 复制到名为 `input.txt` 的文件中：

```
{
```

```
  "eventSourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:canaryclusterb2a659a2-
qo5tcmqkcl03",
  "events": [
    {
      "event": {
        "_id": {
          "_data": "0163eeb6e7000000090100000009000041e1"
        },
        "clusterTime": {
          "$timestamp": {
            "t": 1676588775,
            "i": 9
          }
        },
        "documentKey": {
          "_id": {
            "$oid": "63eeb6e7d418cd98afb1c1d7"
          }
        },
        "fullDocument": {
          "_id": {
            "$oid": "63eeb6e7d418cd98afb1c1d7"
          },
          "anyField": "sampleValue"
        },
        "ns": {
          "db": "docdbdemo",
          "coll": "products"
        },
        "operationType": "insert"
      }
    }
  ],
  "eventSource": "aws:docdb"
}
```

然后，使用以下命令来调用包含此事件的函数：

```
aws lambda invoke \
  --function-name ProcessDocumentDBRecords \
  --cli-binary-format raw-in-base64-out \
  --region us-east-1 \
  --payload file://input.txt out.txt
```

您应该会看到如下响应：

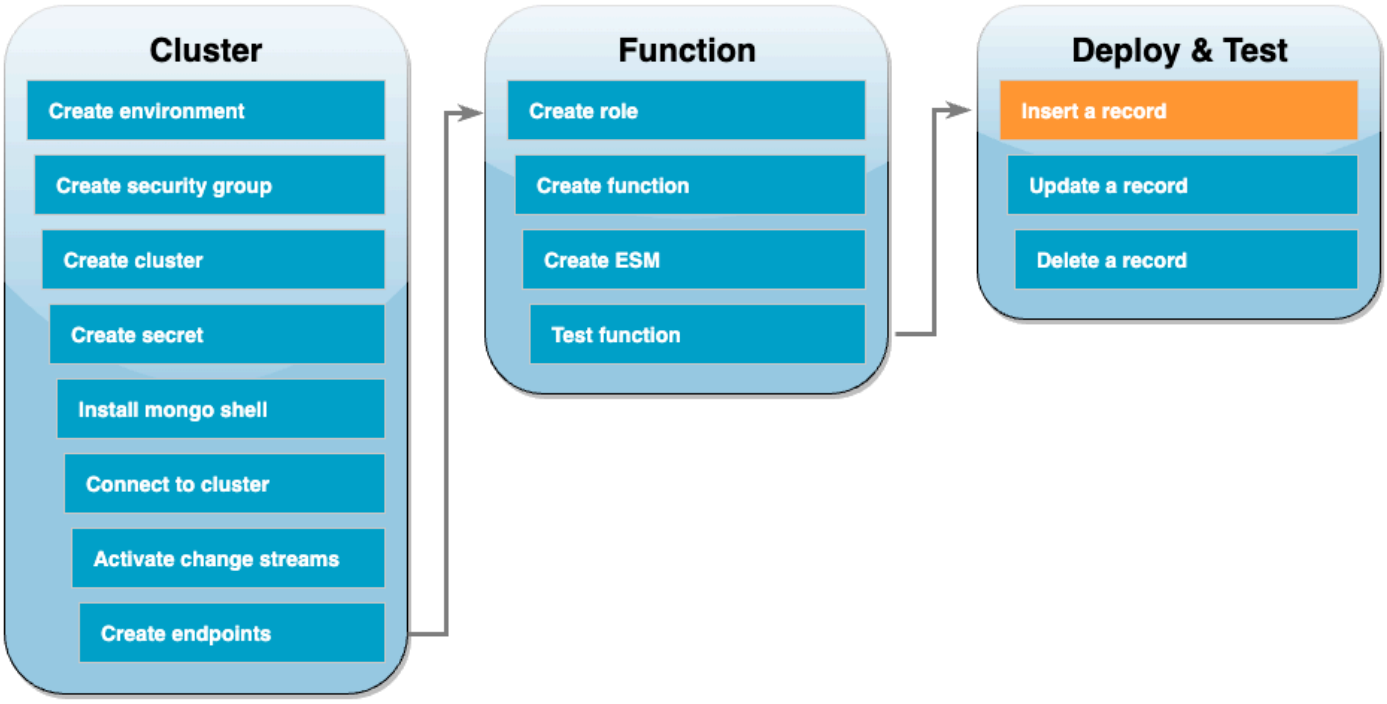
```
{
   "StatusCode": 200,
   "ExecutedVersion": "$LATEST"
}
```

您可以通过查看 CloudWatch Logs 来验证函数是否成功处理该事件。

通过 CloudWatch Logs 验证手动调用

1. 在 Lambda 控制台中打开函数页面。
2. 选择监控选项卡，然后选择查看 CloudWatch 日志。这会将您引导至 CloudWatch 控制台中与函数相关联的特定日志组。
3. 选择最新的日志流。在日志消息中，您应该会看到事件 JSON。

测试函数 – 插入记录



通过直接与 Amazon DocumentDB 数据库交互来测试端到端设置。在接下来的一组步骤中，您将插入记录，对其进行更新，然后将其删除。

插入记录

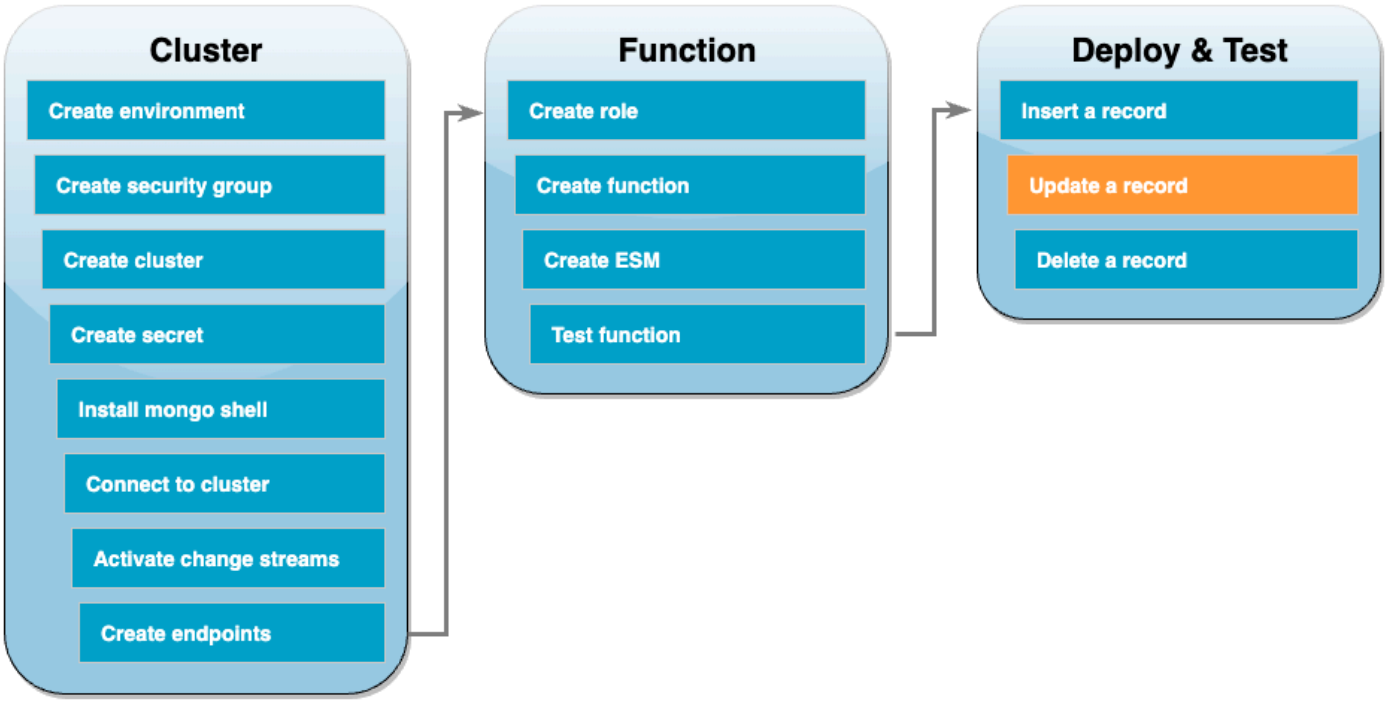
1. 在 AWS Cloud9 环境中重新连接到 Amazon DocumentDB 集群。
2. 使用此命令以确保您当前正在使用 docdbdemo 数据库：

```
use docdbdemo
```

3. 在 docdbdemo 数据库的 products 集合中插入记录：

```
db.products.insert({"name":"Pencil", "price": 1.00})
```

测试函数 – 更新记录

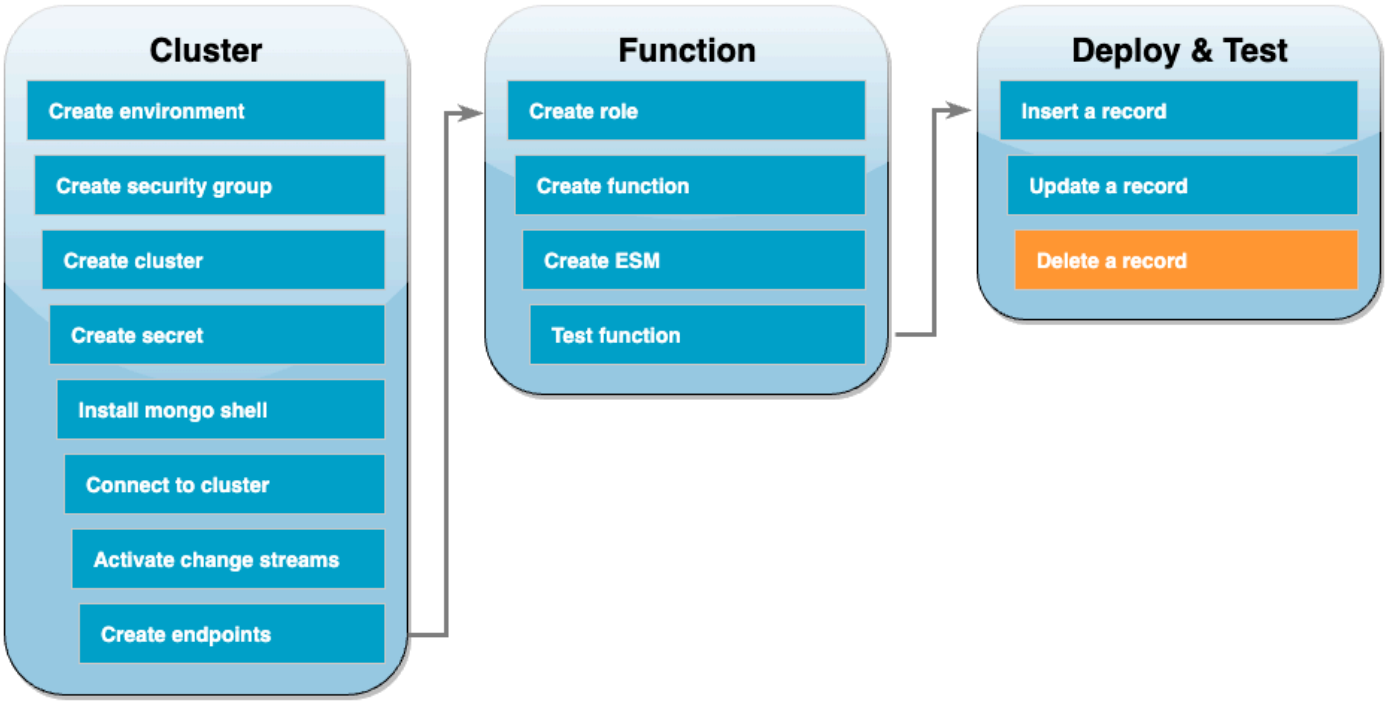


然后，使用以下命令更新您刚刚插入的记录：

```
db.products.update(
    { "name": "Pencil" },
    { $set: { "price": 0.50 }}
)
```

通过查看 CloudWatch Logs 来验证函数是否成功处理该事件。

测试函数 – 删除记录



最后，使用以下命令删除您刚刚更新的记录：

```
db.products.remove( { "name": "Pencil" } )
```

通过查看 CloudWatch Logs 来验证函数是否成功处理该事件。

清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择删除。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。
4. 在文本输入字段中输入角色名称，然后选择删除。

删除 VPC 端点

1. 打开 VPC 控制台。在左侧菜单的虚拟私有云下，选择端点。
2. 选择您创建的端点。
3. 选择 Actions（操作）、Delete VPC Endpoint（删除 VPC 端点）。
4. 在文本输入字段中输入 **`delete`**。
5. 选择删除。

删除 Amazon DocumentDB 集群

1. 打开 Amazon DocumentDB 控制台。
2. 选择您为本教程创建的 Amazon DocumentDB 集群，并禁用删除保护。
3. 在主集群页面中，再次选择 Amazon DocumentDB 集群。
4. 依次选择操作、删除。
5. 对于创建最终集群快照，选择否。
6. 在文本输入字段中输入 **`delete`**。
7. 选择删除。

在 Secrets Manager 中删除密钥

1. 打开 Secrets Manager 控制台。
2. 选择您为本教程创建的密钥。
3. 依次选择操作、删除密钥。
4. 选择计划删除。

删除 Amazon EC2 安全组

1. 打开 EC2 控制台。在网络与安全性下，选择安全组。
2. 选择您为本教程创建的安全组。
3. 依次选择操作、删除安全组。
4. 选择删除。

删除 AWS Cloud9 环境

1. 打开AWS Cloud9控制台。
2. 选择您为本教程创建的环境。
3. 选择删除。
4. 在文本输入字段中输入 **`delete`**。
5. 选择 Delete（删除）。

将 AWS Lambda 与 Amazon DynamoDB 结合使用

> **Note**
>
> 如果想要将数据发送到 Lambda 函数以外的目标，或要在发送数据之前丰富数据，请参阅 Amazon EventBridge Pipes（Amazon EventBridge 管道）。

您可以使用 AWS Lambda 函数来处理 Amazon DynamoDB Streams中的记录。使用 DynamoDB Streams，每次更新 DynamoDB 表时，您都可以触发 Lambda 函数以执行其他工作。

主题

轮询和批处理流

Lambda 以每秒 4 次的基本频率轮询 DynamoDB 流中的分区来获取记录。如果记录可用，Lambda 会调用函数并等待结果。如果处理成功，Lambda 会恢复轮询，直到其收到更多记录。

默认情况下，Lambda 会在记录可用时尽快调用您的函数。如果 Lambda 从事件源中读取的批处理只有一条记录，则 Lambda 将会只向该函数发送一条记录。为避免在记录数量较少的情况下调用该函数，您可以配置 batching window（批处理时段），让事件源缓冲最多五分钟的记录。调用函数

前，Lambda 会持续从事件源中读取记录，直到收集完整批处理、批处理时段到期或批处理达到 6MB 的有效负载时为止。有关更多信息，请参阅 批处理行为。

> **Warning**
>
> Lambda 事件源映射至少处理每个事件一次，有可能出现重复处理记录的情况。为避免与重复事件相关的潜在问题，我们强烈建议您将函数代码设为幂等性。要了解更多信息，请参阅 AWS 知识中心的如何使我的 Lambda 函数具有幂等性。

Lambda 在发送下次批处理之前不会等待任何配置的扩展完成。换句话说，扩展可能会在 Lambda 处理下一批记录时继续运行。如果您违反了账户的任何并发设置或限制，可能会导致节流问题。要检测这是否是潜在问题，请监控函数并检查所显示的并发指标是否高于事件源映射的预期。由于调用间隔时间较短，Lambda 可能会短暂报告高于分片数量的并发使用量。即使对于没有扩展名的 Lambda 函数也是如此。

配置 ParallelizationFactor 设置以同时使用多个 Lambda 调用处理 DynamoDB 流的一个分片。您可以指定 Lambda 通过从 1（默认值）到 10 的并行化因子从分区中轮询的并发批次数。例如，假设您将 `ParallelizationFactor` 设置为 2，则最多可以有 200 个并发 Lambda 调用来处理 100 个 DynamoDB 流分片（但您可能实际上会看到不同的 `ConcurrentExecutions` 指标值）。这有助于在数据量不稳定并且 `IteratorAge` 较高时纵向扩展处理吞吐量。增加每个分片的并发批次数后，Lambda 仍然可以确保项目（分区和排序键）级别的顺序处理。

轮询和流的起始位置

请注意，事件源映射创建和更新期间的流轮询最终是一致的。

- 在事件源映射创建期间，可能需要几分钟才能开始轮询来自流的事件。
- 在事件源映射更新期间，可能需要几分钟才能停止和重新开始轮询来自流的事件。

此行为意味着，如果你指定 `LATEST` 作为流的起始位置，事件源映射可能会在创建或更新期间错过事件。为确保不会错过任何事件，请将流的起始位置指定为 `TRIM_HORIZON`。

DynamoDB Streams 中的分片同时读取器

对于作为非全局表的单区域表，您可以设计最多两个 Lambda 函数来同时从同一个 DynamoDB Streams 分片读取数据。超过此限制会导致请求被拒。对于全局表，我们建议您将并行函数的数量限制为一个，以避免请求节流。

示例事件

Example


```
{
  "Records": [
    {
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-west-2",
      "eventName": "INSERT",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "2",
      "eventVersion": "1.0",
      "dynamodb": {
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
```

```
          }
        },
        "SequenceNumber": "222",
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 59,
        "NewImage": {
          "Message": {
            "S": "This item has changed"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "awsRegion": "us-west-2",
      "eventName": "MODIFY",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    }
  ]}
```

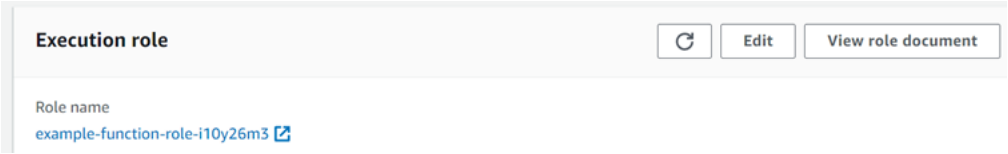
使用 Lambda 处理 DynamoDB 记录

创建事件源映射以指示 Lambda 将流中的记录发送到 Lambda 函数。您可以创建多个事件源映射，以使用多个 Lambda 函数处理相同的数据，或使用单个函数处理来自多个流的项目。

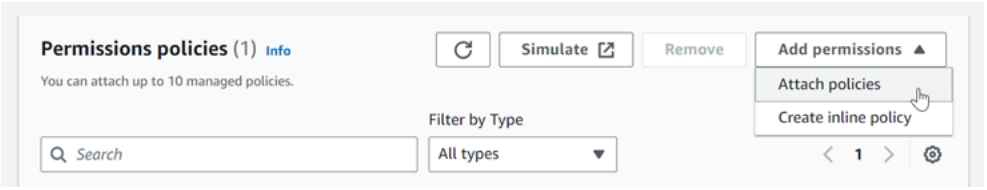
要将函数配置为从 DynamoDB Streams 中读取，请将 AWSLambdaDynamoDBExecutionRole AWS 托管策略附加到执行角色，然后创建 DynamoDB 触发器。

要添加权限并创建触发器

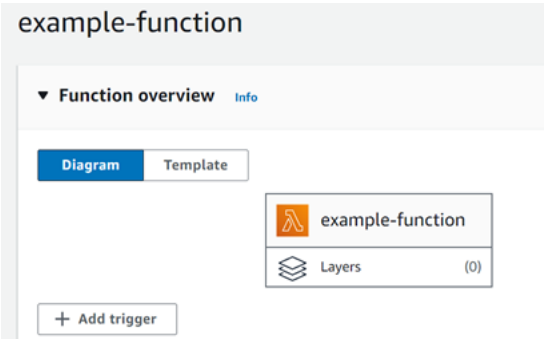
1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数的名称。
3. 选择 Configuration（配置）选项卡，然后选择 Permissions（权限）。
4. 在角色名称下，选择至执行角色的链接。此角色将在 IAM 控制台中打开角色。



5. 选择添加权限，然后选择附加策略。



6. 在搜索字段中输入 AWSLambdaDynamoDBExecutionRole。向执行角色添加此策略。这是一项 AWS 托管策略，其中包含您的函数从 DynamoDB 流中读取所需的权限。有关此策略的更多信息，请参阅《AWS Managed Policy Reference》中的 AWSLambdaDynamoDBExecutionRole。
7. 在 Lambda 控制台中返回您的函数。在 Function overview（函数概览）下，选择 Add trigger（添加触发器）。



8. 选择触发器类型。
9. 配置必填选项，然后选择 Add（添加）。

Lambda 支持 DynamoDB 事件源的以下选项：

事件源选项

- DynamoDB table（DynamoDB 表）– 要从中读取记录的 DynamoDB 表。
- Batch size（批处理大小）– 每个批次中要发送到函数的记录数，最高为 10000。Lambda 通过单个调用将批处理中的所有记录传递给函数，前提是事件的总大小未超出同步调用的有效负载限制 (6 MB)。
- Batch window（批处理时段）– 指定在调用函数之前收集记录的最长时间（以秒为单位）。
- Starting position（开始位置）– 仅处理新记录或所有现有记录。
  - Latest（最新）– 处理已添加到流中的新记录。

- Trim horizon（时间范围）– 处理流中的所有记录。

在处理任何现有记录后，函数将继续处理新记录。

- 失败时的目标 – 无法处理的记录的标准 SQS 队列或标准 SNS 主题。当 Lambda 因为某批记录太旧或已用尽所有重试而将其丢弃时，Lambda 会将有关该批处理的详细信息发送到该队列或主题。
- Retry attempts（重试次数）– 函数返回错误时 Lambda 重试的最大次数。这不适用于批处理未到达函数的服务错误或限制。
- Maximum age of record（记录的最长时限）– Lambda 发送到您的函数的记录的最长期限。
- Split batch on error（出现错误时拆分批）– 当函数返回错误时，在重试之前将批次拆分为两批。原始批量大小设置会保持不变。
- Concurrent batches per shard（每个分片的并发批处理数）– 同时处理来自同一个分片的多个批处理。
- Enabled（已启用）– 设置为 true 可启用事件源映射。设置为 false 可停止处理记录。Lambda 会跟踪已处理的最后一条记录，并在重新启用映射后从停止位置重新开始处理。

> **Note**
>
> 对于 Lambda 作为 DynamoDB 触发器的一部分而调用的 GetRecords API 调用，您无需付费。

之后，要管理事件源配置，请在设计器中选择触发器。

使用 DynamoDB 和 Lambda 配置部分批处理响应

在使用和处理来自事件源的流式数据时，默认情况下，Lambda 仅在批处理完全成功时，才会在批次的最高序列号处设置检查点。Lambda 会将所有其他结果视为完全失败并重试批处理，直至达到重试次数上限。要允许在处理来自流的批次时部分成功，请开启 `ReportBatchItemFailures`。允许部分成功有助于减少对记录重试的次数，尽管这并不能完全阻止在成功记录中重试的可能性。

要开启 `ReportBatchItemFailures`，请在 FunctionResponseTypes 列表中包含枚举值 **`ReportBatchItemFailures`**。此列表指示为函数启用了哪些响应类型。您可以在创建或更新事件源映射时配置此列表。

报告语法

配置批处理项目失败的报告时，将返回 StreamsEventResponse 类，其中包含批处理项目失败列表。您可以使用 StreamsEventResponse 对象返回批处理中第一个失败记录的序列号。您还可以使用正确的响应语法来创建自己的自定义类。以下 JSON 结构显示了所需的响应语法：

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> 如果 batchItemFailures 数组包含多个项目，Lambda 会使用序列号最小的记录作为检查点。然后，Lambda 会重试从该检查点开始的所有记录。

成功和失败的条件

如果返回以下任意一项，则 Lambda 会将批处理视为完全成功：

- 空的 batchItemFailure 列表
- Null batchItemFailure 列表
- 空的 EventResponse
- Null EventResponse

如果返回以下任何一项，则 Lambda 会将批处理视为完全失败：

- 空字符串 itemIdentifier
- Null itemIdentifier
- 包含错误密钥名的 itemIdentifier

Lambda 会根据您的重试策略在失败时重试。

将批次一分为二

如果调用失败并且已开启 BisectBatchOnFunctionError，则无论您的 ReportBatchItemFailures 设置如何，批次都将一分为二。

当收到批处理部分成功响应且同时开启 BisectBatchOnFunctionError 和 ReportBatchItemFailures 时，批次将在返回的序列号处一分为二，并且 Lambda 将仅重试剩余记录。

以下函数代码示例将返回批处理中处理失败消息的 ID 列表：

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 .NET 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)
```

```
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Go 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
```

```
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Java 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 Serializable> {

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
```

```
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 JavaScript 报告 Lambda 的 DynamoDB 批处理项目失败。

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
```

```
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

使用 TypeScript 报告 Lambda 的 DynamoDB 批处理项目失败。

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }

  return { batchItemFailures: batchItemFailures };
};
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 PHP 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
```

```
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Python 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Ruby 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
```

```
    end

    {"batchItemFailures" => []}
  end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Rust 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;
```

```
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();
```

```
    run(service_fn(function_handler)).await
}
```

在 Lambda 中保留 DynamoDB 事件源的丢弃记录

DynamoDB 事件源映射的错误处理取决于错误是在调用函数之前还是在函数调用期间发生的：

- 调用前：如果 Lambda 事件源映射由于节流或其他问题而无法调用该函数，则它会一直重试，直到记录过期或超过事件源映射上配置的最大期限（MaximumRecordAgeInSeconds）。
- 调用期间：如果调用函数但返回错误，Lambda 会重试，直到记录过期、超过最大期限（MaximumRecordAgeInSeconds）或达到配置的重试配额（MaximumRetryAttempts）。对于函数错误，您还可以配置 BisectBatchOnFunctionError，将失败的批次拆分为两个较小的批次，从而隔离错误记录并避免超时。拆分批次不会消耗重试配额。

如果错误处理措施失败，Lambda 将丢弃记录并继续处理数据流中的批次。使用默认设置时，这意味着错误的记录可能会阻止受影响的分区上的处理，时间长达一天。为了避免这种情况，请配置函数的事件源映射，使用合理的重试次数和适合您的使用案例的最长记录期限。

配置失败调用的目标

要保留失败的事件源映射调用的记录，请在函数的事件源映射中添加一个目标。发送到目标的每条记录都是一个 JSON 文档，其中包含有关失败调用的元数据。您可以将任何 Amazon SNS 主题或 Amazon SQS 队列配置为目标。您的执行角色必须具有目标的权限：

- 对于 SQS 目标：sqs:SendMessage
- 对于 SNS 目标：sns:Publish

要使用控制台配置失败时的目标，请执行以下步骤：

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 在 Function overview (函数概览) 下，选择 Add destination (添加目标)。
4. 对于源，请选择事件源映射调用。
5. 对于事件源映射，请选择为此函数配置的事件源。

6. 在条件中，选择失败时。对于事件源映射调用，这是唯一可接受的条件。
7. 对于目标类型，请选择 Lambda 要发送调用记录的目标类型。
8. 对于 Destination (目标)，请选择一个资源。
9. 选择保存。

您还可以使用 AWS Command Line Interface（AWS CLI）配置失败时的目标。例如，以 create-event-source-mapping 命令将带有 SQS 失败时目标的事件源映射添加到 `MyFunction`：

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

以下 update-event-source-mapping 命令更新事件源映射，以在两次重试之后或记录超过一小时后将失败的调用记录发送到 SNS 目标。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

更新的设置是异步应用的，并且直到该过程完成才反映在输出中。使用 get-event-source-mapping 命令查看当前状态。

要移除目标，请提供一个空字符串作为 `destination-config` 参数的实际参数：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

以下示例显示了 DynamoDB 流的调用记录。

Example 调用记录

```
{
    "requestContext": {
```

```
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "800000000003126276362",
        "endSequenceNumber": "800000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    }
}
```

您可以使用此信息从流中检索受影响的记录以进行故障排除。实际记录不包括在内，因此您必须处理此记录并在记录过期和丢失之前从流中检索它们。

使用 CloudWatch Logs 进行监控

在您的函数处理完一批记录后，Lambda 将发出 `IteratorAge` 指标。该指标指示处理完成时批处理中最后一条记录的时间。如果您的函数正在处理新事件，则可使用迭代器期限来估算新记录的添加时间与函数处理新记录的时间之间的延迟。

迭代器期限中的上升趋势可以指示您的函数问题。有关更多信息，请参阅 查看 Lambda 函数的指标。

在 Lambda 中实现有状态的 DynamoDB 流处理

Lambda 函数可以运行连续流处理应用程序。流表示通过您的应用程序持续流动的无边界数据。要分析这种不断更新的输入中的信息，可以使用按时间定义的窗口来限制包含的记录。

滚动窗口是定期打开和关闭的不同窗口。预设情况下，Lambda 调用是无状态的，在没有外部数据库的情况下，无法使用它们跨多次连续调用处理数据。但是，有了滚动窗口后，您可以在不同调用中保持状

态。此状态包含之前为当前窗口处理的消息的汇总结果。您的状态最多可以是每个分片 1MB。如果超过该大小，Lambda 将提前终止窗口。

流中的每条记录都属于特定窗口。Lambda 将至少处理每条记录一次，但不保证每条记录只处理一次。在极少数情况下（例如错误处理），某些记录可能会被多次处理。第一次处理记录时始终按顺序处理。如果多次处理记录，则可能会不按顺序处理。

聚合和处理

系统将调用您的用户托管函数以便聚合和处理该聚合的最终结果。Lambda 汇总在该窗口中接收的所有记录。您可以分多个批次接收这些记录，每个批次都作为单独的调用。每次调用都会收到一个状态。因此，当使用滚动窗口时，Lambda 函数响应必须包含 `state` 属性。如果响应不包含 `state` 属性，Lambda 会将其视作失败的调用。为了满足该条件，您的函数可以返回一个具有以下 JSON 形状的 `TimeWindowEventResponse` 对象：

Example **`TimeWindowEventResponse`** 值

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> 对于 Java 函数，我们建议使用 `Map<String, String>` 来表示状态。

在窗口末尾，标志 `isFinalInvokeForWindow` 被设置 `true`，以表示这是最终状态，并且已准备好进行处理。处理完成后，窗口完成，最终调用完成，然后状态将被删除。

在窗口结束时，Lambda 会对针对聚合结果的操作应用最终处理。您的最终处理将同步调用。成功调用后，函数会检查序列号并继续进行流处理。如果调用失败，则您的 Lambda 函数将暂停进一步处理，直到成功调用为止。

Example DynamodbTimeWindowEvent

```
{
```

```
    "Records":[
       {
          "eventID":"1",
          "eventName":"INSERT",
          "eventVersion":"1.0",
          "eventSource":"aws:dynamodb",
          "awsRegion":"us-east-1",
          "dynamodb":{
             "Keys":{
                "Id":{
                   "N":"101"
                }
             },
             "NewImage":{
                "Message":{
                   "S":"New item!"
                },
                "Id":{
                   "N":"101"
                }
             },
             "SequenceNumber":"111",
             "SizeBytes":26,
             "StreamViewType":"NEW_AND_OLD_IMAGES"
          },
          "eventSourceARN":"stream-ARN"
       },
       {
          "eventID":"2",
          "eventName":"MODIFY",
          "eventVersion":"1.0",
          "eventSource":"aws:dynamodb",
          "awsRegion":"us-east-1",
          "dynamodb":{
             "Keys":{
                "Id":{
                   "N":"101"
                }
             },
             "NewImage":{
                "Message":{
                   "S":"This item has changed"
                },
                "Id":{
```

```
                "N":"101"
             }
          },
          "OldImage":{
             "Message":{
                "S":"New item!"
             },
             "Id":{
                "N":"101"
             }
          },
          "SequenceNumber":"222",
          "SizeBytes":59,
          "StreamViewType":"NEW_AND_OLD_IMAGES"
       },
       "eventSourceARN":"stream-ARN"
    },
    {
       "eventID":"3",
       "eventName":"REMOVE",
       "eventVersion":"1.0",
       "eventSource":"aws:dynamodb",
       "awsRegion":"us-east-1",
       "dynamodb":{
          "Keys":{
             "Id":{
                "N":"101"
             }
          },
          "OldImage":{
             "Message":{
                "S":"This item has changed"
             },
             "Id":{
                "N":"101"
             }
          },
          "SequenceNumber":"333",
          "SizeBytes":38,
          "StreamViewType":"NEW_AND_OLD_IMAGES"
       },
       "eventSourceARN":"stream-ARN"
    }
 ],
```

```
    "window": {
        "start": "2020-07-30T17:00:00Z",
        "end": "2020-07-30T17:05:00Z"
    },
    "state": {
        "1": "state1"
    },
    "shardId": "shard123456789",
    "eventSourceARN": "stream-ARN",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

配置

您可以在创建或更新事件源映射时配置滚动窗口。要配置翻转窗口，请以秒为单位进行指定（TumblingWindowInSeconds）。以下示例 AWS Command Line Interface (AWS CLI) 命令会创建一个滚动窗口为 120 秒的流式事件源映射。为聚合和处理定义的 Lambda 函数被命名为 `tumbling-window-example-function`。

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda 根据记录插入到流的时间来确定滚动窗口的边界。所有记录都有一个大致的时间戳，供 Lambda 在确定边界时使用。

滚动窗口聚合不支持重新分片。当分区结束时，Lambda 会认为窗口已关闭，子分区将以全新的状态启动自己的窗口。

滚动窗口完全支持现有的重试策略 `maxRetryAttempts` 和 `maxRecordAge`。

Example Handler.py – 聚合和处理

以下 Python 函数演示了如何聚合然后处理您的最终状态：

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
```

```
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['dynamodb']['NewImage']['Id']] = state.get(record['dynamodb']
['NewImage']['Id'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Amazon DynamoDB 事件源映射的 Lambda 参数

所有 Lambda 事件源类型共享相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，只有部分参数适用于 DynamoDB Streams。

参数	必需	默认值	备注
BatchSize	否	100	最大值：10000
BisectBatchOnFunctionError	否	false	none
DestinationConfig	否	不适用	丢弃的记录的标准 Amazon SQS 队列或标准 Amazon SNS 主题目标。
启用	否	真实	none

参数	必需	默认值	备注
EventSourceArn	Y	不适用	数据流或流使用者的 ARN
FilterCriteria	否	不适用	控制 Lambda 向您的函数发送的事件
FunctionName	是	不适用	none
FunctionResponseTypes	否	不适用	要使您的函数报告某个批处理中的特定失败，请在 FunctionResponseTypes 中包含值 ReportBatchItemFailures 。有关更多信息，请参阅 使用 DynamoDB 和 Lambda 配置部分批处理响应。
MaximumBatchingWindowInSeconds	否	0	none
MaximumRecordAgeInSeconds	否	–1	-1 表示无限：会一直重试失败的记录，直到记录过期。DynamoDB Streams 的数据留存期限为 24 小时。 最小值：-1 最大值：604800

参数	必需	默认值	备注
MaximumRetryAttempts	否	–1	-1 表示无限：会一直重试失败的记录，直到记录过期。 最小值：0 最大值：10000
ParallelizationFactor	否	1	最大值：10
StartingPosition	Y	不适用	TRIM_HORIZON 或 LATEST（最新）
TumblingWindowInSeconds	否	不适用	最小值：0 最大值：900

对 DynamoDB 事件源使用事件筛选

您可以使用事件筛选，控制 Lambda 将流或队列中的哪些记录发送给函数。有关事件筛选工作原理的一般信息，请参阅 the section called “事件筛选”。

本节重点介绍 DynamoDB 事件源的事件筛选。

主题

- DynamoDB 事件
- 使用表格属性进行筛选
- 使用布尔表达式进行筛选
- 使用 Exists 运算符
- 用于 DynamoDB 筛选的 JSON 格式

DynamoDB 事件

假设您有一个 DynamoDB 表，其中包含主键 `CustomerName`、属性 `AccountManager` 和 `PaymentTerms`。下面显示了来自 DynamoDB 表流的示例记录。

```
{
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
          "ApproximateCreationDateTime": "1678831218.0",
          "Keys": {
              "CustomerName": {
                  "S": "AnyCompany Industries"
              },
              "NewImage": {
                  "AccountManager": {
                      "S": "Pat Candella"
                  },
                  "PaymentTerms": {
                      "S": "60 days"
                  },
                  "CustomerName": {
                      "S": "AnyCompany Industries"
                  }
              },
              "SequenceNumber": "111",
              "SizeBytes": 26,
              "StreamViewType": "NEW_IMAGE"
          }
      }
  }
```

要根据 DynamoDB 表中的键和属性值进行筛选，请使用记录中的 dynamodb 键。以下部分提供了不同筛选条件类型的示例。

使用表键进行筛选

假设您希望函数仅处理主键 CustomerName 为“AnyCompany Industries”的记录。FilterCriteria 对象将如下所示。

```
{
     "Filters": [
          {
              "Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" :
 [ \"AnyCompany Industries\" ] } } } }"
          }
      ]
```

```
 }
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 Pattern 的值。

```
{
     "dynamodb": {
          "Keys": {
              "CustomerName": {
                  "S": [ "AnyCompany Industries" ]
                  }
              }
          }
 }
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
   Filters:
     - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }'
```

使用表格属性进行筛选

借助 DynamoDB，您还可以使用 `NewImage` 和 `OldImage` 键来筛选属性值。假设您要筛选最新表格图像中 `AccountManager` 属性为“Pat Candella”或“Shirley Rodriguez”的记录。`FilterCriteria` 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\", \"Shirley Rodriguez\" ] } } } }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 `Pattern` 的值。

```
{
    "dynamodb": {
        "NewImage": {
            "AccountManager": {
                "S": [ "Pat Candella", "Shirley Rodriguez" ]
            }
        }
    }
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella",
 "Shirley Rodriguez" ] } } } }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella", "Shirley Rodriguez" ] } } } }'
```

使用布尔表达式进行筛选

您也可以使用布尔 AND 表达式创建筛选器。这些表达式可能同时包含表的键和属性参数。假设您想要筛选 AccountManager 的 NewImage 值为“Pat Candella”且 OldImage 值为“Terry Whitlock”的记录。FilterCriteria 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" : { \"OldImage\" : { \"AccountManager
\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 Pattern 的值。

```
{
    "dynamodb" : {
        "NewImage" : {
            "AccountManager" : {
                "S" : [
                    "Pat Candella"
                ]
            }
        }
    },
    "dynamodb": {
        "OldImage": {
            "AccountManager": {
                "S": [
                    "Terry Whitlock"
                ]
            }
        }
    }
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }'
```

> **Note**
>
> DynamoDB 事件筛选不支持使用数字运算符（数字相等和数字范围）。即使表中的项目存储为数字，这些参数也会转换为 JSON 记录对象中的字符串。

使用 Exists 运算符

鉴于 DynamoDB 中 JSON 事件对象的结构方式，使用 Exists 运算符需要特别小心。Exists 运算符仅适用于事件 JSON 中的叶节点，若筛选条件模式使用 Exists 来测试中间节点，则不会起作用。请考虑以下 DynamoDB 表项目：

```
{
  "UserID": {"S": "12345"},
  "Name": {"S": "John Doe"},
  "Organizations": {"L": [
      {"S":"Sales"},
      {"S":"Marketing"},
      {"S":"Support"}
    ]
  }
}
```

您可能需要创建如下所示的筛选条件模式来测试包含 `"Organizations"` 的事件：

```
{ "dynamodb" : { "NewImage" : { "Organizations" : [ { "exists": true } ] } } }
```

不过，此筛选条件模式永远不会返回匹配项，因为 `"Organizations"` 不是叶节点。以下示例展示了如何正确使用 Exists 运算符来构造所需的筛选条件模式：

```
{ "dynamodb" : { "NewImage" : {"Organizations": {"L": {"S": [ {"exists":
 true } ] } } } } }
```

用于 DynamoDB 筛选的 JSON 格式

要正确筛选 DynamoDB 源中的事件，数据字段及其筛选条件 (dynamodb) 都必须为有效的 JSON 格式。如果任一字段不为有效的 JSON 格式，Lambda 将会丢弃消息或引发异常。下表汇总了具体行为：

传入数据格式	数据属性中的筛选条件模式格式	导致的操作
有效 JSON	有效 JSON	Lambda 根据您的筛选条件进行筛选。
有效 JSON	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	非 JSON	Lambda 在事件源映射创建或更新时引发异常。数据属性的筛选条件模式必须为有效的 JSON 格式。
非 JSON	有效 JSON	Lambda 将丢弃记录。
非 JSON	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
非 JSON	非 JSON	Lambda 在事件源映射创建或更新时引发异常。数据属性的筛选条件模式必须为有效的 JSON 格式。

教程：将 AWS Lambda 与 Amazon DynamoDB Streams 结合使用

在本教程中，您将创建 Lambda 函数处理来自 Amazon DynamoDB Streams 的事件。

先决条件

本教程假设您对 Lambda 基本操作和 Lambda 控制台有一定了解。如果您还没有了解，请按照 使用控制台创建 Lambda 函数 中的说明创建您的第一个 Lambda 函数。

要完成以下步骤，您需要 AWS CLI 版本 2。在单独的数据块中列出了命令和预期输出：

```
aws --version
```

您应看到以下输出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

对于长命令，使用转义字符 (\) 将命令拆分为多行。

在 Linux 和 macOS 中，可使用您首选的 shell 和程序包管理器。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常与 Lambda 一起使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。本指南中的示例 CLI 命令使用 Linux 格式。如果您使用的是 Windows CLI，则必须重新格式化包含内联 JSON 文档的命令。

创建执行角色

创建执行角色，向您的函数授予访问 AWS 资源的权限。

创建执行角色

1. 在 IAM 控制台中，打开 Roles（角色）页面。
2. 选择创建角色。
3. 创建具有以下属性的角色。
   - Trusted entity（可信任的实体）– Lambda。
   - Permissions（权限）– AWSLambdaDynamoDBExecutionRole。
   - Role name（角色名称）– **`lambda-dynamodb-role`**。

AWSLambdaDynamoDBExecutionRole 具有该函数所需的权限以从 DynamoDB 中读取项目并将日志写入 CloudWatch Logs。

创建函数

创建一个 Lambda 函数来处理 DynamoDB 事件。函数代码会将一些传入的事件数据写入 CloudWatch Logs。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 .NET 将 DynamoDB 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
```

```
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 DynamoDB 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
```

```
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Java 将 DynamoDB 事件与 Lambda 结合使用。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
```

```
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 JavaScript 将 DynamoDB 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

使用 TypeScript 将 DynamoDB 事件与 Lambda 结合使用。

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
```

```
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 PHP 将 DynamoDB 事件与 Lambda 结合使用。

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
```

```
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Python 将 DynamoDB 事件与 Lambda 结合使用。

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 DynamoDB 事件与 Lambda 结合使用。

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
```

```
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Rust 将 DynamoDB 事件与 Lambda 结合使用。

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }
```

```
    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

创建函数

1. 将示例代码复制到名为 example.js 的文件中。
2. 创建部署程序包。

```
zip function.zip example.js
```

3. 使用 `create-function` 命令创建 Lambda 函数。

```
aws lambda create-function --function-name ProcessDynamoDBRecords \
    --zip-file fileb://function.zip --handler example.handler --runtime nodejs18.x
 \
    --role arn:aws:iam::111122223333:role/lambda-dynamodb-role
```

测试 Lambda 函数

在本步骤中，您将使用 `invoke` AWS Lambda CLI 命令和以下示例 DynamoDB 事件手动调用 Lambda 函数。将以下内容复制到名为 `input.txt` 的文件中。

Example input.txt

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
```

```
    {
       "eventID":"2",
       "eventName":"MODIFY",
       "eventVersion":"1.0",
       "eventSource":"aws:dynamodb",
       "awsRegion":"us-east-1",
       "dynamodb":{
          "Keys":{
             "Id":{
                "N":"101"
             }
          },
          "NewImage":{
             "Message":{
                "S":"This item has changed"
             },
             "Id":{
                "N":"101"
             }
          },
          "OldImage":{
             "Message":{
                "S":"New item!"
             },
             "Id":{
                "N":"101"
             }
          },
          "SequenceNumber":"222",
          "SizeBytes":59,
          "StreamViewType":"NEW_AND_OLD_IMAGES"
       },
       "eventSourceARN":"stream-ARN"
    },
    {
       "eventID":"3",
       "eventName":"REMOVE",
       "eventVersion":"1.0",
       "eventSource":"aws:dynamodb",
       "awsRegion":"us-east-1",
       "dynamodb":{
          "Keys":{
             "Id":{
                "N":"101"
```

```
                }
            },
            "OldImage":{
                "Message":{
                    "S":"This item has changed"
                },
                "Id":{
                    "N":"101"
                }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ]
}
```

运行以下 invoke 命令：

```
aws lambda invoke --function-name ProcessDynamoDBRecords \
    --cli-binary-format raw-in-base64-out \
    --payload file://input.txt outputfile.txt
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

函数在响应正文中返回字符串 message。

在 outputfile.txt 文件中验证输出。

创建启用流的 DynamoDB 表

创建启用流的 Amazon DynamoDB 表。

创建 DynamoDB 表

1. 打开 DynamoDB 控制台。
2. 选择 Create Table。
3. 使用以下设置创建表。

- Table name（表名称）- **lambda-dynamodb-stream**
- Primary key（主键）– **id**（字符串）

4. 选择创建。

启用流

1. 打开 DynamoDB 控制台。
2. 选择表。
3. 选择 lambda-dynamodb-stream 表。
4. 在 Exports and streams（导出和流）下，选择 DynamoDB stream details（DynamoDB 流详细信息）。
5. 选择打开。
6. 对于视图类型，选择仅键属性。
7. 选择开启流。

记下流 ARN。在下一步中将该流与 Lambda 函数关联时，您将需要此类信息。有关启用流的更多信息，请参阅使用 DynamoDB Streams 捕获表活动。

在 AWS Lambda 中添加事件源

在 AWS Lambda 中创建事件源映射。此事件源映射将 DynamoDB Streams 与 Lambda 函数关联。创建此事件源映射后，AWS Lambda 即开始轮询该流。

运行以下 AWS CLI `create-event-source-mapping` 命令。命令运行后，记下 UUID。在任何命令中，如删除事件源映射时，您都需要该 UUID 来引用事件源映射。

```
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
    --batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

这会在指定的 DynamoDB Streams 和 Lambda 函数之间创建映射。您可将一个 DynamoDB Streams 关联到多个 Lambda 函数，也可将同一个 Lambda 函数关联到多个流。但是，Lambda 函数将共享其所共享的流的读取吞吐量。

您可以通过运行以下命令获取事件源映射的列表。

```
aws lambda list-event-source-mappings
```

该列表返回您创建的所有事件源映射，而对于每个映射，它都显示 `LastProcessingResult` 等信息。该字段用于在出现任何问题时提供信息性消息。`No records processed`（指示 AWS Lambda 未开始轮询或流中没有任何记录）和 `OK`（指示 AWS Lambda 已成功读取流中的记录并已调用 Lambda 函数）等值表示未出现任何问题。如果出现问题，您将收到一条错误消息。

如果您有大量事件源映射，请使用函数名称参数缩窄结果范围。

```
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

测试设置

测试端到端体验。当您更新表时，DynamoDB 会将事件记录写入流。当 AWS Lambda 轮询该流时，它将在流中检测新记录并通过向该函数传递事件来代表您调用 Lambda 函数。

1. 在 DynamoDB 控制台中，添加、更新、删除表中的项目。DynamoDB 会将这些操作记录写入流。
2. AWS Lambda 会轮询该流，当检测到流有更新时，它会通过传递在流中发现的事件数据来调用 Lambda 函数。
3. 函数运行并在 Amazon CloudWatch 中创建日志。您可以验证 Amazon CloudWatch 控制台中报告的日志。

清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择删除。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。

4. 在文本输入字段中输入角色名称，然后选择删除。

删除 DynamoDB 表

1. 打开 DynamoDB 控制台中 Tables page（表页面）。
2. 选择您创建的表。
3. 选择删除。
4. 在文本框中输入 **delete**。
5. 选择 删除表。

使用 Lambda 函数处理 Amazon EC2 生命周期事件

您可以使用 AWS Lambda 处理来自 Amazon Elastic Compute Cloud 的生命周期事件并管理 Amazon EC2 资源。Amazon EC2 向 Amazon EventBridge（CloudWatch Events）发送生命周期事件，例如，当实例更改状态时、当 Amazon Elastic Block Store 卷快照完成时或当计划终止竞价型实例时。您可以配置 EventBridge (CloudWatch Events) 以将这些事件转发到 Lambda 函数来进行处理。

EventBridge (CloudWatch Events) 通过来自 Amazon EC2 的事件文档异步调用 Lambda 函数。

Example 实例生命周期事件

```
{
    "version": "0",
    "id": "b6ba298a-7732-2226-xmpl-976312c1a050",
    "detail-type": "EC2 Instance State-change Notification",
    "source": "aws.ec2",
    "account": "111122223333",
    "time": "2019-10-02T17:59:30Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:ec2:us-east-1:111122223333:instance/i-0c314xmplcd5b8173"
    ],
    "detail": {
        "instance-id": "i-0c314xmplcd5b8173",
        "state": "running"
    }
}
```

有关配置事件的详细信息，请参阅 按计划调用 Lambda 函数。有关处理 Amazon EBS 快照通知的示例函数，请参阅 EventBridge Scheduler for Amazon EBS。

您还可以使用 AWS 开发工具包，通过 Amazon EC2 API 管理实例和其他资源。

向 EventBridge（CloudWatch Events）授予权限

要处理来自 Amazon EC2 的生命周期事件，EventBridge（CloudWatch Events）需要权限以调用函数。此权限来自函数的基于资源的策略。如果您使用 EventBridge (CloudWatch Events) 控制台配置事件触发器，则该控制台将代表您更新基于资源的策略。否则，请添加如下所示的语句：

Example 基于资源的策略语句，用于 Amazon EC2 生命周期通知

```
{
```

```
  "Sid": "ec2-events",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": "lambda:InvokeFunction",
  "Resource": "arn:aws:lambda:us-east-1:12456789012:function:my-function",
  "Condition": {
    "ArnLike": {
      "AWS:SourceArn": "arn:aws:events:us-east-1:12456789012:rule/*"
    }
  }
}
```

要添加语句，请使用 `add-permission` AWS CLI 命令。

```
aws lambda add-permission --action lambda:InvokeFunction --statement-id ec2-events \
--principal events.amazonaws.com --function-name my-function --source-arn
 'arn:aws:events:us-east-1:12456789012:rule/*'
```

如果函数使用 AWS 开发工具包来管理 Amazon EC2 资源，请向函数的执行角色添加 Amazon EC2 权限。

使用 Lambda 处理应用程序负载均衡器请求

您可以使用 Lambda 函数，处理来自 Application Load Balancer 的请求。Elastic Load Balancing 支持 Lambda 函数作为 Application Load Balancer 的目标。使用负载均衡器规则，基于路径或标头值将 HTTP 请求路由到一个函数。处理请求并从 Lambda 函数返回 HTTP 响应。

Elastic Load Balancing 将使用包含请求体和元数据的事件同步调用 Lambda 函数。

Example Application Load Balancer 请求事件

```
{
    "requestContext": {
        "elb": {
            "targetGroupArn": "arn:aws:elasticloadbalancing:us-
east-1:123456789012:targetgroup/lambda-279XGJDqGZ5rsrHC2Fjr/49e9d65c45c6791a"
        }
    },
    "httpMethod": "GET",
    "path": "/lambda",
    "queryStringParameters": {
        "query": "1234ABCD"
    },
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8",
        "accept-encoding": "gzip",
        "accept-language": "en-US,en;q=0.9",
        "connection": "keep-alive",
        "host": "lambda-alb-123578498.us-east-1.elb.amazonaws.com",
        "upgrade-insecure-requests": "1",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
        "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
        "x-forwarded-for": "72.12.164.125",
        "x-forwarded-port": "80",
        "x-forwarded-proto": "http",
        "x-imforwards": "20"
    },
    "body": "",
    "isBase64Encoded": False
}
```

您的函数将处理事件并以 JSON 格式向负载均衡器返回响应文档。Elastic Load Balancing 将文档转换为 HTTP 成功或错误响应并将其返回给用户。

Example 响应文档格式

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "isBase64Encoded": False,
    "headers": {
        "Content-Type": "text/html"
    },
    "body": "<h1>Hello from Lambda!</h1>"
}
```

要将 Application Load Balancer 配置为函数触发器，请为 Elastic Load Balancing 授予运行函数的权限、创建向函数发送请求的目标组、并将规则添加到向目标组发送请求的负载均衡器。

使用 `add-permission` 命令将权限语句添加到函数的基于资源的策略。

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

您应看到以下输出：

```
{
    "Statement": "{\"Sid\":\"load-balancer\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"elasticloadbalancing.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-west-2:123456789012:function:alb-function\"}"
}
```

有关配置 Application Load Balancer 侦听器和目标组的说明，请参阅 Application Load Balancer 用户指南中的 Lambda 函数目标。

按计划调用 Lambda 函数

Amazon EventBridge 调度器是一个无服务器调度器，使您能够从一个中央托管服务创建、运行和管理任务。借助 EventBridge 调度器，您可以使用 cron 和 rate 表达式为定期模式创建计划，也可以配置一次性调用。您可以设置灵活的交付时间窗口、定义重试限制，并为未处理的事件设置最长保留时间。

当您使用 Lambda 设置 EventBridge 调度器时，EventBridge 调度器会异步调用您的 Lambda 函数。本页介绍如何使用 EventBridge 调度器按计划调用 Lambda 函数。

设置执行角色

创建新计划时，EventBridge 调度器必须有权代表您调用其目标 API 操作。您可以使用执行角色授予 EventBridge 调度器这些权限。您附加到计划执行角色的权限策略定义了所需权限。这些权限取决于您希望 EventBridge 调度器调用的目标 API。

使用 EventBridge 调度器控制台创建计划时，EventBridge 调度器会根据选择的目标自动设置执行角色，如以下步骤所示。如果您想使用 EventBridge 调度器 SDK（AWS CLI 或 AWS CloudFormation ）之一创建计划，您必须拥有现有的执行角色，以授予 EventBridge 调度器调用目标所需的权限。有关为计划手动设置执行角色的更多信息，请参阅 EventBridge Scheduler User Guide 中的 Setting up an execution role。

创建计划

使用控制台创建计划

1. 打开 Amazon EventBridge 调度器控制台，网址为：https://console.aws.amazon.com/scheduler/home。
2. 在计划页面，选择创建计划。
3. 在指定计划详细信息页面，在计划名称和描述部分中，执行以下操作：

   a. 对于计划名称，输入计划的名称。例如，**`MyTestSchedule`**。

   b. （可选）对于描述，输入对计划的描述。例如，**`My first schedule`**。

   c. 对于计划组，从下拉列表中选择一个计划组。如果您没有计划组，选择默认。要创建计划组，选择创建自己的计划。

      您可以使用计划组将标签添加到计划组。

4. • 选择计划选项。

出现	请执行此操作...	
一次性计划 一次性计划仅在您指定的日期和时间调用一次目标。	对于日期和时间，请执行以下操作： • 输入 YYYY/MM/DD 格式的有效日期。 • 输入 24 小时 hh:mm 格式的时间戳。 • 对于时区，选择时区。	

出现	请执行此操作...	
定期计划 定期计划按照您使用 cron 表达式或 rate 表达式指定的速率调用目标。	a. 对于计划类型，执行以下操作之一： • 要使用 cron 表达式定义计划，请选择基于 cron 的计划并输入 cron 表达式。 • 要使用 rate 表达式定义计划，请选择基于 rate 的计划并输入 rate 表达式。 有关 cron 和 rate 表达式的更多信息，请参阅 Amazon EventBridge 调度器用户指南 中的 EventBridge 调度器的计划类型。 b. 对于灵活的时间窗口，选择关闭以关闭该选项，或者选择一个预定义的时间窗口。例如，如果您选择 15 分钟并且将定期计划设置为每小时调用一次其目标，则该计划将在每小时开始后的 15 分钟内运行。	

5. （可选）如果您在上一步中选择定期计划，在时间范围部分，请执行以下操作：

   a. 对于时区，请选择时区。

   b. 对于开始日期和时间，请输入 YYYY/MM/DD 格式的有效日期，然后指定 24 小时 hh:mm 格式的时间戳。

   c. 对于结束日期和时间，请输入 YYYY/MM/DD 格式的有效日期，然后指定 24 小时 hh:mm 格式的时间戳。

6. 选择下一步。
7. 在选择目标页面，选择 EventBridge 调度器调用的 AWS API 操作：
   a. 选择 AWS Lambda 调用。
   b. 在调用部分，选择功能或选择创建新 Lambda 函数。
   c. （可选）输入 JSON 有效负载。如果您未输入有效负载，EventBridge 调度器将使用空事件来调用函数。
8. 选择下一步。
9. 在 Settings (设置) 页面上，执行以下操作：
   a. 要打开计划，在计划状态下，切换启用计划。
   b. 要为计划配置重试策略，在重试策略和死信队列（DLQ）下，请执行以下操作：
      - 切换重试。
      - 对于事件的最长期限，输入 EventBridge 调度器必须保留未处理事件的最长小时和分钟数。
      - 最长时间为 24 小时。
      - 对于最大重试次数，输入在目标返回错误的情况下，EventBridge 调度器重试计划的最大次数。

        最大值为 185 次重试。

      配置重试策略后，如果计划未能调用其目标，EventBridge 调度器将重新运行该计划。如果已配置，则必须为计划设置最长保留时间和最大重试次数。
   c. 选择 EventBridge 调度器存储未送达事件的位置。

死信队列（DLQ）选项	请执行此操作...
请勿存储	选择 None。
将事件存储在创建计划所在的同一 AWS 账户 中	a. 选择在我的 AWS 账户 中选择一个 Amazon SQS 队列作为 DLQ。 b. 选择 Amazon SQS 队列的 Amazon 资源名称（ARN）。

死信队列（DLQ）选项	请执行此操作...
将事件存储在与创建计划所在不同的 AWS 账户 中	a. 选择在另一个 AWS 账户 中指定一个 Amazon SQS 队列作为 DLQ。 b. 输入 Amazon SQS 队列的 Amazon 资源名称（ARN）。

d. 要使用客户托管密钥加密目标输入，在加密下，选择自定义加密设置（高级）。

   如果选择此选项，请输入现有的 KMS 密钥 ARN 或选择创建一个 AWS KMS key 以导航到 AWS KMS 控制台。有关 EventBridge 调度器如何加密静态数据的更多信息，请参阅 Amazon EventBridge Scheduler User Guide 中的 Encryption at rest。

e. 要让 EventBridge 调度器为您创建新的执行角色，请选择为此计划创建新角色。然后，在角色名称中输入名称。如果您选择此选项，EventBridge 调度器会将模板化目标所需的必要权限附加到该角色。

10. 选择下一步。
11. 在查看并创建计划页面上，查看计划的详细信息。在每个部分中，选择编辑返回到该步骤并编辑其详细信息。
12. 选择创建计划。

    您可以在计划页面上查看新的和现有的计划列表。在状态列下，验证新计划是否已启用。

要确认 EventBridge 调度器是否调用了该函数，查看该函数的 Amazon CloudWatch Logs。

相关资源

有关 EventBridge 调度器的详细信息，请参阅以下内容：

- EventBridge Scheduler User Guide
- EventBridge Scheduler API Reference
- EventBridge Scheduler Pricing

配合使用 AWS Lambda 和 AWS IoT

AWS IoT 提供连接 Internet 的设备（如传感器）与 AWS 云之间的安全通信。这让您能够从多个设备收集、存储和分析遥测数据。

您可以为设备创建 AWS IoT 规则，以便与 AWS 服务进行交互。AWS IoT 规则引擎提供基于 SQL 的语言，用于从消息负载中选择数据，并将数据发送到其他服务，例如 Amazon S3、Amazon DynamoDB 和 AWS Lambda。当您想要调用其他AWS服务或第三方服务时，您可以定义规则以调用 Lambda 函数。

当传入的 IoT 消息触发规则时，AWS IoT 会异步调用 Lambda 函数并将数据从 IoT 消息传递到函数。

以下示例显示了温室传感器的湿度读数。row 和 pos 值标识传感器的位置。此示例事件基于 AWS IoT 规则教程中的温室类型。

Example AWS IoT 消息事件

```
{
    "row" : "10",
    "pos" : "23",
    "moisture" : "75"
}
```

对于异步调用，Lambda 会对消息排队并在函数返回错误时重试。通过 destination 配置函数，保留函数无法处理的事件。

您需要向 AWS IoT 服务授予调用 Lambda 函数的权限。使用 `add-permission` 命令将权限语句添加到函数的基于资源的策略。

```
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal
 iot.amazonaws.com
```

您应看到以下输出：

```
{
    "Statement": "{\"Sid\":\"iot-events\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"iot.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":
\"arn:aws:lambda:us-east-1:123456789012:function:my-function\"}"
```

```
}
```

有关如何将 AWS IoT 与 Lambda 结合使用的更多信息，请参阅[创建 AWS Lambda 规则]。

Lambda 如何处理来自 Amazon Kinesis Data Streams 的记录

您可以使用 Lambda 函数来处理 Amazon Kinesis 数据流中的记录。您可以将 Lambda 函数映射到 Kinesis Data Streams 共享吞吐量使用者（标准迭代器）或具有增强型扇出功能的专用吞吐量使用者。对于标准迭代器，Lambda 使用 HTTP 协议轮询 Kinesis 流中的每个分片以查找记录。事件源映射与分片的其他使用者共享读取吞吐量。

有关 Kinesis 数据流的详细信息，请参阅读取 Amazon Kinesis Data Streams 中的数据。

> **Note**
>
> Kinesis 按每个分区收费；对于增强型扇出功能，从流中读取数据。有关定价详细信息，请参阅 Amazon Kinesis 定价。

主题

轮询和批处理流

Lambda 从数据流中读取记录并同步调用您的函数，带有一个包含流记录的事件。Lambda 分批读取记录并调用您的函数来处理批处理中的记录。每个批处理包含来自单个分区/数据流的记录。

对于标准 Kinesis 数据流，Lambda 将轮询流中的每个分片来获取记录（按照每个分片每秒一次的频率）。对于 Kinesis 增强型扇出功能，Lambda 使用 HTTP/2 连接来监听从 Kinesis 推送的记录。如果记录可用，Lambda 会调用函数并等待结果。

默认情况下，Lambda 会在记录可用时尽快调用您的函数。如果 Lambda 从事件源中读取的批处理只有一条记录，则 Lambda 将会只向该函数发送一条记录。为避免在记录数量较少的情况下调用该函数，您可以配置 batching window（批处理时段），让事件源缓冲最多五分钟的记录。调用函数前，Lambda 会持续从事件源中读取记录，直到收集完整批处理、批处理时段到期或批处理达到 6MB 的有效负载时为止。有关更多信息，请参阅 批处理行为。

> **Warning**
>
> Lambda 事件源映射至少处理每个事件一次，有可能出现重复处理记录的情况。为避免与重复事件相关的潜在问题，我们强烈建议您将函数代码设为幂等性。要了解更多信息，请参阅 AWS 知识中心的如何使我的 Lambda 函数具有幂等性。

Lambda 在发送下次批处理之前不会等待任何配置的扩展完成。换句话说，扩展可能会在 Lambda 处理下一批记录时继续运行。如果您违反了账户的任何并发设置或限制，可能会导致节流问题。要检测这是否是潜在问题，请监控函数并检查所显示的并发指标是否高于事件源映射的预期。由于调用间隔时间较短，Lambda 可能会短暂报告高于分片数量的并发使用量。即使对于没有扩展名的 Lambda 函数也是如此。

配置 ParallelizationFactor 设置以同时使用多个 Lambda 调用处理 Kinesis 数据流的一个分片。您可以指定 Lambda 通过从 1（默认值）到 10 的并行化因子从分区中轮询的并发批次数。例如，假设您将 `ParallelizationFactor` 设置为 2，则最多可以有 200 个并发 Lambda 调用来处理 100 个 Kinesis 数据分片（但您可能实际上会看到不同的 `ConcurrentExecutions` 指标值）。这有助于在数据量不稳定并且 `IteratorAge` 较高时纵向扩展处理吞吐量。增加每个分片的并发批次数后，Lambda 仍然可以确保分区密钥级别的顺序处理。

您还可以将 `ParallelizationFactor` 与 Kinesis 聚合一起使用。事件源映射的行为取决于您是否使用增强型扇出功能：

- 如果没有增强型扇出功能：聚合事件中的所有事件都必须具有相同的分区键。该分区键还必须与聚合事件的分区键相匹配。如果聚合事件中的事件具有不同的分区键，则 Lambda 无法保证按分区键依照顺序处理事件。
- 借助增强型扇出功能：首先，Lambda 将聚合的事件解码为其单个事件。聚合事件可以具有与其包含的事件不同的分区键。但是，与分区键不对应的事件会被丢弃并丢失。Lambda 不处理这些事件，也不会将它们发送到配置的失败目标。

示例事件

Example

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        },
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692540925702759324208523137515618",
                "data": "VGhpcyBpcyBvbmx5IGEgdGVzdC4=",
                "approximateArrivalTimestamp": 1545084711.166
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692540925702759324208523137515618",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
```

```
        }
    ]
}
```

使用 Lambda 处理 Amazon Kinesis Data Streams 记录

要使用 Lambda 处理 Amazon Kinesis Data Streams 记录，请为您的流创建一个使用者，然后创建 Lambda 事件源映射。

配置数据流和函数

您的 Lambda 函数是数据流的用户应用程序。对于每个分片，它一次处理一批记录。您可以将 Lambda 函数映射到共享吞吐量使用者（标准迭代器）或具有增强扇出功能的专用吞吐量使用者。

- 标准迭代器：Lambda 将针对记录轮询 Kinesis 流中的每个分片（按照每秒一次的基本频率）。当有更多记录可用时，Lambda 会继续进行批处理，直到函数赶上流的速度。事件源映射与分区的其他使用者共享读取吞吐量。
- 增强型扇出功能：为了最大限度地减少延迟并最大限度地提高读取吞吐量，请创建具有增强型扇出功能的数据流使用者。增强扇出功能使用者将获得与每个分片的专用连接，这不会影响从流中读取信息的其他应用程序。流使用者使用 HTTP/2 通过长期连接将记录推送到 Lambda 并压缩请求头来减少延迟。您可以使用 Kinesis RegisterStreamConsumer API 创建流使用者。

```
aws kinesis register-stream-consumer \
--consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

您应看到以下输出：

```
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

要提高函数处理记录的速度，请将分片添加到数据流中。Lambda 按顺序处理各个分区中的记录。如果您的函数返回错误，它会停止处理分片中的其他记录。使用更多分片，可以同时处理更多批次，从而降低错误对并发性的影响。

如果您的函数无法扩展以处理并发批处理的总数，请为您的函数请求提高配额或预留并发。

创建一个事件源映射来调用 Lambda 函数

要使用来自数据流的记录调用 Lambda 函数，请创建一个事件源映射。您可以创建多个事件源映射，以使用多个 Lambda 函数处理相同的数据，或使用单个函数处理来自多个数据流的项目。处理来自多个流的项目时，每个批处理将只包含来自单个分片或流的记录。

您可以配置事件源映射来处理来自不同 AWS 账户中的流的记录。要了解更多信息，请参阅 the section called “跨账户映射”。

在创建事件源映射之前，您需要向您的 Lambda 函数授予读取 Kinesis 数据流中数据的权限。Lambda 需要以下权限才能管理与您的 Kinesis 数据流相关的资源：

- kinesis:DescribeStream
- kinesis:DescribeStreamSummary
- kinesis:GetRecords
- kinesis:GetShardIterator
- kinesis:ListShards
- kinesis:ListStreams
- kinesis:SubscribeToShard

AWS 托管式策略 AWSLambdaKinesisExecutionRole 包含这些权限。按照以下过程所述将此托管式策略添加到您的函数。

AWS Management Console

为您的函数添加 Kinesis 权限

1. 打开 Lambda 控制台的“函数”页面，然后选择函数。
2. 在配置选项卡中，选择权限。
3. 在执行角色窗格的角色名称下，选择指向函数的执行角色的链接。此链接将在 IAM 控制台中打开该角色的页面。
4. 在权限策略窗格中，选择添加权限，然后选择附加策略。

5. 在搜索字段中输入 **AWSLambdaKinesisExecutionRole**。
6. 选中该策略名称旁边的复选框，然后选择添加权限。

AWS CLI

为您的函数添加 Kinesis 权限

- 运行以下 CLI 命令，以将 AWSLambdaKinesisExecutionRole 策略附加到函数的执行角色。

```
aws iam attach-role-policy \
--role-name MyFunctionRole \
--policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaKinesisExecutionRole
```

AWS SAM

为您的函数添加 Kinesis 权限

- 在函数定义中添加 Policies 属性，如以下示例所示：

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs20.x
      Policies:
        - AWSLambdaKinesisExecutionRole
```

配置所需的权限后，创建事件源映射。

AWS Management Console

创建 Kinesis 事件源映射

1. 打开 Lambda 控制台的“函数”页面，然后选择函数。
2. 在函数概述窗格中，选择添加触发器。

3. 在触发器配置下，对于源，请选择 Kinesis 。
4. 选择要为其创建事件源映射的 Kinesis 流，也可以选择流的使用者。
5. （可选）编辑事件源映射的批处理大小、起始位置和批处理窗口。
6. 选择添加。

在控制台中创建事件源映射时，您的 IAM 角色必须拥有 kinesis:ListStreams 和 kinesis:ListStreamConsumers 权限。

AWS CLI

创建 Kinesis 事件源映射

- 运行以下 CLI 命令以创建 Kinesis 事件源映射。根据您的应用场景选择自己的批量大小和起始位置。

```
aws lambda create-event-source-mapping \
--function-name MyFunction \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--starting-position LATEST \
--batch-size 100
```

要指定批处理时间窗，请添加 `--maximum-batching-window-in-seconds` 选项。有关使用此参数和其他参数的更多信息，请参阅《AWS CLI Command Reference》中的 create-event-source-mapping。

AWS SAM

创建 Kinesis 事件源映射

- 在函数定义中添加 `KinesisEvent` 属性，如以下示例所示：

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs20.x
      Policies:
        - AWSLambdaKinesisExecutionRole
```

```
      Events:
        KinesisEvent:
          Type: Kinesis
          Properties:
            Stream: !GetAtt MyKinesisStream.Arn
            StartingPosition: LATEST
            BatchSize: 100

  MyKinesisStream:
    Type: AWS::Kinesis::Stream
    Properties:
      ShardCount: 1
```

要了解有关在 AWS SAM 中创建 Kinesis 数据流事件源映射的更多信息，请参阅《AWS Serverless Application Model Developer Guide》中的 Kinesis。

轮询和流的起始位置

请注意，事件源映射创建和更新期间的流轮询最终是一致的。

- 在事件源映射创建期间，可能需要几分钟才能开始轮询来自流的事件。
- 在事件源映射更新期间，可能需要几分钟才能停止和重新开始轮询来自流的事件。

此行为意味着，如果你指定 LATEST 作为流的起始位置，事件源映射可能会在创建或更新期间错过事件。为确保不会错过任何事件，请将流的起始位置指定为 TRIM_HORIZON 或 AT_TIMESTAMP。

创建跨账户事件源映射

Amazon Kinesis Data Streams 支持基于资源的策略。因此，您可以在一个 AWS 账户中使用 Lambda 函数来处理另一个账户的流中摄入的数据。

要使用其他 AWS 账户中的 Kinesis 流为您的 Lambda 函数创建事件源映射，您必须使用基于资源的策略配置该流，以向您的 Lambda 函数授予读取相关项目的权限。要了解如何配置流以允许跨账户存取，请参阅《Amazon Kinesis Streams 开发人员指南》中的 Sharing access with cross-account AWS Lambda functions。

使用基于资源的策略配置流以向您的 Lambda 函数授予所需的权限后，请使用上一节中描述的任何方法创建事件源映射。

如果您选择使用 Lambda 控制台创建事件源映射，请将流的 ARN 直接粘贴到输入字段中。如果您想指定流的使用者，粘贴使用者的 ARN 会自动填充流字段。

使用 Kinesis Data Streams 和 Lambda 配置部分批次响应

在使用和处理来自事件源的流式数据时，默认情况下，Lambda 仅在批处理完全成功时，才会在批次的最高序列号处设置检查点。Lambda 会将所有其他结果视为完全失败并重试批处理，直至达到重试次数上限。要允许在处理来自流的批次时部分成功，请开启 `ReportBatchItemFailures`。允许部分成功有助于减少对记录重试的次数，尽管这并不能完全阻止在成功记录中重试的可能性。

要开启 `ReportBatchItemFailures`，请在 FunctionResponseTypes 列表中包含枚举值 **`ReportBatchItemFailures`**。此列表指示为函数启用了哪些响应类型。您可以在创建或更新事件源映射时配置此列表。

报告语法

配置批处理项目失败的报告时，将返回 `StreamsEventResponse` 类，其中包含批处理项目失败列表。您可以使用 `StreamsEventResponse` 对象返回批处理中第一个失败记录的序列号。您还可以使用正确的响应语法来创建自己的自定义类。以下 JSON 结构显示了所需的响应语法：

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> 如果 `batchItemFailures` 数组包含多个项目，Lambda 会使用序列号最小的记录作为检查点。然后，Lambda 会重试从该检查点开始的所有记录。

成功和失败的条件

如果返回以下任意一项，则 Lambda 会将批处理视为完全成功：

- 空的 `batchItemFailure` 列表
- Null `batchItemFailure` 列表

- 空的 EventResponse
- Null EventResponse

如果返回以下任何一项，则 Lambda 会将批处理视为完全失败：

- 空字符串 itemIdentifier
- Null itemIdentifier
- 包含错误密钥名的 itemIdentifier

Lambda 会根据您的重试策略在失败时重试。

将批次一分为二

如果调用失败并且已开启 BisectBatchOnFunctionError，则无论您的 ReportBatchItemFailures 设置如何，批次都将一分为二。

当收到批处理部分成功响应且同时开启 BisectBatchOnFunctionError 和 ReportBatchItemFailures 时，批次将在返回的序列号处一分为二，并且 Lambda 将仅重试剩余记录。

以下函数代码示例将返回批处理中处理失败消息的 ID 列表：

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 .NET 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
```

```
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
```

```
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告通过 Go 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Java 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Javascript 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
```

```
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

报告使用 TypeScript 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
```

```
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告通过 PHP 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
```

```
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Python 进行 Lambda Kinesis 批处理项目失败。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告通过 Ruby 进行 Lambda Kinesis 批处理项目失败。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end
```

```
def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告通过 Rust 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
```

```
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
```

```
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

在 Lambda 中保留 Kinesis Data Streams 事件源的已丢弃批次记录

Kinesis 事件源映射的错误处理取决于错误是在调用函数之前还是在函数调用期间发生的：

- 调用前：如果 Lambda 事件源映射由于节流或其他问题而无法调用该函数，则它会一直重试，直到记录过期或超过事件源映射上配置的最大期限（MaximumRecordAgeInSeconds）。
- 调用期间：如果调用函数但返回错误，Lambda 会重试，直到记录过期、超过最大期限（MaximumRecordAgeInSeconds）或达到配置的重试配额（MaximumRetryAttempts）。对于函数错误，您还可以配置 BisectBatchOnFunctionError，将失败的批次拆分为两个较小的批次，从而隔离错误记录并避免超时。拆分批次不会消耗重试配额。

如果错误处理措施失败，Lambda 将丢弃记录并继续处理数据流中的批次。使用默认设置时，这意味着错误的记录可能会阻止受影响的分区上的处理，时间最长为一周。为了避免这种情况，请配置函数的事件源映射，使用合理的重试次数和适合您的使用案例的最长记录期限。

配置失败调用的目标

要保留失败的事件源映射调用的记录，请在函数的事件源映射中添加一个目标。发送到目标的每条记录都是一个 JSON 文档，其中包含有关失败调用的元数据。您可以将任何 Amazon SNS 主题或 Amazon SQS 队列配置为目标。您的执行角色必须具有目标的权限：

- 对于 SQS 目标：sqs:SendMessage
- 对于 SNS 目标：sns:Publish

要使用控制台配置失败时的目标，请执行以下步骤：

1. 打开 Lamba 控制台的 Functions（函数）页面。

2. 选择函数。
3. 在 Function overview (函数概览) 下，选择 Add destination (添加目标)。
4. 对于源，请选择事件源映射调用。
5. 对于事件源映射，请选择为此函数配置的事件源。
6. 在条件中，选择失败时。对于事件源映射调用，这是唯一可接受的条件。
7. 对于目标类型，请选择 Lambda 要发送调用记录的目标类型。
8. 对于 Destination (目标)，请选择一个资源。
9. 选择保存。

您还可以使用 AWS Command Line Interface（AWS CLI）配置失败时的目标。例如，以 create-event-source-mapping 命令将带有 SQS 失败时目标的事件源映射添加到 `MyFunction`：

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

以下 update-event-source-mapping 命令更新事件源映射，以在两次重试之后或记录超过一小时后将失败的调用记录发送到 SNS 目标。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

更新的设置是异步应用的，并且直到该过程完成才反映在输出中。使用 get-event-source-mapping 命令查看当前状态。

要移除目标，请提供一个空字符串作为 `destination-config` 参数的实际参数：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

以下示例显示了 Lambda 在 Kinesis 事件源调用失败时向 SQS 队列或 SNS 主题发送的内容。由于 Lambda 仅为这些目标类型发送元数据，因此请使用 `streamArn`、`shardId`、`startSequenceNumber` 和 `endSequenceNumber` 字段获取完整的原始记录。`KinesisBatchInfo` 属性中显示的所有字段始终存在。

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    }
}
```

您可以使用此信息从流中检索受影响的记录以进行故障排除。实际记录不包括在内，因此您必须处理此记录并在记录过期和丢失之前从流中检索它们。

在 Lambda 中实现有状态的 Kinesis Data Streams 处理

Lambda 函数可以运行连续流处理应用程序。流表示通过您的应用程序持续流动的无边界数据。要分析这种不断更新的输入中的信息，可以使用按时间定义的窗口来限制包含的记录。

滚动窗口是定期打开和关闭的不同窗口。预设情况下，Lambda 调用是无状态的，在没有外部数据库的情况下，无法使用它们跨多次连续调用处理数据。但是，有了滚动窗口后，您可以在不同调用中保持状

态。此状态包含之前为当前窗口处理的消息的汇总结果。您的状态最多可以是每个分片 1MB。如果超过该大小，Lambda 将提前终止窗口。

流中的每条记录都属于特定窗口。Lambda 将至少处理每条记录一次，但不保证每条记录只处理一次。在极少数情况下（例如错误处理），某些记录可能会被多次处理。第一次处理记录时始终按顺序处理。如果多次处理记录，则可能会不按顺序处理。

聚合和处理

系统将调用您的用户托管函数以便聚合和处理该聚合的最终结果。Lambda 汇总在该窗口中接收的所有记录。您可以分多个批次接收这些记录，每个批次都作为单独的调用。每次调用都会收到一个状态。因此，当使用滚动窗口时，Lambda 函数响应必须包含 `state` 属性。如果响应不包含 `state` 属性，Lambda 会将其视作失败的调用。为了满足该条件，您的函数可以返回一个具有以下 JSON 形状的 `TimeWindowEventResponse` 对象：

Example **`TimeWindowEventResponse`** 值

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> 对于 Java 函数，我们建议使用 `Map<String, String>` 来表示状态。

在窗口末尾，标志 `isFinalInvokeForWindow` 被设置 `true`，以表示这是最终状态，并且已准备好进行处理。处理完成后，窗口完成，最终调用完成，然后状态将被删除。

在窗口结束时，Lambda 会对针对聚合结果的操作应用最终处理。您的最终处理将同步调用。成功调用后，函数会检查序列号并继续进行流处理。如果调用失败，则您的 Lambda 函数将暂停进一步处理，直到成功调用为止。

Example KinesisTimeWindowEvent

```
{
```

```
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1607497475.000
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-kinesis-role",
            "awsRegion": "us-east-1",
            "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-
stream"
        }
    ],
    "window": {
        "start": "2020-12-09T07:04:00Z",
        "end": "2020-12-09T07:06:00Z"
    },
    "state": {
        "1": 282,
        "2": 715
    },
    "shardId": "shardId-000000000006",
    "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

配置

您可以在创建或更新事件源映射时配置滚动窗口。要配置翻转窗口，请以秒为单位进行指定（TumblingWindowInSeconds）。以下示例 AWS Command Line Interface (AWS CLI) 命令会创建一个滚动窗口为 120 秒的流式事件源映射。为聚合和处理定义的 Lambda 函数被命名为 `tumbling-window-example-function`。

```
aws lambda create-event-source-mapping \
```

```
--event-source-arn arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda 根据记录插入到流的时间来确定滚动窗口的边界。所有记录都有一个大致的时间戳，供 Lambda 在确定边界时使用。

滚动窗口聚合不支持重新分片。当分片结束时，Lambda 认为当前窗口已关闭，并且任何子分片都将以全新状态启动自己的窗口。如果没有向当前窗口添加任何新记录，则 Lambda 会等待最多 2 分钟，然后假定该窗口已结束。这有助于确保函数读取当前窗口中的所有记录，即使这些记录是间歇性添加的。

滚动窗口完全支持现有的重试策略 `maxRetryAttempts` 和 `maxRecordAge`。

Example Handler.py – 聚合和处理

以下 Python 函数演示了如何聚合然后处理您的最终状态：

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['kinesis']['partitionKey']] = state.get(record['kinesis']
['partitionKey'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Amazon Kinesis Data Streams 事件源映射的 Lambda 参数

所有 Lambda 事件源映射共享相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，只有部分参数适用于 Kinesis。

参数	必需	默认值	备注
BatchSize	否	100	最大值：10000
BisectBatchOnFunctionError	否	false	none
DestinationConfig	否	不适用	丢弃的记录的 Amazon SQS 队列或 Amazon SNS 主题目标。有关更多信息，请参阅 配置失败调用的目标。
Enabled (已启用)	否	真实	none
EventSourceArn	Y	不适用	数据流或流使用者的 ARN
FunctionName	是	不适用	none
FunctionResponseTypes	否	不适用	要使您的函数报告某个批处理中的特定失败，请在 `FunctionResponseTypes` 中包含值 `ReportBatchItemFailures` 。有关更多信息，请参阅 使用 Kinesis Data Streams 和 Lambda 配置部分批次响应。
MaximumBatchingWindowInSeconds	否	0	none

参数	必需	默认值	备注
MaximumRecordAgeInSeconds	否	–1	-1 表示无限：Lambda 不会丢弃记录（Kinesis Data Streams 的数据留存设置仍然适用） 最小值：-1 最大值：604800
MaximumRetryAttempts	否	–1	-1 表示无限：会一直重试失败的记录，直到记录过期。 最小值：-1 最大值：10000
ParallelizationFactor	否	1	最大值：10
StartingPosition	Y	不适用	AT_TIMESTAMP、TRIM_HORIZON 或 LATEST
StartingPositionTimestamp	否	不适用	仅当 StartingPosition 设置为 AT_TIMESTAMP 时才有效。开始读取的时间（以 Unix 时间秒为单位）
TumblingWindowInSeconds	否	不适用	最小值：0 最大值：900

对 Kinesis 事件源使用事件筛选

您可以使用事件筛选，控制 Lambda 将流或队列中的哪些记录发送给函数。有关事件筛选工作原理的一般信息，请参阅 the section called “事件筛选”。

本节重点介绍 Kinesis 事件源的事件筛选。

主题

- Kinesis 事件筛选基础知识
- 筛选 Kinesis 聚合记录

Kinesis 事件筛选基础知识

假设创建器将 JSON 格式的数据放入 Kinesis 数据流。示例记录如下所示，data 字段中的 JSON 数据会转换为 Base64 编码字符串。

```
{
    "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "1",
        "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
        "data":
 "eyJSZWNvcmROdW1iZXIiOiAiMDAwMSIsICJUaW1lU3RhbXAiOiAieXl5eS1tbS1kZFRoaDptbTpzcyIsICJSZXF1ZXN0Q
        "approximateArrivalTimestamp": 1545084650.987
        },
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
}
```

只要创建器放入流中的数据是有效的 JSON，您就可以使用 data 键，通过事件筛选来筛选记录。假设创建器将如下 JSON 格式的记录放入 Kinesis 流。

```
{
    "record": 12345,
```

```
    "order": {
        "type": "buy",
        "stock": "ANYCO",
        "quantity": 1000
        }
}
```

要仅筛选订单类型为“购买”的记录，FilterCriteria 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"order\" : { \"type\" : [ \"buy\" ] } } }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 Pattern 的值。

```
{
    "data": {
        "order": {
            "type": [ "buy" ]
            }
      }
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
```

```
    --function-name my-function \
    --event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/my-stream \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "order" : { "type" : [ "buy" ] } } }'
```

要正确筛选 Kinesis 源中的事件，数据字段及其筛选条件都必须为有效的 JSON 格式。如果任一字段不为有效的 JSON 格式，Lambda 将会丢弃消息或引发异常。下表汇总了具体行为：

传入数据格式	数据属性中的筛选条件模式格式	导致的操作
有效 JSON	有效 JSON	Lambda 根据您的筛选条件进行筛选。
有效 JSON	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	非 JSON	Lambda 在事件源映射创建或更新时引发异常。数据属性的筛选条件模式必须为有效的 JSON 格式。
非 JSON	有效 JSON	Lambda 将丢弃记录。

传入数据格式	数据属性中的筛选条件模式格式	导致的操作
非 JSON	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
非 JSON	非 JSON	Lambda 在事件源映射创建或更新时引发异常。数据属性的筛选条件模式必须为有效的 JSON 格式。

筛选 Kinesis 聚合记录

使用 Kinesis，您可以将多条记录聚合到单条 Kinesis Data Streams 记录中，以提高数据吞吐量。使用 Kinesis 增强扇出功能时，Lambda 只能将筛选条件应用于聚合记录。不支持使用标准 Kinesis 筛选聚合记录。使用增强扇出功能时，您可以配置 Kinesis 专用吞吐量使用者作为 Lambda 函数的触发器。然后，Lambda 会筛选聚合记录，并仅传递符合筛选条件的记录。

要了解有关 Kinesis 记录聚合的更多信息，请参阅“Kinesis Producer Library（KPL）关键概念”页面上的聚合部分。要了解有关将 Lambda 与 Kinesis 增强扇出功能结合使用的更多信息，请参阅 AWS 计算博客上的使用 Amazon Kinesis Data Streams 增强扇出功能和 AWS Lambda 提高实时流处理性能。

教程：将 Lambda 与 Kinesis Data Streams 结合使用

在本教程中，您将创建 Lambda 函数来处理 Amazon Kinesis 数据流中的事件。

1. 自定义应用程序将记录写入流。
2. AWS Lambda 轮询流并在检测到流中的新记录时调用 Lambda 函数。
3. AWS Lambda 通过代入您在创建 Lambda 函数时指定的执行角色来运行 Lambda 函数。

先决条件

本教程假设您对 Lambda 基本操作和 Lambda 控制台有一定了解。如果您还没有了解，请按照 使用控制台创建 Lambda 函数 中的说明创建您的第一个 Lambda 函数。

要完成以下步骤，您需要 AWS CLI 版本 2。在单独的数据块中列出了命令和预期输出：

```
aws --version
```

您应看到以下输出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

对于长命令，使用转义字符 (\) 将命令拆分为多行。

在 Linux 和 macOS 中，可使用您首选的 shell 和程序包管理器。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常与 Lambda 一起使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。本指南中的示例 CLI 命令使用 Linux 格式。如果您使用的是 Windows CLI，则必须重新格式化包含内联 JSON 文档的命令。

创建执行角色

创建执行角色，向您的函数授予访问 AWS 资源的权限。

创建执行角色

1. 在 IAM 控制台中，打开 Roles（角色）页面。
2. 选择创建角色。
3. 创建具有以下属性的角色。
   - Trusted entity（可信任的实体）– AWS Lambda。
   - Permissions（权限）– AWSLambdaKinesisExecutionRole。
   - Role name（角色名称）– **`lambda-kinesis-role`**。

AWSLambdaKinesisExecutionRole 策略具有该函数从 Kinesis 中读取项目并将日志写入 CloudWatch Logs 所需的权限。

创建函数

创建一个处理您的 Kinesis 消息的 Lambda 函数。函数代码会将 Kinesis 记录的事件 ID 和事件数据记录到 CloudWatch Logs 中。

本教程使用 Node.js 18.x 运行时系统，但我们还提供了其他运行时系统语言的示例代码。您可以选择以下框中的选项卡，查看适用于您感兴趣的运行时系统的代码。您将在此步骤中使用的 JavaScript 代码是 JavaScript 选项卡中显示的第一个示例。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 .NET 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
```

```
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
```

```
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Java 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
```

```
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 JavaScript 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
```

```
  return data;
}
```

通过 TypeScript 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
```

```
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 PHP 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
```

```
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 Kinesis 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
```

```
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 Kinesis 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
```

```
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Rust 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
```

```
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

创建函数

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir kinesis-tutorial
cd kinesis-tutorial
```

2. 将示例 JavaScript 代码复制到名为 `index.js` 的新文件中。
3. 创建部署程序包。

```
zip function.zip index.js
```

4. 使用 `create-function` 命令创建 Lambda 函数。

```
aws lambda create-function --function-name ProcessKinesisRecords \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::111122223333:role/lambda-kinesis-role
```

测试 Lambda 函数

使用 invoke AWS Lambda CLI 命令和示例 Kinesis 事件手动调用 Lambda 函数。

测试 Lambda 函数

1. 将以下 JSON 复制到文件中并将其保存为 input.txt。

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::111122223333:role/lambda-kinesis-
role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:111122223333:stream/
lambda-stream"
        }
    ]
}
```

2. 使用 invoke 命令将事件发送到该函数。

```
aws lambda invoke --function-name ProcessKinesisRecords \
--cli-binary-format raw-in-base64-out \
```

```
--payload file://input.txt outputfile.txt
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

响应将保存到 `out.txt` 中。

创建 Kinesis 流

使用 `create-stream` 命令创建流。

```
aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

运行下面的 `describe-stream` 命令以获取流 ARN。

```
aws kinesis describe-stream --stream-name lambda-stream
```

您应看到以下输出：

```
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
                    "StartingHashKey": "0",
                    "EndingHashKey": "340282366920746074317682119384634633455"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49591073947768692513481539594623130411957558361251844610"
                }
            }
        ],
        "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream",
        "StreamName": "lambda-stream",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
```

```
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "KeyId": null,
        "StreamCreationTimestamp": 1544828156.0
    }
}
```

您将使用下一步中的流 ARN 来将该流关联到您的 Lambda 函数。

在 AWS Lambda 中添加事件源

运行以下 AWS CLI `add-event-source` 命令。

```
aws lambda create-event-source-mapping --function-name ProcessKinesisRecords \
--event-source  arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream \
--batch-size 100 --starting-position LATEST
```

记下映射 ID 以供将来使用。您可以通过运行 `list-event-source-mappings` 命令获取事件源映射的列表。

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream
```

在该响应中，您可以验证状态值是否为 `enabled`。可以禁用事件源映射，以临时暂停轮询而不丢失任何记录。

测试设置

要测试事件源映射，请将事件记录添加到 Kinesis 流中。`--data` 值是一个字符串，CLI 先将其编码为 base64 字符串，然后才发送到 Kinesis。您可以多次运行同一命令来向流中添加多条记录。

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda 使用执行角色来读取来自流的记录。然后它将调用 Lambda 函数，批量传递记录。该函数解码每条记录中的数据并记录它，将输出发送到 CloudWatch Logs 中。在 CloudWatch 控制台中查看这些日志。

清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。
4. 在文本输入字段中输入角色名称，然后选择 Delete（删除）。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择 Delete（删除）。

删除 Kinesis 流

1. 登录到 AWS Management Console，然后通过以下网址打开 Kinesis 控制台：https://console.aws.amazon.com/kinesisvideo/home。
2. 选择您创建的流。
3. 依次选择 Actions（操作）和 Delete（删除）。
4. 在文本输入字段中输入 **delete**。
5. 选择 Delete（删除）。

结合 Amazon MQ 使用 Lambda

> **Note**
>
> 如果想要将数据发送到 Lambda 函数以外的目标，或要在发送数据之前丰富数据，请参阅 Amazon EventBridge Pipes（Amazon EventBridge 管道）。

Amazon MQ 是一项托管消息代理服务，用于 Apache ActiveMQ 和 RabbitMQ。消息代理允许软件应用程序和组件使用各种编程语言、操作系统和正式消息收发协议，通过主题或队列事件目标进行通信。

Amazon MQ 还可以通过安装 ActiveMQ 代理或 RabbitMQ 代理以及提供不同的网络拓扑和其他基础设施需求来代表您管理 Amazon Elastic Compute Cloud (Amazon EC2) 实例。

您可使用 Lambda 函数处理来自 Amazon MQ 消息代理的记录。Lambda 通过事件源映射调用您的函数，事件源映射是从您的代理读取消息并同步调用函数的一种 Lambda 资源。

> **Warning**
>
> Lambda 事件源映射至少处理每个事件一次，有可能出现重复处理记录的情况。为避免与重复事件相关的潜在问题，我们强烈建议您将函数代码设为幂等性。要了解更多信息，请参阅 AWS 知识中心的如何使我的 Lambda 函数具有幂等性。

Amazon MQ 事件源映射有以下配置限制：

- 并发 – 使用 Amazon MQ 事件源映射的 Lambda 函数具有默认的最大并发设置。对于 ActiveMQ，Lambda 服务将每个 Amazon MQ 事件源映射的并发执行环境数量限制为 5 个。对于 RabbitMQ，每个 Amazon MQ 事件源映射的并发执行环境数量限制为 1 个。即使您更改了函数的预留或预调配并发设置，Lambda 服务也不会提供更多的执行环境。要请求增加单个 Amazon MQ 事件源映射的默认最大并发数，请联系 AWS Support 并提供事件源映射 UUID 和区域。因为是在特定的事件源映射级别而不是账户或区域级别增加，所以需要为每个事件源映射手动请求按比例增加。
- 跨账户 – Lambda 不支持跨账户处理。您不能使用 Lambda 处理来自不同 AWS 账户 账户中的 Amazon MQ 消息代理的记录。
- 身份验证 – 对于 ActiveMQ，仅支持 ActiveMQ SimpleAuthenticationPlugin。对于 RabbitMQ，仅支持 PLAIN 身份验证机制。用户必须使用 AWS Secrets Manager 来管理凭据。有关 ActiveMQ 身份验证的更多信息，请参阅 Amazon MQ 开发人员指南中的使用 LDAP 集成 ActiveMQ 代理。

- 连接配额 – 代理具有每个有线级协议允许的最大连接数。此配额基于代理实例类型。有关更多信息，请参阅 Amazon MQ 开发人员指南中的 Amazon MQ 中的配额的代理部分。
- 连接 – 您可以在公有或私有虚拟私有云（VPC）中创建代理。对于私有 VPC，您的 Lambda 函数需要具备对 VPC 的访问权限才能接收消息。有关更多信息，请参阅此部分后面的the section called “配置网络安全”。
- 事件目标 – 仅支持队列目标。但是，您可以使用虚拟主题，虚拟主题在内部与主题行为一致，在与 Lambda 交互时与队列行为一致。有关更多信息，请参阅 Apache ActiveMQ 网站上的虚拟目标和 RabbitMQ 网站上的虚拟主机。
- 网络拓扑 – 对于 ActiveMQ，每个事件源映射仅支持一个单实例或备用代理。对于 RabbitMQ，每个事件源映射只支持一个单实例代理或集群部署。单实例代理需要一个失效转移端点。有关这些代理部署模式的更多信息，请参阅 Amazon MQ 开发人员指南中的 Active MQ 代理架构和RabbitMQ 代理架构。
- 协议 – 支持的协议取决于 Amazon MQ 集成的类型。
  - 对于 ActiveMQ 集成，Lambda 使用 OpenWire/Java Message Service (JMS) 协议来使用消息。消息的使用不支持任何其他协议。在 JMS 协议中，仅支持 `TextMessage` 和 `BytesMessage`。Lambda 还支持 JMS 自定义属性。有关 OpenWire 协议的更多信息，请参阅 Apache ActiveMQ 网站上的 OpenWire。
  - 对于 RabbitMQ 集成，Lambda 使用 AMQP 0-9-1 协议来使用消息。消息的使用不支持任何其他协议。有关 RabbitMQ 的 AMQP 0-9-1 协议实施的详细信息，请参阅 RabbitMQ 网站上的 AMQP 0-9-1 完整参考指南。

Lambda 自动支持 Amazon MQ 支持的最新版本的 ActiveMQ 和 RabbitMQ。有关受支持的最新版本，请参阅 Amazon MQ 开发人员指南中的 Amazon MQ 发布说明。

> **Note**
>
> 默认情况下，Amazon MQ 代理有一个每周维护时段。代理在该时段内无法使用。对于没有备用代理的代理，Lambda 将无法在该时段处理任何消息。

主题

- Amazon MQ 事件源映射错误排查

了解 Amazon MQ 的 Lambda 使用者组

为了与 Amazon MQ 进行交互，Lambda 会创建一个可以从 Amazon MQ 代理中读取的使用者组。使用与事件源映射 UUID 相同的 ID 创建使用者组。

对于 Amazon MQ 事件源，Lambda 会将记录合并为批处理，然后通过单个有效负载中将其发送到您的函数。要控制行为，您可以配置批处理时段和批处理大小。Lambda 会持续提取消息，直到达到 6 MB 的最大有效负载大小、批处理时段过期或记录数达到完整批处理大小时为止。有关更多信息，请参阅批处理行为。

使用者组将消息作为字节 BLOB 进行检索，然后以 base64 格式编码为单个 JSON 有效负载，接下来调用您的函数。如果函数为批处理中的任何消息返回错误，Lambda 将重试整批消息，直到处理成功或消息过期为止。

> **Note**
>
> 尽管 Lambda 函数的最大超时限制通常为 15 分钟，但 Amazon MSK、自行管理的 Apache Kafka、Amazon DocumentDB、Amazon MQ for ActiveMQ 和 RabbitMQ 的事件源映射，仅支持最大超时限制为 14 分钟的函数。此约束可确保事件源映射可以正确处理函数错误和重试。

您可以使用 Amazon CloudWatch 中的 `ConcurrentExecutions` 指标监控给定函数的并发使用情况。有关并发的更多信息，请参阅 the section called “配置预留并发”。

Example Amazon MQ 记录事件

ActiveMQ

```
{
   "eventSource": "aws:mq",
   "eventSourceArn": "arn:aws:mq:us-
east-2:111122223333:broker:test:b-9bcfa592-423a-4942-879d-eb284b418fc8",
   "messages": [
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/text-message",
        "deliveryMode": 1,
        "replyTo": null,
```

```
        "type": null,
        "expiration": "60000",
        "priority": 1,
        "correlationId": "myJMSCoID",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"QUJDOkFBQUE=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      },
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/bytes-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 2,
        "correlationId": "myJMSCoID1",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"LQaGQ82S48k=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      }
   ]
```

```
}
```

RabbitMQ

```
{
  "eventSource": "aws:rmq",
  "eventSourceArn": "arn:aws:mq:us-
east-2:111122223333:broker:pizzaBroker:b-9bcfa592-423a-4942-879d-eb284b418fc8",
  "rmqMessagesByQueue": {
    "pizzaQueue::/": [
      {
        "basicProperties": {
          "contentType": "text/plain",
          "contentEncoding": null,
          "headers": {
            "header1": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                49
              ]
            },
            "header2": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                50
              ]
            },
            "numberInHeader": 10
          },
          "deliveryMode": 1,
          "priority": 34,
          "correlationId": null,
          "replyTo": null,
          "expiration": "60000",
```

```
          "messageId": null,
          "timestamp": "Jan 1, 1970, 12:33:41 AM",
          "type": null,
          "userId": "AIDACKCEVSQ6C2EXAMPLE",
          "appId": null,
          "clusterId": null,
          "bodySize": 80
        },
        "redelivered": false,
        "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
      }
    ]
  }
}
```

> **Note**
>
> 在 RabbitMQ 示例中，`pizzaQueue` 是 RabbitMQ 队列的名称，`/` 是虚拟主机的名称。接收消息时，事件源会将消息列在 `pizzaQueue::/` 下。

为 Lambda 配置 Amazon MQ 事件源

主题

- 配置网络安全
- 创建事件源映射

配置网络安全

要通过事件源映射向 Lambda 提供对 Amazon MQ 的完全访问权限，代理必须使用公有端点（公有 IP 地址），或者您必须提供对您在其中创建了代理的 Amazon VPC 的访问权限。

将 Amazon MQ 与 Lambda 配合使用时，我们建议您创建 AWS PrivateLinkVPC 端点，并向函数提供对 Amazon VPC 中资源的访问权限。

创建端点以提供对以下资源的访问权限：

- Lambda：为 Lambda 服务主体创建端点。
- AWS STS：为 AWS STS 创建端点，以便服务主体代您代入角色。

- Secrets Manager：如果代理使用 Secrets Manager 来存储凭证，请为 Secrets Manager 创建端点。

也可以在 Amazon VPC 中的每个公有子网上配置 NAT 网关。有关更多信息，请参阅 the section called “VPC 函数的互联网访问权限”。

为 Amazon MQ 创建事件源映射时，Lambda 会检查为 Amazon VPC 配置的子网和安全组是否已经存在弹性网络接口（ENI）。如果 Lambda 发现现有 ENI，则会尝试重用这些 ENI。否则，Lambda 会创建新的 ENI 来连接到事件源并调用函数。

> **Note**
>
> Lambda 函数始终在 Lambda 服务拥有的 Amazon VPC 中运行。函数的 VPC 配置不会影响事件源映射。只有事件源的网络配置才能决定 Lambda 连接到事件源的方式。

为包含代理的 Amazon VPC 配置安全组。默认情况下，Amazon MQ 使用以下端口：61617（Amazon MQ for ActiveMQ）和 5671（Amazon MQ for RabbitMQ）。

- 入站规则：允许与事件源关联安全组的默认代理端口的所有流量。
- 出站规则：允许所有目标的端口 443 上的所有流量。允许与事件源关联安全组的默认代理端口的所有流量。
- Amazon VPC 端点入站规则：如果您正在使用 Amazon VPC 端点，则与 Amazon VPC 端点关联的安全组，必须允许来自代理安全组的端口 443 上的入站流量。

如果代理使用身份验证，您还可以限制 Secrets Manager 端点的端点策略。要调用 Secrets Manager API，Lambda 会使用函数角色而非 Lambda 服务主体。

Example VPC 端点策略：Secrets Manager 端点

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
                  ]
              },
```

```
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

当您使用 Amazon VPC 端点时，AWS 会使用端点的弹性网络接口（ENI）路由 API 调用来调用函数。Lambda 服务主体需要针对使用这些 ENI 的任何函数调用 `lambda:InvokeFunction`。默认情况下，Amazon VPC 端点具有开放的 IAM 策略，允许对资源进行广泛访问。要在生产环境中使用 Amazon MQ 和 Lambda，您可以将这些策略限制为仅允许特定的主体访问特定的角色和函数。

Example 端点策略：Lambda 端点

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "arn:aws::lambda:us-west-2:123456789012:function:my-function"
        }
    ]
}
```

此外，对于事件源映射，如果您希望与 Lambda 集成的资源在 AWS 账户中部署，则 Lambda 服务主体必须执行 `sts:AssumeRole` 才能代入使用弹性网络接口（ENI）的角色。

Example 端点策略：AWS STS 端点

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
```

```
            },
            "Resource": "arn:aws::iam::123456789012:role/my-role"
        }
    ]
}
```

> **Warning**
>
> 将端点策略限制为仅允许来自组织内部的 API 调用，这将阻止事件源映射正常运行。

创建事件源映射

创建事件源映射以指示 Lambda 将 Amazon MQ 代理中的记录发送到 Lambda 函数。您可以创建多个事件源映射，以使用多个函数处理相同的数据，或使用单个函数处理来自多个源的项目。

要将您的函数配置为从 Amazon MQ 中读取，请添加所需权限并在 Lambda 控制台中创建 MQ 触发器。

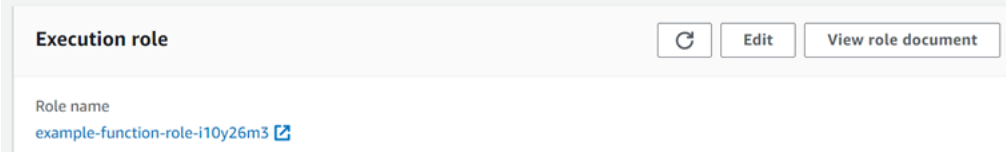
要从 Amazon MQ 代理读取记录，Lambda 函数需要以下权限：通过向函数执行角色添加权限语句，可以授予 Lambda 与 Amazon MQ 代理及其底层资源交互的权限：

- mq:DescribeBroker
- secretsmanager:GetSecretValue
- ec2:CreateNetworkInterface
- ec2:DeleteNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcs
- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

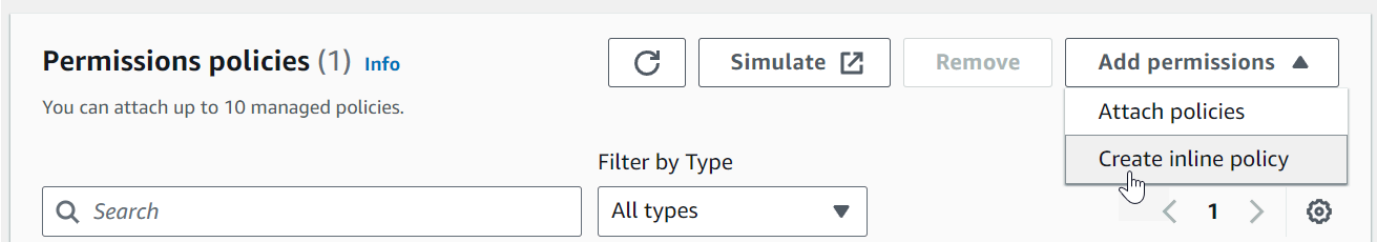
> **Note**
>
> 使用加密的客户托管密钥时，也可以添加 `kms:Decrypt` 权限。

要添加权限并创建触发器

1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数的名称。
3. 选择 Configuration（配置）选项卡，然后选择 Permissions（权限）。
4. 在角色名称下，选择至执行角色的链接。此角色将在 IAM 控制台中打开角色。



5. 选择添加权限，然后选择创建内联策略。



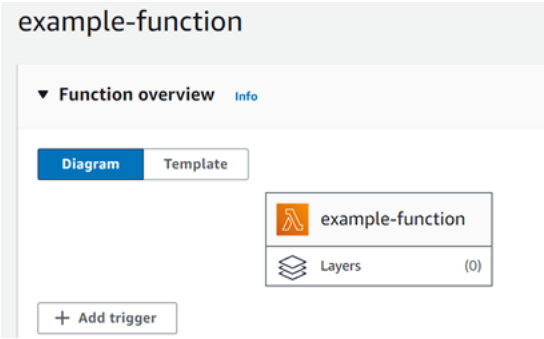
6. 在策略编辑器中，选择 JSON。输入以下策略。您的函数需要这些权限才能从 Amazon MQ 代理读取数据。

```
{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "mq:DescribeBroker",
          "secretsmanager:GetSecretValue",
          "ec2:CreateNetworkInterface",
          "ec2:DeleteNetworkInterface",
          "ec2:DescribeNetworkInterfaces",
          "ec2:DescribeSecurityGroups",
          "ec2:DescribeSubnets",
          "ec2:DescribeVpcs",
```

```
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents"
        ],
        "Resource": "*"
      }
    ]
}
```

> **Note**
>
> 使用加密的客户托管密钥时，还必须添加 `kms:Decrypt` 权限。

7. 选择下一步。输入策略名称，然后选择创建策略。
8. 在 Lambda 控制台中返回您的函数。在 Function overview（函数概览）下，选择 Add trigger（添加触发器）。



9. 选择 MQ 触发器类型。
10. 配置必填选项，然后选择 Add（添加）。

Lambda 支持对 Amazon MQ 事件源使用以下选项：

- MQ broker（MQ 代理）– 选择 Amazon MQ 代理。
- Batch size（批处理大小）– 设置要在单个批次中检索的最大消息数。
- Queue name（队列名称）– 输入要使用的 Amazon MQ 队列。
- Source access configuration（源访问配置）– 输入虚拟主机信息和 Secret Secrets Manager 密钥，用于存储您的代理凭证。
- Enable trigger（启用触发器）– 禁用触发器以停止处理记录。

要启用或禁用触发器（或删除触发器），请在设计器中选择 MQ 触发器。要重新配置触发器，请使用事件源映射 API 操作。

事件源映射 API

所有 Lambda 事件源类型共享相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，只有部分参数适用于 Amazon MQ 和 RabbitMQ。

参数	必需	默认值	备注
BatchSize	否	100	最大值：10000
已启用	否	真实	none
FunctionName	是	不适用	none
FilterCriteria	否	不适用	控制 Lambda 向您的函数发送的事件
MaximumBatchingWindowInSeconds	否	500 毫秒	批处理行为
队列	否	不适用	要使用的 Amazon MQ 代理目标队列的名称。
SourceAccessConfigurations	否	不适用	对于 ActiveMQ 为 BASIC_AUTH 凭证。对于 RabbitMQ，可以同时包含 BASIC_AUTH 凭证和 VIRTUAL_HOST 信息。

筛选来自 Amazon MQ 事件源的事件

您可以使用事件筛选，控制 Lambda 将流或队列中的哪些记录发送给函数。有关事件筛选工作原理的一般信息，请参阅 the section called “事件筛选”。

本节重点介绍 Amazon MQ 事件源的事件筛选。

主题

Amazon MQ 事件筛选基础知识

假设 Amazon MQ 消息队列包含有效 JSON 格式或纯字符串的消息。示例记录如下所示，data 字段中的数据会转换为 Base64 编码字符串。

ActiveMQ

```
{
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
    "messageType": "jms/text-message",
    "deliveryMode": 1,
    "replyTo": null,
    "type": null,
    "expiration": "60000",
    "priority": 1,
    "correlationId": "myJMSCoID",
    "redelivered": false,
    "destination": {
      "physicalName": "testQueue"
    },
    "data":"QUJDOkFBQUE=",
    "timestamp": 1598827811958,
    "brokerInTime": 1598827811958,
    "brokerOutTime": 1598827811959,
    "properties": {
      "index": "1",
      "doAlarm": "false",
      "myCustomProperty": "value"
    }
}
```

RabbitMQ

```
{
    "basicProperties": {
        "contentType": "text/plain",
        "contentEncoding": null,
        "headers": {
```

```
            "header1": {
                "bytes": [
                  118,
                  97,
                  108,
                  117,
                  101,
                  49
                ]
            },
            "header2": {
                "bytes": [
                  118,
                  97,
                  108,
                  117,
                  101,
                  50
                ]
            },
            "numberInHeader": 10
        },
        "deliveryMode": 1,
        "priority": 34,
        "correlationId": null,
        "replyTo": null,
        "expiration": "60000",
        "messageId": null,
        "timestamp": "Jan 1, 1970, 12:33:41 AM",
        "type": null,
        "userId": "AIDACKCEVSQ6C2EXAMPLE",
        "appId": null,
        "clusterId": null,
        "bodySize": 80
        },
    "redelivered": false,
    "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
}
```

对于 Active MQ 和 Rabbit MQ 代理，您可以使用 `data` 键，通过事件筛选来筛选记录。假设 Amazon MQ 队列包含以下 JSON 格式的消息。

```
{
    "timeout": 0,
    "IPAddress": "203.0.113.254"
}
```

要仅筛选 timeout 字段大于 0 的记录，FilterCriteria 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"timeout\" : [ { \"numeric\": [ \">\",
 0] } } ] } }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 Pattern 的值。

```
{
    "data": {
        "timeout": [ { "numeric": [ ">", 0 ] } ]
        }
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
```

```
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }'
```

使用 Amazon MQ，您还可以筛选消息为纯字符串的记录。假设您只想处理消息以“结果：”开头的记录。`FilterCriteria` 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : [ { \"prefix\": \"Result: \" } ] }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 `Pattern` 的值。

```
{
```

```
    "data": [
        {
        "prefix": "Result: "
        }
    ]
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "data" : [ { "prefix": "Result: " } ] }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : [ { "prefix": "Result " } ] }'
```

Amazon MQ 消息必须是 UTF-8 编码的字符串，可以是纯字符串或 JSON 格式。这是因为 Lambda 在应用筛选条件之前将 Amazon MQ 字节数组解码为 UTF-8。如果您的消息使用另一种编码，例如 UTF-16 或 ASCII，或者消息格式与 `FilterCriteria` 格式不匹配，则 Lambda 仅处理元数据筛选条件。下表汇总了具体行为：

传入消息格式	消息属性的筛选条件模式格式	导致的操作
纯字符串	纯字符串	Lambda 根据您的筛选条件进行筛选。
纯字符串	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
纯字符串	有效 JSON	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	纯字符串	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	有效 JSON	Lambda 根据您的筛选条件进行筛选。
非 UTF-8 编码字符串	JSON、纯字符串或无模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。

Amazon MQ 事件源映射错误排查

当 Lambda 函数遇到不可恢复的错误时，您的 Amazon MQ 使用者将停止处理记录。任何其他使用者如果没有遇到相同的错误，都可以继续处理。要确定使用者停止的潜在原因，请检查 `StateTransitionReason` 返回的详细信息中的 `EventSourceMapping` 字段中是否有以下代码：

ESM_CONFIG_NOT_VALID

事件源映射配置无效。

EVENT_SOURCE_AUTHN_ERROR

Lambda 验证事件源失败。

EVENT_SOURCE_AUTHZ_ERROR

Lambda 没有访问事件源所需的权限。

FUNCTION_CONFIG_NOT_VALID

函数的配置无效。

如果记录由于其大小而被 Lambda 丢弃，也将处于未处理状态。Lambda 记录的大小限制为 6 MB。要在函数出错时重新传递消息，您可以使用死信队列 (DLQ)。有关更多信息，请参阅 Apache ActiveMQ 网站上的消息重新传递和 DLQ 处理和 RabbitMQ 网站上的可靠性指南。

> **Note**
>
> Lambda 不支持自定义重新传递策略。相反，Lambda 使用一个策略，其默认值来自 Apache ActiveMQ 网站上的重新传递策略页面。其中，`maximumRedeliveries` 设置为 6。

结合 Amazon MSK 使用 Lambda

> **Note**
>
> 如果想要将数据发送到 Lambda 函数以外的目标，或要在发送数据之前丰富数据，请参阅 Amazon EventBridge Pipes（Amazon EventBridge 管道）。

Amazon Managed Streaming for Apache Kafka (Amazon MSK) 是一项完全托管式服务，可用于构建并运行使用 Apache Kafka 来处理流数据的应用程序。Amazon MSK 简化了运行 Kafka 的集群的设置、扩展和管理。Amazon MSK 还可以更轻松地配置您的应用程序以适用于多个可用区和保证 AWS Identity and Access Management (IAM) 的安全性。Amazon MSK 支持多个开源版本的 Kafka。

Amazon MSK 作为事件源，运行方式与使用 Amazon Simple Queue Service (Amazon SQS) 或 Amazon Kinesis 相似。Lambda 在内部轮询来自事件源的新消息，然后同步调用目标 Lambda 函数。Lambda 批量读取消息，并将这些消息作为事件有效负载提供给您的函数。最大批处理大小是可配置的（默认值为 100 条消息）。有关更多信息，请参阅 批处理行为。

> **Warning**
>
> Lambda 事件源映射至少处理每个事件一次，有可能出现重复处理记录的情况。为避免与重复事件相关的潜在问题，我们强烈建议您将函数代码设为幂等性。要了解更多信息，请参阅 AWS 知识中心的如何使我的 Lambda 函数具有幂等性。

有关如何将 Amazon MSK 配置为事件源的示例，请参阅AWS计算博客上的将 Amazon MSK 用作 AWS Lambda 事件源。要查看完整的教程，请访问 Amazon MSK Labs 中的 Amazon MSK Lambda 集成。

主题

- 示例事件
- 为 Lambda 配置 Amazon MSK 事件源
- 使用 Lambda 处理 Amazon MSK 消息
- 对 Amazon MSK 事件源使用事件筛选
- 捕获 Amazon MSK 事件源丢弃的批处理
- 教程：使用 Amazon MSK 事件源映射调用 Lambda 函数

示例事件

Lambda 调用函数时会在事件参数中发送一批消息。事件负载包含一个消息数组。每个数组项目都包含 Amazon MSK 主题和分区标识符的详细信息，以及时间戳和 base64 编码的消息。


```
{
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
         }
      ]
   }
}
```

为 Lambda 配置 Amazon MSK 事件源

在为 Amazon MSK 集群创建事件源映射之前，必须确保集群及其所在的 VPC 配置正确。您还要确保 Lambda 函数的执行角色具有必要的 IAM 权限。

按照以下章节中的说明配置 Amazon MSK 集群、VPC 和 Lambda 函数。要了解如何创建事件源映射，请参阅the section called “将 Amazon MSK 添加为事件源”。

主题

- MSK 集群身份验证
- 管理 API 访问和权限
- 身份验证和授权错误
- 配置网络安全

MSK 集群身份验证

Lambda 需要访问 Amazon MSK 集群、检索记录和执行其他任务的权限。Amazon MSK 支持通过多种选项来控制客户端对 MSK 集群的访问。

集群访问选项

- 未经身份验证的访问
- SASL/SCRAM 身份验证
- 基于 IAM 角色的身份验证
- 双向 TLS 身份验证
- 配置 mTLS 密钥
- Lambda 如何选择引导代理

未经身份验证的访问

如果没有客户端会通过互联网访问集群，则可以使用未经身份验证访问。

SASL/SCRAM 身份验证

Amazon MSK 支持使用传输层安全性协议（TLS）加密进行简单身份验证和安全层/加盐质疑应答身份验证机制（SASL/SCRAM）身份验证。为使 Lambda 连接到集群，您可以将身份验证凭证（用户名和密码）存储在 AWS Secrets Manager 密钥中。

有关使用 Secrets Manager 的更多信息，请参阅《Amazon Managed Streaming for Apache Kafka 开发人员指南》中的使用 AWS Secrets Manager 进行用户名和密码身份验证。

Amazon MSK 不支持 SASL/PLAIN 身份验证。

基于 IAM 角色的身份验证

您可以使用 IAM 来验证连接到 MSK 集群的客户端的身份。如果 IAM 身份验证在您的 MSK 集群上处于活动状态，并且您没有为身份验证提供密钥，则 Lambda 自动默认使用 IAM 身份验证。要创建和部署用户或基于角色的策略，请使用 IAM 控制台或 API。有关更多信息，请参阅《Amazon Managed Streaming for Apache Kafka 开发人员指南》中的 IAM 访问控制。

要允许 Lambda 连接到 MSK 集群、读取记录和执行其他所需操作，请将以下权限添加到函数的执行角色。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
                "arn:aws:kafka:region:account-id:cluster/cluster-name/cluster-uuid",
                "arn:aws:kafka:region:account-id:topic/cluster-name/cluster-uuid/topic-name",
                "arn:aws:kafka:region:account-id:group/cluster-name/cluster-uuid/consumer-group-id"
            ]
        }
    ]
}
```

您可以将这些权限范围限定为特定集群、主题和组。有关更多信息，请参阅《Amazon Managed Streaming for Apache Kafka 开发人员指南》中的 Amazon MSK Kafka 操作。

双向 TLS 身份验证

双向 TLS（mTLS）在客户端和服务器之间提供双向身份验证。客户端向服务器发送证书以便服务器验证客户端，而服务器又向客户端发送证书以便客户端验证服务器。

对于 Amazon MSK，Lambda 充当客户端。您可以配置客户端证书（作为 Secrets Manager 中的密钥），以使用 MSK 集群中的代理对 Lambda 进行身份验证。客户端证书必须由服务器信任存储中的 CA 签名。MSK 集群会向 Lambda 发送服务器证书，以便使用 Lambda 对代理进行身份验证。服务器证书必须由 AWS 信任存储中的证书颁发机构（CA）签名。

有关如何生成客户端证书的说明，请参阅为作为事件源的 Amazon MSK 引入双向 TLS 身份验证。

Amazon MSK 不支持自签名服务器证书，因为 Amazon MSK 中的所有代理都使用由 Amazon Trust Services CA 签名的公有证书，预设情况下 Lambda 信任这些证书。

有关适用于 Amazon MSK 的 mTLS 的更多信息，请参阅《Amazon Managed Streaming for Apache Kafka 开发人员指南》中的双向 TLS 身份验证。

配置 mTLS 密钥

CLIENT_CERTIFICATE_TLS_AUTH 密钥需要证书字段和私有密钥字段。对于加密的私有密钥，密钥需要私有密钥密码。证书和私有密钥必须采用 PEM 格式。

> **Note**
>
> Lambda 支持 PBES1（而不是 PBES2）私有密钥加密算法。

证书字段必须包含证书列表，首先是客户端证书，然后是任何中间证书，最后是根证书。每个证书都必须按照以下结构在新行中启动：

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager 支持最多包含 65536 字节的密钥，这为长证书链提供了充足的空间。

私有密钥必须采用 PKCS #8 格式，并具有以下结构：

```
-----BEGIN PRIVATE KEY-----
```

```
         <private key contents>
-----END PRIVATE KEY-----
```

对于加密的私有密钥，请使用以下结构：

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

以下示例显示使用加密私有密钥进行 mTLS 身份验证的密钥内容。对于加密的私有密钥，您可以在密钥中包含私有密钥密码。

```
{
 "privateKeyPassword": "testpassword",
 "certificate": "-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
 "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

Lambda 如何选择引导代理

Lambda 根据集群上可用的身份验证方法以及您是否提供用于身份验证的密钥来选择引导代理。如果您为 mTLS 或 SASL/SCRAM 提供了密钥，Lambda 将自动选择该身份验证方法。如果不提供密钥，Lambda 会选择在集群上处于活动状态的最强身份验证方法。下面是 Lambda 选择代理的优先级顺序，从最强到最弱的身份验证：

- mTLS（为 mTLS 提供的密钥）

- SASL/SCRAM（为 SASL/SCRAM 提供的密钥）
- SASL IAM（未提供密钥，IAM 身份验证处于活动状态）
- 未经身份验证的 TLS（未提供密钥，IAM 身份验证未处于活动状态）
- 纯文本（未提供密钥，IAM 身份验证和未经身份验证的 TLS 均未处于活动状态）

> **Note**
>
> 如果 Lambda 无法连接到最安全的代理类型，则 Lambda 不会尝试连接到其他（较弱）代理类型。如果希望 Lambda 选择较弱的代理类型，请停用集群上所有更强的身份验证方法。

管理 API 访问和权限

除了访问 Amazon MSK 集群外，您的函数还需要具有执行各种 Amazon MSK API 操作的权限。您可以为函数的执行角色添加这些权限。如果您的用户需要访问任何 Amazon MSK API 操作，请将所需权限添加到用户或角色的身份策略中。

您可以将以下各项权限手动添加到您的执行角色。您也可以将 AWS 托管式策略 AWSLambdaMSKExecutionRole 附加到您的执行角色。`AWSLambdaMSKExecutionRole` 策略包含了下面列出的所有必需 API 操作和 VPC 权限。

需要的 Lambda 函数执行角色权限

要在 Amazon CloudWatch Logs 中创建日志并将日志存储到日志组，Lambda 函数必须在它的执行角色中具有以下权限：

- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

要使 Lambda 能够代表您访问您的 Amazon MSK 集群，您的 Lambda 函数执行角色必须具有以下权限。

- kafka:DescribeCluster
- kafka:DescribeClusterV2
- kafka:GetBootstrapBrokers
- kafka:DescribeVpcConnection：只有跨账户事件源映射才需要。

- kafka:ListVpcConnections：在执行角色中不是必需的，但对于正在创建跨账户事件源映射的 IAM 主体是必需的。

您只需要添加 kafka:DescribeCluster 或 kafka:DescribeClusterV2 中的一个。对于预调配的 MSK 集群，任何一个权限均有效。对于无服务器 MSK 集群，必须使用 kafka:DescribeClusterV2。

> **Note**
>
> Lambda 最终计划从关联的 AWSLambdaMSKExecutionRole 托管式策略中移除 kafka:DescribeCluster 权限。您使用此策略，则应迁移任何使用 kafka:DescribeCluster 的应用程序，以便改用 kafka:DescribeClusterV2。

VPC 权限

如果只有 VPC 内的用户才能访问您的 Amazon MSK 集群，则您的 Lambda 函数必须具有访问您的 Amazon VPC 资源的权限。这些资源包括您的 VPC、子网、安全组和网络接口。要连接到这些资源，函数的执行角色必须具有以下权限。这些权限包含在 AWSLambdaMSKExecutionRole AWS 托管式策略中。

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups

可选的 Lambda 函数权限

您的 Lambda 函数还可能需要权限来：

- 访问您的 SCRAM 密钥（如果使用 SASL/SCRAM 身份验证）。
- 描述您的 Secrets Manager 密钥。
- 访问 AWS Key Management Service（AWS KMS）客户管理的密钥。
- 将失败调用的记录发送到目标。

Secrets Manager 和 AWS KMS 权限

根据您为 Amazon MSK 代理配置的访问控制类型，Lambda 函数可能需要具有访问您的 SCRAM 密钥（如果使用 SASL/SCRAM 身份验证）或 Secrets Manager 密钥，来解密您的 AWS KMS 客户自主管理型密钥的权限。要连接到这些资源，函数的执行角色必须具有以下权限：

- kafka:ListScramSecrets
- secretsmanager:GetSecretValue
- kms:Decrypt

向执行角色添加权限

请按照以下步骤使用 IAM 控制台将 AWS 托管策略 AWSLambdaMSKExecutionRole 添加到执行角色。

添加 AWS 托管策略

1. 打开 IAM 控制台的 Policies（策略）页面。
2. 在搜索框中，输入策略名称 (AWSLambdaMSKExecutionRole)。
3. 从列表中选择策略，然后依次选择 Policy actions（策略操作）、Attach（附加）。
4. 在 Attach policy（附加策略）页面，从列表中选择您的执行角色，然后选择 Attach policy（附加策略）。

使用 IAM policy 授予用户访问权限

预设情况下，用户和角色无权执行 Amazon MSK API 操作。要向组织或账户中的用户授予访问权限，您可以添加或更新基于身份的策略。有关更多信息，请参阅 Amazon Managed Streaming for Apache Kafka 开发人员指南中的 Amazon MSK 基于身份的策略示例。

身份验证和授权错误

如果缺少使用来自 Amazon MSK 集群的数据所需的任何权限，Lambda 会在 LastProcessingResult 下的事件源映射中显示以下错误消息。

错误消息

- 集群未能授权 Lambda
- SASL 身份验证失败
- 服务器未能通过 Lambda 的身份验证

- 提供的证书或私有密钥无效

集群未能授权 Lambda

对于 SASL/SCRAM 或 mTLS，此错误表明提供的用户不具有以下所有必需的 Kafka 访问控制列表（ACL）权限：

- DescribeConfigs 集群
- 描述组
- 读取组
- 描述主题
- 读取主题

对于 IAM 访问控制，此错误表明函数的执行角色缺少访问组或主题所需的一个或多个权限。查看 the section called “基于 IAM 角色的身份验证” 中的所需权限列表。

当您使用所需的 Kafka 集群权限创建 Kafka ACL 或 IAM policy 时，请将主题和组指定为资源。主题名称必须与事件源映射中的主题一致。组名称必须与事件源映射的 UUID 一致。

向执行角色添加所需的权限后，更改可能需要几分钟才会生效。

SASL 身份验证失败

对于 SASL/SCRAM，此错误表明提供的用户名和密码无效。

对于 IAM 访问控制，此错误表明执行角色缺少 MSK 集群的 `kafka-cluster:Connect` 权限。将此权限添加到该角色并将集群的 Amazon Resource Name（ARN）指定为资源。

您可能会看到此错误间歇性发生。在 TCP 连接数超过 Amazon MSK 服务限额后，集群将拒绝连接。Lambda 会退回并重试，直到连接成功为止。在 Lambda 连接到集群并轮询记录后，最后的处理结果将更改为 0K。

服务器未能通过 Lambda 的身份验证

此错误表明 Amazon MSK Kafka 未能通过 Lambda 的身份验证。出现此错误的可能原因如下：

- 您没有为 mTLS 身份验证提供客户端证书。
- 您提供了客户端证书，但未将代理配置为使用 mTLS。
- 代理不信任客户端证书。

提供的证书或私有密钥无效

此错误表明 Amazon MSK 使用者无法使用提供的证书或私有密钥。确保证书和密钥使用 PEM 格式，并且私有密钥加密使用 PBES1 算法。

配置网络安全

要通过事件源映射向 Lambda 提供对 Amazon MSK 的完全访问权限，集群必须使用公有端点（公有 IP 地址），或者您必须提供对您在其中创建了集群的 Amazon VPC 的访问权限。

将 Amazon MSK 与 Lambda 配合使用时，我们建议您创建 AWS PrivateLinkVPC 端点，并向函数提供对 Amazon VPC 中资源的访问权限。

创建端点以提供对以下资源的访问权限：

- Lambda：为 Lambda 服务主体创建端点。
- AWS STS：为 AWS STS 创建端点，以便服务主体代您代入角色。
- Secrets Manager：如果集群使用 Secrets Manager 来存储凭证，请为 Secrets Manager 创建端点。

也可以在 Amazon VPC 中的每个公有子网上配置 NAT 网关。有关更多信息，请参阅 the section called “VPC 函数的互联网访问权限”。

为 Amazon MSK 创建事件源映射时，Lambda 会检查为 Amazon VPC 配置的子网和安全组是否已经存在弹性网络接口（ENI）。如果 Lambda 发现现有 ENI，则会尝试重用这些 ENI。否则，Lambda 会创建新的 ENI 来连接到事件源并调用函数。

> **Note**
>
> Lambda 函数始终在 Lambda 服务拥有的 Amazon VPC 中运行。函数的 VPC 配置不会影响事件源映射。只有事件源的网络配置才能决定 Lambda 连接到事件源的方式。

为包含集群的 Amazon VPC 配置安全组。默认情况下，Amazon MSK 使用以下端口：纯文本为 9092，TLS 为 9094，SASL 为 9096，IAM 为 9098。

- 入站规则：允许与事件源关联安全组的默认集群端口的所有流量。
- 出站规则：允许所有目标的端口 443 上的所有流量。允许与事件源关联安全组的默认集群的所有流量。

- Amazon VPC 端点入站规则：如果您正在使用 Amazon VPC 端点，则与 Amazon VPC 端点关联的安全组，必须允许来自集群安全组的端口 443 上的入站流量。

如果集群使用身份验证，您还可以限制 Secrets Manager 端点的端点策略。要调用 Secrets Manager API，Lambda 会使用函数角色而非 Lambda 服务主体。

Example VPC 端点策略：Secrets Manager 端点

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
                  ]
              },
              "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
          }
      ]
  }
```

当您使用 Amazon VPC 端点时，AWS 会使用端点的弹性网络接口（ENI）路由 API 调用来调用函数。Lambda 服务主体需要针对使用这些 ENI 的任何函数调用 `lambda:InvokeFunction`。默认情况下，Amazon VPC 端点具有开放的 IAM 策略，允许对资源进行广泛访问。要在生产环境中配合使用 Amazon MSK 和 Lambda，您可以将这些策略限制为仅允许特定的主体访问特定的角色和函数。

Example 端点策略：Lambda 端点

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
```

```
            "Resource": "arn:aws::lambda:us-west-2:123456789012:function:my-function"
        }
    ]
}
```

此外，对于事件源映射，如果您希望与 Lambda 集成的资源在 AWS 账户中部署，则 Lambda 服务主体必须执行 `sts:AssumeRole` 才能代入使用弹性网络接口（ENI）的角色。

Example 端点策略：AWS STS 端点

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "arn:aws::iam::123456789012:role/my-role"
        }
    ]
}
```

Warning

将端点策略限制为仅允许来自组织内部的 API 调用，这将阻止事件源映射正常运行。

使用 Lambda 处理 Amazon MSK 消息

Note

如果想要将数据发送到 Lambda 函数以外的目标，或要在发送数据之前丰富数据，请参阅 Amazon EventBridge Pipes（Amazon EventBridge 管道）。

主题

- 将 Amazon MSK 添加为事件源

- Amazon MSK 配置参数
- 创建跨账户事件源映射
- 将 Amazon MSK 集群用作事件源
- 轮询和流的起始位置
- Amazon CloudWatch 指标
- Amazon MSK 事件源的自动伸缩

将 Amazon MSK 添加为事件源

创建事件源映射，使用 Lambda 控制台、AWS开发工具包，或 AWS Command Line Interface (AWS CLI) 将您的 Amazon MSK 添加为 Lambda 函数触发器。请注意，当您将 Amazon MSK 添加为触发器时，Lambda 将假定 Amazon MSK 集群的 VPC 设置，而不是 Lambda 函数的 VPC 设置。

本节介绍了如何使用 Lambda 控制台和 AWS CLI 创建事件源映射。

先决条件

- 一个 Amazon MSK 集群和一个 Kafka 主题。有关更多信息，请参阅 Amazon Managed Streaming for Apache Kafka 开发人员指南中的开始使用 Amazon MSK。
- 一个有权访问 MSK 集群所用 AWS 资源的执行角色。

可自定义的使用者组 ID

将 Kafka 设置为事件源时，您可以指定使用者组 ID。此使用者组 ID 是您希望 Lambda 函数加入的 Kafka 使用者组的现有标识符。您可以使用此功能将任何正在进行的 Kafka 记录处理设置从其他使用者无缝迁移到 Lambda。

如果指定了使用者组 ID，并且该使用者组中还有其他活跃的轮询器，则 Kafka 会向所有使用者分发消息。换句话说，Lambda 不会收到 Kafka 主题的所有消息。如果希望 Lambda 处理主题中的所有消息，请关闭该使用者组中的任何其他轮询器。

此外，如果指定了使用者组 ID，而 Kafka 找到了具有相同 ID 的有效现有使用者组，则 Lambda 会忽略事件源映射的 `StartingPosition` 参数。相反，Lambda 开始根据使用者组的已提交偏移量处理记录。如果指定了使用者组 ID，而 Kafka 找不到现有使用者组，则 Lambda 会使用指定的 `StartingPosition` 配置事件源。

在所有 Kafka 事件源中，您指定的使用者组 ID 必须是唯一的。在使用指定的使用者组 ID 创建 Kafka 事件源映射后，无法更新此值。

添加 Amazon MSK 触发器（控制台）

按照以下步骤将 Amazon MSK 集群和 Kafka 主题添加为 Lambda 函数的触发器。

将 Amazon MSK 触发器添加到 Lambda 函数（控制台）

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择 Lambda 函数的名称。
3. 在 Function overview（函数概览）下，选择 Add trigger（添加触发器）。
4. 在 Trigger configuration（触发配置）下，执行以下操作：
   a. 选择 MSK 触发器类型。
   b. 对于 MSK cluster（MSK 集群），选择您的集群。
   c. 对于 Batch size（批处理大小），输入要在单个批次中接收的最大消息数。
   d. 对于 Batch window（批处理时段），输入 Lambda 在调用函数之前收集记录所花费的最大秒数。
   e. 对于 Topic name（主题名称），输入 Kafka 主题名称。
   f. （可选）对于 Consumer group ID（使用者组 ID），输入要加入的 Kafka 使用者组的 ID。
   g. （可选）对于起始位置，选择最新即可从最新记录开始读取流，选择最早即可从最早的可用记录开始读取流，选择在时间戳处即可从指定的时间戳开始读取流。
   h. （可选）对于 Authentication（身份验证），选择用于通过 MSK 集群中的代理进行身份验证的密钥。
   i. 要在禁用状态下创建触发器以进行测试（推荐），请清除 Enable trigger（启用触发器）。或者，要立即启用该触发器，请选择 Enable trigger（启用触发器）。
5. 要创建触发器，请选择 Add（添加）。

添加 Amazon MSK 触发器（AWS CLI）

使用以下示例 AWS CLI 命令为 Lambda 函数创建和查看 Amazon MSK 触发器。

使用 AWS CLI 创建触发器

Example — 为使用 IAM 身份验证的集群创建事件源映射

以下示例使用 create-event-source-mapping AWS CLI 命令将名为 `my-kafka-function` 的 Lambda 函数映射至名为 `AWSKafkaTopic` 的 Kafka 主题。将主题的起始位置设置为 `LATEST`。当集群使用基于 IAM 角色的身份验证时，您不需要 SourceAccessConfiguration 对象。例如：

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example — 为使用 SASL/SCRAM 身份验证的集群创建事件源映射

如果集群使用 SASL/SCRAM 身份验证，则必须包含指定 SASL_SCRAM_512_AUTH 的 SourceAccessConfiguration 对象以及 Secrets Manager 密钥 ARN。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

Example — 为使用 mTLS 身份验证的集群创建事件源映射

如果集群使用 mTLS 身份验证，则必须包含指定 CLIENT_CERTIFICATE_TLS_AUTH 的 SourceAccessConfiguration 对象以及 Secrets Manager 密钥 ARN。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

有关更多信息，请参阅 CreateEventSourceMapping API 参考文档。

使用 AWS CLI 查看状态

以下示例使用 get-event-source-mapping AWS CLI 命令来描述您创建的事件源映射的状态。

```
aws lambda get-event-source-mapping \
  --uuid 6d9bce8e-836b-442c-8070-74e77903c815
```

Amazon MSK 配置参数

所有 Lambda 事件源类型共享相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，只有部分参数适用于Amazon MSK。

参数	必需	默认值	注意
AmazonManagedKafkaEventSourceConfig	否	包含 ConsumerGroupId 字段，该字段默认为唯一值。	只能在 Create（创建）设置
BatchSize	否	100	最大值：10000
已启用	否	已启用	none
EventSourceArn	Y	不适用	只能在 Create（创建）设置
FunctionName	是	不适用	none
FilterCriteria	否	不适用	控制 Lambda 向您的函数发送的事件
MaximumBatchingWindowInSeconds	否	500 毫秒	批处理行为
SourceAccessConfigurations	否	无凭证	事件源的 SASL/SCRAM 或 CLIENT_CERTIFICATE_TLS_AUTH (MutualTLS) 身份验证凭证
StartingPosition	Y	不适用	AT_TIMESTAMP、TRIM_HORIZON 或 LATEST

参数	必需	默认值	注意
			只能在 Create（创建）设置
StartingPositionTimestamp	否	不适用	当 StartingPosition 设置为 AT_TIMESTAMP 时，为必需项
主题	Y	不适用	Kafka 主题名称 只能在 Create（创建）设置

创建跨账户事件源映射

您可以使用多 VPC 私有连接将 Lambda 函数连接到不同 AWS 账户 中的预置 MSK 集群。多 VPC 连接使用 AWS PrivateLink，可将所有流量保持在 AWS 网络内。

> **Note**
>
> 您无法为无服务器 MSK 集群创建跨账户事件源映射。

要创建跨账户事件源映射，必须先为 MSK 集群配置多 VPC 连接。创建事件源映射时，请使用托管 VPC 连接 ARN 而非集群 ARN，如以下示例所示。CreateEventSourceMapping 操作也因 MSK 集群使用的身份验证类型而异。

Example — 为使用 IAM 身份验证的集群创建跨账户事件源映射

当集群使用基于 IAM 角色的身份验证时，您不需要 SourceAccessConfiguration 对象。例如：

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example — 为使用 SASL/SCRAM 身份验证的集群创建跨账户事件源映射

如果集群使用 SASL/SCRAM 身份验证，则必须包含指定 SASL_SCRAM_512_AUTH 的 SourceAccessConfiguration 对象以及 Secrets Manager 密钥 ARN。

有两种方法可以通过 SASL/SCRAM 身份验证将密钥用于跨账户 Amazon MSK 事件源映射：

- 在 Lambda 函数账户中创建密钥并将其与集群密钥同步。创建轮换以使两个密钥保持同步。此选项允许您控制来自函数账户的密钥。
- 使用与 MSK 集群关联的密钥。此密钥必须可用于跨账户存取 Lambda 函数账户。有关更多信息，请参阅不同账户中用户对 AWS Secrets Manager 密钥的访问权限。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

Example — 为使用 mTLS 身份验证的集群创建跨账户事件源映射

如果集群使用 mTLS 身份验证，则必须包含指定 CLIENT_CERTIFICATE_TLS_AUTH 的 SourceAccessConfiguration 对象以及 Secrets Manager 密钥 ARN。密钥可以存储在集群账户或 Lambda 函数账户中。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

将 Amazon MSK 集群用作事件源

当您添加 Apache Kafka 或 Amazon MSK 集群作为 Lambda 函数的触发器时，该集群将用作事件源。

Lambda 根据您指定的 `StartingPosition`，从您在 CreateEventSourceMapping 请求中指定为 `Topics` 的 Kafka 主题读取事件数据。成功进行处理后，会将 Kafka 主题提交给 Kafka 集群。

如果您指定 `StartingPosition` 作为 `LATEST`，则 Lambda 开始读取主题下每个分区中的最新消息。由于在触发器配置后 Lambda 开始读取消息之前可能会有一些延迟，因此 Lambda 不会读取在此窗口中生成的任何消息。

Lambda 按顺序读取每个 Kafka 主题分区的消息。单个 Lambda 负载可以包含来自多个分区的消息。当有更多记录可用时，Lambda 根据您在 CreateEventSourceMapping 中指定的 `BatchSize` 值，继续对记录进行批处理，直到函数赶上主题的速度。

Lambda 处理各个批次后，会提交该批次中消息的偏移量。如果函数为批处理中的任何消息返回错误，Lambda 将重试整批消息，直到处理成功或消息过期为止。您可以将所有重试都失败的记录发送到失败时的目标，以供日后处理。

> **Note**
>
> 尽管 Lambda 函数的最大超时限制通常为 15 分钟，但 Amazon MSK、自行管理的 Apache Kafka、Amazon DocumentDB、Amazon MQ for ActiveMQ 和 RabbitMQ 的事件源映射，仅支持最大超时限制为 14 分钟的函数。此约束可确保事件源映射可以正确处理函数错误和重试。

轮询和流的起始位置

请注意，事件源映射创建和更新期间的流轮询最终是一致的。

- 在事件源映射创建期间，可能需要几分钟才能开始轮询来自流的事件。
- 在事件源映射更新期间，可能需要几分钟才能停止和重新开始轮询来自流的事件。

此行为意味着，如果你指定 `LATEST` 作为流的起始位置，事件源映射可能会在创建或更新期间错过事件。为确保不会错过任何事件，请将流的起始位置指定为 `TRIM_HORIZON` 或 `AT_TIMESTAMP`。

Amazon CloudWatch 指标

Lambda 会在您的函数处理记录时发出 `OffsetLag` 指标。此指标的值是写入 Kafka 事件源主题的最后一条记录与函数的使用者组处理的最后一条记录之间的偏移量差值。您可以使用 `OffsetLag` 来估计添加记录和使用者组处理记录之间的延迟。

如果 `OffsetLag` 呈上升趋势，则可能表明函数的使用者组中的轮询器存在问题。有关更多信息，请参阅 查看 Lambda 函数的指标。

Amazon MSK 事件源的自动伸缩

当您最初创建 Amazon MSK 事件源时，Lambda 会分配一个使用者来处理 Kafka 主题中的所有分区。每个使用者都使用多个并行运行的处理器来处理增加的工作负载。此外，Lambda 会根据工作负载自动增加或缩减使用者的数量。要保留每个分区中的消息顺序，使用者的最大数量为主题中每个分区一个使用者。

Lambda 会按一分钟的间隔时间来评估主题中所有分区的使用者偏移滞后。如果延迟太高，则分区接收消息的速度比 Lambda 处理消息的速度更快。如有必要，Lambda 会在主题中添加或删除使用者。增加或移除使用者的扩缩过程会在评估完成后的三分钟内进行。

如果目标 Lambda 函数受到限制，Lambda 会减少使用者的数量。此操作通过减少使用者可以检索和发送到函数的消息数来减少函数的工作负载。

要监控 Kafka 主题的吞吐量，请查看 Lambda 在您的函数处理记录时发出的偏移滞后指标。

要检查并行发生的函数调用次数，还可以监控函数的并发指标。

对 Amazon MSK 事件源使用事件筛选

您可以使用事件筛选，控制 Lambda 将流或队列中的哪些记录发送给函数。有关事件筛选工作原理的一般信息，请参阅 the section called “事件筛选”。

本节重点介绍 Amazon MSK 事件源的事件筛选。

主题

- Amazon MSK 事件筛选基础知识

Amazon MSK 事件筛选基础知识

假设创建者以有效的 JSON 格式或纯字符串的形式将消息写入 Amazon MSK 集群中的主题。示例记录将如下所示，`value` 字段中的消息会转换为 Base64 编码字符串。

```
{
    "mytopic-0":[
        {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
```

```
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

假设 Apache Kafka 创建器以如下 JSON 格式将消息写入主题。

```
{
    "device_ID": "AB1234",
    "session":{
        "start_time": "yyyy-mm-ddThh:mm:ss",
        "duration": 162
    }
}
```

您可以使用 `value` 键筛选记录。假设您只想筛选 `device_ID` 以字母 AB 开头的记录。`FilterCriteria` 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\": \"AB\" } ] } }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 `Pattern` 的值。

```
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
      }
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }'
```

通过 Amazon MSK，您还可以筛选消息为纯字符串的记录。假设您想忽略字符串为“错误”的消息。`FilterCriteria` 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 Pattern 的值。

```
{
    "value": [
        {
        "anything-but": [ "error" ]
        }
    ]
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

Amazon MSK 消息必须是 UTF-8 编码的字符串，可以是纯字符串或 JSON 格式。这是因为 Lambda 在应用筛选条件之前将 Amazon MSK 字节数组解码为 UTF-8。如果您的消息使用另一种编码，例如 UTF-16 或 ASCII，或者消息格式与 `FilterCriteria` 格式不匹配，则 Lambda 仅处理元数据筛选条件。下表汇总了具体行为：

传入消息格式	消息属性的筛选条件模式格式	导致的操作
纯字符串	纯字符串	Lambda 根据您的筛选条件进行筛选。
纯字符串	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
纯字符串	有效 JSON	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	纯字符串	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	有效 JSON	Lambda 根据您的筛选条件进行筛选。
非 UTF-8 编码字符串	JSON、纯字符串或无模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。

捕获 Amazon MSK 事件源丢弃的批处理

要保留失败的事件源映射调用的记录，请在函数的事件源映射中添加一个目标。发送到目标的每条记录都是一个 JSON 文档，其中包含有关失败调用的元数据。您可以将任何 Amazon SNS 主题、Amazon SQS 队列或 S3 存储桶配置为目标。您的执行角色必须具有目标的权限：

- 对于 SQS 目标：sqs:SendMessage
- 对于 SNS 目标：sns:Publish
- 对于 S3 存储桶目标：s3:PutObject 和 s3:ListBuckets

您必须在 Amazon MSK 集群 VPC 中为故障目标服务部署 VPC 端点。

此外，如果您在目标上配置了 KMS 密钥，则根据具体目标类型，Lambda 需要以下权限：

- 如果您已使用自己的 KMS 密钥为 S3 目标启用加密，则需要 kms:GenerateDataKey。如果 KMS 密钥和 S3 存储桶目标与您的 Lambda 函数和执行角色位于不同的账户中，请将 KMS 密钥配置为信任执行角色以允许 kms:GenerateDataKey。
- 如果您已使用自己的 KMS 密钥为 SQS 目标启用加密，则需要 kms:Decrypt 和 kms:GenerateDataKey。如果 KMS 密钥和 SQS 队列目标与您的 Lambda 函数和执行角色位于不同的账户中，请将 KMS 密钥配置为信任执行角色以允许 kms:Decrypt、kms:GenerateDataKey、kms:DescribeKey 和 kms:ReEncrypt。
- 如果您已使用自己的 KMS 密钥为 SNS 目标启用加密，则需要 kms:Decrypt 和 kms:GenerateDataKey。如果 KMS 密钥和 SNS 主题目标与您的 Lambda 函数和执行角色位于不同的账户中，请将 KMS 密钥配置为信任执行角色以允许 kms:Decrypt、kms:GenerateDataKey、kms:DescribeKey 和 kms:ReEncrypt。

为 Amazon MSK 事件源映射配置失败时的目标

要使用控制台配置失败时的目标，请执行以下步骤：

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 在 Function overview (函数概览) 下，选择 Add destination (添加目标)。
4. 对于源，请选择事件源映射调用。
5. 对于事件源映射，请选择为此函数配置的事件源。
6. 在条件中，选择失败时。对于事件源映射调用，这是唯一可接受的条件。

7. 对于目标类型，请选择 Lambda 要发送调用记录的目标类型。
8. 对于 Destination (目标)，请选择一个资源。
9. 选择保存。

您还可以使用 AWS CLI 配置失败时的目标。例如，以 create-event-source-mapping 命令将带有 SQS 失败时目标的事件源映射添加到 `MyFunction`：

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

以下 update-event-source-mapping 命令将 S3 失败时目标添加到与输入 `uuid` 关联的事件源：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

要移除目标，请提供一个空字符串作为 `destination-config` 参数的实际参数：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

SNS 和 SQS 示例调用记录

以下示例显示了 Lambda 在 Kafka 事件源调用失败时向 SNS 主题或 SQS 队列目标发送的内容。`recordsInfo` 下面的每个密钥都包含 Kafka 主题和分区，用连字符分隔。例如，对于密钥 `"Topic-0"`，`Topic` 是 Kafka 主题，`0` 是分区。对于每个主题和分区，可以使用偏移量和时间戳数据来查找原始调用记录。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
```

```
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    }
}
```

S3 目标示例调用记录

对于 S3 目标，Lambda 会将整个调用记录以及元数据发送到目标。以下示例显示了 Lambda 因调用 Kafka 事件源失败而向 S3 存储桶目标发送消息。除了针对 SQS 和 SNS 目标的上一示例中的所有字段外，`payload` 字段还包含作为转义 JSON 字符串的原始调用记录。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
```

```
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

> **Tip**
>
> 我们建议在目标存储桶上启用 S3 版本控制。

教程：使用 Amazon MSK 事件源映射调用 Lambda 函数

在本教程中，您将执行以下操作：

- 在与现有 Amazon MSK 集群相同的 AWS 账户中创建 Lambda 函数。
- 为 Lambda 配置联网和身份验证，以便与 Amazon MSK 通信。
- 设置 Lambda Amazon MSK 事件源映射，其在主题中出现事件时运行您的 Lambda 函数。

完成这些步骤后，当事件发送到 Amazon MSK 时，您将能够设置 Lambda 函数，以使用自己的自定义 Lambda 代码自动处理这些事件。

您可以用这项功能做什么？

示例解决方案：使用 MSK 事件源映射向您的客户提供实时比分。

请考虑以下应用场景：您的公司托管了一个 Web 应用程序，您的客户可以在其中查看有关直播活动（例如体育比赛）的信息。比赛的信息更新将通过 Amazon MSK 上的 Kafka 主题提供给您的团队。您想设计一个解决方案，使用来自 MSK 主题的更新在您开发的应用程序中向客户提供直播活动的更新视图。您已决定采用以下设计方法：您的客户端应用程序将与在 AWS 中托管的无服务器后端进行通信。客户端将使用 Amazon API Gateway WebSocket API 通过 websocket 会话进行连接。

在此解决方案中，您需要一个组件来读取 MSK 事件，执行一些自定义逻辑为应用程序层准备这些事件，然后将该信息转发到 API Gateway API。您可以使用 AWS Lambda 实现此组件，方法是在 Lambda 函数中提供自定义逻辑，然后使用 AWS Lambda Amazon MSK 事件源映射对其进行调用。

有关使用 Amazon API Gateway WebSocket API 实施解决方案的更多信息，请参阅 API Gateway 文档中的 WebSocket API 教程。

先决条件

具有以下预配置资源的 AWS 账户：

要满足这些先决条件，建议按照 Amazon MSK 文档中的 Getting started using Amazon MSK 进行操作。

- Amazon MSK 集群。请参阅 Getting started using Amazon MSK 中的 Create an Amazon MSK cluster。
- 以下配置：

  - 确保在集群安全设置中启用基于 IAM 角色的身份验证。这会将您的 Lambda 函数限制为仅访问所需的 Amazon MSK 资源，从而提高安全性。默认情况下会对新的 Amazon MSK 集群启用此设置。
  - 确保集群网络设置中的公有访问已关闭。通过限制处理数据的中介数量，限制 Amazon MSK 集群对互联网的访问，以提高您的安全性。默认情况下会对新的 Amazon MSK 集群启用此设置。
- 您的 Amazon MSK 集群中用于此解决方案的 Kafka 主题。请参阅 Getting started using Amazon MSK 中的 Create a topic。
- 设置 Kafka 管理主机以从您的 Kafka 集群检索信息并将 Kafka 事件发送到您的主题进行测试，例如安装了 Kafka 管理 CLI 和 Amazon MSK IAM 库的 Amazon EC2 实例。请参阅 Getting started using Amazon MSK 中的 Create a client machine。

设置完这些资源后，请从您的 AWS 账户中收集以下信息，以确认您已准备好继续。

- Amazon MSK 集群的名称。您可以在 Amazon MSK 控制台中找到这些信息。
- 集群 UUID，您的 Amazon MSK 集群 ARN 的一部分，您可以在 Amazon MSK 控制台中找到它。按照 Amazon MSK 文档中 Listing clusters 中的步骤查找此信息。
- 与您的 Amazon MSK 集群关联的安全组。您可以在 Amazon MSK 控制台中找到这些信息。在以下步骤中，这些安全组称为 *`clusterSecurityGroups`*。
- 包含 Amazon MSK 集群的 Amazon VPC 的 ID。您可以通过在 Amazon MSK 控制台中识别与您的 Amazon MSK 集群关联的子网，然后在 Amazon VPC 控制台中识别与该子网关联的 Amazon VPC 来找到此信息。
- 解决方案中使用的 Kafka 主题名称。您可以通过从 Kafka 管理主机使用 Kafka `topics` CLI 调用您的 Amazon MSK 集群来找到此信息。有关主题 CLI 的更多信息，请参阅 Kafka 文档中的 Adding and removing topics。
- 您的 Kafka 主题使用者组的名称，适合您的 Lambda 函数使用。Lambda 可以自动创建该组，因此您无需使用 Kafka CLI 创建该组。如果您确实需要管理使用者组，了解有关使用者组 CLI 的更多信息，则请参阅 Kafka 文档中的 Managing Consumer Groups。

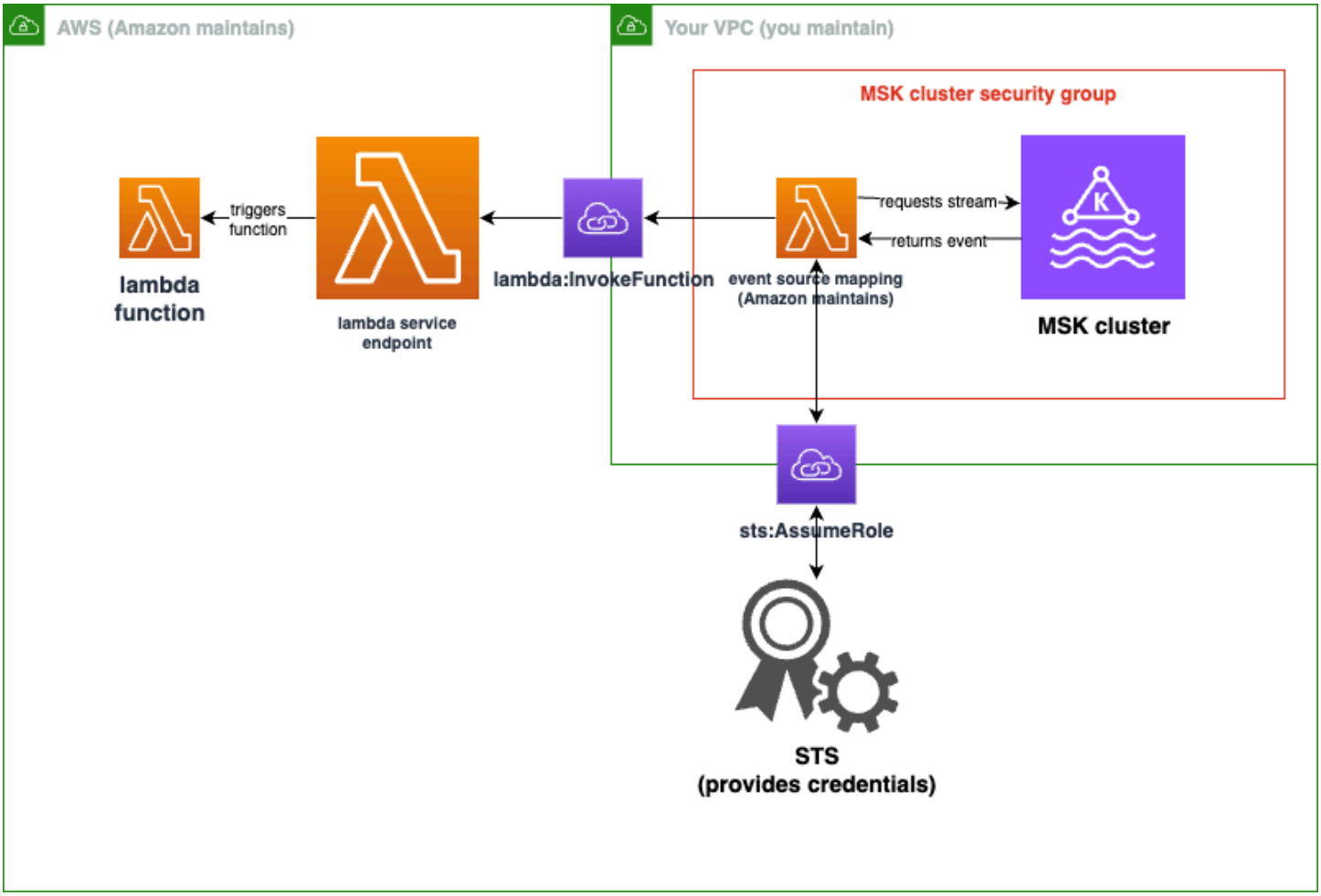
您 AWS 账户中有以下权限：

- 创建和管理 Lambda 函数的权限。
- 创建 IAM 策略并将其与您的 Lambda 函数关联的权限。
- 在托管您的 Amazon MSK 集群的 Amazon VPC 中创建 Amazon VPC 端点和更改网络配置的权限。

为 Lambda 配置网络连接以与 Amazon MSK 通信

使用 AWS PrivateLink 连接 Lambda 和 Amazon MSK。您可以通过在 Amazon VPC 控制台中创建接口 Amazon VPC 端点来实现此目的。有关联网配置的更多信息，请参阅 the section called “配置网络安全”。

当 Amazon MSK 事件源映射代表 Lambda 函数运行时，它会担任 Lambda 函数的执行角色。此 IAM 角色授权映射访问受 IAM 保护的资源，例如您的 Amazon MSK 集群。尽管这些组件共享执行角色，但 Amazon MSK 映射和您的 Lambda 函数对各自的任务有不同的连接要求，如下图所示。



您的事件源映射属于 Amazon MSK 集群安全组。在此联网步骤中，从您的 Amazon MSK 集群 VPC 创建 Amazon VPC 端点，将事件源映射连接到 Lambda 和 STS 服务。保护这些端点，以接受来自您的 Amazon MSK 集群安全组的流量。然后，调整 Amazon MSK 集群安全组，以允许事件源映射与 Amazon MSK 集群进行通信。

您可以使用 AWS Management Console 配置以下步骤。

配置接口 Amazon VPC 端点以连接 Lambda 和 Amazon MSK

1. 为您的接口 Amazon VPC 端点创建一个安全组 *endpointSecurityGroup*，以允许来自 *clusterSecurityGroups* 端口 443 的入站 TCP 流量。按照 Amazon EC2 文档中创建安全组的步骤创建安全组。然后，按照 Amazon EC2 文档中向安全组添加规则的步骤添加相应的规则。

   使用以下信息创建安全组：

   添加入站规则时，为 *clusterSecurityGroups* 中的每个安全组创建一条规则。对于每条规则：

   - 对于类型，选择 HTTPS。
   - 对于源，选择其中一个 *clusterSecurityGroups*。

2. 创建一个端点，将 Lambda 服务连接到包含 Amazon MSK 集群的 Amazon VPC。按照创建接口端点中的步骤进行操作。

   使用以下信息创建接口端点：

   - 对于服务名称，选择 `com.amazonaws.`*regionName*`.lambda`，其中 *regionName* 托管您的 Lambda 函数。
   - 对于 VPC，选择包含您的 Amazon MSK 集群的 Amazon VPC。
   - 对于安全组，选择您之前创建的 *endpointSecurityGroup*。
   - 对于子网，选择托管您的 Amazon MSK 集群的子网。
   - 对于策略，请提供以下策略文档，其保护端点以供 Lambda 服务主体用于执行 `lambda:InvokeFunction` 操作。

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
```

```
    ]
}
```

- 确保启用 DNS 名称保持设置状态。

3. 创建一个端点，将 AWS STS 服务连接到包含 Amazon MSK 集群的 Amazon VPC。按照创建接口端点中的步骤进行操作。

   使用以下信息创建接口端点：

   - 对于服务名称，选择 AWS STS。
   - 对于 VPC，选择包含您的 Amazon MSK 集群的 Amazon VPC。
   - 对于安全组，选择 *endpointSecurityGroup*。
   - 对于子网，选择托管您的 Amazon MSK 集群的子网。
   - 对于策略，请提供以下策略文档，其保护端点以供 Lambda 服务主体用于执行 `sts:AssumeRole` 操作。

   ```
   {
       "Statement": [
           {
               "Action": "sts:AssumeRole",
               "Effect": "Allow",
               "Principal": {
                   "Service": [
                       "lambda.amazonaws.com"
                   ]
               },
               "Resource": "*"
           }
       ]
   }
   ```

   - 确保启用 DNS 名称保持设置状态。

4. 对于与您的 Amazon MSK 集群关联的每个安全组（即 *clusterSecurityGroups*），允许执行以下操作：

   - 允许端口 9098 上到所有 *clusterSecurityGroups*（包括其自身内部）的所有入站和出站 TCP 流量。
   - 允许端口 443 上的所有出站 TCP 流量。

默认安全组规则允许部分流量，因此，如果您的集群连接到单个安全组，并且该组有默认规则，则不需要其他规则。要调整安全组规则，请按照 Amazon EC2 文档中向安全组添加规则的步骤进行操作。

使用以下信息向您的安全组添加规则：

- 对于端口 9098 的每条入站规则或出站规则，请提供
  - 对于 Type (类型)，选择 Custom TCP (自定义 TCP)。
  - 对于端口范围，请提供 9098。
  - 对于源，提供其中一个 *clusterSecurityGroups*。
- 对于端口 443 的每条入站规则的类型，选择 HTTPS。

为 Lambda 创建 IAM 角色，以从您的 Amazon MSK 主题中读取

确定 Lambda 从 Amazon MSK 主题读取的身份验证要求，然后在策略中进行定义。创建一个角色 *lambdaAuthRole*，授权 Lambda 使用这些权限。使用 `kafka-cluster` IAM 操作在您的 Amazon MSK 集群上授权操作。然后，授权 Lambda 执行发现和连接到您的 Amazon MSK 集群所需的 Amazon MSK `kafka` 和 Amazon EC2 操作以及 CloudWatch 操作，这样 Lambda 便可记录其所执行的操作。

描述 Lambda 从 Amazon MSK 读取的身份验证要求

1. 编写一个 IAM 策略文档（JSON 文档）*clusterAuthPolicy*，允许 Lambda 使用您的 Kafka 使用者组从 Amazon MSK 集群中的 Kafka 主题进行读取。Lambda 要求在读取时设置一个 Kafka 使用者组。

   修改以下模板以符合您的先决条件：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
                "kafka-cluster:DescribeTopic",
```

```
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
                "arn:aws:kafka:region:account-id:cluster/mskClusterName/cluster-
uuid",
                "arn:aws:kafka:region:account-id:topic/mskClusterName/cluster-
uuid/mskTopicName",
                "arn:aws:kafka:region:account-id:group/mskClusterName/cluster-
uuid/mskGroupName"
            ]
        }
    ]
}
```

有关更多信息，请参阅 the section called “基于 IAM 角色的身份验证”。编写策略时：

- 对于 *region* 和 *account-id*，提供托管您的 Amazon MSK 集群的区域和账户 ID。
- 对于 *mskClusterName*，提供您的 Amazon MSK 集群的名称。
- 对于 *cluster-uuid*，提供您的 Amazon MSK 集群的 ARN 中 UUID。
- 对于 *mskTopicName*，提供您的 Kafka 主题的名称。
- 对于 *mskGroupName*，提供您的 Kafka 使用者组的名称。

2. 确定 Lambda 发现和连接您的 Amazon MSK 集群所需的 Amazon MSK、Amazon EC2 和 CloudWatch 权限，并记录这些事件。

   `AWSLambdaMSKExecutionRole` 托管策略宽松地定义所需的权限。在以下步骤中使用该策略。

   在生产环境中，评测 `AWSLambdaMSKExecutionRole` 以根据最低权限原则限制您的执行角色策略，然后为您的角色编写一个策略来取代此托管策略。

有关 IAM 策略语言的详细信息，请参阅 IAM 文档。

现在，您已经编写了策略文档，请创建一个 IAM 策略，这样便可将其附加到您的角色中。您可以使用控制台按以下步骤完成此操作。

从策略文档创建 IAM 策略

1. 登录 AWS Management Console，然后通过以下网址打开 IAM 控制台：https://console.aws.amazon.com/iam/。

2. 在左侧的导航窗格中，选择策略。
3. 选择创建策略。
4. 在策略编辑器部分，选择 JSON 选项。
5. 粘贴 *clusterAuthPolicy*。
6. 向策略添加完权限后，选择下一步。
7. 在查看和创建页面上，为创建的策略键入策略名称和描述（可选）。查看此策略中定义的权限以查看策略授予的权限。
8. 选择创建策略可保存新策略。

有关更多信息，请参阅 IAM 文档中的创建 IAM 策略。

既然您已经有了适当的 IAM 策略，请创建一个角色并将其附加到该角色。您可以使用控制台按以下步骤完成此操作。

在 IAM 控制台中创建执行角色

1. 在 IAM 控制台中，打开 Roles（角色）页面。
2. 选择 Create role（创建角色）。
3. 在可信实体类型下，选择 AWS 服务。
4. 在 Use case（使用案例）下，选择 Lambda。
5. 选择下一步。
6. 选择以下策略：
   - *clusterAuthPolicy*
   - AWSLambdaMSKExecutionRole
7. 选择下一步。
8. 对于角色名称，输入 *lambdaAuthRole*，然后选择创建角色。

有关更多信息，请参阅 the section called “执行角色（函数访问其他资源的权限）”。

创建 Lambda 函数以从您的 Amazon MSK 主题中读取

创建配置为使用您的 IAM 角色的 Lambda 函数。您可以使用控制台创建您的 Lambda 函数。

使用您的身份验证配置创建 Lambda 函数

1. 打开 Lambda 控制台并从标题中选择创建函数。
2. 选择从头开始编写。
3. 对于函数名称，请提供您选择的相应名称。
4. 对于运行时，选择最新支持版本的 Node.js 以使用本教程中提供的代码。
5. 选择更改默认执行角色。
6. 选择使用现有角色。
7. 对于现有角色，选择 *lambdaAuthRole*。

在生产环境中，您通常需要向 Lambda 函数的执行角色添加更多策略，以便有效地处理您的 Amazon MSK 事件。有关向角色添加策略的更多信息，请参阅 IAM 文档中的添加或删除身份权限。

创建到 Lambda 函数的事件源映射

您的 Amazon MSK 事件源映射为 Lambda 服务提供了必要的信息，以在发生相应 Amazon MSK 事件时调用您的 Lambda。您可以使用控制台创建 Amazon MSK 映射。创建 Lambda 触发器，然后事件源映射会自动设置。

创建 Lambda 触发器（和事件源映射）

1. 导航到您的 Lambda 函数的概述页面。
2. 在函数概述部分中，选择左下角的添加触发器。
3. 在选择源下拉列表中，选择 Amazon MSK。
4. 请勿设置身份验证。
5. 对于 MSK 集群，选择集群的名称。
6. 对于批次大小，输入 1。此步骤使该功能更易于测试，但并非生产中的理想值。
7. 对于主题名称，输入 Kafka 主题名称。
8. 对于使用者组 ID，请提供您的 Kafka 使用者组的 ID。

更新您的 Lambda 函数以读取流数据

Lambda 通过事件方法参数提供有关 Kafka 事件的信息。有关 Amazon MSK 事件的示例结构，请参阅 the section called “ 示例事件”。在您了解如何解读 Lambda 转发的 Amazon MSK 事件后，您可以修改您的 Lambda 函数代码以使用其提供的信息。

向您的 Lambda 函数提供以下代码，用于记录 Lambda Amazon MSK 事件的内容以进行测试：

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 .NET 将 Amazon MSK 事件与 Lambda 结合使用。

```
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KafkaEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace MSKLambda;

public class Function
{


    /// <param name="input">The event for the Lambda function handler to
 process.</param>
    /// <param name="context">The ILambdaContext that provides methods for
 logging and describing the Lambda environment.</param>
    /// <returns></returns>
    public void FunctionHandler(KafkaEvent evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Records)
        {
            Console.WriteLine("Key:" + record.Key);
            foreach (var eventRecord in record.Value)
            {
```

```
                var valueBytes = eventRecord.Value.ToArray();
                var valueText = Encoding.UTF8.GetString(valueBytes);

                Console.WriteLine("Message:" + valueText);
            }
        }
    }


}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Go 将 Amazon MSK 事件与 Lambda 结合使用。

```
package main

import (
 "encoding/base64"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.KafkaEvent) {
 for key, records := range event.Records {
  fmt.Println("Key:", key)

  for _, record := range records {
   fmt.Println("Record:", record)
```

```
   decodedValue, _ := base64.StdEncoding.DecodeString(record.Value)
   message := string(decodedValue)
   fmt.Println("Message:", message)
  }
 }
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Java 将 Amazon MSK 事件与 Lambda 结合使用。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent.KafkaEventRecord;

import java.util.Base64;
import java.util.Map;

public class Example implements RequestHandler<KafkaEvent, Void> {

    @Override
    public Void handleRequest(KafkaEvent event, Context context) {
        for (Map.Entry<String, java.util.List<KafkaEventRecord>> entry :
 event.getRecords().entrySet()) {
            String key = entry.getKey();
            System.out.println("Key: " + key);

            for (KafkaEventRecord record : entry.getValue()) {
```

```
                System.out.println("Record: " + record);

                byte[] value = Base64.getDecoder().decode(record.getValue());
                String message = new String(value);
                System.out.println("Message: " + message);
            }
        }

        return null;
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 JavaScript 将 Amazon MSK 事件与 Lambda 结合使用。

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 PHP 将 Amazon MSK 事件与 Lambda 结合使用。

```
<?php
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

// using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kafka\KafkaEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): void
    {
        $kafkaEvent = new KafkaEvent($event);
        $this->logger->info("Processing records");
        $records = $kafkaEvent->getRecords();
```

```
        foreach ($records as $record) {
            try {
                $key = $record->getKey();
                $this->logger->info("Key: $key");

                $values = $record->getValue();
                $this->logger->info(json_encode($values));

                foreach ($values as $value) {
                    $this->logger->info("Value: $value");
                }

            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Python 将 Amazon MSK 事件与 Lambda 结合使用。

```
import base64

def lambda_handler(event, context):
    # Iterate through keys
```

```
    for key in event['records']:
        print('Key:', key)
        # Iterate through records
        for record in event['records'][key]:
            print('Record:', record)
            # Decode base64
            msg = base64.b64decode(record['value']).decode('utf-8')
            print('Message:', msg)
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 Amazon MSK 事件与 Lambda 结合使用。

```
require 'base64'

def lambda_handler(event:, context:)
  # Iterate through keys
  event['records'].each do |key, records|
    puts "Key: #{key}"

    # Iterate through records
    records.each do |record|
      puts "Record: #{record}"

      # Decode base64
      msg = Base64.decode64(record['value'])
      puts "Message: #{msg}"
    end
  end
end
```

您可以使用控制台向 Lambda 提供函数代码。

更新 Lambda 函数代码

1. 导航到您的 Lambda 函数的概述页面。
2. 选择节点选项卡。
3. 将提供的代码复制到代码源代码编辑器中，替换 Lambda 创建的代码。
4. 在主侧栏中，展开部署部分，然后选择部署。

测试您的 Lambda 函数以验证其是否连接到您的 Amazon MSK 主题

现在，您可以通过查看 CloudWatch 事件日志来验证事件源是否正在调用您的 Lambda。

验证是否正在调用您的 Lambda 函数

1. 使用您的 Kafka 管理主机通过 CLI `kafka-console-producer` 生成 Kafka 事件。有关更多信息，请参阅 Kafka 文档中的 Write some events into the topic。发送足够的事件以填充上一步中定义的事件源映射批次大小所定义的批次，否则 Lambda 将等待更多信息来调用。
2. 如果您的函数运行，则 Lambda 会将发生的事件写入 CloudWatch。在控制台中，导航到您的 Lambda 函数的详细信息页面。
3. 选择 Configuration（配置）选项卡。
4. 在侧栏中，选择监控和操作工具。
5. 在日志记录配置下确定 CloudWatch 日志组。日志组应以 `/aws/lambda` 开头。选择日志组的链接。
6. 在 CloudWatch 控制台中，检查日志事件，查看 Lambda 已发送到日志流的日志事件。确定是否存在包含来自 Kafka 事件消息的日志事件，如下图所示。如果有，则表示您已成功使用 Lambda 事件源映射将 Lambda 函数连接到 Amazon MSK。

```
2020-08-06T15:06:18.861-04:00	START RequestId: 88ebae59-be0c-4e22-9db7-4154b437e43a Version: $LATEST
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z 88ebae59-be0c-4e22-9db7-4154b437e43a INFO Key: mytopic-0
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z	88ebae59-be0c-4e22-9db7-4154b437e43a	INFO	Record: {
  topic: 'mytopic',
  partition: 0,
  offset: 38,
  timestamp: 1596740777633,
  timestampType: 'CREATE_TIME',
  value: 'TWVzc2FnZSAjMQ=='
}
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z 88ebae59-be0c-4e22-9db7-4154b437e43a INFO Message: Message #1
2020-08-06T15:06:18.890-04:00	END RequestId: 88ebae59-be0c-4e22-9db7-4154b437e43a
```

将 AWS Lambda 与 Amazon RDS 结合使用

您可以将 Lambda 函数直接连接到 Amazon Relational Database Service（Amazon RDS），也可以通过 Amazon RDS 代理连接。直接连接适用于简单的场景，而在生产中则推荐使用代理。数据库代理管理共享数据库连接池，从而让函数能够在不耗尽数据库连接的情况下达到高并发级别。

对于频繁建立短数据库连接或打开和关闭大量数据库连接的 Lambda 函数，我们建议使用 Amazon RDS 代理。有关更多信息，请参阅《Amazon Relational Database Service 开发人员指南》中的自动连接 Lambda 函数和数据库实例。

配置函数以使用 RDS 资源

在 Lambda 控制台中，您可以预置和配置 Amazon RDS 数据库实例和代理资源。您可以导航到配置选项卡下的 RDS 数据库完成此操作。或者，您也可以在 Amazon RDS 控制台中创建和配置到 Lambda 函数的连接。配置 RDS 数据库实例以便与 Lambda 配合使用时，请注意以下条件：

- 要连接到数据库，您的函数必须位于数据库在其中运行的 Amazon VPC。
- 您可以将 Amazon RDS 数据库与 MySQL、MariaDB、PostgreSQL 或 Microsoft SQL Server 引擎一起使用。
- 您也可以将 Aurora 数据库集群与 MySQL 或 PostgreSQL 引擎一起使用。
- 您需要提供用于数据库身份验证的 Secrets Manager 密钥。
- IAM 角色必须提供使用密钥的权限和必须允许 Amazon RDS 代入角色的信任策略。
- 使用控制台配置 Amazon RDS 资源并将其连接到函数的 IAM 主体必须具有以下权限：

> **Note**
>
> 只有在配置 Amazon RDS 代理来管理数据库连接池时，才需要 Amazon RDS 代理权限。

Example 权限策略

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateSecurityGroup",
```

```
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ec2:AuthorizeSecurityGroupIngress",
        "ec2:AuthorizeSecurityGroupEgress",
        "ec2:RevokeSecurityGroupEgress",
        "ec2:CreateNetworkInterface",
        "ec2:DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "rds-db:connect",
        "rds:CreateDBProxy",
        "rds:CreateDBInstance",
        "rds:CreateDBSubnetGroup",
        "rds:DescribeDBClusters",
        "rds:DescribeDBInstances",
        "rds:DescribeDBSubnetGroups",
        "rds:DescribeDBProxies",
        "rds:DescribeDBProxyTargets",
        "rds:DescribeDBProxyTargetGroups",
        "rds:RegisterDBProxyTargets",
        "rds:ModifyDBInstance",
        "rds:ModifyDBProxy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:ListFunctions",
        "lambda:UpdateFunctionConfiguration"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:AttachRolePolicy",
```

```
        "iam:AttachPolicy",
        "iam:CreateRole",
        "iam:CreatePolicy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetResourcePolicy",
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret",
        "secretsmanager:ListSecretVersionIds",
        "secretsmanager:CreateSecret"
      ],
      "Resource": "*"
    }
  ]
}
```

Amazon RDS 根据数据库实例大小按小时收取代理费率，详情请参阅 RDS 代理定价。有关通用代理连接的更多信息，请参阅《Amazon RDS 用户指南》中的使用 Amazon RDS 代理。

> **Lambda 和 Amazon RDS 设置**
>
> Lambda 和 Amazon RDS 控制台都将协助您自动配置一些必要资源，以便在 Lambda 和 Amazon RDS 之间建立连接。

使用 Lambda 函数连接到 Amazon RDS 数据库

以下代码示例显示如何实现连接到 Amazon RDS 数据库的 Lambda 函数。该函数发出一个简单的数据库请求并返回结果。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 Go 连接到 Amazon RDS 数据库。

```
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"
 "os"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = os.Getenv("DatabaseName")
 var dbUser string = os.Getenv("DatabaseUser")
 var dbHost string = os.Getenv("DBHost") // Add hostname without https
 var dbPort int = os.Getenv("Port")      // Add port number
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
 var region string = os.Getenv("AWS_REGION")
```

```
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }

 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}
```

```
func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 Java 连接到 Amazon RDS 数据库。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rdsdata.RdsDataClient;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementRequest;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementResponse;
import software.amazon.awssdk.services.rdsdata.model.Field;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class RdsLambdaHandler implements
 RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
 event, Context context) {
        APIGatewayProxyResponseEvent response = new
 APIGatewayProxyResponseEvent();
```

```
        try {
            // Obtain auth token
            String token = createAuthToken();

            // Define connection configuration
            String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true",
                    System.getenv("ProxyHostName"),
                    System.getenv("Port"),
                    System.getenv("DBName"));

            // Establish a connection to the database
            try (Connection connection =
 DriverManager.getConnection(connectionString, System.getenv("DBUserName"),
 token);
                 PreparedStatement statement =
 connection.prepareStatement("SELECT ? + ? AS sum")) {

                statement.setInt(1, 3);
                statement.setInt(2, 2);

                try (ResultSet resultSet = statement.executeQuery()) {
                    if (resultSet.next()) {
                        int sum = resultSet.getInt("sum");
                        response.setStatusCode(200);
                        response.setBody("The selected sum is: " + sum);
                    }
                }
            }

        } catch (Exception e) {
            response.setStatusCode(500);
            response.setBody("Error: " + e.getMessage());
        }

        return response;
    }

    private String createAuthToken() {
        // Create RDS Data Service client
        RdsDataClient rdsDataClient = RdsDataClient.builder()
                .region(Region.of(System.getenv("AWS_REGION")))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();
```

```
        // Define authentication request
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
                .resourceArn(System.getenv("ProxyHostName"))
                .secretArn(System.getenv("DBUserName"))
                .database(System.getenv("DBName"))
                .sql("SELECT 'RDS IAM Authentication'")
                .build();

        // Execute request and obtain authentication token
        ExecuteStatementResponse response =
 rdsDataClient.executeStatement(request);
        Field tokenField = response.records().get(0).get(0);

        return tokenField.stringValue();
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 JavaScript 连接到 Amazon RDS 数据库。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
```

```
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
```

```
  }
};
```

在 Lambda 函数中使用 TypeScript 连接到 Amazon RDS 数据库。

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that
 the DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!


async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });

    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
            host: proxy_host_name,
            user: db_user_name,
            password: token,
```

```
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
        const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
        console.log('result:', rows);
        return rows;
    }
    catch (err) {
        console.log(err);
    }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number;
 body: string }> => {
    // Execute database flow
    const result = await dbOps();

    // Return error is result is undefined
    if (result == undefined)
        return {
            statusCode: 500,
            body: JSON.stringify(`Error with connection to DB host`)
        }

    // Return result
    return {
        statusCode: 200,
        body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
    };
};
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 PHP 连接到 Amazon RDS 数据库。

```
<?php
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;
use Aws\Rds\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }


    private function getAuthToken(): string {
        // Define connection authentication parameters
        $dbConnection = [
            'hostname' => getenv('DB_HOSTNAME'),
            'port' => getenv('DB_PORT'),
            'username' => getenv('DB_USERNAME'),
            'region' => getenv('AWS_REGION'),
        ];

        // Create RDS AuthTokenGenerator object
        $generator = new
 AuthTokenGenerator(CredentialProvider::defaultProvider());

        // Request authorization token from RDS, specifying the username
        return $generator->createToken(
            $dbConnection['hostname'] . ':' . $dbConnection['port'],
            $dbConnection['region'],
            $dbConnection['username']
        );
    }
```

```
    private function getQueryResults() {
        // Obtain auth token
        $token = $this->getAuthToken();

        // Define connection configuration
        $connectionConfig = [
            'host' => getenv('DB_HOSTNAME'),
            'user' => getenv('DB_USERNAME'),
            'password' => $token,
            'database' => getenv('DB_NAME'),
        ];

        // Create the connection to the DB
        $conn = new PDO(

"mysql:host={$connectionConfig['host']};dbname={$connectionConfig['database']}",
            $connectionConfig['user'],
            $connectionConfig['password'],
            [
                PDO::MYSQL_ATTR_SSL_CA => '/path/to/rds-ca-2019-root.pem',
                PDO::MYSQL_ATTR_SSL_VERIFY_SERVER_CERT => true,
            ]
        );

        // Obtain the result of the query
        $stmt = $conn->prepare('SELECT ?+? AS sum');
        $stmt->execute([3, 2]);

        return $stmt->fetch(PDO::FETCH_ASSOC);
    }

    /**
     * @param mixed $event
     * @param Context $context
     * @return array
     */
    public function handle(mixed $event, Context $context): array
    {
        $this->logger->info("Processing query");

        // Execute database flow
        $result = $this->getQueryResults();
```

```
        return [
            'sum' => $result['sum']
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 Python 连接到 Amazon RDS 数据库。

```
import json
import os
import boto3
import pymysql

# RDS settings
proxy_host_name = os.environ['PROXY_HOST_NAME']
port = int(os.environ['PORT'])
db_name = os.environ['DB_NAME']
db_user_name = os.environ['DB_USER_NAME']
aws_region = os.environ['AWS_REGION']


# Fetch RDS Auth Token
def get_auth_token():
    client = boto3.client('rds')
    token = client.generate_db_auth_token(
        DBHostname=proxy_host_name,
        Port=port
        DBUsername=db_user_name
        Region=aws_region
```

```
    )
    return token

def lambda_handler(event, context):
    token = get_auth_token()
    try:
        connection = pymysql.connect(
            host=proxy_host_name,
            user=db_user_name,
            password=token,
            db=db_name,
            port=port,
            ssl={'ca': 'Amazon RDS'}  # Ensure you have the CA bundle for SSL
 connection
        )

        with connection.cursor() as cursor:
            cursor.execute('SELECT %s + %s AS sum', (3, 2))
            result = cursor.fetchone()

        return result

    except Exception as e:
        return (f"Error: {str(e)}")  # Return an error message if an exception
 occurs
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 Ruby 连接到 Amazon RDS 数据库。

```
# Ruby code here.

require 'aws-sdk-rds'
```

```
require 'json'
require 'mysql2'

def lambda_handler(event:, context:)
  endpoint = ENV['DBEndpoint'] # Add the endpoint without https"
  port = ENV['Port']           # 3306
  user = ENV['DBUser']
  region = ENV['DBRegion']     # 'us-east-1'
  db_name = ENV['DBName']

  credentials = Aws::Credentials.new(
    ENV['AWS_ACCESS_KEY_ID'],
    ENV['AWS_SECRET_ACCESS_KEY'],
    ENV['AWS_SESSION_TOKEN']
  )
  rds_client = Aws::RDS::AuthTokenGenerator.new(
    region: region,
    credentials: credentials
  )

  token = rds_client.auth_token(
    endpoint: endpoint+ ':' + port,
    user_name: user,
    region: region
  )

  begin
    conn = Mysql2::Client.new(
      host: endpoint,
      username: user,
      password: token,
      port: port,
      database: db_name,
      sslca: '/var/task/global-bundle.pem',
      sslverify: true,
      enable_cleartext_plugin: true
    )
    a = 3
    b = 2
    result = conn.query("SELECT #{a} + #{b} AS sum").first['sum']
    puts result
    conn.close
    {
      statusCode: 200,
```

```
      body: result.to_json
    }
  rescue => e
    puts "Database connection failed due to #{e}"
  end
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 Rust 连接到 Amazon RDS 数据库。

```
use aws_config::BehaviorVersion;
use aws_credential_types::provider::ProvideCredentials;
use aws_sigv4::{
    http_request::{sign, SignableBody, SignableRequest, SigningSettings},
    sign::v4,
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
use sqlx::postgres::PgConnectOptions;
use std::env;
use std::time::{Duration, SystemTime};

const RDS_CERTS: &[u8] = include_bytes!("global-bundle.pem");

async fn generate_rds_iam_token(
    db_hostname: &str,
    port: u16,
    db_username: &str,
) -> Result<String, Error> {
    let config = aws_config::load_defaults(BehaviorVersion::v2024_03_28()).await;

    let credentials = config
        .credentials_provider()
```

```
        .expect("no credentials provider found")
        .provide_credentials()
        .await
        .expect("unable to load credentials");
    let identity = credentials.into();
    let region = config.region().unwrap().to_string();

    let mut signing_settings = SigningSettings::default();
    signing_settings.expires_in = Some(Duration::from_secs(900));
    signing_settings.signature_location =
 aws_sigv4::http_request::SignatureLocation::QueryParams;

    let signing_params = v4::SigningParams::builder()
        .identity(&identity)
        .region(&region)
        .name("rds-db")
        .time(SystemTime::now())
        .settings(signing_settings)
        .build()?;

    let url = format!(
        "https://{db_hostname}:{port}/?Action=connect&DBUser={db_user}",
        db_hostname = db_hostname,
        port = port,
        db_user = db_username
    );

    let signable_request =
        SignableRequest::new("GET", &url, std::iter::empty(),
 SignableBody::Bytes(&[]))
            .expect("signable request");

    let (signing_instructions, _signature) =
        sign(signable_request, &signing_params.into())?.into_parts();

    let mut url = url::Url::parse(&url).unwrap();
    for (name, value) in signing_instructions.params() {
        url.query_pairs_mut().append_pair(name, &value);
    }

    let response = url.to_string().split_off("https://".len());

    Ok(response)
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(handler)).await
}

async fn handler(_event: LambdaEvent<Value>) -> Result<Value, Error> {
    let db_host = env::var("DB_HOSTNAME").expect("DB_HOSTNAME must be set");
    let db_port = env::var("DB_PORT")
        .expect("DB_PORT must be set")
        .parse::<u16>()
        .expect("PORT must be a valid number");
    let db_name = env::var("DB_NAME").expect("DB_NAME must be set");
    let db_user_name = env::var("DB_USERNAME").expect("DB_USERNAME must be set");

    let token = generate_rds_iam_token(&db_host, db_port, &db_user_name).await?;

    let opts = PgConnectOptions::new()
        .host(&db_host)
        .port(db_port)
        .username(&db_user_name)
        .password(&token)
        .database(&db_name)
        .ssl_root_cert_from_pem(RDS_CERTS.to_vec())
        .ssl_mode(sqlx::postgres::PgSslMode::Require);

    let pool = sqlx::postgres::PgPoolOptions::new()
        .connect_with(opts)
        .await?;

    let result: i32 = sqlx::query_scalar("SELECT $1 + $2")
        .bind(3)
        .bind(2)
        .fetch_one(&pool)
        .await?;

    println!("Result: {:?}", result);

    Ok(json!({
        "statusCode": 200,
        "content-type": "text/plain",
        "body": format!("The selected sum is: {result}")
    }))
}
```

处理来自 Amazon RDS 的事件通知

您可以使用 Lambda 处理 Amazon RDS 数据库中的事件通知。Amazon RDS 将通知发送到 Amazon Simple Notification Service (Amazon SNS) 主题，您可以将其配置为调用 Lambda 函数。Amazon SNS 将来自 Amazon RDS 的消息封装在其自己的事件文档中，并将其发送到您的函数。

有关配置 Amazon RDS 数据库以发送通知的详细信息，请参阅使用 Amazon RDS 事件通知。

Example Amazon SNS 事件中的 Amazon RDS 消息

```
{
        "Records": [
          {
            "EventVersion": "1.0",
            "EventSubscriptionArn": "arn:aws:sns:us-east-2:123456789012:rds-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
            "EventSource": "aws:sns",
            "Sns": {
              "SignatureVersion": "1",
              "Timestamp": "2023-01-02T12:45:07.000Z",
              "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi
+tE/1+82j...65r==",
              "SigningCertUrl": "https://sns.us-east-2.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
              "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
              "Message": "{\"Event Source\":\"db-instance\",\"Event Time\":\"2023-01-02
 12:45:06.000\",\"Identifier Link\":\"https://console.aws.amazon.com/rds/home?
region=eu-west-1#dbinstance:id=dbinstanceid\",\"Source ID\":\"dbinstanceid\",\"Event ID
\":\"http://docs.amazonwebservices.com/AmazonRDS/latest/UserGuide/USER_Events.html#RDS-
EVENT-0002\",\"Event Message\":\"Finished DB Instance backup\"}",
              "MessageAttributes": {},
              "Type": "Notification",
              "UnsubscribeUrl": "https://sns.us-east-2.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-2:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
              "TopicArn":"arn:aws:sns:us-east-2:123456789012:sns-lambda",
              "Subject": "RDS Notification Message"
            }
          }
        ]
```

```
    }
```

Lambda 和 Amazon RDS 完整教程

- 使用 Lambda 函数访问 Amazon RDS 数据库：在《Amazon RDS 用户指南》中，了解如何使用 Lambda 函数通过 Amazon RDS 代理将数据写入 Amazon RDS 数据库。Lambda 函数将从 Amazon SQS 队列中读取记录，每当添加消息时，都将新的项目写入数据库的表中。

使用 Lambda 处理 Amazon S3 事件通知

您可以使用 Lambda 来处理来自 Amazon Simple Storage Service 的事件通知。Amazon S3 可以在创建或删除对象时向 Lambda 函数发送事件。您在存储桶上配置通知设置，并向 Amazon S3 授予权限来根据函数的基于资源的权限策略调用函数。

> **Warning**
>
> 如果您的 Lambda 函数使用触发它的同一存储桶，则会导致在一个循环中运行该函数。例如，如果每当上传一个对象，存储桶就触发某个函数，而该函数又上传一个对象给存储桶，则该函数间接触发了自身。为避免这种情况，请使用两个存储桶，或将触发器配置为仅适用于传入对象所用的前缀。

Amazon S3 使用包含有关对象的详细信息的事件异步调用您的函数。以下示例显示了在将部署包上载到 Amazon S3 时 Amazon S3 发送的事件。

Example Amazon S3 通知事件

```
{
  "Records": [
    {
      "eventVersion": "2.1",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-2",
      "eventTime": "2019-09-03T19:37:27.192Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AWS:AIDAINPONIXQXHT3IKHL2"
      },
      "requestParameters": {
        "sourceIPAddress": "205.255.255.255"
      },
      "responseElements": {
        "x-amz-request-id": "D82B88E5F771F645",
        "x-amz-id-2":
 "vlR7PnpV2Ce81l0PRw6jlUpck7Jo5ZsQjryTjKlc5aLWGVHPZLj5NeC6qMa0emYBDXOo6QBU0Wo="
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",
```

```
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "A3I5XTEXAMAI3E"
          },
          "arn": "arn:aws:s3:::lambda-artifacts-deafc19498e3f2df"
        },
        "object": {
          "key": "b21b84d653bb07b05b1e6b33684dc11b",
          "size": 1305107,
          "eTag": "b21b84d653bb07b05b1e6b33684dc11b",
          "sequencer": "0C0F6F405D6ED209E1"
        }
      }
    }
  ]
}
```

要调用您的函数，Amazon S3 需要来自该函数的基于资源的策略的权限。当您在 Lambda 控制台中配置 Amazon S3 触发器时，该控制台将修改基于资源的策略以允许 Amazon S3 在存储桶名称和账户 ID 匹配时调用函数。如果您在 Amazon S3 中配置通知，请使用 Lambda API 更新策略。您还可以使用 Lambda API 向另一个账户授予权限，或将权限限制到指定的别名。

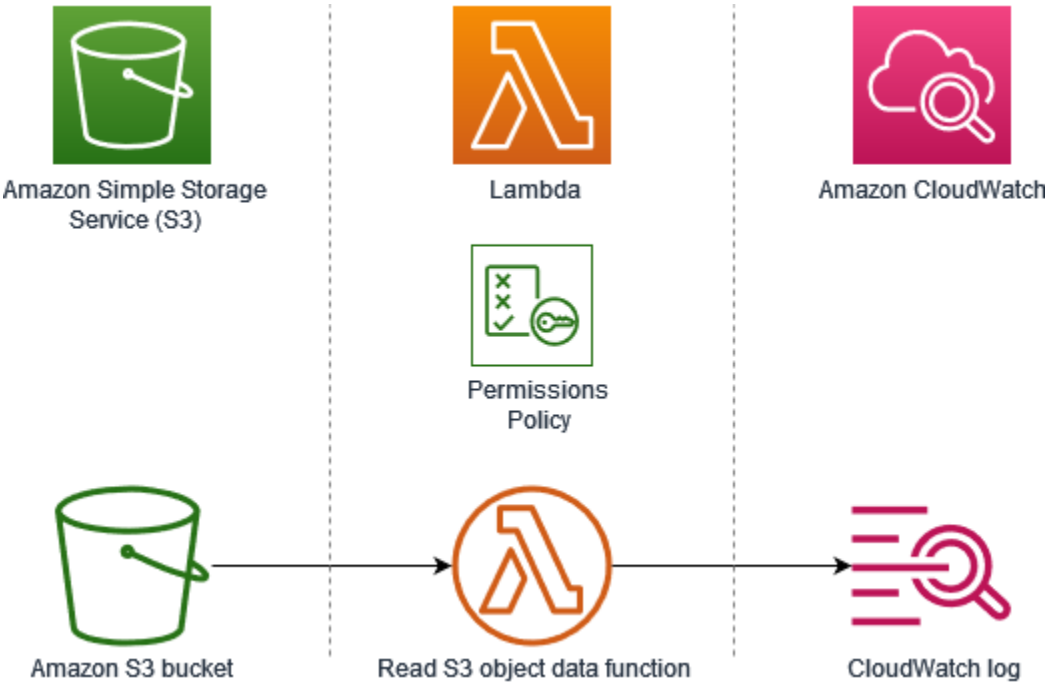
如果您的函数使用 AWS 开发工具包来管理 Amazon S3 资源，则其执行角色也需要 Amazon S3 权限。

主题

- 教程：使用 Amazon S3 触发器调用 Lambda 函数
- 教程：使用 Amazon S3 触发器创建缩略图

教程：使用 Amazon S3 触发器调用 Lambda 函数

在本教程中，您将使用控制台创建 Lambda 函数，然后为 Amazon Simple Storage Service（Amazon S3）存储桶配置触发器。每次向 Amazon S3 存储桶添加对象时，函数都会运行并将该对象类型输出到 Amazon CloudWatch Logs 中。



本教程演示如何：

1. 创建 Amazon S3 存储桶。
2. 创建一个 Lambda 函数，该函数会在 Amazon S3 存储桶中返回对象的类型。
3. 配置一个 Lambda 触发器，该触发器将在对象上传到存储桶时调用函数。
4. 先后使用虚拟事件和触发器测试函数。

完成这些步骤后，您将了解如何配置 Lambda 函数，使其在向 Amazon S3 存储桶添加或删除对象时运行。您仅可以使用 AWS Management Console 完成此教程。

先决条件

注册 AWS 账户

如果您还没有 AWS 账户，请完成以下步骤来创建一个。

注册 AWS 账户

1. 打开 https://portal.aws.amazon.com/billing/signup。
2. 按照屏幕上的说明进行操作。

   在注册时，将接到一通电话，要求使用电话键盘输入一个验证码。

当您注册 AWS 账户时，系统将会创建一个 AWS 账户根用户。根用户有权访问该账户中的所有 AWS 服务 和资源。作为安全最佳实践，请为用户分配管理访问权限，并且只使用根用户来执行需要根用户访问权限的任务。

注册过程完成后，AWS 会向您发送一封确认电子邮件。在任何时候，您都可以通过转至 https://aws.amazon.com/ 并选择我的账户来查看当前的账户活动并管理您的账户。

创建具有管理访问权限的用户

注册 AWS 账户 后，请保护好您的 AWS 账户根用户，启用 AWS IAM Identity Center，并创建一个管理用户，以避免使用根用户执行日常任务。

保护您的 AWS 账户根用户

1. 选择根用户并输入您的 AWS 账户电子邮件地址，以账户拥有者身份登录 AWS Management Console。在下一页上，输入您的密码。

   要获取使用根用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的以根用户身份登录。
2. 为您的根用户启用多重身份验证 (MFA)。

   有关说明，请参阅《IAM 用户指南》中的为 AWS 账户 根用户启用虚拟 MFA 设备（控制台）。

创建具有管理访问权限的用户

1. 启用 IAM Identity Center。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的启用 AWS IAM Identity Center。
2. 在 IAM Identity Center 中，为用户授予管理访问权限。

   有关如何使用 IAM Identity Center 目录 作为身份源的教程，请参阅《AWS IAM Identity Center 用户指南》中的使用默认的 IAM Identity Center 目录 配置用户访问权限。

以具有管理访问权限的用户身份登录

- 要使用您的 IAM Identity Center 用户身份登录，请使用您在创建 IAM Identity Center 用户时发送到您的电子邮件地址的登录网址。

要获取使用 IAM Identity Center 用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的登录 AWS 访问门户。

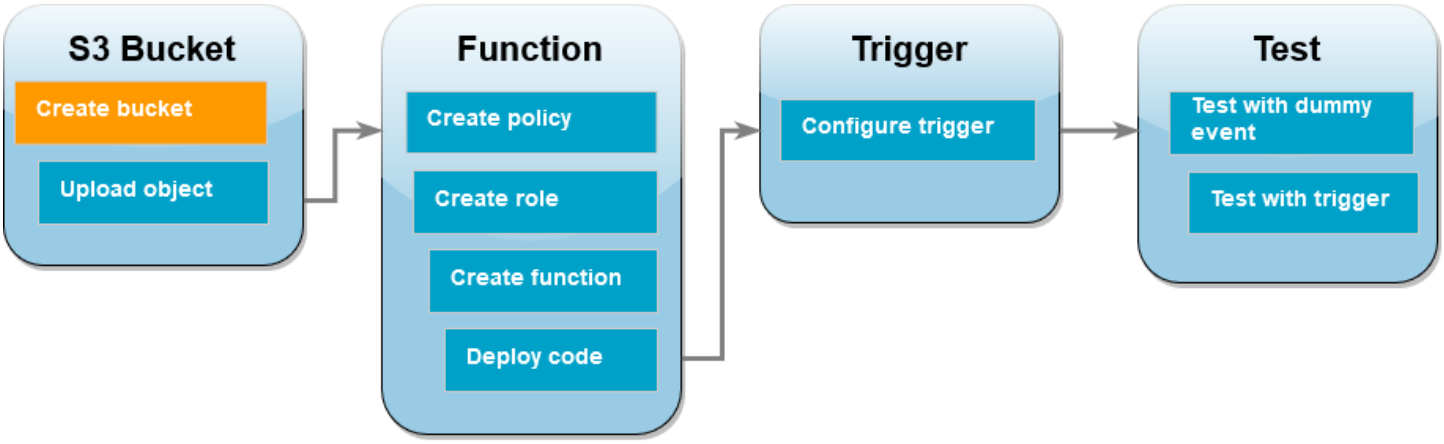
将访问权限分配给其他用户

1. 在 IAM Identity Center 中，创建一个权限集，该权限集遵循应用最低权限的最佳做法。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的创建权限集。
2. 将用户分配到一个组，然后为该组分配单点登录访问权限。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的添加组。

创建 Amazon S3 存储桶



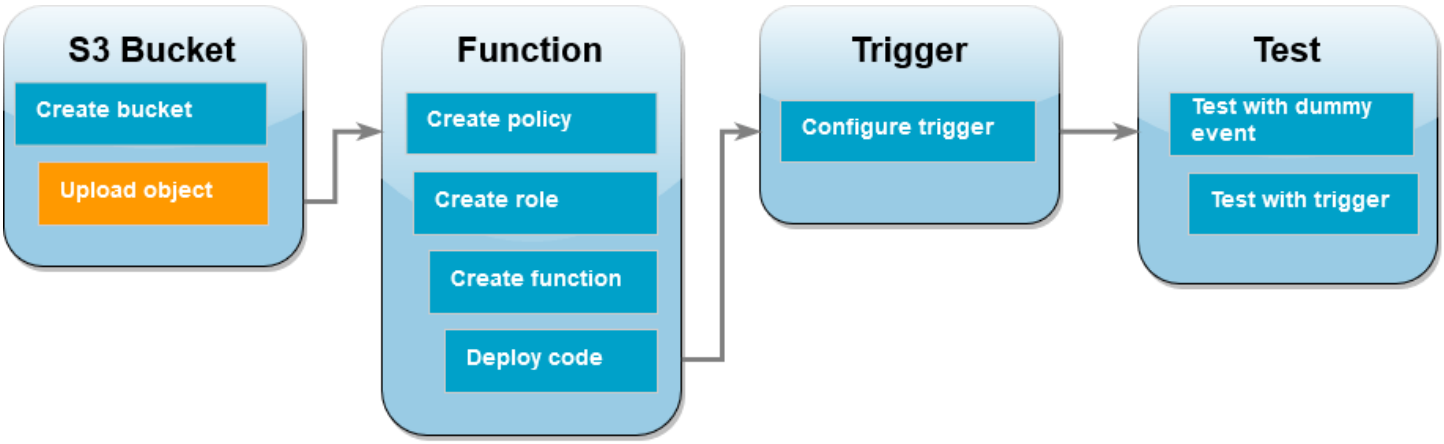
创建 Amazon S3 存储桶

1. 打开 Amazon S3 控制台并选择存储桶页面。
2. 选择 Create bucket（创建存储桶）。
3. 在 General configuration（常规配置）下，执行以下操作：

   a. 对于存储桶名称，输入符合 Amazon S3 存储桶命名规则的全局唯一名称。存储桶名称只能由小写字母、数字、句点（.）和连字符（-）组成。

   b. 对于 AWS Region（亚马逊云科技区域），选择一个区域。在本教程的后面部分，您必须在同个区域中创建 Lambda 函数。
4. 将所有其他选项设置为默认值并选择创建存储桶。

将测试对象上传到存储桶

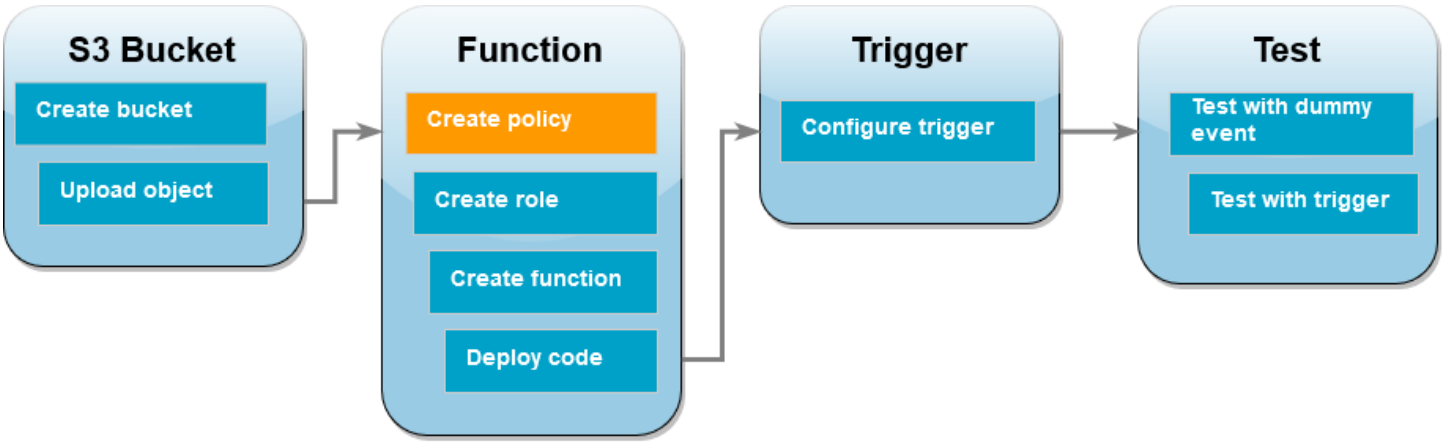


要上传测试对象

1. 打开 Amazon S3 控制台的存储桶页面，选择您在上一步中创建的存储桶。
2. 选择上传。
3. 选择添加文件，然后选择要上传的对象。您可以选择任何文件（例如 `HappyFace.jpg`）。
4. 选择打开，然后选择上传。

在本教程的后面部分，您要使用此对象测试 Lambda 函数。

创建权限策略



创建权限策略，允许 Lambda 从 Amazon S3 存储桶获取对象并写入 Amazon CloudWatch Logs。

创建策略

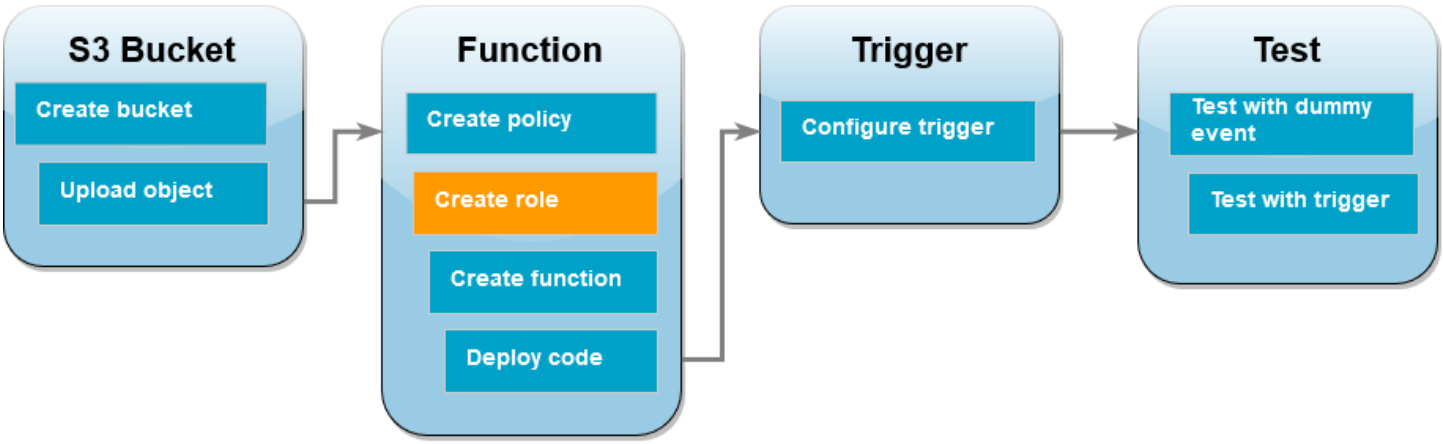
1. 打开 IAM 控制台的 Policies（策略）页面。

2. 选择创建策略。
3. 选择 JSON 选项卡，然后将以下自定义策略粘贴到 JSON 编辑器中。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. 选择下一步：标签。
5. 选择下一步：审核。
6. 在 Review policy (查看策略) 下，为策略 Name (名称) 输入 **s3-trigger-tutorial**。
7. 选择创建策略。

创建执行角色

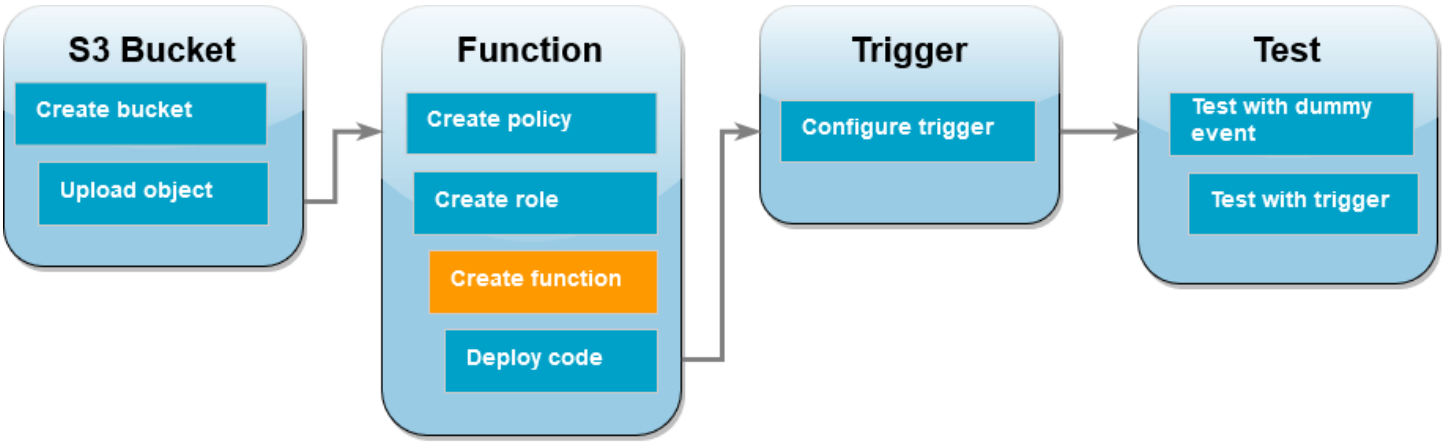


执行角色是一个 AWS Identity and Access Management（IAM）角色，用于向 Lambda 函数授予访问 AWS 服务和资源的权限。在此步骤中，您要使用在之前步骤中创建的权限策略来创建执行角色。

创建执行角色并附加自定义权限策略

1. 打开 IAM 控制台的角色页面。
2. 选择 Create role（创建角色）。
3. 对于可信实体，选择 AWS 服务，对于使用案例，选择 Lambda。
4. 选择下一步。
5. 在策略搜索框中，输入 **`s3-trigger-tutorial`**。
6. 在搜索结果中，选择您创建的策略（`s3-trigger-tutorial`），然后选择 Next（下一步）。
7. 在 Role details（角色详细信息）下，为 Role name（角色名称）输入 **`lambda-s3-trigger-role`**，然后选择 Create role（创建角色）。

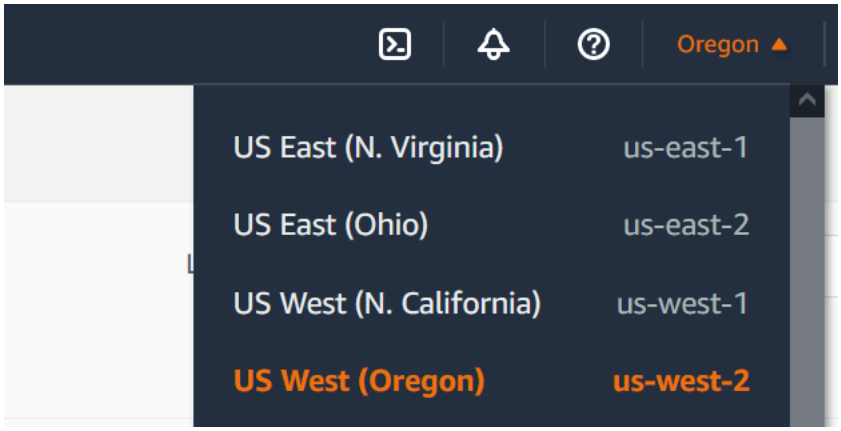
创建 Lambda 函数



使用 Python 3.12 运行时系统在控制台中创建 Lambda 函数。

创建 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 确保您在创建 Amazon S3 存储桶所在的同一 AWS 区域 内操作。您可以使用屏幕顶部的下拉列表更改区域。



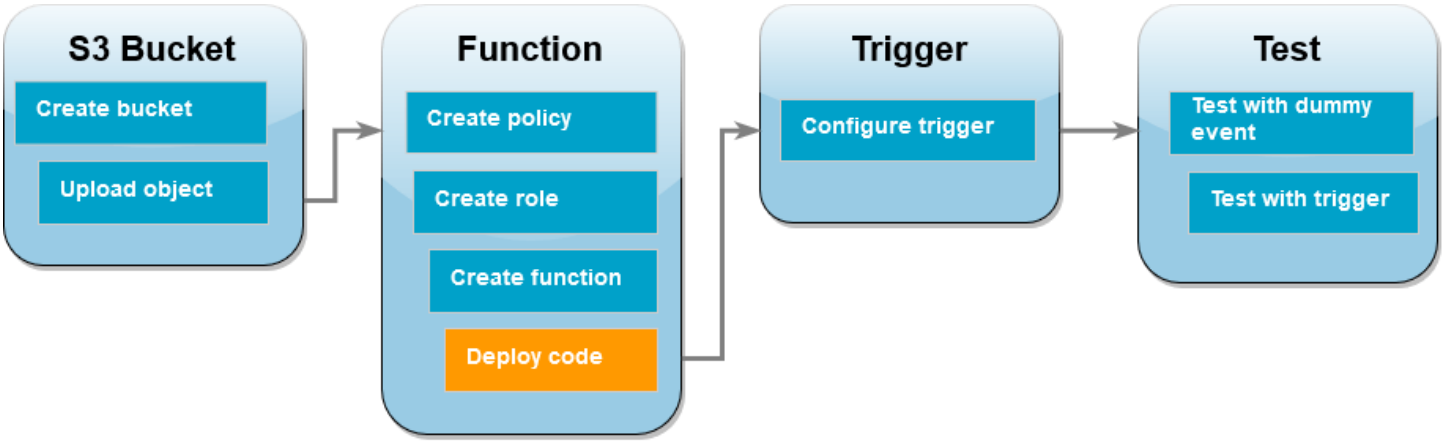
3. 选择 Create function (创建函数)。
4. 选择从头开始编写。
5. 在基本信息中，执行以下操作：
   a. 对于函数名称，输入 `s3-trigger-tutorial`。
   b. 对于运行时系统，选择 Python 3.12。
   c. 对于架构，选择 x86_64。
6. 在更改默认执行角色选项卡中，执行以下操作：

a. 展开选项卡，然后选择使用现有角色。

b. 选择您之前创建的 `lambda-s3-trigger-role`。

7. 选择 Create function (创建函数)。

部署函数代码



本教程使用 Python 3.12 运行时系统，但我们还提供了适用于其他运行时系统的示例代码文件。您可以选择以下框中的选项卡，查看适用于您感兴趣的运行时系统的代码。

Lambda 函数检索已上传对象的键名称和来自该对象从 Amazon S3 收到的 `event` 参数的存储桶名称。然后，该函数使用 AWS SDK for Python (Boto3) 中的 get_object 方法来检索对象的元数据，包括已上传对象的内容类型（MIME 类型）。

要部署函数代码

1. 在下框中选择 Python 选项卡并复制代码。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 .NET 将 S3 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext
 context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key =
 HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);
```

```
                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket,
 key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 S3 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
```

```
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Java 将 S3 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNo

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger =
 LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
```

```
            HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

            logger.info("Successfully retrieved " + srcBucket + "/" + srcKey +
 " of type " + headObject.contentType());

            return "Ok";
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String
 bucket, String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 JavaScript 将 S3 事件与 Lambda 结合使用。

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

    // Get the object from the event and show its content type
```

```
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\
+/g, ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}.
 Make sure they exist and your bucket is in the same region as this
 function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

使用 TypeScript 将 S3 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> =>
 {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
```

```
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 PHP 将 S3 事件与 Lambda 结合使用。

```
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
```

```
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as
 this function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 S3 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']
['key'], encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they
 exist and your bucket is in the same region as this function.'.format(key,
 bucket))
        raise e
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 S3 事件与 Lambda 结合使用。

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']
['key'], Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they
 exist and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

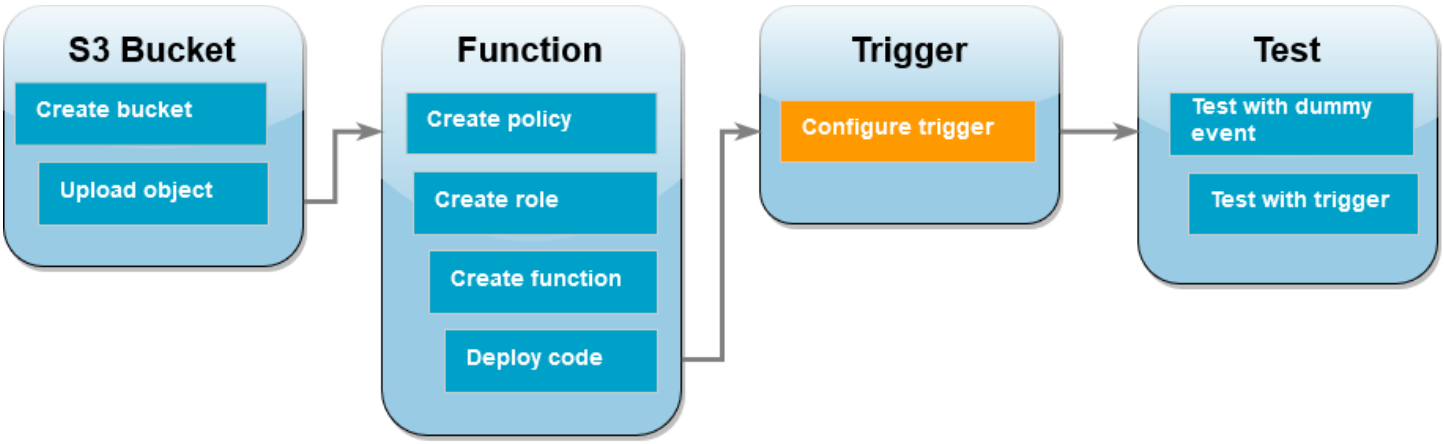
使用 Rust 将 S3 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
```

```
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request
 from SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket =
 evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket name to
 exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object
 key to exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

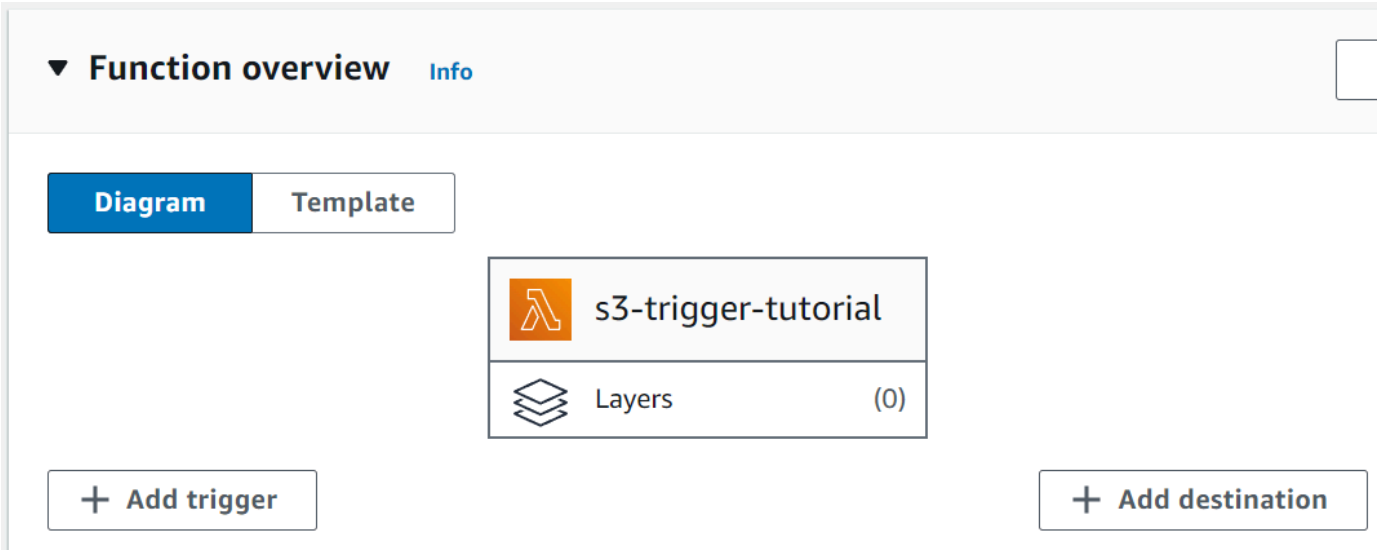
2. 在 Lambda 控制台的代码源窗格中，将代码粘贴到代码编辑器中，替换 Lambda 创建的代码。
3. 在主侧栏中，展开部署部分，然后选择部署。

创建 Amazon S3 触发器



创建 Amazon S3 触发器

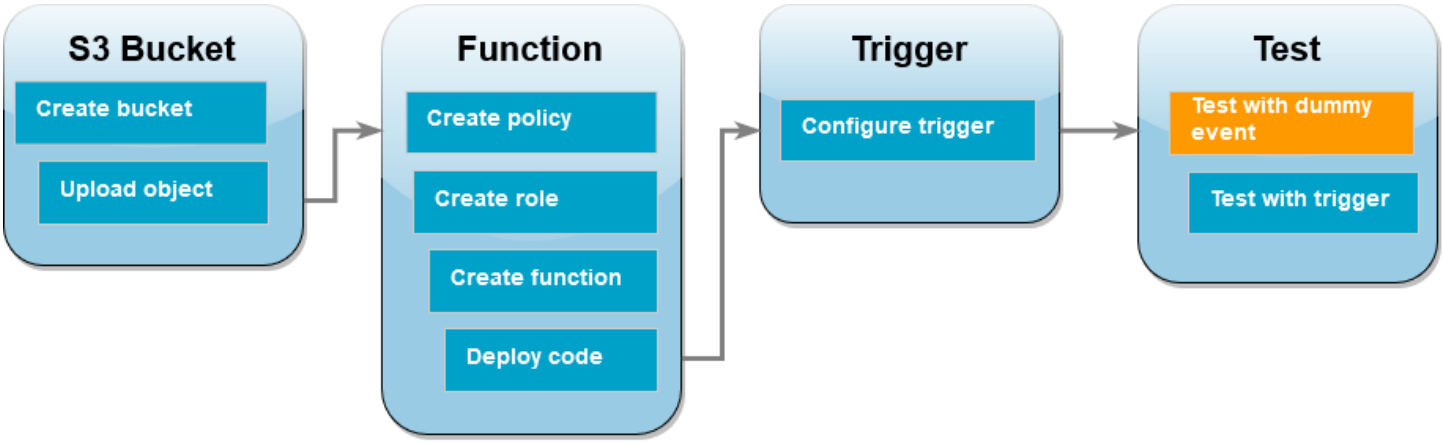
1. 在函数概述窗格中，选择添加触发器。



2. 选择 S3。
3. 在存储桶下，选择您在本教程前面步骤中创建的存储桶。
4. 在事件类型下，确保已选择所有对象创建事件。
5. 在递归调用下，选中复选框以确认知晓不建议使用相同的 Amazon S3 存储桶用于输入和输出。
6. 选择添加。

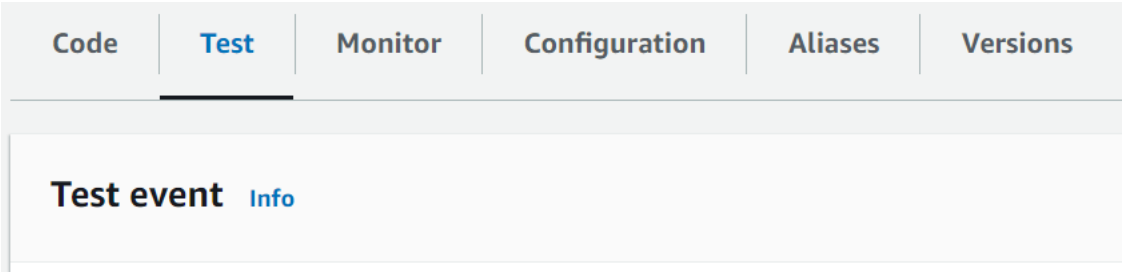
> **Note**
>
> 当您使用 Lambda 控制台为 Lambda 函数创建 Amazon S3 触发器时，Amazon S3 会在您指定的存储桶上配置事件通知。在配置此事件通知之前，Amazon S3 会执行一系列检查以确认事件目标是否存在并具有所需的 IAM 策略。Amazon S3 还会对为该存储桶配置的任何其他事件通知执行这些测试。
>
> 由于这项检查，如果存储桶之前为已不再存在的资源或没有所需权限策略的资源配置了事件目标，则 Amazon S3 将无法创建新的事件通知。您将看到以下错误消息，表明无法创建触发器：
>
> ```
> An error occurred when creating the trigger: Unable to validate the following
>  destination configurations.
> ```
>
> 如果您之前使用同一存储桶为另一个 Lambda 函数配置了触发器，并且此后又删除了该函数或修改了其权限策略，则会看到此错误。

使用虚拟事件测试 Lambda 函数



要使用虚拟事件测试 Lambda 函数

1. 在函数的 Lambda 控制台页面中，选择测试选项卡。



2. 对于事件名称，输入 MyTestEvent。
3. 在事件 JSON 中，粘贴以下测试事件。请务必替换以下值：
   - 使用 us-east-1 替换要在其中创建 Amazon S3 存储桶的区域。
   - 将 amzn-s3-demo-bucket 的两个实例都替换为 Amazon S3 存储桶的名称。
   - 将 test%2FKey 替换为您之前上传到存储桶的测试对象的名称（例如，HappyFace.jpg）。

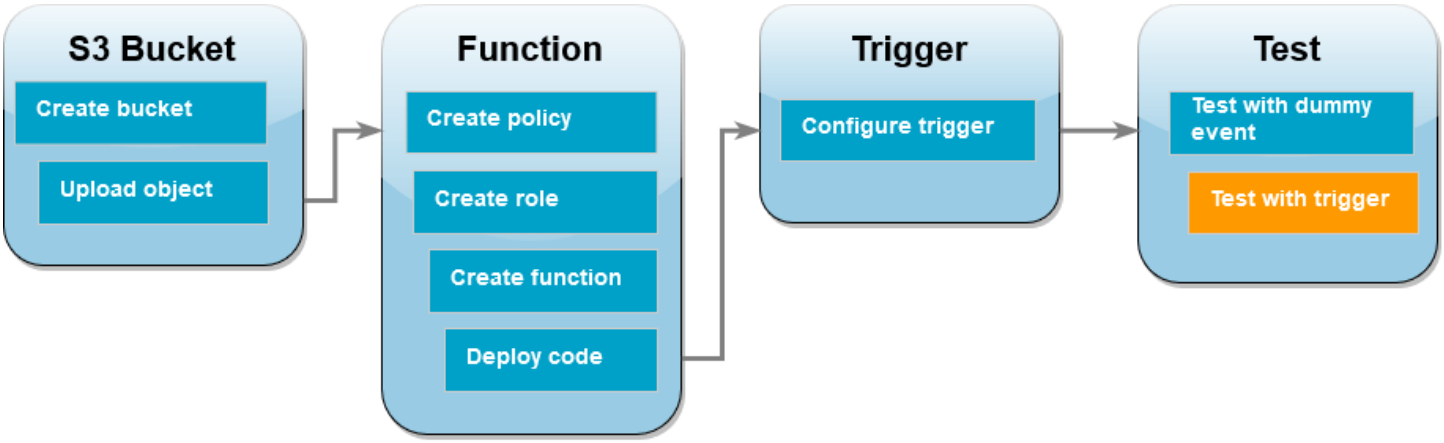
```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
```

```
        }
      }
    }
  ]
}
```

4. 选择保存。
5. 选择测试。
6. 如果函数成功运行，您将在执行结果选项卡中看到如下输出。

```
Response
"image/jpeg"

Function Logs
START RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6 Version: $LATEST
2021-02-18T21:40:59.280Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    INPUT
 BUCKET AND KEY:  { Bucket: 'amzn-s3-demo-bucket', Key: 'HappyFace.jpg' }
2021-02-18T21:41:00.215Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    CONTENT
 TYPE: image/jpeg
END RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6
REPORT RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6    Duration: 976.25 ms
 Billed Duration: 977 ms    Memory Size: 128 MB    Max Memory Used: 90 MB    Init
 Duration: 430.47 ms

Request ID
12b3cae7-5f4e-415e-93e6-416b8f8b66e6
```

使用 Amazon S3 触发器测试 Lambda 函数



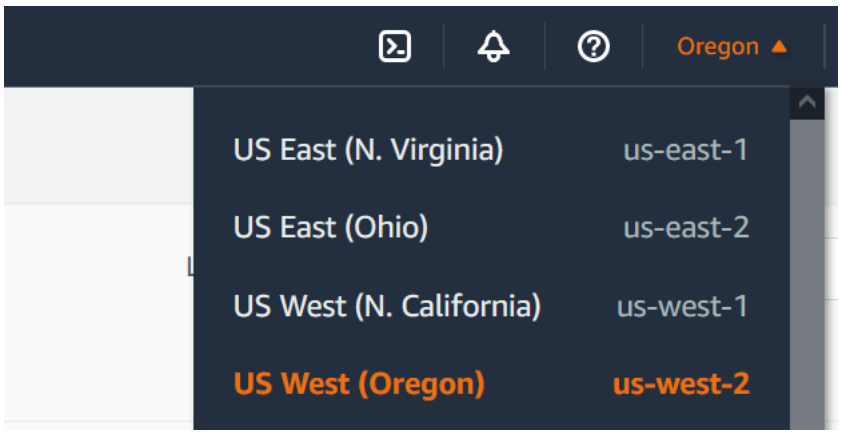
要使用配置的触发器测试函数，请使用控制台将对象上传到 Amazon S3 存储桶。要验证 Lambda 函数是否按预期运行，请使用 CloudWatch Logs 查看函数的输出。

要将对象上传到 Amazon S3 存储桶

1. 打开 Amazon S3 控制台的存储桶页面，选择之前创建的存储桶。
2. 选择上传。
3. 选择添加文件，然后使用文件选择器选择要上传的对象。此对象可以是您选择的任何文件。
4. 选择打开，然后选择上传。

使用 CloudWatch Logs 验证函数调用情况

1. 打开 CloudWatch 控制台。
2. 确保您在创建 Lambda 函数所在相同的 AWS 区域 操作。您可以使用屏幕顶部的下拉列表更改区域。



3. 选择日志，然后选择日志组。
4. 选择函数 (`/aws/lambda/s3-trigger-tutorial`) 的日志组。
5. 在日志流下，选择最新的日志流。
6. 如果已正确调用函数来响应 Amazon S3 触发器，您会看到如下输出。您看到的 `CONTENT TYPE` 取决于上传到存储桶的文件类型。

```
2022-05-09T23:17:28.702Z 0cae7f5a-b0af-4c73-8563-a3430333cc10 INFO CONTENT
 TYPE: image/jpeg
```

清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择删除。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。
4. 在文本输入字段中输入角色名称，然后选择 Delete（删除）。

删除 S3 存储桶

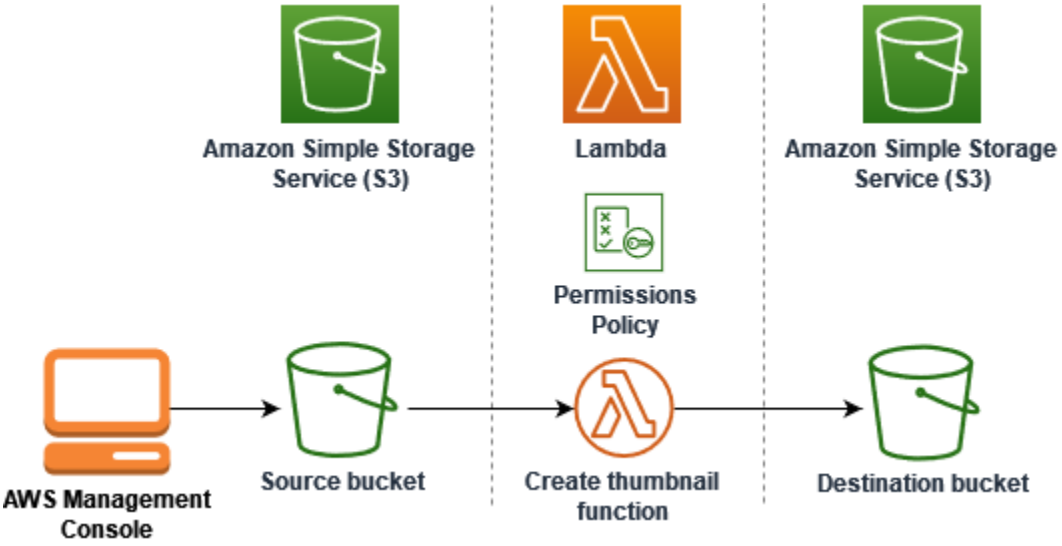
1. 打开 Amazon S3 控制台。
2. 选择您创建的存储桶。
3. 选择删除。
4. 在文本输入字段中输入存储桶的名称。
5. 选择删除存储桶。

后续步骤

在 教程：使用 Amazon S3 触发器创建缩略图 中，Amazon S3 触发器会调用一个函数，该函数会为上传到存储桶的每个图像文件创建缩略图。本教程需要适度的AWS和 Lambda 领域知识水平。其展示了如何使用 AWS Command Line Interface（AWS CLI）来创建资源，以及如何为函数及其依赖项创建 .zip 文件存档部署包。

教程：使用 Amazon S3 触发器创建缩略图

在本教程中，您将创建并配置 Lambda 函数，用于调整添加到 Amazon Simple Storage Service（Amazon S3）存储桶的图像大小。当您向存储桶添加图像文件时，Amazon S3 会调用您的 Lambda 函数。然后，该函数会创建图像的缩略图版本，并将其输出到不同的 Amazon S3 存储桶。



要完成本教程，请执行以下步骤：

1. 创建源和目标 Amazon S3 存储桶并上传示例图片。
2. 创建一个 Lambda 函数，用于调整图像大小并将缩略图输出到 Amazon S3 存储桶。
3. 配置一个 Lambda 触发器，该触发器将在对象上传到源存储桶时调用函数。
4. 若要测试您的函数，先使用虚拟事件，然后将图像上传到源存储桶。

完成这些步骤后，您将了解如何使用 Lambda 对添加到 Amazon S3 存储桶的对象执行文件处理任务。您可以使用 AWS Command Line Interface 或 AWS CLI (AWS Management Console) 完成此教程。

如果您相同通过更简单的示例来了解如何为 Lambda 配置 Amazon S3 触发器，则可以参阅教程：使用 Amazon S3 触发器调用 Lambda 函数。

主题

- 创建执行角色
- 创建函数部署包
- 创建 Lambda 函数
- 配置 Amazon S3 来调用函数
- 使用虚拟事件测试 Lambda 函数
- 使用 Amazon S3 触发器测试函数
- 清除资源

先决条件

注册 AWS 账户

如果您还没有 AWS 账户，请完成以下步骤来创建一个。

注册 AWS 账户

1. 打开 https://portal.aws.amazon.com/billing/signup。
2. 按照屏幕上的说明进行操作。

   在注册时，将接到一通电话，要求使用电话键盘输入一个验证码。

   当您注册 AWS 账户时，系统将会创建一个 AWS 账户根用户。根用户有权访问该账户中的所有 AWS 服务 和资源。作为安全最佳实践，请为用户分配管理访问权限，并且只使用根用户来执行需要根用户访问权限的任务。

注册过程完成后，AWS 会向您发送一封确认电子邮件。在任何时候，您都可以通过转至 https://aws.amazon.com/ 并选择我的账户来查看当前的账户活动并管理您的账户。

创建具有管理访问权限的用户

注册 AWS 账户 后，请保护好您的 AWS 账户根用户，启用 AWS IAM Identity Center，并创建一个管理用户，以避免使用根用户执行日常任务。

保护您的 AWS 账户根用户

1. 选择根用户并输入您的 AWS 账户电子邮件地址，以账户拥有者身份登录 AWS Management Console。在下一页上，输入您的密码。

要获取使用根用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的以根用户身份登录。

2. 为您的根用户启用多重身份验证 (MFA)。

   有关说明，请参阅《IAM 用户指南》中的为 AWS 账户 根用户启用虚拟 MFA 设备（控制台）。

创建具有管理访问权限的用户

1. 启用 IAM Identity Center。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的启用 AWS IAM Identity Center。

2. 在 IAM Identity Center 中，为用户授予管理访问权限。

   有关如何使用 IAM Identity Center 目录 作为身份源的教程，请参阅《AWS IAM Identity Center 用户指南》中的使用默认的 IAM Identity Center 目录 配置用户访问权限。

以具有管理访问权限的用户身份登录

- 要使用您的 IAM Identity Center 用户身份登录，请使用您在创建 IAM Identity Center 用户时发送到您的电子邮件地址的登录网址。

  要获取使用 IAM Identity Center 用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的登录 AWS 访问门户。

将访问权限分配给其他用户

1. 在 IAM Identity Center 中，创建一个权限集，该权限集遵循应用最低权限的最佳做法。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的创建权限集。

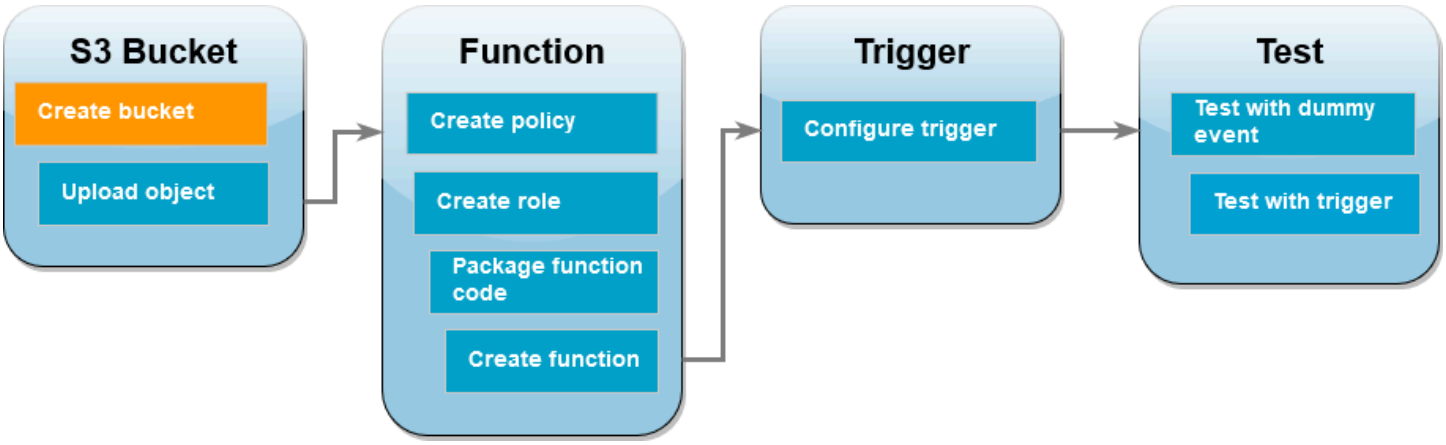
2. 将用户分配到一个组，然后为该组分配单点登录访问权限。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的添加组。

如果您想使用 AWS CLI 来完成教程，请安装最新版本的 AWS Command Line Interface。

对于 Lambda 函数代码，您可以使用 Python 或 Node.js。为您要使用的语言安装语言支持工具和软件包管理器。

创建两个 Amazon S3 存储桶



先创建两个 Amazon S3 存储桶。第一个存储桶是您要向其上传图像的源存储桶。第二个存储桶是 Lambda 用来保存调用函数时调整大小后的缩略图。

AWS Management Console

创建 Amazon S3 存储桶（控制台）

1. 打开 Amazon S3 控制台的存储桶页面。
2. 选择 Create bucket（创建存储桶）。
3. 在 General configuration（常规配置）下，执行以下操作：
   a. 对于存储桶名称，输入符合 Amazon S3 存储桶命名规则的全局唯一名称。存储桶名称只能由小写字母、数字、句点（.）和连字符（-）组成。
   b. 对于 AWS 区域，请选择最接近您地理位置的 AWS 区域。在本教程的后面部分，您必须在同个 AWS 区域 中创建 Lambda 函数，因此请记下您选择的区域。
4. 将所有其他选项设置为默认值并选择创建存储桶。
5. 重复步骤 1 到 4 以创建自己的目标存储桶。在存储桶名称中输入 **`amzn-s3-demo-source-bucket-resized`**，其中 **`amzn-s3-demo-source-bucket`** 是您刚刚创建的源存储桶的名称。

AWS CLI

创建 Amazon S3 存储桶（AWS CLI）

1. 运行以下 CLI 命令来创建自己的源存储桶。您为存储桶选择的名称必须具有全局唯一性，并遵守 Amazon S3 存储桶命名规则。名称只能由小写字母、数字、句点（.）和连字符（-）组成。对于 region 和 LocationConstraint，请选择最接近您地理位置的 AWS 区域。

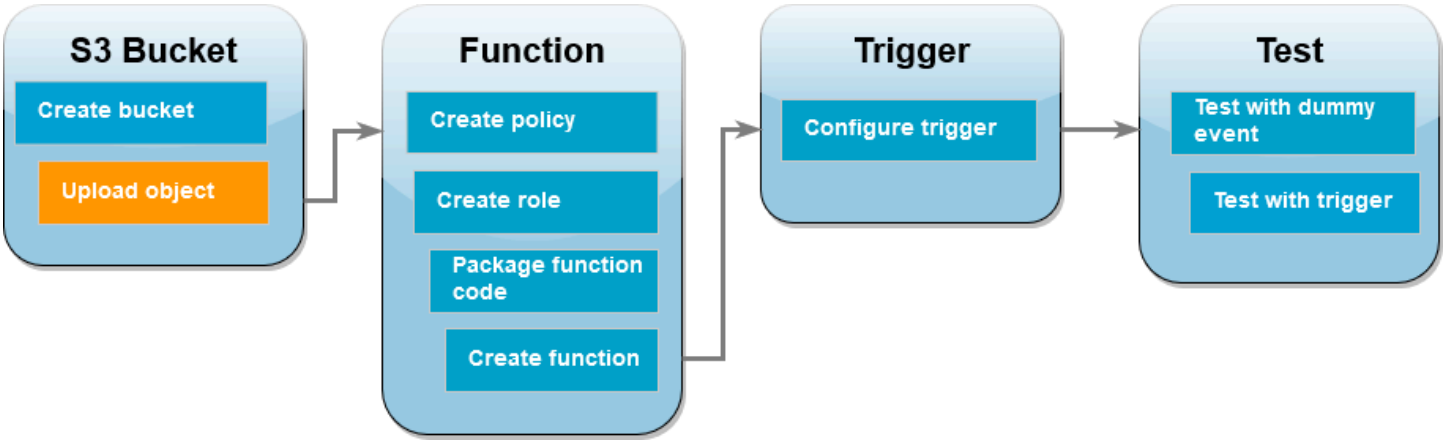
```
aws s3api create-bucket --bucket amzn-s3-demo-source-bucket --region us-east-1 \
--create-bucket-configuration LocationConstraint=us-east-1
```

   在本教程的后面部分，您必须在与源存储桶相同的 AWS 区域 中创建 Lambda 函数，因此请记下您选择的区域。

2. 运行以下命令来创建自己的目标存储桶。对于存储桶名称，必须使用 **amzn-s3-demo-source-bucket-resized**，其中 **amzn-s3-demo-source-bucket** 是您在步骤 1 中创建的源存储桶名称。对于 region 和 LocationConstraint，请选择与用于创建源存储桶时相同的 AWS 区域。

```
aws s3api create-bucket --bucket amzn-s3-demo-source-bucket-resized --region us-
east-1 \
--create-bucket-configuration LocationConstraint=us-east-1
```

将测试图片上传到源存储桶



在本教程的后面部分，您将使用 AWS CLI 或 Lambda 控制台调用 Lambda 函数以对其进行测试。要确认函数是否正常运行，您的源存储桶需要包含测试图片。此图像可以是您选择的任何 JPG 或 PNG 文件。

AWS Management Console

将测试图片上传到源存储桶（控制台）

1. 打开 Amazon S3 控制台的存储桶页面。
2. 选择您在上一步中创建的源存储桶。
3. 选择上传。
4. 选择添加文件，然后使用文件选择器选择要上传的对象。
5. 选择打开，然后选择上传。

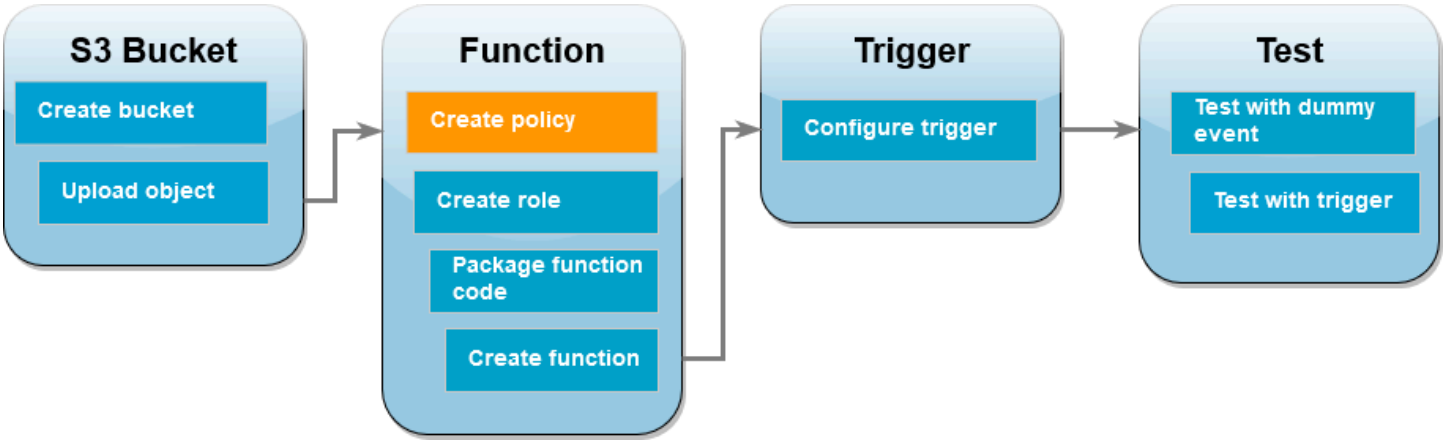
AWS CLI

将测试图片上传到源存储桶（AWS CLI）

- 在包含要上传的图像的目录中，运行以下 CLI 命令。将 `--bucket` 参数替换为源存储桶的名称。对于`--key` 和 `--body` 参数，请使用测试图像的文件名。

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key HappyFace.jpg --
body ./HappyFace.jpg
```

创建权限策略



创建 Lambda 函数的第一步是创建权限策略。此策略为函数提供访问其他 AWS 资源所需的权限。在本教程中，此策略授予了 Lambda 对 Amazon S3 存储桶的读取和写入权限，并允许其写入 Amazon CloudWatch Logs。

AWS Management Console

创建策略（控制台）

1. 打开 AWS Identity and Access Management IAM 控制台的 Policies（策略）页面。
2. 选择创建策略。
3. 选择 JSON 选项卡，然后将以下自定义策略粘贴到 JSON 编辑器中。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. 选择下一步。
5. 在策略详细信息下，为策略名称输入 **LambdaS3Policy**。
6. 选择创建策略。

AWS CLI

创建策略（AWS CLI）

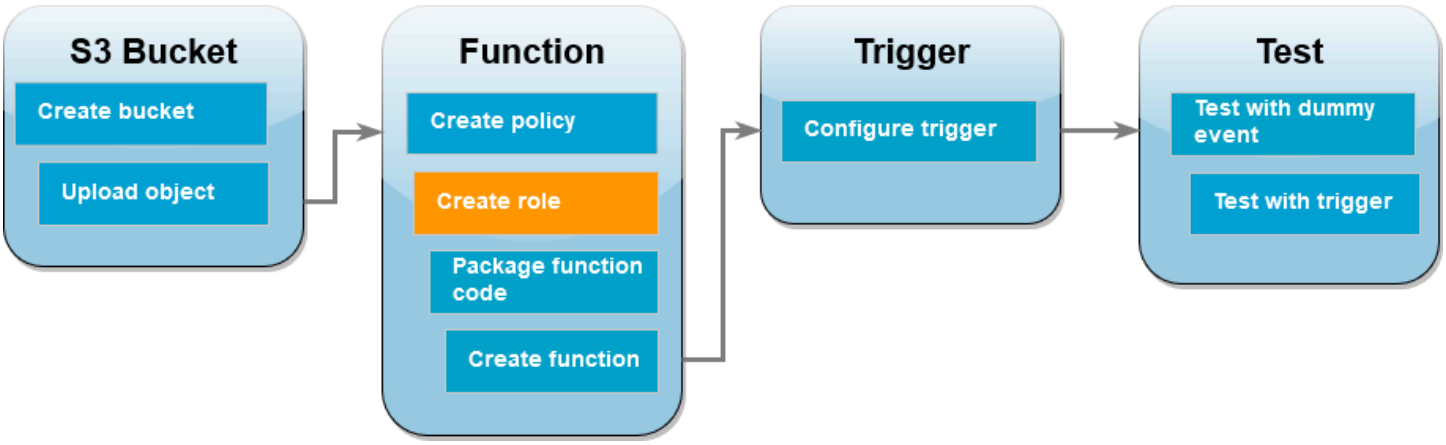
1. 将下列 JSON 保存在名为 policy.json 的文件中。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

2. 在保存 JSON 策略文档的目录中，运行以下 CLI 命令。

```
aws iam create-policy --policy-name LambdaS3Policy --policy-document file://
policy.json
```

创建执行角色



执行角色是一个 IAM 角色，用于向 Lambda 函数授予访问 AWS 服务 和资源的权限。要授予函数读取和写入 Amazon S3 存储桶的权限，您需要附加上一步中创建的权限策略。

AWS Management Console

创建执行角色并附加权限策略（控制台）

1. 打开（IAM）控制台的角色页面。
2. 选择 Create role（创建角色）。
3. 在可信实体类型中选择 AWS 服务，在使用案例中选择 Lambda。
4. 选择下一步。
5. 执行以下操作添加您在上一步中创建的权限策略：
   a. 在策略搜索框中，输入 **`LambdaS3Policy`**。
   b. 在搜索结果中，选中 `LambdaS3Policy` 的复选框。
   c. 选择下一步。
6. 在角色详细信息下的角色名称中输入 **`LambdaS3Role`**。
7. 选择 Create role（创建角色）。

AWS CLI

创建执行角色并附加权限策略（AWS CLI）

1. 将下列 JSON 保存在名为 `trust-policy.json` 的文件中。此信任策略允许 Lambda 通过向服务主体 `lambda.amazonaws.com` 授予调用 AWS Security Token Service（AWS STS）`AssumeRole` 操作的权限来使用该角色的权限。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

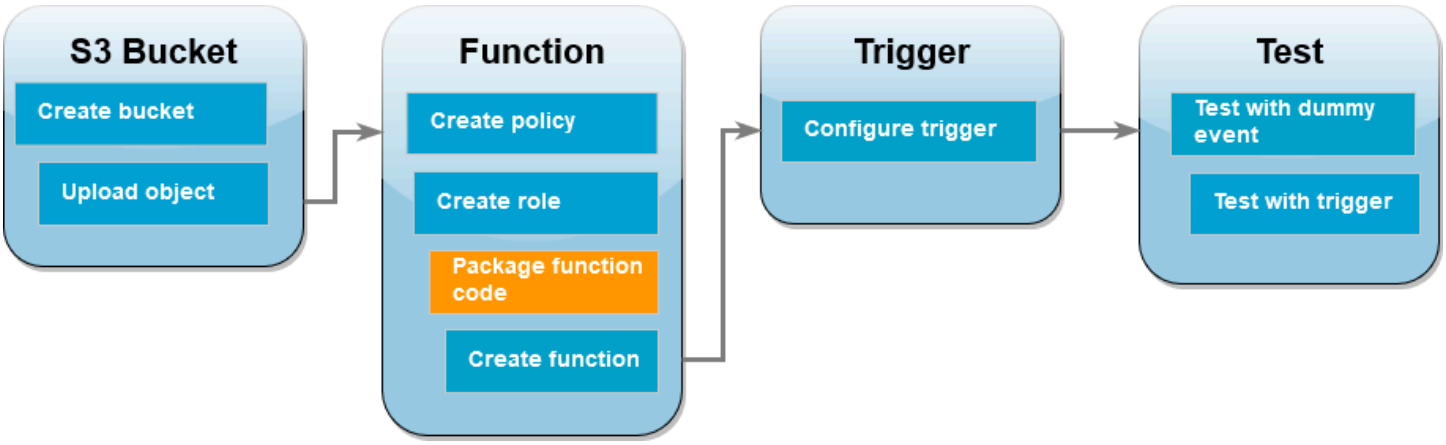
2. 在保存 JSON 信任策略文档的目录中，运行以下 CLI 命令来创建执行角色。

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. 要附加您在上一步中创建的权限策略，请运行以下 CLI 命令。将策略 ARN 中的 AWS 账户 号码替换为自己的账号。

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::123456789012:policy/LambdaS3Policy
```

创建函数部署包



要创建函数，您需要创建包含函数代码和所有依赖项的部署包。对于此 `CreateThumbnail` 函数，您的函数代码使用单独的库来调整图像大小。按照所选语言的说明创建包含所需库的部署包。

Node.js

创建部署包（Node.js）

1. 为您的函数代码和依赖项创建一个名为 `lambda-s3` 的目录并导航到此目录。

```
mkdir lambda-s3
cd lambda-s3
```

2. 使用 npm 创建新的 Node.js 项目。要在交互式体验中接受提供的默认选项，请按 `Enter`。

```
npm init
```

3. 将以下函数代码保存在名为 `index.mjs` 的文件中。请务必将 `us-east-1` 替换您创建源存储桶和目标存储桶时所在的 AWS 区域。

```
// dependencies
import { S3Client, GetObjectCommand, PutObjectCommand } from '@aws-sdk/client-
s3';

import { Readable } from 'stream';

import sharp from 'sharp';
import util from 'util';
```

```
// create S3 client
const s3 = new S3Client({region: 'us-east-1'});

// define the handler function
export const handler = async (event, context) => {

// Read options from the event parameter and get the source bucket
console.log("Reading options from event:\n", util.inspect(event, {depth: 5}));
  const srcBucket = event.Records[0].s3.bucket.name;

// Object key may have spaces or unicode non-ASCII characters
const srcKey    = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, " "));
const dstBucket = srcBucket + "-resized";
const dstKey    = "resized-" + srcKey;

// Infer the image type from the file suffix
const typeMatch = srcKey.match(/\.([^.]*)$/);
if (!typeMatch) {
  console.log("Could not determine the image type.");
  return;
}

// Check that the image type is supported
const imageType = typeMatch[1].toLowerCase();
if (imageType != "jpg" && imageType != "png") {
  console.log(`Unsupported image type: ${imageType}`);
  return;
}

// Get the image from the source bucket. GetObjectCommand returns a stream.
try {
  const params = {
    Bucket: srcBucket,
    Key: srcKey
  };
  var response = await s3.send(new GetObjectCommand(params));
  var stream = response.Body;

// Convert stream to buffer to pass to sharp resize function.
  if (stream instanceof Readable) {
    var content_buffer = Buffer.concat(await stream.toArray());

  } else {
```

```
    throw new Error('Unknown object stream type');
  }


} catch (error) {
  console.log(error);
  return;
}


// set thumbnail width. Resize will set the height automatically to maintain
 aspect ratio.
const width  = 200;

// Use the sharp module to resize the image and save in a buffer.
try {
  var output_buffer = await sharp(content_buffer).resize(width).toBuffer();

} catch (error) {
  console.log(error);
  return;
}

// Upload the thumbnail image to the destination bucket
try {
  const destparams = {
    Bucket: dstBucket,
    Key: dstKey,
    Body: output_buffer,
    ContentType: "image"
  };

  const putResult = await s3.send(new PutObjectCommand(destparams));

  } catch (error) {
    console.log(error);
    return;
  }

  console.log('Successfully resized ' + srcBucket + '/' + srcKey +
    ' and uploaded to ' + dstBucket + '/' + dstKey);
  };
```

4. 在 lambda-s3 目录中，使用 npm 安装 sharp 库。请注意，最新版本的 sharp（0.33）与 Lambda 不兼容。安装版本 0.32.6 以完成本教程。

```
npm install sharp@0.32.6
```

npm install 命令会为模块创建 node_modules 目录。完成此步骤后，目录结构应如下所示：

```
lambda-s3
|- index.mjs
|- node_modules
|  |- base64js
|  |- bl
|  |- buffer
...
|- package-lock.json
|- package.json
```

5. 创建一个包含函数代码和依赖项的部署包。在 MacOS 或 Linux 中，运行以下命令。

```
zip -r function.zip .
```

在 Windows 中，使用您首选的 ZIP 实用工具来创建 .zip 文件。确保您的 index.mjs、package.json、和 package-lock.json 文件以及您的 node_modules 目录都在.zip 文件的根目录中。

Python

创建部署包（Python）

1. 将代码示例保存为名为 lambda_function.py 的文件。

```
import boto3
import os
import sys
import uuid
from urllib.parse import unquote_plus
from PIL import Image
import PIL.Image
```

```
s3_client = boto3.client('s3')

def resize_image(image_path, resized_path):
  with Image.open(image_path) as image:
    image.thumbnail(tuple(x / 2 for x in image.size))
    image.save(resized_path)

def lambda_handler(event, context):
  for record in event['Records']:
    bucket = record['s3']['bucket']['name']
    key = unquote_plus(record['s3']['object']['key'])
    tmpkey = key.replace('/', '')
    download_path = '/tmp/{}{}'.format(uuid.uuid4(), tmpkey)
    upload_path = '/tmp/resized-{}'.format(tmpkey)
    s3_client.download_file(bucket, key, download_path)
    resize_image(download_path, upload_path)
    s3_client.upload_file(upload_path, '{}-resized'.format(bucket), 'resized-
{}'.format(key))
```

2. 在创建 `lambda_function.py` 文件的同一目录中，创建一个名为 `package` 的新目录并安装 Pillow（PIL）库和 AWS SDK for Python (Boto3)。虽然 Lambda Python 运行时系统包含 Boto3 SDK 的一个版本，但建议您将所有函数的依赖项添加到部署包中，即使这些依赖项已经包含在了运行时系统中。有关更多信息，请参阅 Python 中的运行时系统依赖项。

```
mkdir package
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.12 \
--only-binary=:all: --upgrade \
pillow boto3
```

Pillow 库包含 C/C++ 代码。通过使用 `--platform manylinux_2014_x86_64` 和 `--only-binary=:all:` 选项，pip 将下载并安装包含与 Amazon Linux 2 操作系统兼容的预编译二进制文件的 Pillow 版本。这可以确保无论本地构建计算机的操作系统和架构如何，部署包都能在 Lambda 执行环境中正常发挥作用。

3. 创建包含应用程序代码和 Pillow 以及 Boto3 库的 zip 文件。在 Linux 或 MacOS 中，从命令行界面运行以下命令。

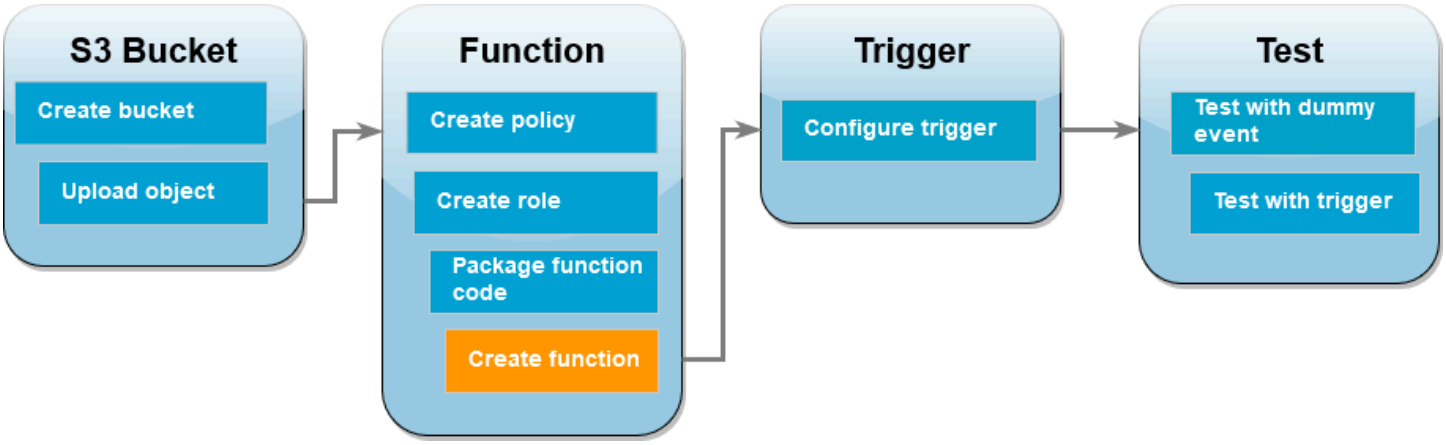
```
cd package
```

```
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

在 Windows 中，使用您首选的压缩工具来创建 `lambda_function.zip` 文件。确保您的 `lambda_function.py` 文件和包含依赖项的文件夹都位于.zip 文件的根目录下。

您也可以使用 Python 虚拟环境创建部署包。请参阅 将 .zip 文件归档用于 Python Lambda 函数。

创建 Lambda 函数



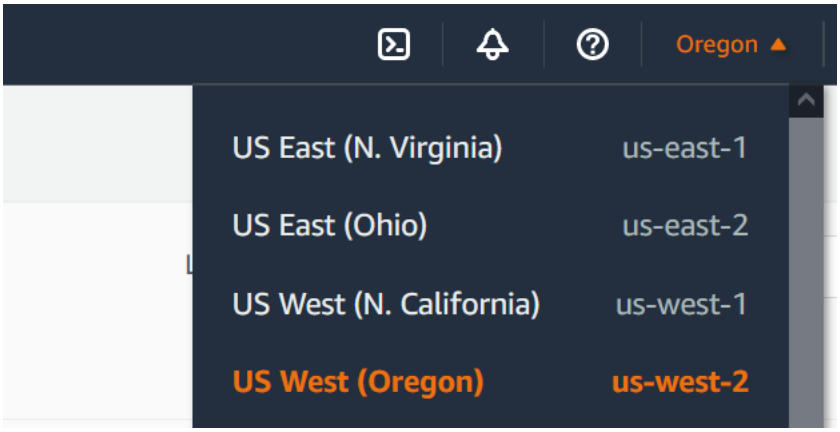
您也可以使用 AWS CLI 或 Lambda 控制台创建 Lambda 函数。按照所选语言的说明创建函数。

AWS Management Console

创建函数（控制台）

要使用控制台创建 Lambda 函数，首先要创建包含一些“Hello world”代码的基本函数。然后，通过上传在上一步中创建的 .zip 或 JAR 文件，将此代码替换为自己的函数代码。

1. 打开 Lamba 控制台的函数页面。
2. 确保您在创建 Amazon S3 存储桶所在的同一 AWS 区域 内操作。您可以使用屏幕顶部的下拉列表更改区域。



3. 选择 Create function (创建函数)。
4. 选择从头开始创作。
5. 在基本信息中，执行以下操作：
   a. 对于 Function name（函数名称），请输入 **`CreateThumbnail`**。
   b. 对于运行时，根据您为函数选择的语言，选择 Node.js 20.x 或 Python 3.12。
   c. 对于架构，选择 x86_64。
6. 在更改默认执行角色选项卡中，执行以下操作：
   a. 展开选项卡，然后选择使用现有角色。
   b. 选择您之前创建的 `LambdaS3Role`。
7. 选择 Create function (创建函数)。

上传函数代码（控制台）

1. 在代码源窗格中，选择上传自。
2. 选择 .zip 文件。
3. 选择上传。
4. 在文件选择器中，选择 .zip 文件，然后选择打开。
5. 选择保存。

AWS CLI

创建函数（AWS CLI）

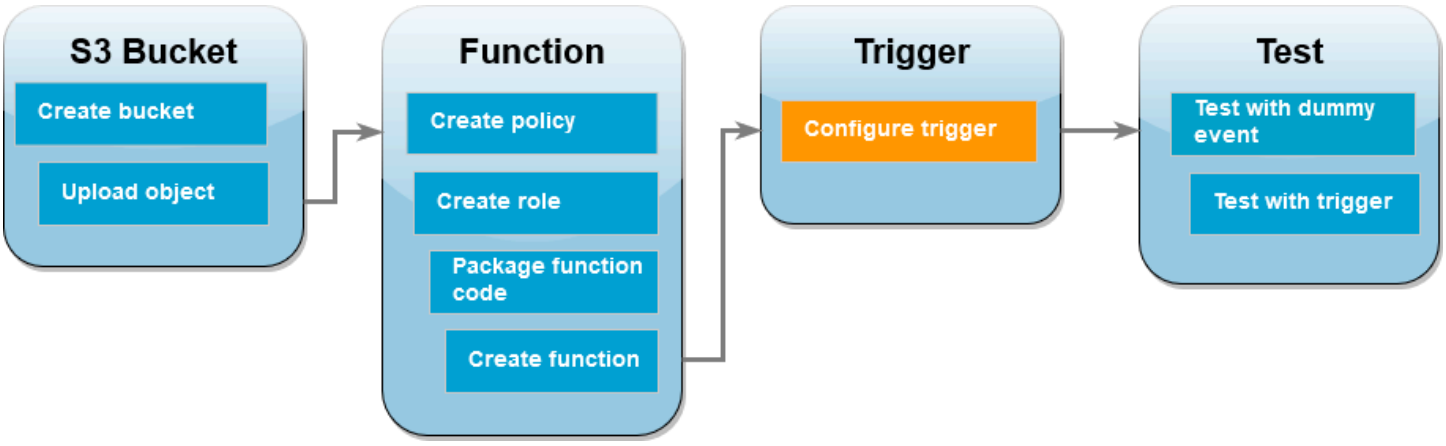
- 运行您所选语言的 CLI 命令。对于 `role` 参数，确保将 `123456789012` 替换为自己的 AWS 账户 ID。对于 `region` 参数，将 `us-east-1` 替换为在其中创建 Amazon S3 存储桶的区域。
  - 对于 Node.js，从包含 `function.zip` 文件的目录中运行以下命令。

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs20.x \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-1
```

  - 对于 Python，从包含 `lambda_function.zip` 文件的目录中运行以下命令。

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://lambda_function.zip --handler
 lambda_function.lambda_handler \
--runtime python3.12 --timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-1
```

配置 Amazon S3 来调用函数



要在将图像上传到源存储桶时运行 Lambda 函数，您需要为函数配置触发器。您可以使用控制台或 AWS CLI 配置 Amazon S3 触发器。

> **Important**
>
> 此程序将 Amazon S3 存储桶配置为每次在此存储桶中创建对象时调用您的函数。请确保仅在源存储桶上配置。如果您的 Lambda 函数在调用此函数的同一个存储桶中创建对象，则可以在循环中持续调用您的函数。这可能会导致您的 AWS 账户 产生额外费用。

AWS Management Console

配置 Amazon S3 触发器（控制台）

1. 打开 Lambda 控制台的函数页面，然后选择函数 (`CreateThumbnail`)。
2. 选择添加触发器。
3. 选择 S3。
4. 在存储桶下，选择自己的源存储桶。
5. 在事件类型下，选择所有对象创建事件。
6. 在递归调用下，选中复选框以确认知晓不建议使用相同的 Amazon S3 存储桶用于输入和输出。您可以阅读 Serverless Land 中的 Recursive patterns that cause run-away Lambda functions，进一步了解 Lambda 中的递归调用模式。
7. 选择添加。

   在您使用 Lambda 控制台创建触发器时，Lambda 会自动创建基于资源的策略，授予您选择的服务调用函数的权限。

AWS CLI

配置 Amazon S3 触发器（AWS CLI）

1. 要让 Amazon S3 源存储桶在添加图像文件时调用函数，您首先需要使用基于资源的策略为函数配置权限。基于资源的策略声明授予其他 AWS 服务 调用您函数的权限。要授予 Amazon S3 调用函数的权限，请运行以下 CLI 命令。请务必将 `source-account` 参数替换为您的 AWS 账户 ID 并使用自己的源存储桶名称。

```
aws lambda add-permission --function-name CreateThumbnail \
--principal s3.amazonaws.com --statement-id s3invoke --action
 "lambda:InvokeFunction" \
--source-arn arn:aws:s3:::amzn-s3-demo-source-bucket \
```

```
--source-account 123456789012
```

您使用此命令定义的策略允许 Amazon S3 仅在源存储桶上执行操作时调用函数。

> **Note**
>
> 虽然 Amazon S3 存储桶名称具有全局唯一性，但在使用基于资源的策略时，最佳做法是指定存储桶必须属于您的账户。这是因为，如果删除一个存储桶，则另一个 AWS 账户 账户可能会创建具有相同 Amazon 资源名称（ARN）的存储桶。

2. 将下列 JSON 保存在名为 `notification.json` 的文件中。在应用到您的源存储桶时，此 JSON 会将存储桶配置为在每次添加新对象时向 Lambda 函数发送通知。将 Lambda 函数 ARN 中的 AWS 账户 号码和 AWS 区域 替换为您自己的账号和区域。

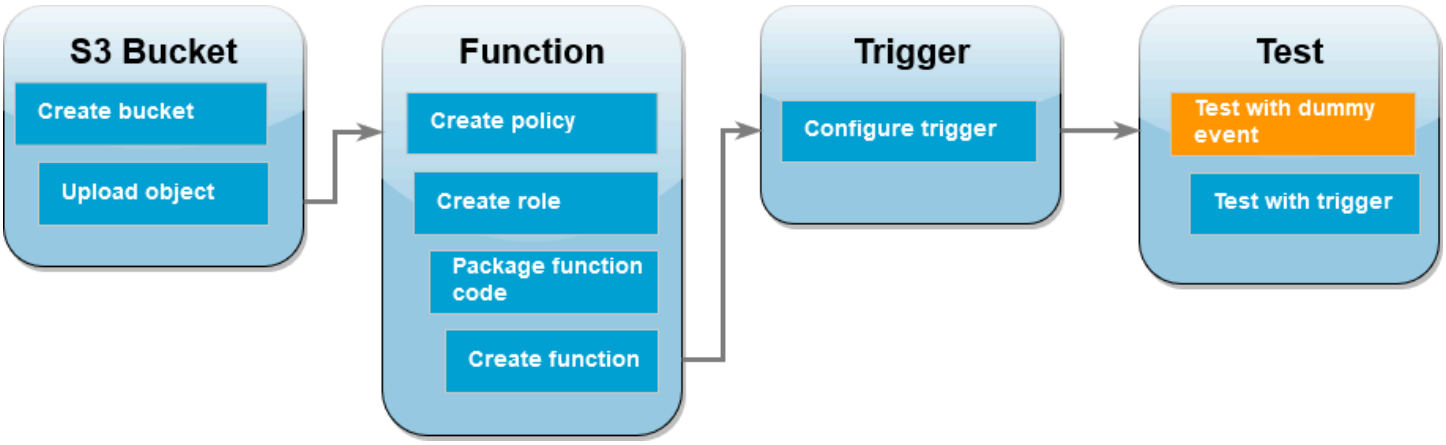
```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "CreateThumbnailEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:CreateThumbnail",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. 运行以下 CLI 命令，将您创建的 JSON 文件中的通知设置应用到源存储桶。将 `amzn-s3-demo-source-bucket` 替换为源存储桶的名称。

```
aws s3api put-bucket-notification-configuration --bucket amzn-s3-demo-source-
bucket \
--notification-configuration file://notification.json
```

要了解有关 `put-bucket-notification-configuration` 命令和 `notification-configuration` 选项的更多信息，请参阅 AWS CLI 命令参考中的 put-bucket-notification-configuration。

使用虚拟事件测试 Lambda 函数



在通过向 Amazon S3 源存储桶添加图像文件来测试整个设置之前，您可以通过使用虚拟事件调用 Lambda 函数来测试此函数是否正常运行。Lambda 中的事件是 JSON 格式的文档，其中包含要处理的函数数据。Amazon S3 调用您的函数时，发送到函数的事件包含存储桶名称、存储桶 ARN 和对象键等信息。

AWS Management Console

使用虚拟事件测试 Lambda 函数（控制台）

1. 打开 Lambda 控制台的函数页面，然后选择函数 (`CreateThumbnail`)。
2. 选择测试选项卡。
3. 要创建测试事件，在测试事件窗格中，执行以下操作：
   a. 在测试事件操作下，选择创建新事件。
   b. 对于事件名称，输入 **`myTestEvent`**。
   c. 在模板中，选择 S3 Put。
   d. 将以下参数的值替换为您自己的值。
      - 对于 `awsRegion`，将 `us-east-1` 替换为在其中创建 Amazon S3 存储桶的 AWS 区域。
      - 对于 `name`，将 `amzn-s3-demo-bucket` 替换为您自己的 Amazon S3 源存储桶的名称。
      - 对于 `key`，将 `test%2Fkey` 替换为您在步骤 将测试图片上传到源存储桶 中上传到源存储桶的测试对象的文件名。

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

e. 选择保存。

4. 在测试事件窗格中，选择测试。
5. 要检查您的函数是否已创建图像的调整大小版本并将其存储在目标 Amazon S3 存储桶中，请执行以下操作：

   a. 打开 Amazon S3 控制台的存储桶页面。

   b. 选择目标存储桶并确认调整大小的文件已在对象窗格中列出。

AWS CLI

使用虚拟事件测试 Lambda 函数（AWS CLI）

1. 将下列 JSON 保存在名为 dummyS3Event.json 的文件中。将以下参数的值替换为您自己的值：

   - 对于 awsRegion，将 us-east-1 替换为在其中创建 Amazon S3 存储桶的 AWS 区域。
   - 对于 name，将 amzn-s3-demo-bucket 替换为您自己的 Amazon S3 源存储桶的名称。
   - 对于 key，将 test%2Fkey 替换为您在步骤 将测试图片上传到源存储桶 中上传到源存储桶的测试对象的文件名。

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
```

```
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

2. 在保存 dummyS3Event.json 文件的目录中，运行以下 CLI 命令来调用函数。此命令通过指定 RequestResponse 作为调用类型参数的值来同步调用 Lambda 函数。要了解有关同步和异步调用的更多信息，请参阅调用 Lambda 函数。

```
aws lambda invoke --function-name CreateThumbnail \
--invocation-type RequestResponse --cli-binary-format raw-in-base64-out \
--payload file://dummyS3Event.json outputfile.txt
```

如果您使用的是 AWS CLI 版本 2，则 cli-binary-format 选项必填。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅AWS CLI 支持的全局命令行选项。

3. 验证您的函数是否已创建图像的缩略图版本并已保存到目标 Amazon S3 存储桶中。运行以下 CLI 命令，将 amzn-s3-demo-source-bucket-resized 替换为自己的目标存储桶的名称。

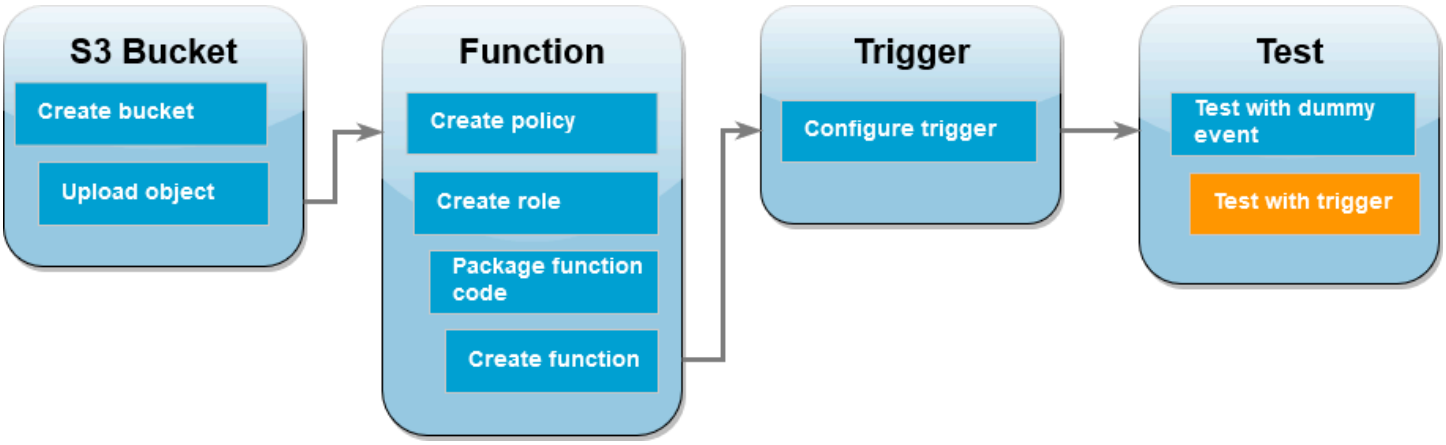
```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

您应该可以看到类似于如下所示的输出内容。Key 参数显示调整大小后的图像文件的文件名。

```
{
```

```
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-06T21:40:07+00:00",
            "ETag": "\"d8ca652ffe83ba6b721ffc20d9d7174a\"",
            "Size": 2633,
            "StorageClass": "STANDARD"
        }
    ]
}
```

使用 Amazon S3 触发器测试函数



此时，您已确认 Lambda 函数运行正常，您可以通过向 Amazon S3 源存储桶添加图像文件来测试完整设置。将图像添加到源存储桶时，应自动调用 Lambda 函数。函数会创建文件已调整大小的版本并将其存储在目标存储桶中。

AWS Management Console

使用 Amazon S3 触发器测试 Lambda 函数（控制台）

1. 要将图像上传到 Amazon S3 存储桶，请执行以下操作：
   a. 打开 Amazon S3 控制台的存储桶页面，然后选择您的源存储桶。
   b. 选择上传。
   c. 选择添加文件，然后使用文件选择器选择要上传的图像文件。您的图像对象可以是任何.jpg 或.png 文件。
   d. 选择打开，然后选择上传。
2. 执行以下操作，验证 Lambda 是否已将调整大小后的图像文件保存在目标存储桶中：

a. 导航回 Amazon S3 控制台的存储桶页面，然后选择您的目标存储桶。

b. 在对象窗格中，您现在应该能看到两个已调整大小的图像文件，其中一个来自 Lambda 函数的每次测试。要下载已调整大小的图像，请选择所需文件，然后选择下载。

AWS CLI

使用 Amazon S3 触发器测试 Lambda 函数（AWS CLI）

1. 在包含要上传的图像的目录中，运行以下 CLI 命令。将 `--bucket` 参数替换为源存储桶的名称。对于`--key` 和 `--body` 参数，请使用测试图像的文件名。您的测试图像可以是任何.jpg 或.png 文件。

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key SmileyFace.jpg --body ./SmileyFace.jpg
```

2. 验证您的函数是否已创建图像的缩略图版本并已保存到目标 Amazon S3 存储桶中。运行以下 CLI 命令，将 `amzn-s3-demo-source-bucket-resized` 替换为自己的目标存储桶的名称。

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

如果函数成功运行，您将看到类似于以下内容的输出。您的目标存储桶现在应该包含两个已调整大小的文件。

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        },
        {
            "Key": "resized-SmileyFace.jpg",
            "LastModified": "2023-06-07T00:13:18+00:00",
            "ETag": "\"ca536e5a1b9e32b22cd549e18792cdbc\"",
            "Size": 1245,
            "StorageClass": "STANDARD"
```

```
        }
    ]
}
```

清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择 Delete（删除）。

删除您创建的策略

1. 打开 IAM 控制台的 Policies（策略）页面。
2. 选择您创建的策略 (AWSLambdaS3Policy)。
3. 选择 Policy actions（策略操作）、Delete（删除）。
4. 选择 Delete（删除）。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。
4. 在文本输入字段中输入角色名称，然后选择 Delete（删除）。

删除 S3 存储桶

1. 打开 Amazon S3 控制台。
2. 选择您创建的存储桶。
3. 选择删除。

4. 在文本输入字段中输入存储桶的名称。
5. 选择删除存储桶。

将 Lambda 与 Amazon SQS 结合使用

> **Note**
>
> 如果想要将数据发送到 Lambda 函数以外的目标，或要在发送数据之前丰富数据，请参阅 Amazon EventBridge Pipes（Amazon EventBridge 管道）。

您可以使用 Lambda 函数来处理某个 Amazon Simple Queue Service（Amazon SQS）队列中的消息。Lambda 支持事件源映射的标准队列和先进先出（FIFO）队列。Lambda 函数和 Amazon SQS 队列必须位于同一 AWS 区域，即便二者可能位于不同的 AWS 账户。

主题

了解 Amazon SQS 事件源映射的轮询和批处理行为

使用 Amazon SQS 事件源映射，Lambda 会轮询队列并通过事件同步调用您的函数。每个事件可以包含来自队列的一批多条消息。Lambda 每次接收一批这些事件，并为每个批次调用一次您的函数。当您的函数成功处理一个批次后，Lambda 就会将其消息从队列中删除。

当 Lambda 接收某个批次时，消息将保留在队列中，但会根据队列的可见性超时长度隐藏。如果您的函数成功处理了批次中的所有消息，Lambda 会将消息从队列中删除。预设情况下，如果您的函数在处理某个批处理时遇到错误，则该批处理中的所有消息都会在可见性超时过期后在队列中重新可见。因此，函数代码必须能够多次处理同一条消息，而不会产生意外的副作用。

> **Warning**
>
> Lambda 事件源映射至少处理每个事件一次，有可能出现重复处理记录的情况。为避免与重复事件相关的潜在问题，我们强烈建议您将函数代码设为幂等性。要了解更多信息，请参阅 AWS 知识中心的如何使我的 Lambda 函数具有幂等性。

要防止 Lambda 多次处理消息，您可以将事件源映射配置为在函数响应中包含批次项目失败，也可以在 Lambda 函数成功处理消息后使用 DeleteMessage API 将消息从队列中删除。

有关 Lambda 支持的 SQS 事件源映射配置参数的更多信息，请参阅 the section called “创建 SQS 事件源映射”。

示例标准队列消息事件

Example Amazon SQS 消息事件（标准队列）

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        },
        {
            "messageId": "2e1424d4-f796-459a-8184-9c92662be6da",
            "receiptHandle": "AQEBzWwaftRI0KuVm4tP+/7q1rGgNqicHq...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
```

```
                "SentTimestamp": "1545082650636",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082650649"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        }
    ]
}
```

默认情况下，Lambda 将一次性轮询队列中最多 10 条消息，并将该批次发送到函数。为避免在记录数量较少的情况下调用该函数，您可以配置批次时段，将事件源配置为缓冲最多五分钟的记录。在调用函数之前，Lambda 将继续轮询标准队列中的消息，直到批次时段到期、达到调用有效负载大小配额或达到配置的最大批次大小为止。

如果您使用的是批处理窗口，并且 SQS 队列包含的流量非常低，Lambda 可能会等待最多 20 秒钟才能调用您的函数。即使您将批处理窗口设置为低于 20 秒，情况依然如此。

> **Note**
>
> 在 Java 中，反序列化 JSON 时可能会遇到空指针错误。这可能要归因于 JSON 对象映射器转换“Records”和“eventSourceARN”大小写的方式。

示例 FIFO 队列消息事件

对于 FIFO 队列，记录包含与重复数据消除和顺序相关的其他属性。

Example Amazon SQS 消息事件（FIFO 队列）

```
{
    "Records": [
        {
            "messageId": "11d6ee51-4cc7-4302-9e22-7cd8afdaadf5",
            "receiptHandle": "AQEBBX8nesZEXmkhsmZeyIE8iQAMig7qw...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
```

```
                "SentTimestamp": "1573251510774",
                "SequenceNumber": "18849496460467696128",
                "MessageGroupId": "1",
                "SenderId": "AIDAIO23YVJENQZJOL4VO",
                "MessageDeduplicationId": "1",
                "ApproximateFirstReceiveTimestamp": "1573251510774"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:fifo.fifo",
            "awsRegion": "us-east-2"
        }
    ]
}
```

创建和配置 Amazon SQS 事件源映射

要使用 Lambda 处理 Amazon SQS 消息，请使用适当的设置配置您的队列，然后创建 Lambda 事件源映射。

配置队列以便用于 Lambda

如果您没有现有的 Amazon SQS 队列，则创建一个队列，用作 Lambda 函数的事件源。Lambda 函数和 Amazon SQS 队列必须位于同一 AWS 区域，即便二者可能位于不同的 AWS 账户。

为使函数有时间处理每批记录，请将源队列的可见性超时设置为函数配置超时的至少六倍。这一额外的时间允许 Lambda 在您的函数处理之前的批次期间遇到限流时进行重试。

默认情况下，如果 Lambda 在处理某个批次期间的任何时候遇到错误，则该批次中的所有消息都会返回到队列。可见性超时后，Lambda 将再次看到这些消息。您可以将事件源映射配置为使用部分批次响应，以仅使失败的消息返回队列。此外，如果函数多次都未能处理某条消息，则 Amazon SQS 可以将其发送到某个死信队列。建议将源队列的重新驱动策略的 `maxReceiveCount` 设置为至少 5。这让 Lambda 在将失败的消息直接发送到死信队列之前有几次重试的机会。

设置 Lambda 执行角色权限

AWSLambdaSQSQueueExecutionRole AWS 托管策略包含 Lambda 从您的 Amazon SQS 队列中读取所需的权限。您可以将此托管策略添加到您的函数的执行角色。

或者，如果您使用的是加密队列，则还需要为执行角色添加以下权限：

- kms:Decrypt

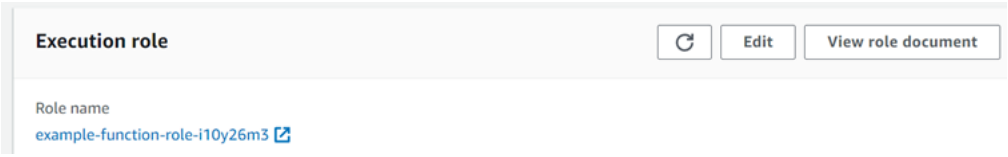
创建 SQS 事件源映射

创建事件源映射以指示 Lambda 将队列中的项目发送到 Lambda 函数。您可以创建多个事件源映射，以使用单个函数处理来自多个队列的项目。当 Lambda 调用目标函数时，事件可以包含多个项目（最多为可配置的最大批处理大小）。

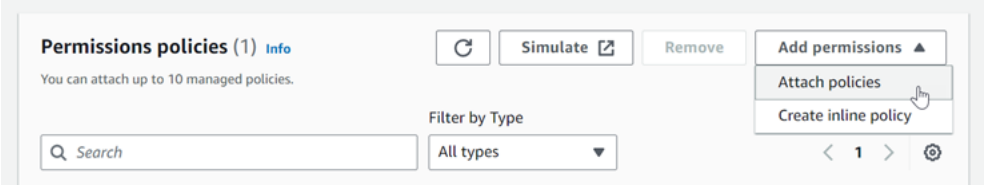
要将您的函数配置为从 Amazon SQS 中读取，请将 AWSLambdaSQSQueueExecutionRole AWS 托管策略附加到您的执行角色。然后，使用以下步骤从控制台创建 SQS 事件源映射。

要添加权限并创建触发器

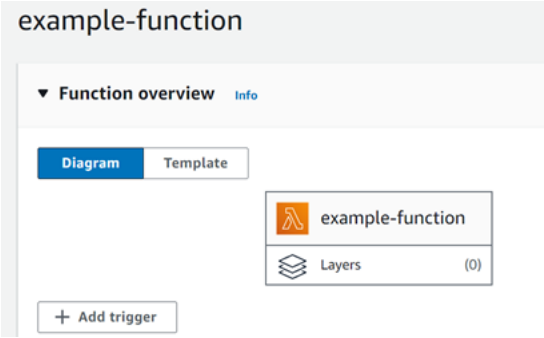
1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数的名称。
3. 选择 Configuration（配置）选项卡，然后选择 Permissions（权限）。
4. 在角色名称下，选择至执行角色的链接。此角色将在 IAM 控制台中打开角色。



5. 选择添加权限，然后选择附加策略。



6. 在搜索字段中输入 AWSLambdaSQSQueueExecutionRole。向执行角色添加此策略。这是一项 AWS 托管策略，其中包含您的函数从 Amazon SQS 队列中读取所需的权限。有关此策略的更多信息，请参阅《AWS Managed Policy Reference》中的 AWSLambdaSQSQueueExecutionRole。
7. 在 Lambda 控制台中返回您的函数。在 Function overview（函数概览）下，选择 Add trigger（添加触发器）。



8. 选择触发器类型。
9. 配置必填选项，然后选择 Add（添加）。

Lambda 支持 Amazon SQS 事件源的以下配置选项：

SQS 队列

要从中读取记录的 Amazon SQS 队列。Lambda 函数和 Amazon SQS 队列必须位于同一 AWS 区域，即便二者可能位于不同的 AWS 账户。

启用触发器

事件源映射的状态。Enable trigger（启用触发器）默认处于选中状态。

批次大小

每个批次中要发送给函数的最大记录数。对于标准队列，这最高可为 10,000 条记录。对于 FIFO 队列，最大值为 10。对于超过 10 的批处理大小，还必须将批处理时间（`MaximumBatchingWindowInSeconds`）设置为至少 1 秒。

配置函数超时，以允许有足够的时间来处理整个批次的项目。如果项目处理需要很长时间，请选择一个较小的批处理大小。大批量处理可以提高非常快速或拥有大量开销的工作负载的效率。如果您在函数上配置了预留并发，请将最小并发执行数设置为 5，以降低在 Lambda 调用函数时出现节流错误的几率。

Lambda 通过单个调用将批次中的所有记录传递给函数，前提是事件的总大小未超出同步调用的调用有效负载大小配额（6 MB）。Lambda 和 Amazon SQS 都会为每条记录生成元数据。这一额外的元数据将会计入总有效负载大小，并且可能导致批处理中发送的记录总数低于配置的批处理大小。Amazon SQS 发送的元数据字段的长度是可变的。有关 Amazon SQS 元数据字段的更多信息，请参阅《Amazon Simple Queue Service API 参考》中的 ReceiveMessage API 操作文档。

Batch 时间

在调用函数之前收集记录的最长时间（以秒为单位）。它仅适用于标准队列。

如果您使用的批次时段大于 0 秒，则必须考虑队列可见性超时中增加的处理时间。我们建议将队列可见性超时设置为函数超时的六倍，加上 `MaximumBatchingWindowInSeconds` 的值。这使 Lambda 函数有时间处理每个批次的事件，并在出现节流错误时重试。

当消息可用时，Lambda 开始批量处理消息。Lambda 通过五次并发调用您的函数开始一次处理五个批处理。如果仍有消息可用，则 Lambda 最多每分钟添加 300 个函数实例，最多为 1000 个函数实例。要了解函数扩展和并发的更多信息，请参阅 Lambda 函数扩展。

要处理更多消息，您可以优化 Lambda 函数以提高吞吐量。有关更多信息，请参阅了解 AWS Lambda 如何使用 Amazon SQS 标准队列进行扩展。

最大并发数量

事件源可调用的最大并发函数数量。有关更多信息，请参阅 为 Amazon SQS 事件源配置最大并发。

筛选条件

添加筛选条件以控制 Lambda 将哪些事件发送给函数进行处理。有关更多信息，请参阅 控制 Lambda 向您的函数发送的事件。

为 SQS 事件源映射配置扩展行为

对于标准队列，Lambda 使用长轮询来轮询一个队列，直到它变为活动状态。当消息可用时，Lambda 会通过函数的五次并发调用开始一次处理五个批次。如果仍有消息可用，则 Lambda 增加批量读取的进程数，最多每分钟增加 300 个实例。事件源映射可以同时处理的最大批处理数量为 1,000。

对于 FIFO 队列，Lambda 按照接收消息的顺序向函数发送消息。向 FIFO 队列发送消息时，需要指定消息组 ID。Amazon SQS 确保同一组中的消息按顺序传递到 Lambda。当 Lambda 按批次读取消息时，每个批次可能包含来自多个消息组的消息，但消息的顺序保持不变。如果函数返回错误，函数会对受影响的消息尝试所有重试，然后 Lambda 才会接收来自同一个组的其他消息。

为 Amazon SQS 事件源配置最大并发

您可以使用最大并发设置来控制 SQS 事件源的扩展行为。最大并发设置限制了 Amazon SQS 事件源可以调用的函数的并发实例数。最大并发属于事件源级别的设置。如果您将多个 Amazon SQS 事件源映射到一个函数，则每个事件源均可进行单独的最大并发设置。您可以使用最大并发，从而防止某个队列使用函数的所有预留并发或账户的其余并发限额。在 Amazon SQS 事件源上配置最大并发不收取任何费用。

重要的是，最大并发和预留并发为两个独立的设置。不要将最大并发设置为高于函数的预留并发。配置最大并发后，请确保您的函数的预留并发大于或等于该函数上所有 Amazon SQS 事件源的最大总并发数。否则，Lambda 可能会限制您的消息。

当您账户的并发配额设置为默认值 1000 时，Amazon SQS 事件源映射可以扩展为调用最高此值的函数实例，除非您指定最大并发量。

如果您账户的默认并发配额有所增加，Lambda 可能无法调用最高达到新配额的并发函数实例。默认情况下，Lambda 可以扩展为针对 Amazon SQS 事件源映射调用最高 1250 个并发函数实例。如果这不足以满足您的应用场景，请联系 AWS 支持人员，讨论如何提高您账户的 Amazon SQS 事件源映射并发度。

> **Note**
>
> 对于 FIFO 队列，并发调用的上限为消息组 ID 的数量 (`messageGroupId`) 或最大并发设置，以较低者为准。例如，如果您有六个消息组 ID 并且最大并发设置为 10，则函数最多可以进行六次并发调用。

您可以在新的和现有的 Amazon SQS 事件源映射上配置最大并发。

使用 Lambda 控制台配置最大并发

1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数的名称。
3. 在 Function overview（函数概览）下，选择 SQS。此操作将打开 Configuration（配置）选项卡。
4. 选择 Amazon SQS 触发器，然后选择 Edit（编辑）。
5. 对于 Maximum concurrency（最大并发），输入 2 到 1,000 之间的数字。要关闭最大并发，将该框保留为空。
6. 选择保存。

使用 AWS Command Line Interface（AWS CLI）配置最大并发

使用带 `--scaling-config` 选项的 update-event-source-mapping 命令。例如：

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
```

```
    --scaling-config '{"MaximumConcurrency":5}'
```

要关闭最大并发，请为 `--scaling-config` 输入一个空值：

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config "{}"
```

使用 Lambda API 配置最大并发

将 CreateEventSourceMapping 或 UpdateEventSourceMapping 操作与 ScalingConfig 对象结合使用。

在 Lambda 中处理 SQS 事件源错误

为了处理与 SQS 事件源相关的错误，Lambda 会自动使用带有回退策略的重试策略。您还可以通过配置 SQS 事件源映射来返回部分批次响应，从而自定义错误处理行为。

失败调用的回退策略

如果调用失败，Lambda 会在实施回退策略时尝试重试调用。回退策略略有不同，具体取决于 Lambda 的故障原因是函数代码中的错误还是节流所致。

- 如果您的函数代码导致了该错误，Lambda 将停止处理并重试调用。同时，Lambda 会逐渐退出，以减少分配给 Amazon SQS 事件源映射的并发量。队列的可见性超时结束后，消息将再次显示在队列中。
- 如果调用失败是节流造成的，Lambda 会通过减少分配给 Amazon SQS 事件源映射的并发量来逐渐停止重试。Lambda 会继续重试该消息，直到消息的时间戳超过队列的可见性超时，此时 Lambda 会删除该消息。

实施部分批处理响应

预设情况下,如果您的 Lambda 函数在处理某个批处理时遇到错误，则该批处理中的所有消息都会在队列中重新可见，包括 Lambda 已经成功处理的消息。因此，您的函数最终可能会多次处理同一消息。

对于处理失败的批处理，要避免重新处理其中已经成功处理的消息，您可以将事件源映射配置为仅使失败的消息重新可见。这称为部分批处理响应。要开启部分批处理响应，请在配置事件源映射时为 FunctionResponseTypes 操作指定 `ReportBatchItemFailures`。这可以让您的函数返回部分成功，从而有助于减少对记录进行不必要的重试次数。

激活 ReportBatchItemFailures 后，当函数调用失败时，Lambda 不会 缩减消息轮询范围。如果您预计某些消息会失败，并且不希望这些失败影响消息处理速率，请使用 ReportBatchItemFailures。

> **Note**
>
> 在使用部分批处理响应时，请记住以下几点：
>
> - 如果您函数发现了一个异常，则整个批处理将被视为完全失败。
> - 如果您将此功能与 FIFO 队列结合使用，则您的函数应在第一次失败后停止处理消息，并返回 batchItemFailures 中的所有失败和未处理的消息。这有助于保持队列中消息的顺序。

激活部分批处理报告

1. 查看 实施部分批处理响应的最佳实践。
2. 运行以下命令来为您的函数激活 ReportBatchItemFailures。要检索事件源映射的 UUID，请运行 list-event-source-mappings AWS CLI 命令。

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--function-response-types "ReportBatchItemFailures"
```

3. 更新您的函数代码以捕获所有异常并在 batchItemFailures JSON 响应中返回处理失败的消息。batchItemFailures 响应必须包含消息 ID 列表，以作为 itemIdentifier JSON 值。

   例如，假设一个批处理有五条消息，消息 ID 分别为 id1、id2、id3、id4 和 id5。您的函数成功处理了 id1、id3 和 id5。要使消息 id2 和 id4 在队列中重新可见，您的函数应运行以下响应：

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "id2"
        },
        {
            "itemIdentifier": "id4"
        }
    ]
```

```
}
```

以下函数代码示例将返回批处理中处理失败消息的 ID 列表：

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 .NET 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
```

```
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Go 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "encoding/json"
```

```
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {

  if /* Your message processing condition here */ {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Java 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context)
 {

        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();
        String messageId = "";
        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
                //process your message
                messageId = message.getMessageId();
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures
 list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(messageId));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 JavaScript 进行 Lambda SQS 批处理项目失败。

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
    const batchItemFailures = [];
    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
    console.log(`Processed message: ${record.body}`);
}
```

报告使用 TypeScript 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure,
 SQSRecord } from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
```

```
    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 PHP 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }
```

```
    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS
 records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Python 进行 Lambda SQS 批处理项目失败。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event["Records"]:
            try:
                # process message
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Ruby 进行 Lambda SQS 批处理项目失败。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
```

```
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
  end
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Rust 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}
```

```
async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

如果处理失败的事件没有返回到队列，请参阅 AWS 知识中心中的 [如何排查 Lambda 函数 SQS ReportBatchItemFailures 问题？]。

成功和失败的条件

如果您的函数返回以下任意一项，则 Lambda 会将批处理视为完全成功：

- 空的 `batchItemFailures` 列表
- Null `batchItemFailures` 列表
- 空的 `EventResponse`
- Null `EventResponse`

如果您的函数返回以下任意一项，则 Lambda 会将批处理视为完全失败：

- JSON 响应无效
- 空字符串 `itemIdentifier`
- Null `itemIdentifier`

- 包含错误密钥名的 `itemIdentifier`
- 具有某个消息 ID 的 `itemIdentifier` 值不存在

CloudWatch 指标

要确定函数是否在正确报告批处理项目失败情况，您可以监控 Amazon CloudWatch 中的 `NumberOfMessagesDeleted` 和 `ApproximateAgeOfOldestMessage` Amazon SQS 指标。

- `NumberOfMessagesDeleted` 会跟踪从队列中删除的消息数量。如果该值降至 0，则表明您的函数响应没有正确返回失败的消息。
- `ApproximateAgeOfOldestMessage` 会跟踪最早的消息在队列中停留的时间。如果此指标急剧增加，则可能表明您的函数没有正确返回失败的消息。

Amazon SQS 事件源映射的 Lambda 参数

所有 Lambda 事件源类型共享相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，只有部分参数适用于 Amazon SQS。

参数	必需	默认值	备注
BatchSize	否	10	对于标准队列，最大值为 10000。对于 FIFO 队列，最大值为 10。
启用	否	真实	none
EventSourceArn	Y	不适用	数据流或流使用者的 ARN
FunctionName	是	不适用	none
FilterCriteria	否	不适用	控制 Lambda 向您的函数发送的事件
FunctionResponseTypes	否	不适用	要使您的函数报告某个批处理中的特定失败，请在 FunctionR

参数	必需	默认值	备注
			esponseTypes 中包含值 ReportBatchItemFailures 。有关更多信息，请参阅 实施部分批处理响应。
MaximumBatchingWindowInSeconds	否	0	none
ScalingConfig	否	不适用	为 Amazon SQS 事件源配置最大并发

对 Amazon SQS 事件源使用事件筛选

您可以使用事件筛选，控制 Lambda 将流或队列中的哪些记录发送给函数。有关事件筛选工作原理的一般信息，请参阅 the section called “事件筛选”。

本节重点介绍 Amazon MSK 事件源的事件筛选。

主题

- Amazon SQS 事件筛选基础知识

Amazon SQS 事件筛选基础知识

假设 Amazon SQS 队列包含以下 JSON 格式的消息。

```
{
    "RecordNumber": 1234,
    "TimeStamp": "yyyy-mm-ddThh:mm:ss",
    "RequestCode": "AAAA"
}
```

此队列的示例记录将如下所示。

```
{
```

```
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\n "RecordNumber": 1234,\n "TimeStamp": "yyyy-mm-ddThh:mm:ss",\n
 "RequestCode": "AAAA"\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
        },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-west-2:123456789012:my-queue",
    "awsRegion": "us-west-2"
}
```

要根据 Amazon SQS 消息的内容进行筛选，请使用 Amazon SQS 消息记录中的 body 键。假设您只想处理 Amazon SQS 消息中 RequestCode 为“BBBB”的记录。FilterCriteria 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RequestCode\" : [ \"BBBB\" ] } }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 Pattern 的值。

```
{
    "body": {
        "RequestCode": [ "BBBB" ]
        }
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```

假设您希望函数只处理 `RecordNumber` 大于 9999 的记录。`FilterCriteria` 对象将如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RecordNumber\" : [ { \"numeric\": [ \">\",
 9999 ] } ] } }"
        }
    ]
}
```

为了更清楚起见，以下是在纯 JSON 中展开的筛选条件 `Pattern` 的值。

```
{
    "body": {
        "RecordNumber": [
            {
                "numeric": [ ">", 9999 ]
            }
        ]
    }
}
```

您可以使用控制台、AWS CLI 或 AWS SAM 模板添加筛选条件。

Console

要使用控制台添加此筛选条件，请按照 将筛选条件附加到事件源映射（控制台） 中的说明，为筛选条件输入以下字符串。

```
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

AWS CLI

要使用 AWS Command Line Interface（AWS CLI）创建包含这些筛选条件的新事件源映射，请运行以下命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

要将这些筛选条件添加到现有事件源映射中，请运行以下命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

AWS SAM

要使用 AWS SAM 添加此筛选条件，请将以下代码段添加到事件源的 YAML 模板中。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }'
```

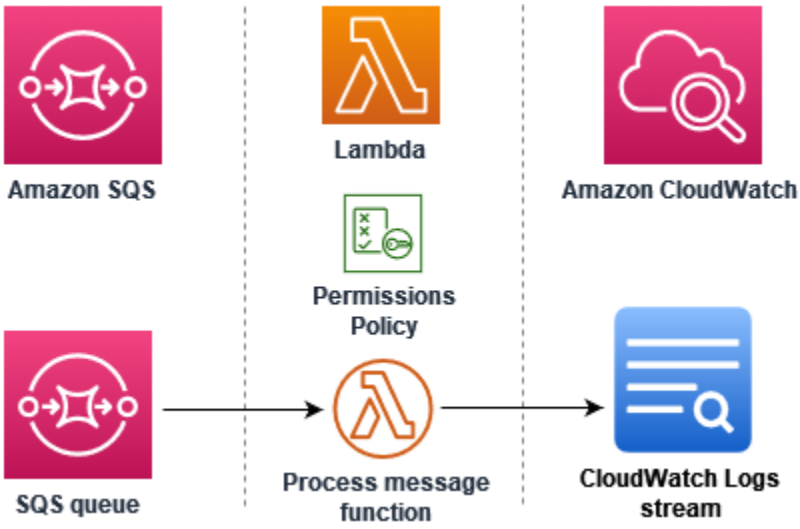
对于 Amazon SQS，消息正文可以是任何字符串。但如果您的 `FilterCriteria` 期望 body 为有效的 JSON 格式，则可能会导致问题。反之亦然：如果传入的消息正文为 JSON 格式，但筛选条件期望 body 为纯字符串，这可能会导致出现意外行为。

要避免此问题，请确保 `FilterCriteria` 中的正文格式与您从队列中收到的消息中的 body 的期望格式一致。在筛选消息之前，Lambda 会自动评估传入消息正文的格式以及 body 的筛选条件模式的格式。如果不一致，Lambda 将会删除此消息。下表汇总了此评估：

传入消息 **body** 格式	筛选条件模式 **body** 格式	导致的操作
纯字符串	纯字符串	Lambda 根据您的筛选条件进行筛选。
纯字符串	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
纯字符串	有效 JSON	Lambda 将会丢弃消息。
有效 JSON	纯字符串	Lambda 将会丢弃消息。
有效 JSON	数据属性中没有筛选条件模式	Lambda 根据您的筛选条件进行筛选（仅限其他元数据属性）。
有效 JSON	有效 JSON	Lambda 根据您的筛选条件进行筛选。

教程：将 Lambda 与 Amazon SQS 结合使用

在此教程中，您将创建一个会使用来自某个 Amazon Simple Queue Service（Amazon SQS）队列的消息的 Lambda 函数。只要有新消息添加到队列，Lambda 函数就会运行。函数会将消息写入 Amazon CloudWatch Logs 日志流。下图显示了您用于完成教程的 AWS 资源。



要完成本教程，请执行以下步骤：

1. 创建将消息写入 CloudWatch Logs 的 Lambda 函数。
2. 创建 Amazon SQS 队列。
3. 创建 Lambda 事件源映射。事件源映射会读取 Amazon SQS 队列并在添加新消息时调用 Lambda 函数。
4. 将消息添加到队列并在 CloudWatch Logs 中监控结果来测试设置。

先决条件

注册 AWS 账户

如果您还没有 AWS 账户，请完成以下步骤来创建一个。

注册 AWS 账户

1. 打开 https://portal.aws.amazon.com/billing/signup。
2. 按照屏幕上的说明进行操作。

   在注册时，将接到一通电话，要求使用电话键盘输入一个验证码。

   当您注册 AWS 账户时，系统将会创建一个 AWS 账户根用户。根用户有权访问该账户中的所有 AWS 服务 和资源。作为安全最佳实践，请为用户分配管理访问权限，并且只使用根用户来执行需要根用户访问权限的任务。

注册过程完成后，AWS 会向您发送一封确认电子邮件。在任何时候，您都可以通过转至 https://aws.amazon.com/ 并选择我的账户来查看当前的账户活动并管理您的账户。

创建具有管理访问权限的用户

注册 AWS 账户 后，请保护好您的 AWS 账户根用户，启用 AWS IAM Identity Center，并创建一个管理用户，以避免使用根用户执行日常任务。

保护您的 AWS 账户根用户

1. 选择根用户并输入您的 AWS 账户电子邮件地址，以账户拥有者身份登录 AWS Management Console。在下一页上，输入您的密码。

   要获取使用根用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的以根用户身份登录。
2. 为您的根用户启用多重身份验证 (MFA)。

   有关说明，请参阅《IAM 用户指南》中的为 AWS 账户 根用户启用虚拟 MFA 设备（控制台）。

创建具有管理访问权限的用户

1. 启用 IAM Identity Center。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的启用 AWS IAM Identity Center。
2. 在 IAM Identity Center 中，为用户授予管理访问权限。

   有关如何使用 IAM Identity Center 目录 作为身份源的教程，请参阅《AWS IAM Identity Center 用户指南》中的使用默认的 IAM Identity Center 目录 配置用户访问权限。

以具有管理访问权限的用户身份登录

- 要使用您的 IAM Identity Center 用户身份登录，请使用您在创建 IAM Identity Center 用户时发送到您的电子邮件地址的登录网址。

  要获取使用 IAM Identity Center 用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的登录 AWS 访问门户。

将访问权限分配给其他用户

1. 在 IAM Identity Center 中，创建一个权限集，该权限集遵循应用最低权限的最佳做法。

有关说明，请参阅《AWS IAM Identity Center 用户指南》中的创建权限集。

2. 将用户分配到一个组，然后为该组分配单点登录访问权限。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的添加组。

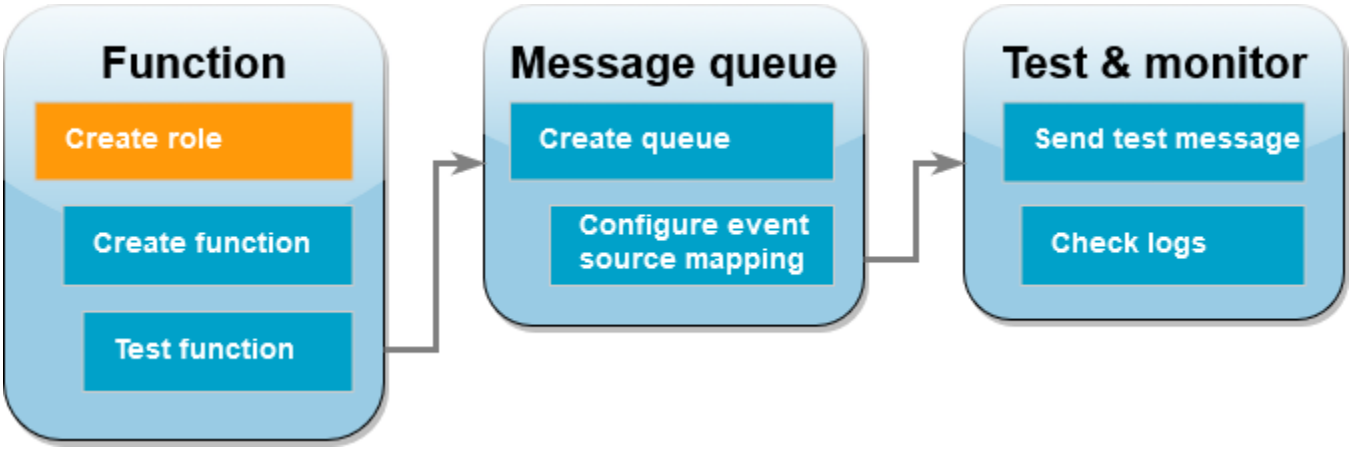
安装 AWS Command Line Interface

如果您尚未安装 AWS Command Line Interface，请按照安装或更新最新版本的 AWS CLI 中的步骤进行安装。

本教程需要命令行终端或 Shell 来运行命令。在 Linux 和 macOS 中，可使用您首选的 Shell 和程序包管理器。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常与 Lambda 一起使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。

创建执行角色



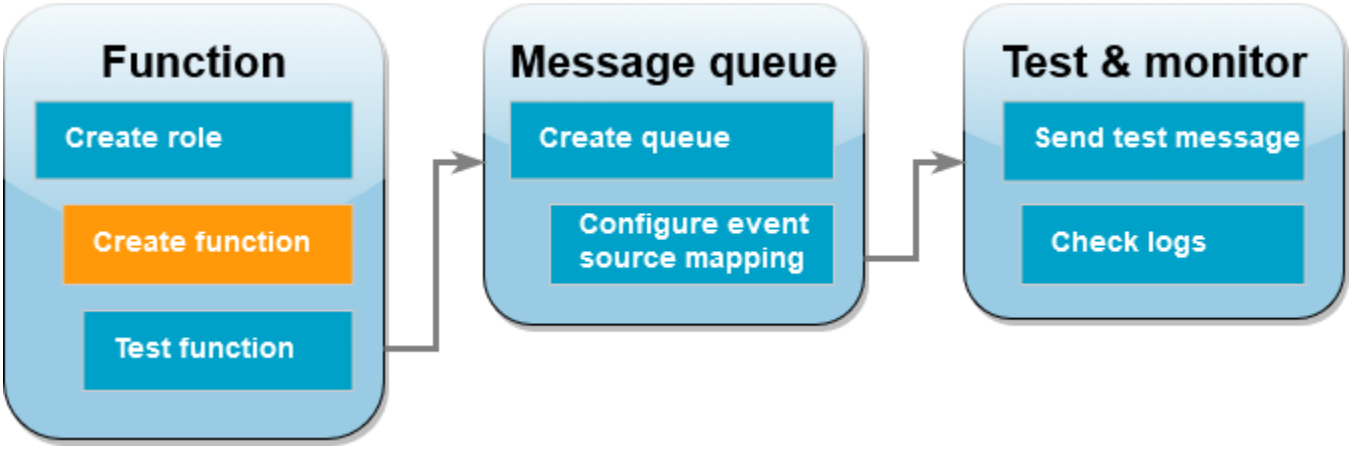
执行角色是一个 AWS Identity and Access Management（IAM）角色，用于向 Lambda 函数授予访问 AWS 服务和资源的权限。要允许函数从 Amazon SQS 中读取项目，请附加 AWSLambdaSQSQueueExecutionRole 权限策略。

创建执行角色并附加 Amazon SQS 权限策略

1. 打开 IAM 控制台的角色页面。
2. 选择 Create role（创建角色）。
3. 在可信实体类型中选择 AWS 服务。
4. 在使用案例中选择 Lambda。
5. 选择下一步。
6. 在权限策略搜索框中输入 **`AWSLambdaSQSQueueExecutionRole`**。
7. 选择 AWSLambdaSQSQueueExecutionRole 策略，然后选择下一步。
8. 在角色详细信息下，为角色名称输入 **`lambda-sqs-role`**，然后选择创建角色。

角色创建后，记下执行角色的 Amazon 资源名称（ARN）。您将在后面的步骤中用到它。

创建函数



创建一个处理您的 Amazon SQS 消息的 Lambda 函数。函数代码将 Amazon SQS 消息的正文记录到 CloudWatch Logs 中。

本教程使用 Node.js 18.x 运行时系统，但我们还提供了其他运行时系统语言的示例代码。您可以选择以下框中的选项卡，查看适用于您感兴趣的运行时系统的代码。您将在此步骤中使用的 JavaScript 代码是 JavaScript 选项卡中显示的第一个示例。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 .NET 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
```

```
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
```

```
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Java 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
```

```
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }

    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 JavaScript 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
```

```
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

通过 TypeScript 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";

export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 PHP 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
```

```
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 SQS 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Ruby 将 SQS 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Rust 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

创建 Node.js Lambda 函数

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir sqs-tutorial
cd sqs-tutorial
```

2. 将示例 JavaScript 代码复制到名为 `index.js` 的新文件中。
3. 使用以下 `zip` 命令创建部署包。

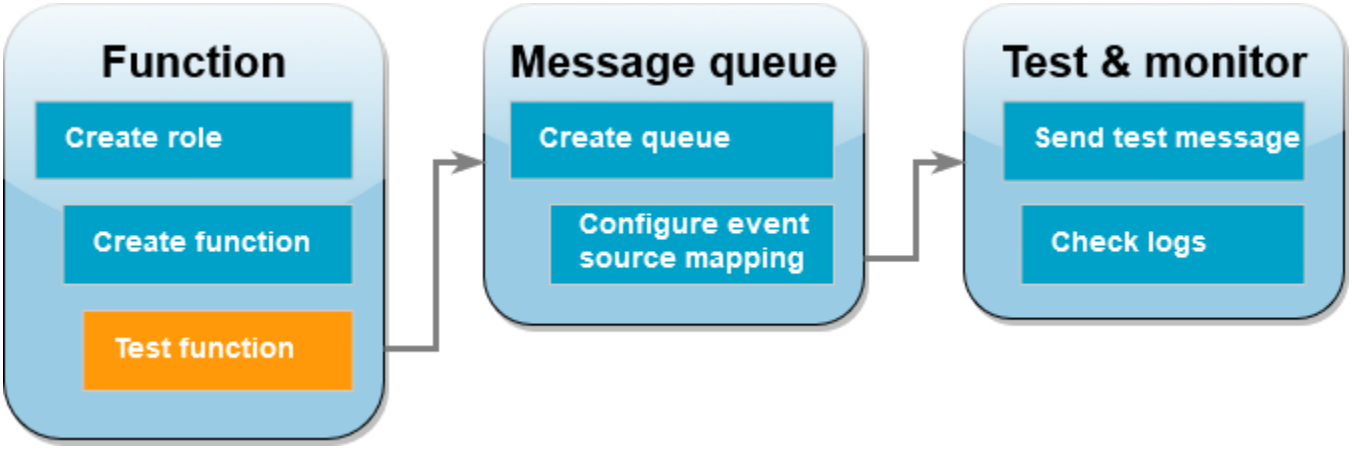
```
zip function.zip index.js
```

4. 使用 create-function AWS CLI 命令创建 Lambda 函数。对于 `role` 参数，请输入您之前创建的执行角色的 ARN。

> **Note**
>
> Lambda 函数和 Amazon SQS 队列必须位于同一 AWS 区域。

```
aws lambda create-function --function-name ProcessSQSRecord \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::111122223333:role/lambda-sqs-role
```

测试此函数



使用 `invoke` AWS CLI 命令和一个示例 Amazon SQS 事件手动调用您的 Lambda 函数。

使用示例事件调用 Lambda 函数

1. 将下列 JSON 保存为名为 `input.json` 的文件。此 JSON 模拟 Amazon SQS 可能发送到 Lambda 函数的事件，其中 `"body"` 包含来自该队列的实际消息。在本示例中，消息为 `"test"`。

   Example Amazon SQS 事件

   此为测试事件，无需您更改消息或账号。

```
{
```

```
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:my-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

2. 运行以下调用 AWS CLI 命令。此命令将在响应中返回 CloudWatch 日志。有关检索日志的更多信息，请参阅使用 AWS CLI 访问日志。

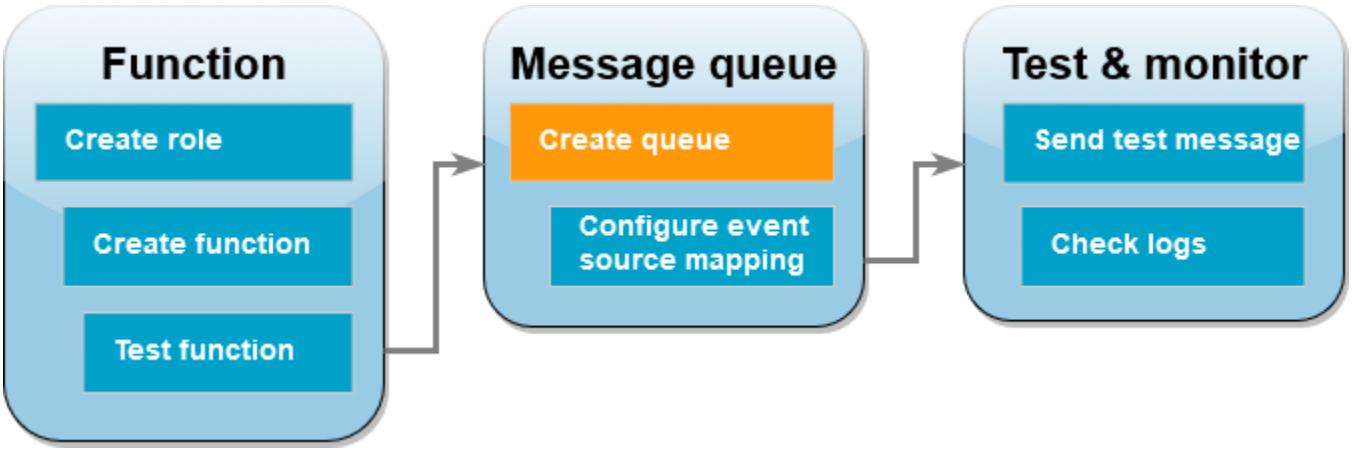
```
aws lambda invoke --function-name ProcessSQSRecord --payload file://input.json out
 --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

3. 在响应中查找 INFO 日志。Lambda 函数会在此处记录消息正文。应看到类似如下内容的日志：

```
2023-09-11T22:45:04.271Z 348529ce-2211-4222-9099-59d07d837b60 INFO Processed
 message test
2023-09-11T22:45:04.288Z 348529ce-2211-4222-9099-59d07d837b60 INFO done
```

创建 Amazon SQS 队列



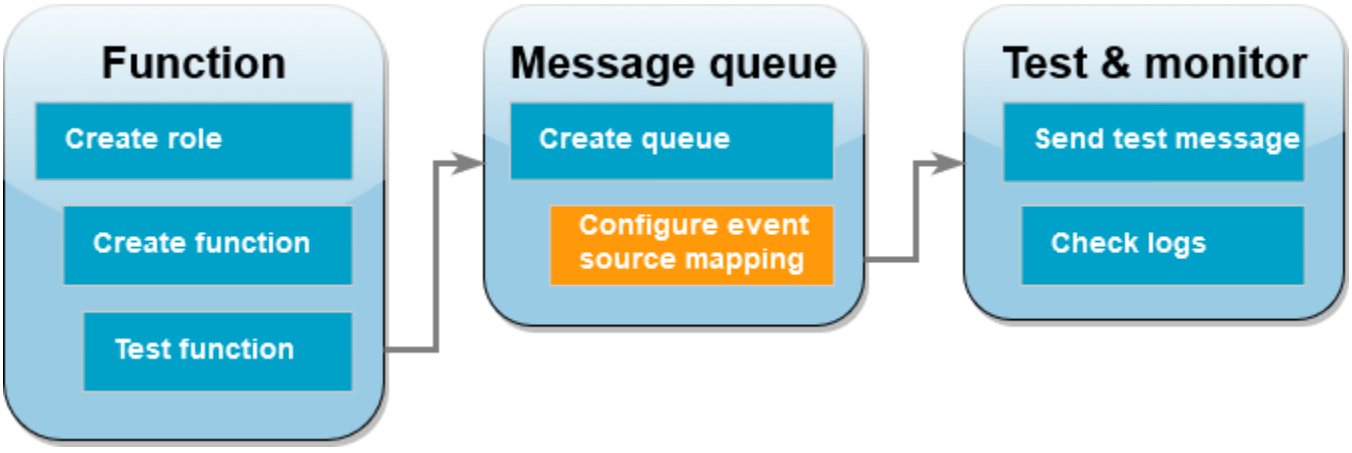
创建一个可由 Lambda 函数用作事件源的 Amazon SQS 队列。Lambda 函数和 Amazon SQS 队列必须位于同一 AWS 区域。

创建队列

1. 打开 Amazon SQS 控制台。
2. 选择创建队列。
3. 输入队列名称。将所有其他选项保留为默认设置。
4. 选择创建队列。

在创建队列后，记下其 ARN。在下一步中将该队列与您的 Lambda 函数关联时，您将需要此类信息。

配置事件源



创建事件源映射，将 Amazon SQS 队列连接到 Lambda 函数。事件源映射会读取 Amazon SQS 队列并在添加新消息时调用 Lambda 函数。

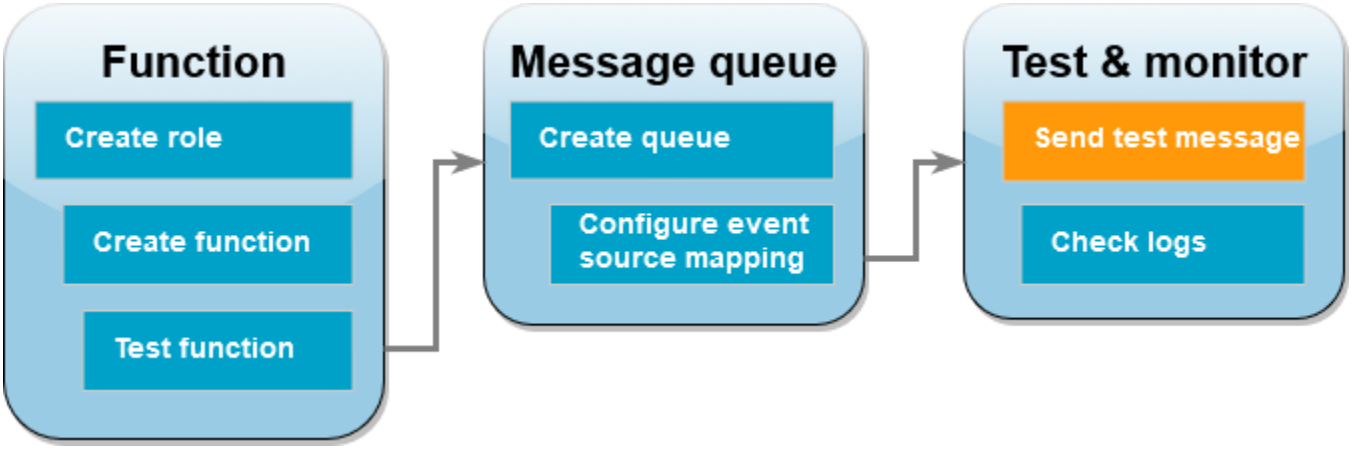
要在 Amazon SQS 队列与 Lambda 函数之间创建映射，请使用以下 create-event-source-mapping AWS CLI 命令。例如：

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord  --batch-size
 10 \
--event-source-arn arn:aws:sqs:us-east-1:111122223333:my-queue
```

要获取事件源映射列表，请使用 list-event-source-mappings 命令。例如：

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord
```

发送测试消息

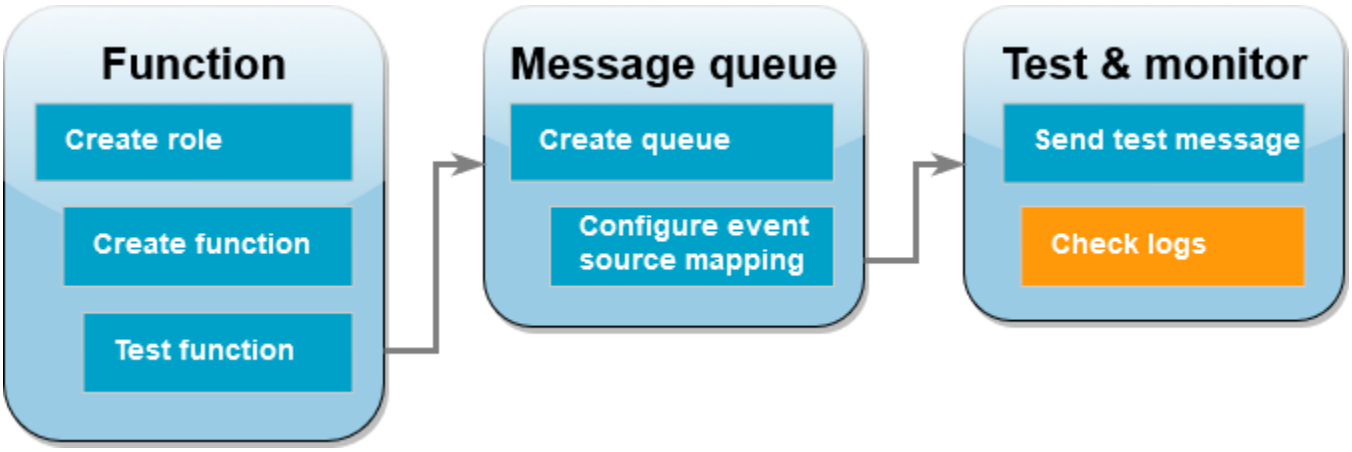


将 Amazon SQS 消息发送到 Lambda 函数

1. 打开 Amazon SQS 控制台。
2. 选择您之前创建的队列。
3. 选择发送和接收消息。
4. 在消息正文下输入测试消息，例如“这是一条测试消息”。
5. 选择 Send message（发送消息）。

Lambda 轮询队列以获取更新。当有新消息时，Lambda 会使用该新的事件数据从队列中调用您的函数。如果该函数处理程序正常返回并且没有异常，则 Lambda 认为该消息得到成功处理并开始读取队列中的新消息。成功处理消息后，Lambda 从队列中将其自动删除。如果该处理程序引发异常，则 Lambda 认为消息的批量处理并未成功进行，并且 Lambda 会用相同的批量消息调用该函数。

查看 CloudWatch 日志



确认函数已处理消息

1. 打开 Lamba 控制台的函数页面。
2. 选择 ProcessSQSRecord 函数。
3. 选择 Monitor (监控)。
4. 选择查看 CloudWatch 日志。
5. 在 CloudWatch 控制台中，为该函数选择日志流。
6. 查找 INFO 日志。Lambda 函数会在此处记录消息正文。您应该能看到从 Amazon SQS 队列发送的消息。例如：

```
2023-09-11T22:49:12.730Z b0c41e9c-0556-5a8b-af83-43e59efeec71 INFO Processed
 message this is a test message.
```

清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。
4. 在文本输入字段中输入角色名称，然后选择 Delete（删除）。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择 Delete（删除）。

删除 Amazon SQS 队列

1. 登录到 AWS Management Console 并打开 Amazon SQS 控制台，网址：https://console.aws.amazon.com/vpc/。
2. 选择创建的队列。
3. 选择删除。
4. 在文本输入字段中输入 **confirm**。
5. 选择 Delete（删除）。

教程：将跨账户 Amazon SQS 队列用作事件源

在此教程中，您将创建一个使用来自不同 AWS 账户中 Amazon Simple Queue Service (Amazon SQS) 队列的消息的 Lambda 函数。本教程涉及两个 AWS 账户：账户 A 指包含您的 Lambda 函数的账户，而账户 B 指包含 Amazon SQS 队列的账户。

先决条件

本教程假设您对 Lambda 基本操作和 Lambda 控制台有一定了解。如果您还没有了解，请按照 使用控制台创建 Lambda 函数 中的说明创建您的第一个 Lambda 函数。

要完成以下步骤，您需要 AWS CLI 版本 2。在单独的数据块中列出了命令和预期输出：

```
aws --version
```

您应看到以下输出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

对于长命令，使用转义字符 (\) 将命令拆分为多行。

在 Linux 和 macOS 中，可使用您首选的 shell 和程序包管理器。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常与 Lambda 一起使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。本指南中的示例 CLI 命令使用 Linux 格式。如果您使用的是 Windows CLI，则必须重新格式化包含内联 JSON 文档的命令。

创建执行角色（账户 A）

在账户 A 中，创建一个执行角色，该角色向您的函数授予访问所需 AWS 资源的权限。

创建执行角色

1. 在 AWS Identity and Access Management IAM 控制台中，打开 Roles (角色) 页面。
2. 选择 Create role（创建角色）。
3. 创建具有以下属性的角色。
   - Trusted entity (可信任的实体) – AWS Lambda
   - Permissions（权限） – AWSLambdaSQSQueueExecutionRole
   - Role name（角色名称）– **`cross-account-lambda-sqs-role`**

AWSLambdaSQSQueueExecutionRole 策略具有该函数从 Amazon SQS 中读取项目并将日志写入 Amazon CloudWatch Logs 所需的权限。

创建函数（账户 A）

在账户 A 中，创建一个处理您 Amazon SQS 消息的 Lambda 函数。Lambda 函数和 Amazon SQS 队列必须位于同一 AWS 区域。

下列 Node.js 18 代码示例将每条消息写入 CloudWatch Logs 中的日志。

Example index.mjs

```
export const handler = async function(event, context) {
  event.Records.forEach(record => {
```

```
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

创建函数

> **Note**
>
> 按照这些步骤在 Node.js 18 中创建一个函数。对于其他语言，步骤类似，但有些细节不同。

1. 将代码示例保存为名为 `index.mjs` 的文件。
2. 创建部署程序包。

```
zip function.zip index.mjs
```

3. 使用 `create-function` AWS Command Line Interface (AWS CLI) 命令创建函数。

```
aws lambda create-function --function-name CrossAccountSQSExample \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::<AccountA_ID>:role/cross-account-lambda-sqs-role
```

测试函数（账户 A）

在账户 A 中，使用 `invoke` AWS CLI 命令和示例 Amazon SQS 事件手动调用 Lambda 函数。

如果该处理程序正常返回并且没有异常，则 Lambda 认为该消息得到成功处理并开始读取队列中的新消息。成功处理消息后，Lambda 从队列中将其自动删除。如果该处理程序引发异常，则 Lambda 认为消息的批量处理并未成功进行，并且 Lambda 会用相同的批量消息调用该函数。

1. 将下列 JSON 保存为名为 `input.txt` 的文件。

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
```

```
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:example-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

上述 JSON 模拟 Amazon SQS 可能发送到您的 Lambda 函数的事件，其中 "body" 包含来自该队列的实际消息。

2. 运行以下 invoke AWS CLI 命令。

```
aws lambda invoke --function-name CrossAccountSQSExample \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 `aws configure set cli-binary-format raw-in-base64-out`。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

3. 在 `outputfile.txt` 文件中验证输出。

创建 Amazon SQS 队列（账户 B）。

在账户 B 中，创建一个可由账户 A 中 Lambda 函数用作事件源的 Amazon SQS 队列。Lambda 函数和 Amazon SQS 队列必须位于同一 AWS 区域。

创建队列

1. 打开 Amazon SQS 控制台。
2. 选择创建队列。
3. 创建具有以下属性的队列。

- 类型 - 标准
- 名称 - LambdaCrossAccountQueue
- 配置 - 保留默认设置。
- 访问策略 - 选择 Advanced (高级)。粘贴以下 JSON 策略：

```
{
   "Version": "2012-10-17",
   "Id": "Queue1_Policy_UUID",
   "Statement": [{
      "Sid":"Queue1_AllActions",
      "Effect": "Allow",
      "Principal": {
         "AWS": [
            "arn:aws:iam::<AccountA_ID>:role/cross-account-lambda-sqs-role"
         ]
      },
      "Action": "sqs:*",
      "Resource": "arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue"
    }
  ]
}
```

此策略授予账户 A 中 Lambda 执行角色权限，以使用 Amazon SQS 队列中的消息。

4. 创建队列后，记录其 Amazon Resource Name (ARN)。在下一步中将该队列与您的 Lambda 函数关联时，您将需要此类信息。

配置事件源（帐户 A）

在账户 A 中，通过运行以下 `create-event-source-mapping` AWS CLI 命令创建账户 B 中 Amazon SQS 队列和 Lambda 函数之间的事件源映射。

```
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-
size 10 \
--event-source-arn arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

要获取您的事件源映射的列表，请运行下列命令。

```
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample \
```

```
--event-source-arn arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

测试设置

现在，可以按以下方式测试设置：

1. 在账户 B 中，打开Amazon SQS 控制台。
2. 选择之前创建的 LambdaCrossAccountQueue。
3. 选择 Send and receive messages（发送和接收消息）。
4. 在 Message body (消息正文) 中，输入测试消息。
5. 选择 Send message（发送消息）。

您在账户 A 中的 Lambda 函数应该收到消息。Lambda 将继续轮询队列以获取更新。当有新消息时，Lambda 会使用该新的事件数据从队列中调用您的函数。您的函数运行并在 Amazon CloudWatch 中创建日志。您可以在 CloudWatch 控制台中查看这些日志。

清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

在账户 A 中，清理您的执行角色和 Lambda 函数。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。
4. 在文本输入字段中输入角色名称，然后选择 Delete（删除）。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **`delete`**，然后选择 Delete（删除）。

在账户 B 中，清理 Amazon SQS 队列。

删除 Amazon SQS 队列

1. 登录到 AWS Management Console 并打开 Amazon SQS 控制台，网址：https://console.aws.amazon.com/vpc/。
2. 选择创建的队列。
3. 选择删除。
4. 在文本输入字段中输入 **confirm**。
5. 选择 Delete（删除）。

通过 Amazon S3 批处理事件调用 Lambda 函数

可以使用 Amazon S3 分批操作对一大组 Amazon S3 对象调用 Lambda 函数。Amazon S3 将跟踪批处理操作的进度，发送通知，并存储显示每个操作的状态的完成报告。

要运行分批操作，请创建 Amazon S3 分批操作作业。在创建作业时，您将提供清单（对象列表）并配置要对这些对象执行的操作。

在批处理作业启动时，Amazon S3 会为清单中的每个对象同步调用 Lambda 函数。事件参数包含存储桶和对象的名称。

以下示例显示了对于 amzn-s3-demo-bucket 存储桶中名为 customerImage1.jpg 的对象，Amazon S3 发送到 Lambda 函数的事件。

Example Amazon S3 批处理请求事件

```
{
"invocationSchemaVersion": "1.0",
    "invocationId": "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
    "job": {
        "id": "f3cc4f60-61f6-4a2b-8a21-d07600c373ce"
    },
    "tasks": [
        {
            "taskId": "dGFza2lkZ29lc2hlcmUK",
            "s3Key": "customerImage1.jpg",
            "s3VersionId": "1",
            "s3BucketArn": "arn:aws:s3:::amzn-s3-demo-bucket"
        }
    ]
}
```

您的 Lambda 函数必须返回带字段的 JSON 对象，如以下示例所示。您可以从事件参数复制 `invocationId` 和 `taskId`。您可以在 `resultString` 内返回一个字符串。Amazon S3 会保存完成报告中的 `resultString` 值。

Example Amazon S3 批处理请求响应

```
{
```

```
  "invocationSchemaVersion": "1.0",
  "treatMissingKeysAs" : "PermanentFailure",
  "invocationId" : "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
  "results": [
    {
      "taskId": "dGFza2lkZ29lc2hlcmUK",
      "resultCode": "Succeeded",
      "resultString": "[\"Alice\", \"Bob\"]"
    }
  ]
}
```

从 Amazon S3 分批操作调用 Lambda 函数

您可以使用非限定的或限定的函数 ARN 调用 Lambda 函数。如果要对整个批处理作业使用同一函数版本，请在创建作业时在 FunctionARN 参数中配置特定的函数版本。在配置别名或 $LATEST 限定符的情况下，如果在作业执行期间更新别名或 $LATEST，则批处理作业会立即开始调用函数的新版本。

请注意，您不能对分批操作重用现有的 Amazon S3 的基于事件的函数。这是因为 Amazon S3 分批操作会将不同的事件参数传递给 Lambda 函数，并且需要一条带特定 JSON 结构的返回消息。

在为 Amazon S3 Batch Job 创建的基于资源的策略中，请确保为任务设置调用 Lambda 函数的权限。

在函数的执行角色中，为 Amazon S3 设置一个信任策略以便它在运行函数时代入该角色。

如果您的函数使用AWS开发工具包来管理 Amazon S3 资源，则您需要在执行角色中添加 Amazon S3 权限。

在作业运行时，Amazon S3 会启动多个函数实例来并行处理 Amazon S3 对象，直至函数的并发限制。Amazon S3 会限制实例的初始增加以避免较小作业的额外成本。

如果 Lambda 函数返回 TemporaryFailure 响应代码，则 Amazon S3 会重试操作。

有关 Amazon S3 分批操作的更多信息，请参阅 Amazon S3 开发人员指南中的执行分批操作。

有关如何在 Amazon S3 分批操作中使用 Lambda 函数的示例，请参阅开发人员指南中的从 Amazon S3 分批操作调用 Lambda 函数。

使用 Amazon SNS 通知调用 Lambda 函数

您可以使用 Lambda 函数处理 Amazon Simple Notification Service (Amazon SNS) 通知。Amazon SNS 支持将 Lambda 函数作为发送到主题的消息的目标。您可以将函数订阅到同一账户或其他 AWS 账户中的主题。有关详细的演练过程，请参阅the section called “教程”。

Lambda 仅支持标准 SNS 主题的 SNS 触发器。不支持 FIFO 主题。

对于异步调用，Lambda 对消息排队并处理重试。如果无法联系到 Lambda 或消息被拒绝，Amazon SNS 将在几个小时内以递增的间隔重试。有关详细信息，请参阅 Amazon SNS 常见问题中的可靠性。

> **Warning**
>
> Lambda 事件源映射至少处理每个事件一次，有可能出现重复处理记录的情况。为避免与重复事件相关的潜在问题，我们强烈建议您将函数代码设为幂等性。要了解更多信息，请参阅 AWS 知识中心的如何使我的 Lambda 函数具有幂等性。

主题

使用控制台为 Lambda 函数添加 Amazon SNS 主题触发器

要添加 SNS 主题作为 Lambda 函数的触发器，最简单的方法是使用 Lambda 控制台。当您通过控制台添加触发器时，Lambda 会自动设置必要的权限和订阅以开始从 SNS 主题接收事件。

添加 SNS 主题作为 Lambda 函数的触发器（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择您要为其添加触发器的函数的名称。
3. 选择配置，然后选择触发器。
4. 选择添加触发器。
5. 在触发器配置下的下拉菜单中，选择 SNS。
6. 对于 SNS 主题，请选择要订阅的 SNS 主题。

为 Lambda 函数手动添加 Amazon SNS 主题触发器

要手动为 Lambda 函数设置 SNS 触发器，您需要完成以下步骤：

- 为函数定义基于资源的策略，以允许 SNS 调用该函数。
- 将 Lambda 函数订阅至 Amazon SNS 主题。

> **Note**
>
> 如果您的 SNS 主题和 Lambda 函数位于不同的 AWS 账户中，则还需要授予额外权限以允许跨账户订阅 SNS 主题。有关更多信息，请参阅授予 Amazon SNS 订阅的跨账户权限。

您可以使用 AWS Command Line Interface（AWS CLI）来完成这两个步骤。首先，要为允许 SNS 调用的 Lambda 函数定义基于资源的策略，请使用以下 AWS CLI 命令。请务必将 `--function-name` 的值替换为您的 Lambda 函数名称，将 `--source-arn` 的值替换为您的 SNS 主题 ARN。

```
aws lambda add-permission --function-name example-function \
    --source-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com
```

要将您的函数订阅到 SNS 主题，请使用以下 AWS CLI 命令。将 `--topic-arn` 的值替换为您的 SNS 主题 ARN，将 `--notification-endpoint` 的值替换为您的 Lambda 函数 ARN。

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-east-1:123456789012:function:example-
function
```

示例 SNS 事件形状

Amazon SNS 通过包含消息和元数据的事件异步调用您的函数。

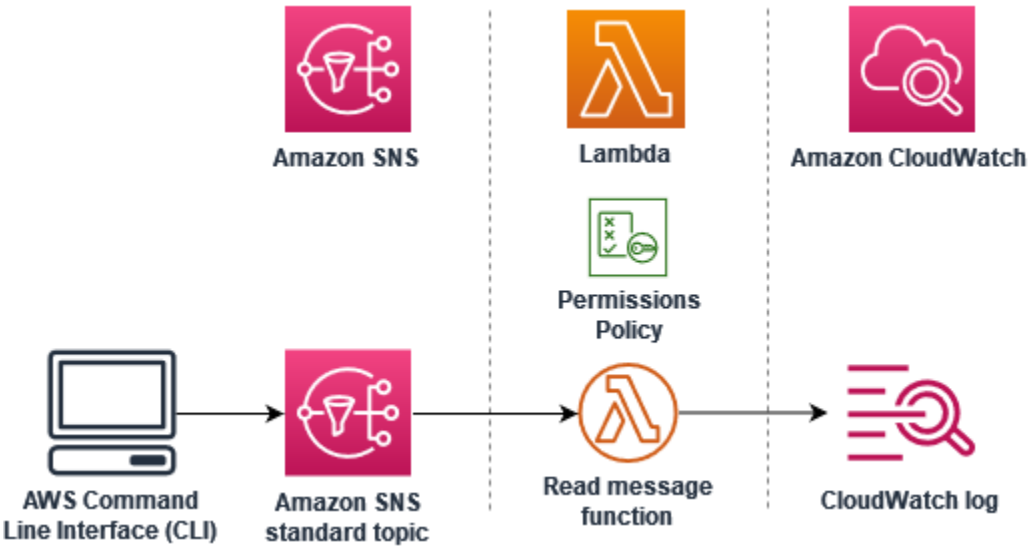
Example Amazon SNS 消息事件

```
{
```

```
  "Records": [
    {
      "EventVersion": "1.0",
      "EventSubscriptionArn": "arn:aws:sns:us-east-1:123456789012:sns-lambda:21be56ed-
a058-49f5-8c98-aedd2564c486",
      "EventSource": "aws:sns",
      "Sns": {
        "SignatureVersion": "1",
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "SigningCertURL": "https://sns.us-east-1.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        "MessageAttributes": {
          "Test": {
            "Type": "String",
            "Value": "TestString"
          },
          "TestBinary": {
            "Type": "Binary",
            "Value": "TestBinary"
          }
        },
        "Type": "Notification",
        "UnsubscribeUrl": "https://sns.us-east-1.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-1:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
        "TopicArn":"arn:aws:sns:us-east-1:123456789012:sns-lambda",
        "Subject": "TestInvoke"
      }
    }
  ]
}
```

教程：将 AWS Lambda 与 Amazon Simple Notification Service 结合使用

在本教程中，您将使用某个 AWS 账户 中的 Lambda 函数，订阅独立 AWS 账户 中的 Amazon Simple Notification Service（Amazon SNS）主题。将消息发布到 Amazon SNS 主题时，Lambda 函数会读取消息内容并将其输出到 Amazon CloudWatch Logs。要完成此教程，需要使用 AWS Command Line Interface（AWS CLI）。



要完成本教程，请执行以下步骤：

- 在账户 A 中，创建 Amazon SNS 主题。
- 在账户 B 中，创建可从该主题读取消息的 Lambda 函数。
- 在账户 B 中，创建该主题的订阅。
- 在账户 A 中将消息发布到 Amazon SNS 主题，并确认账户 B 中的 Lambda 函数将其输出到 CloudWatch Logs。

通过完成这些步骤，您将了解如何配置 Amazon SNS 主题以调用 Lambda 函数。您还将了解如何创建 AWS Identity and Access Management（IAM）策略，向其他 AWS 账户 中的资源授予调用 Lambda 的权限。

在本教程中，您将使用两个独立的 AWS 账户。AWS CLI 命令通过以下方式对此进行说明：使用两个名为 `accountA` 和 `accountB` 的命名配置文件，每个文件配置为用于不同的 AWS 账户。要了解如何配置 AWS CLI 以使用不同的配置文件，请参阅《AWS Command Line Interface User Guide for Version 2》中的 Configuration and credential file settings。确保为两个配置文件配置相同的默认值 AWS 区域。

如果您为两个 AWS 账户 创建的 AWS CLI 配置文件使用不同名称，或者使用默认配置文件和一个命名配置文件，请根据需要按照以下步骤修改 AWS CLI 命令。

先决条件

注册 AWS 账户

如果您还没有 AWS 账户，请完成以下步骤来创建一个。

注册 AWS 账户

1. 打开 https://portal.aws.amazon.com/billing/signup。
2. 按照屏幕上的说明进行操作。

   在注册时，将接到一通电话，要求使用电话键盘输入一个验证码。

   当您注册 AWS 账户时，系统将会创建一个 AWS 账户根用户。根用户有权访问该账户中的所有 AWS 服务 和资源。作为安全最佳实践，请为用户分配管理访问权限，并且只使用根用户来执行需要根用户访问权限的任务。

注册过程完成后，AWS 会向您发送一封确认电子邮件。在任何时候，您都可以通过转至 https://aws.amazon.com/ 并选择我的账户来查看当前的账户活动并管理您的账户。

创建具有管理访问权限的用户

注册 AWS 账户 后，请保护好您的 AWS 账户根用户，启用 AWS IAM Identity Center，并创建一个管理用户，以避免使用根用户执行日常任务。

保护您的 AWS 账户根用户

1. 选择根用户并输入您的 AWS 账户电子邮件地址，以账户拥有者身份登录 AWS Management Console。在下一页上，输入您的密码。

   要获取使用根用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的以根用户身份登录。
2. 为您的根用户启用多重身份验证 (MFA)。

   有关说明，请参阅《IAM 用户指南》中的为 AWS 账户 根用户启用虚拟 MFA 设备（控制台）。

创建具有管理访问权限的用户

1. 启用 IAM Identity Center。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的启用 AWS IAM Identity Center。
2. 在 IAM Identity Center 中，为用户授予管理访问权限。

   有关如何使用 IAM Identity Center 目录 作为身份源的教程，请参阅《AWS IAM Identity Center 用户指南》中的使用默认的 IAM Identity Center 目录 配置用户访问权限。

以具有管理访问权限的用户身份登录

- 要使用您的 IAM Identity Center 用户身份登录，请使用您在创建 IAM Identity Center 用户时发送到您的电子邮件地址的登录网址。

  要获取使用 IAM Identity Center 用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的登录 AWS 访问门户。

将访问权限分配给其他用户

1. 在 IAM Identity Center 中，创建一个权限集，该权限集遵循应用最低权限的最佳做法。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的创建权限集。
2. 将用户分配到一个组，然后为该组分配单点登录访问权限。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的添加组。

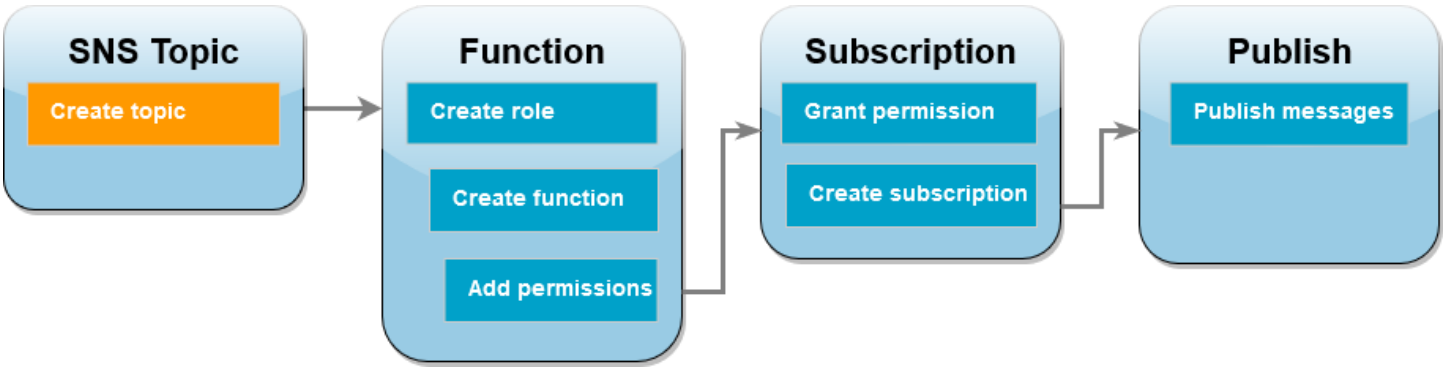
安装 AWS Command Line Interface

如果您尚未安装 AWS Command Line Interface，请按照安装或更新最新版本的 AWS CLI 中的步骤进行安装。

本教程需要命令行终端或 Shell 来运行命令。在 Linux 和 macOS 中，可使用您首选的 Shell 和程序包管理器。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常与 Lambda 一起使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。

创建 Amazon SNS 主题（账户 A）



创建 主题

- 在账户 A 中，使用以下 AWS CLI 命令创建 Amazon SNS 标准主题。

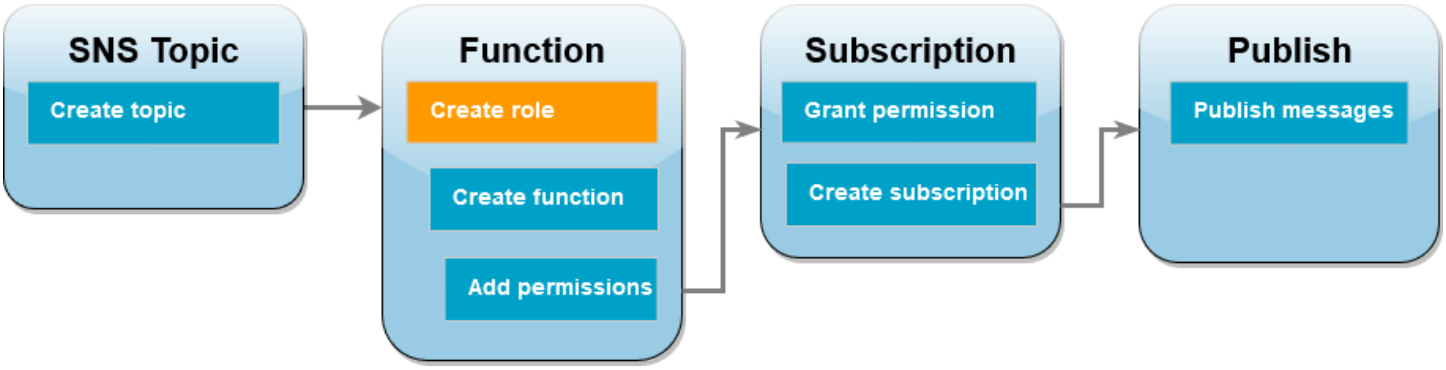
  ```
  aws sns create-topic --name sns-topic-for-lambda --profile accountA
  ```

  您应该可以看到类似于如下所示的输出内容。

  ```
  {
      "TopicArn": "arn:aws:sns:us-west-2:123456789012:sns-topic-for-lambda"
  }
  ```

  记下主题的 Amazon 资源名称（ARN）。稍后在本教程中向 Lambda 函数添加权限以订阅主题时，将需要使用此 ARN。

创建函数执行角色（账户 B）

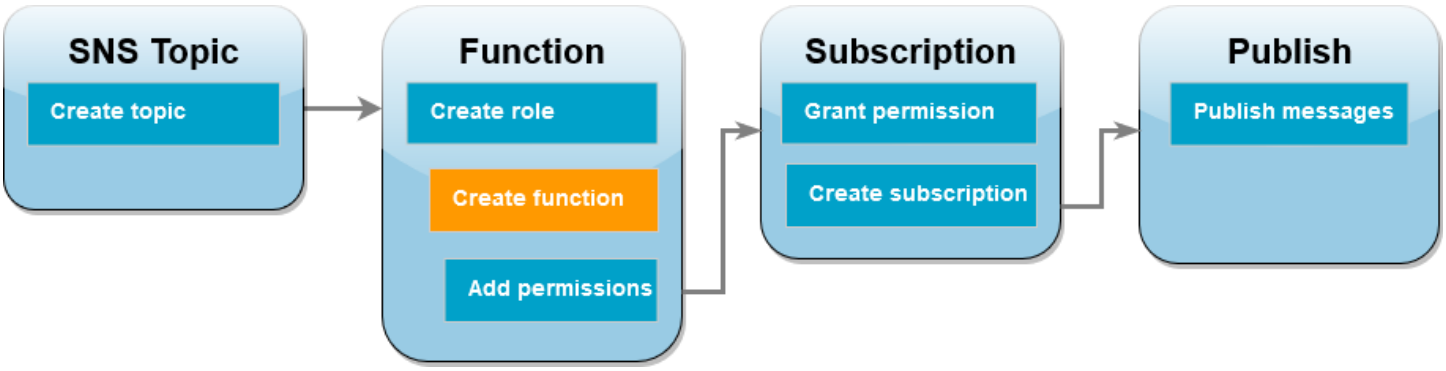


执行角色是一个 IAM 角色，用于向 Lambda 函数授予访问 AWS 服务和资源的权限。在账户 B 中创建函数之前，您需要创建一个角色，向该函数授予将日志写入 CloudWatch Logs 的基本权限。我们将在后续步骤中添加读取 Amazon SNS 主题的权限。

创建执行角色

1. 在账户 B 中，打开 IAM 控制台中的角色页面。
2. 选择 Create role（创建角色）。
3. 在可信实体类型中选择 AWS 服务。
4. 在使用案例中选择 Lambda。
5. 选择下一步。
6. 通过执行以下操作，向角色添加基本权限策略：
   a. 在权限策略搜索框中输入 **AWSLambdaBasicExecutionRole**。
   b. 选择下一步。
7. 通过执行以下操作，完成角色创建：
   a. 在角色详细信息下的角色名称中输入 **lambda-sns-role**。
   b. 选择 Create role（创建角色）。

创建 Lambda 函数（账户 B）



创建一个处理您的 Amazon SNS 消息的 Lambda 函数。函数代码会将每条记录的消息内容记录到 Amazon CloudWatch Logs 中。

本教程使用 Node.js 18.x 运行时系统，但我们还提供了其他运行时系统语言的示例代码。您可以选择以下框中的选项卡，查看适用于您感兴趣的运行时系统的代码。您将在此步骤中使用的 JavaScript 代码是 JavaScript 选项卡中显示的第一个示例。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 .NET 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");
```

```
            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
```

```
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Java 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;
```

```
    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 JavaScript 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

使用 TypeScript 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
```

```
    throw err;
  }
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 PHP 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
```

```
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 SNS 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here
```

```
    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 SNS 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Rust 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

创建函数

1. 为项目创建一个目录，然后切换到该目录。

```
mkdir sns-tutorial
cd sns-tutorial
```

2. 将示例 JavaScript 代码复制到名为 `index.js` 的新文件中。
3. 使用以下 `zip` 命令创建部署包。

```
zip function.zip index.js
```

4. 运行以下 AWS CLI 命令，在账户 B 中创建 Lambda 函数。

```
aws lambda create-function --function-name Function-With-SNS \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
    --role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role  \
    --timeout 60 --profile accountB
```

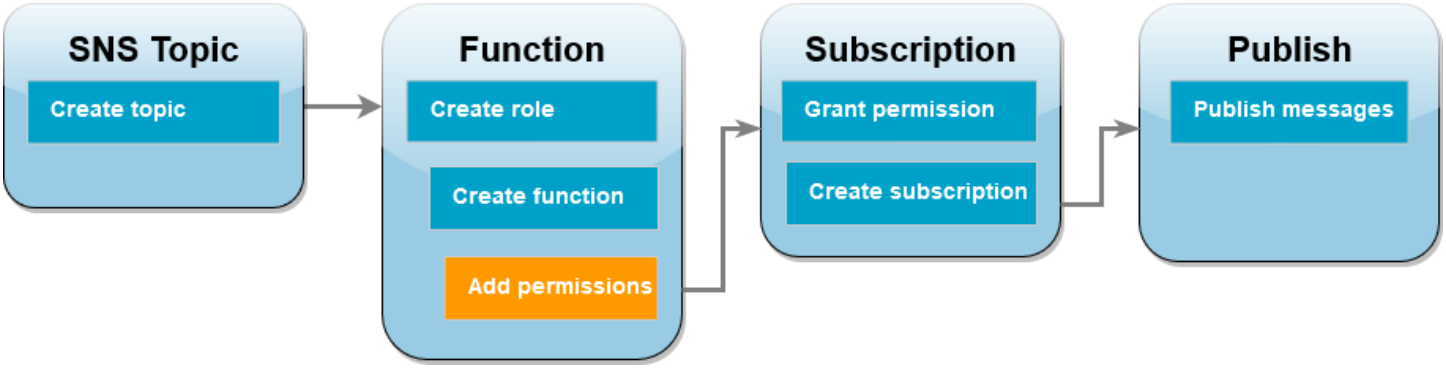
您应该可以看到类似于如下所示的输出内容。

```
{
    "FunctionName": "Function-With-SNS",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:Function-With-
SNS",
    "Runtime": "nodejs18.x",
    "Role": "arn:aws:iam::123456789012:role/lambda_basic_role",
    "Handler": "index.handler",
    ...
```

```
    "RuntimeVersionConfig": {
        "RuntimeVersionArn": "arn:aws:lambda:us-
west-2::runtime:7d5f06b69c951da8a48b926ce280a9daf2e8bb1a74fc4a2672580c787d608206"
    }
}
```

5. 记下函数的 Amazon 资源名称（ARN）。稍后在本教程中添加权限以允许 Amazon SNS 调用函数时，将需要使用此 ARN。

向函数添加权限（账户 B）



要让 Amazon SNS 调用函数，您需要在基于资源的策略语句中向其授予权限。您可以使用 AWS CLI `add-permission` 命令添加此语句。

授予 Amazon SNS 调用函数的权限

- 在账户 B 中，使用之前记下的 Amazon SNS 主题的 ARN 运行以下 AWS CLI 命令。

```
aws lambda add-permission --function-name Function-With-SNS \
    --source-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com --profile accountB
```

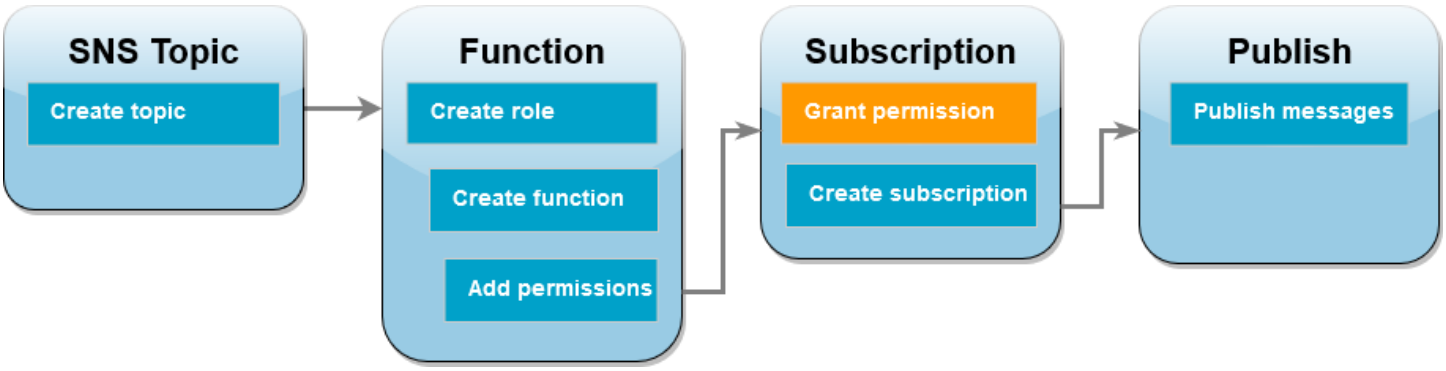
您应该可以看到类似于如下所示的输出内容。

```
{
    "Statement": "{\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
      \"arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda\"}},
      \"Action\":[\"lambda:InvokeFunction\"],
      \"Resource\":\"arn:aws:lambda:us-east-1:<AccountB_ID>:function:Function-With-
SNS\",
      \"Effect\":\"Allow\",\"Principal\":{\"Service\":\"sns.amazonaws.com\"},
```

```
      \"Sid\":\"function-with-sns\"}"
}
```

> **Note**
>
> 如果在选择加入型 AWS 区域 中托管具有 Amazon SNS 主题的账户，则需要在主体中指定区域。例如，假设您正在亚太地区（香港）区域使用一个 Amazon SNS 主题，则需要为主体指定 `sns.ap-east-1.amazonaws.com`，而不是 `sns.amazonaws.com`。

授予 Amazon SNS 订阅的跨账户权限（账户 A）



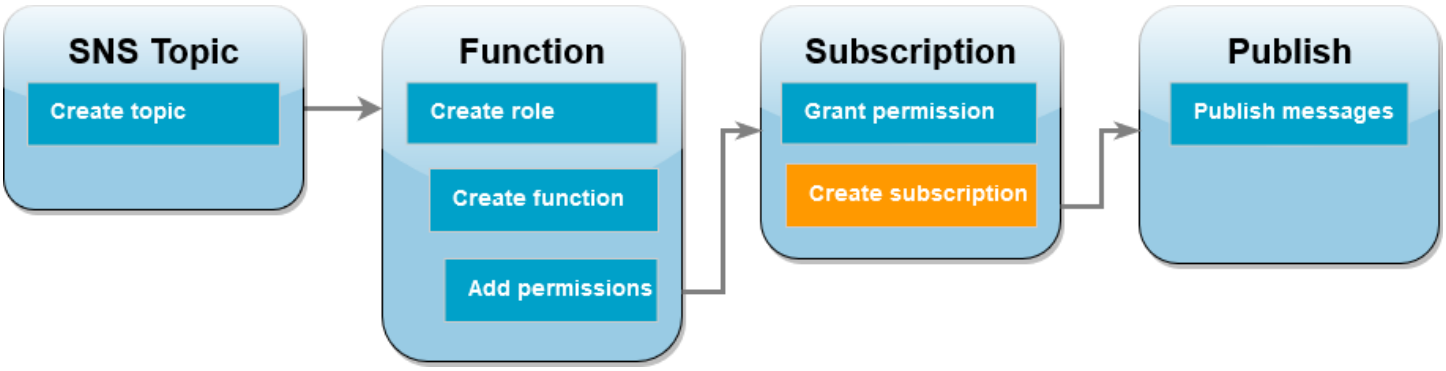
要让账户 B 中的 Lambda 函数订阅您在账户 A 中创建的 Amazon SNS 主题，您需要向账户 B 授予订阅主题的权限。您可以使用 AWS CLI `add-permission` 命令授予此权限。

授予账户 B 订阅主题的权限

- 在账户 A 中，运行以下 AWS CLI 命令。使用之前记下的 Amazon SNS 主题的 ARN。

```
aws sns add-permission --label lambda-access --aws-account-id <AccountB_ID> \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --action-name Subscribe ListSubscriptionsByTopic --profile accountA
```

创建订阅（账户 B）



在账户 B 中，您现在将 Lambda 函数订阅到教程开始时您在账户 A 中创建的 Amazon SNS 主题。当消息发送到该主题 (`sns-topic-for-lambda`) 后，Amazon SNS 会调用账户 B 中的 Lambda 函数 `Function-With-SNS`。

创建订阅

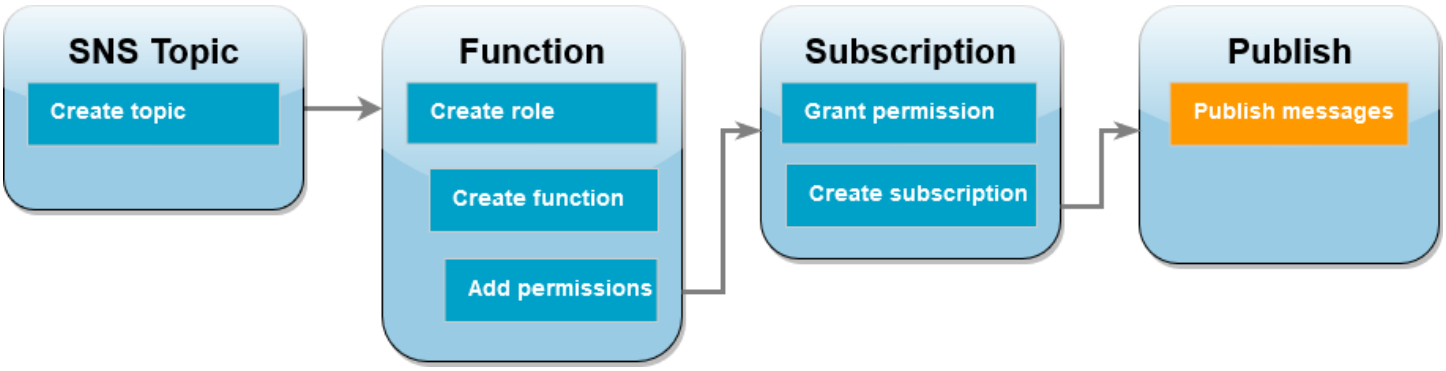
- 在账户 B 中，运行以下 AWS CLI 命令。使用您在其中创建主题的默认区域以及主题和 Lambda 函数的 ARN。

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-
east-1:<AccountB_ID>:function:Function-With-SNS \
    --profile accountB
```

您应该可以看到类似于如下所示的输出内容。

```
{
    "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-
lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

将消息发布到主题（账户 A 和账户 B）



账户 B 中的 Lambda 函数现已订阅账户 A 中的 Amazon SNS 主题，现在可以通过将消息发布到主题来测试设置了。要确认 Amazon SNS 是否已调用 Lambda 函数，可以使用 CloudWatch Logs 查看函数的输出。

将消息发布到主题并查看函数的输出

1. 在文本文件中输入 `Hello World`，并将其另存为 `message.txt`。
2. 在保存文本文件的同一目录中，在账户 A 中运行以下 AWS CLI 命令。将 ARN 用于您自己的主题。

```
aws sns publish --message file://message.txt --subject Test \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --profile accountA
```

   这将返回一个具有唯一标识符的消息 ID，表明 Amazon SNS 服务已接受该消息。然后，Amazon SNS 则尝试将消息传输给主题订阅用户。要确认 Amazon SNS 是否已调用 Lambda 函数，可以使用 CloudWatch Logs 查看函数的输出：

3. 在账户 B 中，打开 Amazon CloudWatch 控制台的日志组页面。
4. 选择函数 (`/aws/lambda/Function-With-SNS`) 的日志组。
5. 选择最新的日志流。
6. 如果函数已正确调用，您将看到类似于以下内容的输出，其中会显示发布到主题的消息内容。

```
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO Processed
 message Hello World
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO done
```

清除资源

除非您想要保留为本教程创建的资源，否则可立即将其删除。通过删除您不再使用的 AWS 资源，可防止您的 AWS 账户 产生不必要的费用。

在账户 A 中，清理您的 Amazon SNS 主题。

删除 Amazon SNS 主题

1. 打开 Amazon SNS 控制台中的 Topics（主题）页面。
2. 选择您已创建的主题。
3. 选择删除。
4. 在文本输入字段中输入 **delete me**。
5. 选择删除。

在账户 B 中，清理您的执行角色、Lambda 函数和 Amazon SNS 订阅。

删除执行角色

1. 打开 IAM 控制台的角色页面。
2. 选择您创建的执行角色。
3. 选择删除。
4. 在文本输入字段中输入角色名称，然后选择 Delete（删除）。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **delete**，然后选择 Delete（删除）。

删除 Amazon SNS 订阅

1. 在 Amazon SNS 控制台中打开 Subscription（订阅）页面。
2. 选择您已创建的订阅。
3. 选择 Delete（删除），Delete（删除）。

在 AWS Lambda 中管理权限

您可以使用 AWS Lambda（IAM）管理 AWS Identity and Access Management 中的权限。使用 Lambda 函数时需要考虑两类主要权限：

- 您的 Lambda 函数执行 API 操作和访问其他 AWS 资源所需的权限
- 其他 AWS 用户和实体访问您的 Lambda 函数所需的权限

Lambda 函数通常需要访问其他 AWS 资源，并对这些资源执行各种 API 操作。例如，您可能会有想通过更新 Amazon DynamoDB 数据库中的条目来响应事件的 Lambda 函数。在这种情况下，您的函数需要访问数据库的权限，以及在该数据库中放置或更新项目的权限。

您可以在名为执行角色的特殊 IAM 角色中定义 Lambda 函数所需的权限。在此角色中，您可以附加一个策略，该策略定义您的函数访问其他 AWS 资源以及从事件源读取所需的所有权限。每个 Lambda 函数都必须有一个执行角色。您的执行角色必须至少有权访问 Amazon CloudWatch，因为默认情况下，Lambda 函数会记录到 CloudWatch Logs 中。您可以将 `AWSLambdaBasicExecutionRole` 托管策略附加到执行角色以满足此要求。

要授予其他 AWS 账户 账户、组织和服务访问 Lambda 资源的权限，您有以下几种选择：

- 您可以使用基于身份的策略授予其他用户访问 Lambda 资源的权限。基于身份的策略可以直接应用于用户，也可以应用于与用户相关的组和角色。
- 您可以使用基于资源的策略授予其他账户和 AWS 服务权限以访问您的 Lambda 资源。当用户尝试访问 Lambda 资源时，Lambda 会同时考虑用户的基于身份的策略和资源的基于资源的策略。当 Amazon Simple Storage Service (Amazon S3) 等 AWS 服务调用您的 Lambda 函数时，Lambda 仅考虑基于资源的策略。
- 您可以使用基于属性的访问权限控制（ABAC）模型来控制对 Lambda 函数的访问。使用 ABAC，您可以将标签附加到 Lambda 函数，在某些 API 请求中传递它们，或将它们附加到发出请求的 IAM 主体。在 IAM 策略的条件元素中指定相同的标签来控制功能访问。

在 AWS 中，最佳实践是仅授予执行任务所需的权限（最低权限）。要在 Lambda 中实现这一点，建议从 AWS 托管策略开始。您可以按原样使用这些托管策略，也可以作为编写自己的限制性更强策略的起点。

为了帮助您微调最低权限访问的权限，Lambda 提供了一些您可以包含在策略中的其他条件。有关更多信息，请参阅 the section called “资源和条件”。

有关 IAM 的更多信息，请参阅《IAM 用户指南》https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html。

使用执行角色定义 Lambda 函数权限

Lambda 函数的执行角色是一个 AWS Identity and Access Management (IAM) 角色，用于向函数授予访问AWS服务和资源的权限。例如，您可以创建一个有权将日志发送到 Amazon CloudWatch 并将跟踪数据上传到 AWS X-Ray 的执行角色。此页面提供有关如何创建、查看和管理 Lambda 函数执行角色的信息。

当您调用函数时，Lambda 会自动代入您的执行角色。您应该避免在函数代码中手动调用 `sts:AssumeRole` 来担任执行角色。如果您的使用案例要求角色能代入自己，则必须将角色本身作为可信主体包含在角色的信任策略中。有关如何修改角色信任策略的更多信息，请参阅《IAM 用户指南》中的修改角色信任策略（控制台）。

为了让 Lambda 正确担任执行角色，该角色的信任策略必须将 Lambda 服务主体（`lambda.amazonaws.com`）指定为可信服务。

主题

- 在 IAM 控制台中创建执行角色
- 使用 AWS CLI 创建和管理角色
- 授予对 Lambda 执行角色的最低访问权限
- 查看和更新执行角色中的权限
- 在执行角色中使用 AWS 托管策略
- 使用源函数 ARN 控制函数访问行为

在 IAM 控制台中创建执行角色

预设情况下，当您在 Lambda 控制台中创建函数时，Lambda 会创建具有最少权限的执行角色。具体而言，此执行角色包括 `AWSLambdaBasicExecutionRole` 托管策略，该策略授予您的函数将事件记录到 Amazon CloudWatch Logs 的基本权限。

您的函数通常需要额外的权限才能执行更有意义的任务。例如，您可能会有想通过更新 Amazon DynamoDB 数据库中的条目来响应事件的 Lambda 函数。您可以使用 IAM 控制台创建具有必要权限的执行角色。

在 IAM 控制台中创建执行角色

1. 在 IAM 控制台中，打开 Roles（角色）页面。
2. 选择 Create role（创建角色）。

3. 在可信实体类型下，选择 AWS 服务。
4. 在 Use case（使用案例）下，选择 Lambda。
5. 选择下一步。
6. 选择您想要附加到角色的 AWS 托管策略。例如，如果您的函数需要访问 DynamoDB，则请选择 AWSLambdaDynamoDBExecutionRole 托管策略。
7. 选择下一步。
8. 输入角色名称，然后选择创建角色。

有关详细说明，请参阅 IAM 用户指南中的为 AWS 服务（控制台）创建一个角色。

创建执行角色后，将其附加到您的函数。当您在 Lambda 控制台中创建函数时，您可以将之前创建的任何执行角色附加到该函数。如果要将新的执行角色附加到现有函数，则请按照 the section called “更新函数的执行角色” 中的步骤操作。

使用 AWS CLI 创建和管理角色

要使用 AWS Command Line Interface (AWS CLI) 创建执行角色，请使用 create-role 命令。在使用此命令时，您可以指定内联信任策略。角色的信任策略会向指定主体授予代入该角色的权限。在以下示例中，您向 Lambda 服务主体授予代入角色的权限。请注意，JSON 字符串中对转义引号的要求可能因 shell 而异。

```
aws iam create-role \
  --role-name lambda-ex \
  --assume-role-policy-document '{"Version": "2012-10-17","Statement":
 [{ "Effect": "Allow", "Principal": {"Service": "lambda.amazonaws.com"}, "Action":
 "sts:AssumeRole"}]}'
```

您还可以使用单独的 JSON 文件为角色定义信任策略。在下面的示例中，`trust-policy.json` 是位于当前目录中的一个文件。

Example trust-policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
aws iam create-role \
  --role-name lambda-ex \
  --assume-role-policy-document file://trust-policy.json
```

您应看到以下输出：

```
{
    "Role": {
        "Path": "/",
        "RoleName": "lambda-ex",
        "RoleId": "AROAQFOXMPL6TZ6ITKWND",
        "Arn": "arn:aws:iam::123456789012:role/lambda-ex",
        "CreateDate": "2020-01-17T23:19:12Z",
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "lambda.amazonaws.com"
                    },
                    "Action": "sts:AssumeRole"
                }
            ]
        }
    }
}
```

要向角色添加权限，请使用 attach-policy-to-role 命令。以下命令将 AWSLambdaBasicExecutionRole 托管策略添加到 lambda-ex 执行角色。

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

创建执行角色后，将其附加到您的函数。当您在 Lambda 控制台中创建函数时，您可以将之前创建的任何执行角色附加到该函数。如果要将新的执行角色附加到现有函数，则请按照 the section called “更新函数的执行角色” 中的步骤操作。

授予对 Lambda 执行角色的最低访问权限

在开发阶段首次为 Lambda 函数创建 IAM 角色时，有时授予的权限可能超出所需权限。在生产环境中发布函数之前，最佳实践是调整策略，使其仅包含所需权限。有关更多信息，请参阅《IAM 用户指南》中的应用最低权限许可。

使用 IAM 访问分析器帮助确定 IAM 执行角色策略所需的权限。IAM 访问分析器将检查您指定的日期范围内的 AWS CloudTrail 日志，并生成仅具有该函数在该时间内使用的权限的策略模板。您可以使用模板创建具有精细权限的托管策略，然后将其附加到 IAM 角色。这样，您仅需授予角色与特定使用案例中的 AWS 资源进行交互所需的权限。

有关更多信息，请参阅《IAM 用户指南》中的基于访问活动生成策略。

查看和更新执行角色中的权限

本主题介绍如何查看和更新函数的执行角色。

主题

- 查看函数的执行角色
- 更新函数的执行角色

查看函数的执行角色

要查看函数的执行角色，请使用 Lambda 控制台。

查看函数的执行角色（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数的名称。
3. 选择 Configuration（配置），然后选择 Permissions（权限）。
4. 在执行角色下，您可以查看当前用作函数执行角色的角色。为了方便起见，您可以在资源摘要部分下查看该函数可以访问的所有资源和操作。您还可以从下拉列表中选择一项服务以查看与该服务相关的所有权限。

更新函数的执行角色

可以随时在函数的执行角色中添加或删除权限，或配置您的函数以使用不同的角色。如果函数需要访问任何其他服务或资源，则必须向执行角色添加必要的权限。

向函数添加权限时，请同时简单更新其代码或配置。这会强制停止并替换函数的具有过时凭证的运行实例。

要更新函数的执行角色，您可以使用 Lambda 控制台。

更新函数的执行角色（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数的名称。
3. 选择 Configuration（配置），然后选择 Permissions（权限）。
4. 在执行角色下，选择编辑。
5. 如果要更新函数以使用其他角色作为执行角色，则请在现有角色下的下拉菜单中选择新角色。

   > **Note**
   >
   > 如果要更新现有执行角色中的权限，则只能在 AWS Identity and Access Management（IAM）控制台中执行此操作。

   如果要创建用作执行角色的新角色，则请在执行角色下选择从 AWS 策略模板创建新角色。然后，在角色名称下输入新角色的名称，并在策略模板下指定要附加到新角色的任何策略。
6. 选择保存。

在执行角色中使用 AWS 托管策略

以下 AWS 托管式策略提供使用 Lambda 函数所需的权限。

更改	描述	日期
AWSLambdaMSKExecutionRole：Lambda 将	`AWSLambdaMSKExecutionRole` 授予读取和访问 Amazon Managed Streaming	2022 年 6 月 17 日

更改	描述	日期
kafka:DescribeClusterV2 权限添加到了此政策。	for Apache Kafka（Amazon MSK）集群中的记录、管理网络接口 (ENI) 并写入到 CloudWatch Logs 的权限。	
AWSLambdaBasicExecutionRole – Lambda 开始跟踪对此策略所做的更改。	AWSLambdaBasicExecutionRole 授予将日志上传至 CloudWatch 的权限。	2022 年 2 月 14 日
AWSLambdaDynamoDBExecutionRole – Lambda 开始跟踪对此策略所做的更改。	AWSLambdaDynamoDBExecutionRole 授予读取 Amazon DynamoDB 流中的记录并写入到 CloudWatch Logs 的权限。	2022 年 2 月 14 日
AWSLambdaKinesisExecutionRole – Lambda 开始跟踪对此策略所做的更改。	AWSLambdaKinesisExecutionRole 授予读取 Amazon Kinesis 数据流中的事件并写入到 CloudWatch Logs 的权限。	2022 年 2 月 14 日
AWSLambdaMSKExecutionRole – Lambda 开始跟踪对此策略所做的更改。	AWSLambdaMSKExecutionRole 授予读取和访问 Amazon Managed Streaming for Apache Kafka (Amazon MSK) 集群中的记录、管理网络接口 (ENI) 并写入到 CloudWatch Logs 的权限。	2022 年 2 月 14 日
AWSLambdaSQSQueueExecutionRole – Lambda 开始跟踪对此策略所做的更改。	AWSLambdaSQSQueueExecutionRole 授予读取 Amazon Simple Queue Service (Amazon SQS) 队列中的消息并写入到 CloudWatch Logs 的权限。	2022 年 2 月 14 日

更改	描述	日期
AWSLambdaVPCAccessExecutionRole – Lambda 开始跟踪对此策略所做的更改。	AWSLambdaVPCAccessExecutionRole 授予管理 Amazon VPC 中的 ENI 并写入到 CloudWatch Logs 的权限。	2022 年 2 月 14 日
AWSXRayDaemonWriteAccess – Lambda 开始跟踪对此策略所做的更改。	AWSXRayDaemonWriteAccess 授予将跟踪数据上传到 X-Ray 的权限。	2022 年 2 月 14 日
CloudWatchLambdaInsightsExecutionRolePolicy – Lambda 开始跟踪对此策略所做的更改。	CloudWatchLambdaInsightsExecutionRolePolicy 授予将运行时指标写入到 CloudWatch Lambda Insights 的权限。	2022 年 2 月 14 日
AmazonS3ObjectLambdaExecutionRolePolicy – Lambda 开始跟踪对此策略所做的更改。	AmazonS3ObjectLambdaExecutionRolePolicy 授予与 Amazon Simple Storage Service (Amazon S3) 对象 Lambda 交互并写入到 CloudWatch Logs 的权限。	2022 年 2 月 14 日

对于某些功能，Lambda 控制台会尝试在客户管理型策略中向执行角色添加缺失的权限。这些策略可能会变得很多。为避免创建额外的策略，在启用功能之前，请将相关的 AWS 托管策略添加到您的执行角色。

当您使用事件源映射调用您的函数时，Lambda 将使用执行角色读取事件数据。例如，Kinesis 的事件源映射从数据流读取事件并将事件成批发送到您的函数。

当服务在您的账户中担任角色时，您的角色信任策略中可以包含 `aws:SourceAccount` 和 `aws:SourceArn` 全局条件上下文密钥，以将对角色的访问限制为仅由预期资源生成的请求。有关更多信息，请参阅防止 AWS Security Token Service 跨服务混淆代理。

除了AWS托管式策略，Lambda 控制台还提供模板，以创建包含用于额外使用案例的权限的自定义策略。当您在 Lambda 控制台中创建函数时，可以选择利用来自一个或多个模板的权限创建新的执行角

色。当您从蓝图创建函数，或者配置需要访问其他服务的选项时，也会自动应用这些模板。示例模板可从本指南的 GitHub 存储库中找到。

使用源函数 ARN 控制函数访问行为

您的 Lambda 函数代码通常会向其他 AWS 服务发出 API 请求。为了发出这些请求，Lambda 通过承担您的函数的执行角色来生成一组临时凭证。这些凭证在函数调用期间可用作环境变量。使用 AWS 开发工具包时，无需直接在代码中为开发工具包提供凭证。默认情况下，凭证提供程序链会按顺序检查每个可以设置凭证的位置，然后选择第一个可用位置，通常是环境变量（`AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY` 和 `AWS_SESSION_TOKEN`）。

如果请求是来自您执行环境中的 AWS API 请求，Lambda 会将源函数 ARN 注入到凭证上下文中。Lambda 还会为其在执行环境之外代表您发出的以下 AWS API 请求注入源函数 ARN：

服务	操作	Reason
CloudWatch Logs	`CreateLogGroup`，`CreateLogStream`，`PutLogEvents`	将日志存储到 CloudWatch Logs 日志组
X-Ray	`PutTraceSegments`	将跟踪数据发送到 X-Ray
Amazon EFS	`ClientMount`	将函数连接到 Amazon Elastic File System（Amazon EFS）文件系统

使用相同执行角色在执行环境之外代表您进行的其他 AWS API 调用不包含源函数 ARN。执行环境之外的此类 API 调用示例包括：

- 调用 AWS Key Management Service (AWS KMS) 以自动加密和解密您的环境变量。
- 调用 Amazon Elastic Compute Cloud（Amazon EC2），为启用 VPC 的函数创建弹性网络接口（ENI）。
- 调用 Amazon Simple Queue Service（Amazon SQS）等 AWS 服务，从设置为事件源映射的事件源进行读取。

使用凭证上下文中的源函数 ARN，您可以验证对您的资源的调用是否来自特定 Lambda 函数的代码。要对此进行验证，请在 IAM 基于身份的策略或服务控制策略（SCP）中使用 `lambda:SourceFunctionArn` 条件键。

> **Note**
>
> 您不能在基于资源的策略中使用 `lambda:SourceFunctionArn` 条件键。

在基于身份的策略或 SCP 中使用此条件键，您可以为您的函数代码对其他 AWS 服务执行的 API 操作实施安全控制。这有一些关键的安全应用程序，例如帮助您识别凭证泄漏的来源。

> **Note**
>
> `lambda:SourceFunctionArn` 条件键与 `lambda:FunctionArn` 和 `aws:SourceArn` 条件键不同。`lambda:FunctionArn` 条件键仅适用于事件源映射，并帮助定义您的事件源可以调用哪些函数。`aws:SourceArn` 条件键仅适用于以您的 Lambda 函数为目标资源的策略，并帮助定义哪些其他 AWS 服务和资源可以调用该函数。`lambda:SourceFunctionArn` 条件键可应用于任何基于身份的策略或 SCP，以定义有权对其他资源进行特定 AWS API 调用的特定 Lambda 函数。

要在您的策略中使用 `lambda:SourceFunctionArn`，请将其作为条件包含在任何 ARN 条件运算符中。密钥的值必须是有效的 ARN。

例如，假设您的 Lambda 函数代码进行了针对特定 Amazon S3 存储桶的 `s3:PutObject` 调用。您可能希望仅允许一个特定 Lambda 函数让 `s3:PutObject` 访问该存储桶。在这种情况下，您的函数的执行角色应附加如下所示的策略：

Example 授予特定 Lambda 函数访问 Amazon S3 资源的权限的策略

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleSourceFunctionArn",
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Condition": {
```

```
                "ArnEquals": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

如果源是具有 ARN `arn:aws:lambda:us-east-1:123456789012:function:source_lambda` 的 Lambda 函数，则此策略仅允许 `s3:PutObject` 访问。此策略不允许 `s3:PutObject` 访问任何其他调用身份。即使不同的函数或实体使用相同的执行角色进行 `s3:PutObject` 调用也是如此。

> **Note**
>
> `lambda:SourceFunctionARN` 条件键不支持 Lambda 函数版本或函数别名。如果您将 ARN 用于特定函数版本或别名，则函数将无权执行您指定的操作。确保使用函数的非限定 ARN（不带版本或别名后缀）。

您也可以在 SCP 中使用 `lambda:SourceFunctionArn`。例如，假设您希望将对存储桶的访问限制为单个 Lambda 函数的代码或来自特定 Amazon 虚拟私有云（VPC）的调用。以下 SCP 对此进行了说明。

Example 在特定条件下拒绝访问 Amazon S3 的策略

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "StringNotEqualsIfExists": {
                    "aws:SourceVpc": [
                        "vpc-12345678"
                    ]
                }
```

```
            }
        },
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "ArnNotEqualsIfExists": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

此策略将拒绝所有 S3 操作，除非它们来自具有 ARN `arn:aws:lambda:*:123456789012:function:source_lambda` 的特定 Lambda 函数，或者它们来自指定的 VPC。`StringNotEqualsIfExists` 运算符告诉 IAM 仅当请求中存在 `aws:SourceVpc` 键时才处理此条件。同样，仅当存在 `lambda:SourceFunctionArn` 时，IAM 才会考虑 `ArnNotEqualsIfExists` 运算符。

向其他 AWS 实体授予访问您的 Lambda 函数的权限

要授予其他 AWS 账户 账户、组织和服务访问 Lambda 资源的权限，您有以下几种选择：

- 您可以使用基于身份的策略授予其他用户访问 Lambda 资源的权限。基于身份的策略可以直接应用于用户，也可以应用于与用户相关的组和角色。
- 您可以使用基于资源的策略授予其他账户和 AWS 服务权限以访问您的 Lambda 资源。当用户尝试访问 Lambda 资源时，Lambda 会同时考虑用户的基于身份的策略和资源的基于资源的策略。当 Amazon Simple Storage Service (Amazon S3) 等 AWS 服务调用您的 Lambda 函数时，Lambda 仅考虑基于资源的策略。
- 您可以使用基于属性的访问权限控制（ABAC）模型来控制对 Lambda 函数的访问。使用 ABAC，您可以将标签附加到 Lambda 函数，在某些 API 请求中传递它们，或将它们附加到发出请求的 IAM 主体。在 IAM 策略的条件元素中指定相同的标签来控制功能访问。

为了帮助您微调最低权限访问的权限，Lambda 提供了一些您可以包含在策略中的其他条件。有关更多信息，请参阅 the section called “资源和条件”。

Lambda 的基于身份的 IAM policy

您可以使用 AWS Identity and Access Management (IAM) 中基于身份的策略授予您账户中的用户访问 Lambda 的权限。基于身份的策略可以应用于用户、用户组或角色。您也可以授予另一个账户中的用户在您的账户中代入角色和访问您的 Lambda 资源的权限。

Lambda 提供AWS托管策略，授予对 Lambda API 操作的访问权限，在某些情况下还可以访问其他AWS服务，用于开发和管理 Lambda 资源。Lambda 根据需要更新这些托管策略，确保您的用户有权在新功能发布时进行访问。

- AWSLambda_FullAccess - 授予对 Lambda 操作和其他用于开发和维护 Lambda 资源的 AWS 服务的完全访问权。
- AWSLambda_ReadOnlyAccess – 授予对 Lambda 资源的只读访问权限。
- AWSLambdaRole – 授予调用 Lambda 函数的权限。

AWS 托管策略授予 API 操作的权限，而不限制用户可以修改的 Lambda 函数或层。要进行更精细的控制，您可以创建自己的策略来限制用户权限的范围。

主题

- 授予用户对 Lambda 函数的访问权限

- 授予用户对 Lambda 层的访问权限

授予用户对 Lambda 函数的访问权限

使用基于身份的策略允许用户、用户组或角色对 Lambda 函数执行操作。

> **Note**
>
> 对于定义为容器映像的函数，必须在 Amazon Elastic container Registry（Amazon ECR）中配置访问映像的用户权限。有关示例，请参阅 Amazon ECR 存储库策略。

以下显示具有有限范围的权限策略示例。该策略允许用户创建和管理名称前带指定前缀 (`intern-`) 并用指定执行角色配置的 Lambda 函数。

Example 函数开发策略

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyPermissions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetAccountSettings",
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
        },
        {
            "Sid": "DevelopFunctions",
            "Effect": "Allow",
            "NotAction": [
                "lambda:AddPermission",
```

```
                "lambda:PutFunctionConcurrency"
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
        },
        {
            "Sid": "DevelopEventSourceMappings",
            "Effect": "Allow",
            "Action": [
                "lambda:DeleteEventSourceMapping",
                "lambda:UpdateEventSourceMapping",
                "lambda:CreateEventSourceMapping"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
                }
            }
        },
        {
            "Sid": "PassExecutionRole",
            "Effect": "Allow",
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
                "iam:PassRole",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
        },
        {
            "Sid": "ViewLogs",
            "Effect": "Allow",
            "Action": [
                "logs:*"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
        }
    ]
}
```

策略中的权限基于它们所支持的资源和条件组织成语句。

- ReadOnlyPermissions – 当您浏览和查看函数时，Lambda 控制台使用这些权限。它们不支持资源模式或条件。

```
        "Action": [
            "lambda:GetAccountSettings",
            "lambda:GetEventSourceMapping",
            "lambda:GetFunction",
            "lambda:GetFunctionConfiguration",
            "lambda:GetFunctionCodeSigningConfig",
            "lambda:GetFunctionConcurrency",
            "lambda:ListEventSourceMappings",
            "lambda:ListFunctions",
            "lambda:ListTags",
            "iam:ListRoles"
        ],
        "Resource": "*"
```

- DevelopFunctions：使用任何对前缀为 intern- 的函数执行的 Lambda 操作，但 AddPermission 和 PutFunctionConcurrency 除外。AddPermission 修改函数上基于资源的策略并可能影响安全性。PutFunctionConcurrency 保留函数的扩展容量，并可以从其他函数取得容量。

```
        "NotAction": [
            "lambda:AddPermission",
            "lambda:PutFunctionConcurrency"
        ],
        "Resource": "arn:aws:lambda:*:*:function:intern-*"
```

- DevelopEventSourceMappings – 管理前缀为 intern- 的函数的事件源映射。虽然这些操作在事件源映射上运行，但您可以通过附带条件的函数限制它们。

```
        "Action": [
            "lambda:DeleteEventSourceMapping",
            "lambda:UpdateEventSourceMapping",
            "lambda:CreateEventSourceMapping"
        ],
        "Resource": "*",
```

```
            "Condition": {
                "StringLike": {
                    "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
                }
            }
```

- `PassExecutionRole` – 查看并仅传递名为 `intern-lambda-execution-role` 的角色，该角色必须由具有 IAM 权限的用户创建和管理。当您为函数分配执行角色时，使用 `PassRole`。

```
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
                "iam:PassRole",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
```

- `ViewLogs` – 使用 CloudWatch Logs 查看前缀为 `intern-` 的函数的日志。

```
            "Action": [
                "logs:*"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
```

该策略允许用户开始使用 Lambda，而不会将其他用户的资源置于风险之下。它不允许用户配置能被触发或调用其他 AWS 服务的函数，这需要更广的 IAM 权限。它也不包含对于不支持有限范围策略的服务（如 CloudWatch 和 X-Ray）的权限。将只读策略用于这些服务以便让用户能够访问指标和跟踪数据。

当您为您的函数配置触发器时，需要有权使用调用您的函数的 AWS 服务。例如，要配置 Amazon S3 触发器，您需要权限来使用管理存储桶通知的 Amazon S3 操作。其中很多权限包含在 AWSLambda_FullAccess 托管策略中。

授予用户对 Lambda 层的访问权限

使用基于身份的策略允许用户、用户组或角色在 Lambda 层上执行操作。以下策略授予用户创建层并通过函数使用层的权限。资源模式允许用户在任何 AWS 区域 和任何层版本中工作，只要层的名称以 `test-` 开头即可。

Example 层开发策略

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublishLayers",
            "Effect": "Allow",
            "Action": [
                "lambda:PublishLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*"
        },
        {
            "Sid": "ManageLayerVersions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetLayerVersion",
                "lambda:DeleteLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*:*"
        }
    ]
}
```

您还可以附加 `lambda:Layer` 条件以在函数创建和配置过程中强制使用层。例如，您可以防止用户使用其他账户发布的层。以下策略在 `CreateFunction` 和 `UpdateFunctionConfiguration` 操作中添加一个条件，以要求任何指定层来自账户 `123456789012`。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ConfigureFunctions",
            "Effect": "Allow",
            "Action": [
```

```
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringLike": {
                    "lambda:Layer": [
                        "arn:aws:lambda:*:123456789012:layer:*:*"
                    ]
                }
            }
        }
    ]
}
```

为确保条件应用，应验证没有其他语句向用户授予这些操作的权限。

在 Lambda 中使用基于资源的 IAM 策略

Lambda 支持将基于资源的权限策略用于 Lambda 函数和层。您可以使用基于资源的策略向其他 AWS 账户、组织或服务授予访问权限。基于资源的策略应用于单个函数、版本、别名或层版本。

Console

查看函数的基于资源的策略

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration (配置)，然后选择 Permissions (权限)。
4. 向下滚动到 Resource-based policy (基于资源的策略)，然后选择 View policy document (查看策略文档)。基于资源的策略显示了在其他账户或 AWS 服务尝试访问该函数时应用的权限。以下示例显示了一个语句，该语句允许 Amazon S3 调用为账户 123456789012 中名为 amzn-s3-demo-bucket 的存储桶调用名为 my-function 的函数。

   Example 基于资源的策略

   ```
   {
       "Version": "2012-10-17",
       "Id": "default",
       "Statement": [
           {
   ```

```
            "Sid": "lambda-allow-s3-my-function",
            "Effect": "Allow",
            "Principal": {
              "Service": "s3.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource":  "arn:aws:lambda:us-east-2:123456789012:function:my-
function",
            "Condition": {
              "StringEquals": {
                "AWS:SourceAccount": "123456789012"
              },
              "ArnLike": {
                "AWS:SourceArn": "arn:aws:s3:::amzn-s3-demo-bucket"
              }
            }
        }
    ]
}
```

AWS CLI

要查看函数的基于资源的策略，请使用 get-policy 命令。

```
aws lambda get-policy \
  --function-name my-function \
  --output text
```

您应看到以下输出：

```
{"Version":"2012-10-17","Id":"default","Statement":
[{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"s3.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:u
east-2:123456789012:function:my-function","Condition":{"ArnLike":
{"AWS:SourceArn":"arn:aws:sns:us-east-2:123456789012:lambda*"}}}]}      7c681fc9-
b791-4e91-acdf-eb847fdaa0f0
```

对于版本和别名，请在函数名后面附加版本号或别名。

```
aws lambda get-policy --function-name my-function:PROD
```

要从函数中删除权限，请使用 remove-permission。

```
aws lambda remove-permission \
  --function-name example \
  --statement-id sns
```

使用 get-layer-version-policy 命令可查看层上的权限。

```
aws lambda get-layer-version-policy \
  --layer-name my-layer \
  --version-number 3 \
  --output text
```

您应看到以下输出：

```
b0cd9796-d4eb-4564-939f-de7fe0b42236    {"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:
west-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

使用 remove-layer-version-permission 可从策略中删除语句。

```
aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3
 --statement-id engineering-org
```

支持的 API 操作

以下 Lambda API 操作支持基于资源的策略：

- CreateAlias
- DeleteAlias
- DeleteFunction
- DeleteFunctionConcurrency
- DeleteFunctionEventInvokeConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias
- GetFunction

- GetFunctionConcurrency
- GetFunctionConfiguration
- GetFunctionEventInvokeConfig
- GetPolicy
- GetProvisionedConcurrencyConfig
- Invoke
- ListAliases
- ListFunctionEventInvokeConfigs
- ListProvisionedConcurrencyConfigs
- ListTags
- ListVersionsByFunction
- PublishVersion
- PutFunctionConcurrency
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- TagResource
- UntagResource
- UpdateAlias
- UpdateFunctionCode
- UpdateFunctionEventInvokeConfig

授予 Lambda 函数访问 AWS 服务的权限

当您使用 AWS 服务调用您的函数时，可以用基于资源的策略语句授权。您可以将语句应用于整个函数，也可以将语句限制为单个版本或别名。

> **Note**
>
> 当您通过 Lambda 控制台向函数添加触发器时，该控制台会更新函数的基于资源的策略以允许服务调用它。要向 Lambda 控制台中不可用的其他账户或服务授予权限，可以使用 AWS CLI。

使用 add-permission 命令添加一条语句。最简单的基于资源的策略语句是允许一个服务调用某个函数。以下命令授予 Amazon Simple Notification Service 调用名为 `my-function` 的函数的权限。

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id sns \
  --principal sns.amazonaws.com \
  --output text
```

您应看到以下输出：

```
{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function"}
```

这允许 Amazon SNS 在函数上调用 Invoke API 操作，但不会限制触发调用的 Amazon SNS 主题。为确保您的函数只被特定资源调用，请使用 `source-arn` 选项指定资源的 Amazon Resource Name (ARN)。以下命令只允许 Amazon SNS 调用名为 `my-topic` 的主题的订阅函数。

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id sns-my-topic \
  --principal sns.amazonaws.com \
  --source-arn arn:aws:sns:us-east-2:123456789012:my-topic
```

有些服务可以调用其他账户中的函数。如果您指定的一个源 ARN 中包含您的账户 ID，这不是问题。但对于 Amazon S3 来说，源是其 ARN 中不包含账户 ID 的存储桶。有可能是您删除了该存储桶，而另一个账户用同样的名称创建了这样一个存储桶。使用 `source-account` 选项以及您的账户 ID 以确保只有您账户中的资源可以调用该函数。

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id s3-account \
  --principal s3.amazonaws.com \
  --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
  --source-account 123456789012
```

向组织授予函数访问权限

要向 AWS Organizations 中的组织授予权限，请将组织 ID 指定为 `principal-org-id`。以下 add-permission 命令向组织 `o-a1b2c3d4e5f` 中的所有用户授予调用访问权限。

```
aws lambda add-permission \
  --function-name example \
  --statement-id PrincipalOrgIDExample \
  --action lambda:InvokeFunction \
  --principal * \
  --principal-org-id o-a1b2c3d4e5f
```

> **Note**
>
> 在此命令中，`Principal` 为 `*`。这意味着组织 `o-a1b2c3d4e5f` 中的所有用户都获得了函数调用权限。如果您将某个 AWS 账户 或角色指定为 `Principal`，则只有该主体会获得函数调用权限，但前提是他们也是 `o-a1b2c3d4e5f` 组织的成员。

此命令会创建一个基于资源的策略，如下所示：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PrincipalOrgIDExample",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:example",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
                }
            }
        }
    ]
}
```

有关更多信息，请参阅《IAM 用户指南》中的 aws:PrincipalOrgID。

向其他账户授予 Lambda 函数访问权限

要与其他 AWS 账户 共享函数，请在该函数的基于资源的策略中添加跨账户权限语句。执行 add-permission 命令，并将账户 ID 指定为 `principal`。以下示例向账户 `111122223333` 授权以 `my-function` 别名调用 `prod`。

```
aws lambda add-permission \
  --function-name my-function:prod \
  --statement-id xaccount \
  --action lambda:InvokeFunction \
  --principal 111122223333 \
  --output text
```

您应看到以下输出：

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:l
east-1:123456789012:function:my-function"}
```

基于资源的策略向另一个账户授予访问此函数的权限，但不允许该账户中的用户超出其权限。另一个账户中的用户必须具有相应的用户权限才能使用 Lambda API。

要限制对另一个账户中用户或角色的访问，请将身份的完整 ARN 指定为主体。例如：`arn:aws:iam::123456789012:user/developer`。

别名限制了其他账户可以调用哪个版本。它要求另一个账户在函数 ARN 中包括该别名。

```
aws lambda invoke \
  --function-name arn:aws:lambda:us-east-2:123456789012:function:my-function:prod out
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "1"
}
```

然后，函数所有者可以更新别名以指向新版本，而无需调用者更改他们调用函数的方式。这可以确保另一个账户不需要更改其代码即可使用新版本，并且它只具有调用与别名关联的函数版本的权限。

您可以授予对作用于现有函数的大多数 API 操作的跨账户访问权。例如，您可以授予 `lambda:ListAliases` 权限，以允许一个账户获得别名列表，或授予 `lambda:GetFunction` 权限，以让它们下载您的函数代码。分别添加每个权限，或使用 `lambda:*` 授予有关指定函数的所有操作的权限。

要授予其他账户对多个函数的权限，或不对某个函数执行的操作的权限，请使用 IAM 角色。

向其他账户授予 Lambda 层访问权限

要与其他 AWS 账户 共享层，请在该层的基于资源的策略中添加跨账户权限语句。执行 add-layer-version-permission 命令，并将账户 ID 指定为 `principal`。在每个语句中，您可以向 AWS Organizations 中的单个账户、所有账户或组织授予权限。

以下示例向账户 111122223333 授予访问 `bash-runtime` 层版本 2 的权限。

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
  --output text
```

您应该可以看到类似于如下所示的输出内容：

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

权限仅适用于单个层版本。每次创建新的层版本时都需重复此过程。

要向 AWS Organizations 组织中的所有账户授予权限，请使用 `organization-id` 选项。以下示例授予组织 `o-t194hfs8cz` 中的所有账户使用 `my-layer` 版本 3 的权限。

```
aws lambda add-layer-version-permission \
  --layer-name my-layer \
  --version-number 3 \
  --statement-id engineering-org \
  --principal '*' \
  --action lambda:GetLayerVersion \
```

```
  --organization-id o-t194hfs8cz \
  --output text
```

您应看到以下输出：

```
{"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
east-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

要向多个账户或组织授予权限，您必须添加多个语句。

在 Lambda 中使用基于属性的访问控制

借助基于属性的访问权限控制（ABAC），您可以使用标签来控制对 Lambda 资源的访问。您可以将标签附加到某些 Lambda 资源，附加到某些 API 请求，或附加到发出请求的 AWS Identity and Access Management（IAM）主体。有关 AWS 如何授予基于属性的访问权限的更多信息，请参阅《IAM 用户指南》中的使用标签控制对 AWS 资源的访问。

您可以使用 ABAC 授予最低权限，而无需在 IAM policy 中指定 Amazon 资源名称 (ARN) 或 ARN 模式。相反，您可以在 IAM policy 的条件元素中指定一个标签来控制访问。使用 ABAC 可以更轻松地进行扩展，因为您无需在创建新资源时更新 IAM 策略。相反，将标签添加到新资源即可控制访问权限。

在 Lambda 中，标签可用于以下资源：

- 函数 – 有关标记函数的更多信息，请参阅the section called “标签”。
- 代码签名配置 – 有关标记代码签名配置的更多信息，请参阅the section called “代码签名配置标签”。
- 事件源映射 – 有关标记事件源映射的更多信息，请参阅the section called “事件源映射标签”。

层不支持标签。

可以使用以下条件键根据标签编写 IAM 策略规则：

- aws:ResourceTag/tag-key：根据附加到 Lambda 资源的标签控制访问权限。
- aws:RequestTag/tag-key：要求请求中存在标签，例如在创建新函数时。
- aws:PrincipalTag/tag-key：根据附加到其 IAM 用户或角色的标签，控制允许 IAM 主体（发出请求的人）执行的操作。

- aws:TagKeys：控制是否可以在请求中使用特定的标签键。

只能为支持这些条件的操作指定条件。有关各 Lambda 操作支持的条件列表，请参阅《Service Authorization Reference》中的 Actions, resources, and condition keys for AWS Lambda。有关 aws:ResourceTag/tag-key 支持，请参阅该文档中的“Resource types defined by AWS Lambda”。有关 aws:RequestTag/tag-key 和 aws:TagKeys 支持，同样请参阅该文档中的“Actions defined by AWS Lambda”。

主题

通过标签保护函数

以下步骤演示了一种使用 ABAC 设置函数权限的方法。在此示例方案中，您将创建四个 IAM 权限策略。然后，您会将这些策略附加到新的 IAM 角色。最后，您将创建一个 IAM 用户并授予该用户担任新角色的权限。

主题

先决条件

确保您具有 Lambda 执行角色。当您授予 IAM 权限和创建 Lambda 函数时，您将使用此角色。

步骤 1：要求新函数具有标签

当将 ABAC 与 Lambda 配合使用时，最佳做法是要求所有函数都具有标签。这有助于确保您的 ABAC 权限策略按预期工作。

创建类似于以下示例的 IAM policy。此策略使用 aws:RequestTag/tag-key、aws:ResourceTag/tag-key 和 aws:TagKeys 条件键来要求新函数和创建函数的 IAM 主体都具有 `project` 标签。`ForAllValues` 修饰符确保 `project` 是唯一允许的标签。如果您未包括 `ForAllValues` 修饰符，则用户可以将其他标签添加到函数中，只要它们也传递 `project`。

Example – 要求新函数具有标签

```
{
    "Version": "2012-10-17",
    "Statement": {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:TagResource"
      ],
      "Resource": "arn:aws:lambda:*:*:function:*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/project": "${aws:PrincipalTag/project}",
          "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
        },
        "ForAllValues:StringEquals": {
          "aws:TagKeys": "project"
        }
      }
    }
  }
```

步骤 2：允许基于附加到 Lambda 函数和 IAM 主体的标签执行操作

使用 aws:ResourceTag/tag-key 条件键创建第二个 IAM policy，以要求主体的标签与附加到函数的标签匹配。以下示例策略允许具有 `project` 标签的委托人调用具有 `project` 标签的函数。如果函数具有任何其他标签，则该操作将被拒绝。

Example – 要求函数和 IAM 主体的标签匹配

```
{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
```

```
          "lambda:InvokeFunction",
          "lambda:GetFunction"
        ],
        "Resource": "arn:aws:lambda:*:*:function:*",
        "Condition": {
          "StringEquals": {
            "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
          }
        }
      }
    ]
  }
```

步骤 3：授予列表权限

创建允许主体列出 Lambda 函数和 IAM 角色的策略。这样，主体就可以在控制台上以及调用 API 操作时查看所有 Lambda 函数和 IAM 角色。

Example – 授予 Lambda 和 IAM 列表权限

```
{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Sid": "AllResourcesLambdaNoTags",
        "Effect": "Allow",
        "Action": [
          "lambda:GetAccountSettings",
          "lambda:ListFunctions",
          "iam:ListRoles"
        ],
        "Resource": "*"
      }
    ]
  }
```

步骤 4：授予 IAM 权限

创建允许 iam:PassRole 的策略。当您将执行角色分配给函数时，需要此权限。在以下示例策略中，将示例 ARN 替换为 Lambda 执行角色的 ARN。

> **Note**
>
> 不要在策略中将 ResourceTag 条件键与 iam:PassRole 操作一起使用。您无法在 IAM 角色上使用标签以控制可以传递该角色的用户的访问权限。有关将角色传递给服务所需权限的更多信息，请参阅授予用户将角色传递给 AWS 服务的权限。

Example – 授予传递执行角色的权限

```
{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Sid": "VisualEditor0",
        "Effect": "Allow",
        "Action": [
          "iam:PassRole"
        ],
        "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
      }
    ]
  }
```

步骤 5：创建 IAM 角色

使用角色委托权限是最佳实践。创建名为 abac-project-role 的 IAM 角色：

- 在步骤 1：选择可信实体中：选择 AWS 账户，然后选择此账户。
- 在步骤 2：添加权限上：附加您在前面的步骤中创建的四个 IAM policy。
- 在步骤 3：命名、查看和创建上：选择 Add tag（添加标签）。对于键，输入 project。不要输入 Value（值）。

步骤 6：创建 IAM 用户

创建名为 abac-test-user 的 IAM 用户。在 Set permissions（设置权限）部分中，选择 Attach existing policies directly（直接附加现有策略），然后选择 Create policy（创建策略）。输入以下策略定义。将 *111122223333* 替换为您的 AWS 账户 ID。此策略允许 abac-test-user 担任 abac-project-role。

Example – 允许 IAM 用户担任 ABAC 角色

```
{
    "Version": "2012-10-17",
    "Statement": {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
    }
  }
```

步骤 7：测试权限

1. 以 abac-test-user 身份登录到 AWS 控制台。有关更多信息，请参阅作为 IAM 用户登录。
2. 切换到 abac-project-role 角色。有关更多信息，请参阅切换到角色（控制台）。
3. 创建 Lambda 函数：
   - 在 Permissions（权限）下，选择 Change default execution role（更改默认执行角色），然后对于 Execution role（执行角色），选择 Use an existing role（使用现有角色）。选择您在 步骤 4：授予 IAM 权限 中使用的相同执行角色。
   - 在 Advanced settings（高级设置）下，选择 Enable tags（启用标签），然后选择 Add new tag（添加新标签）。对于键，输入 project。不要输入 Value（值）。
4. 测试函数。
5. 创建第二个 Lambda 函数并添加其他标签，例如 environment。此操作应会失败，因为您在 步骤 1：要求新函数具有标签 中创建的 ABAC 策略仅允许主体创建具有 project 标签的函数。
6. 创建第三个没有标签的函数。此操作应会失败，因为您在 步骤 1：要求新函数具有标签 中创建的 ABAC 策略不允许主体创建没有标签的函数。

此授权策略允许您控制访问权限，而无需为每个新用户创建新的策略。要向新用户授予访问权限，只需授予他们担任与其所分配项目相对应的角色的权限。

步骤 8：清理资源

要删除 IAM 角色

1. 打开 IAM 控制台的角色页面。
2. 选择您在步骤 5 中创建的角色。
3. 选择删除。

4. 要确认删除，在文本输入字段中输入角色名称。
5. 选择删除。

删除 IAM 用户

1. 打开 IAM 控制台的用户页面。
2. 选择您在步骤 6 中创建的 IAM 用户。
3. 选择删除。
4. 要确认删除，在文本输入字段中输入用户名。
5. 选择删除用户。

删除 Lambda 函数

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择您创建的函数。
3. 依次选择操作和删除。
4. 在文本输入字段中键入 **`delete`**，然后选择 Delete（删除）。

微调策略的“资源和条件”部分

您可以通过在 AWS Identity and Access Management (IAM) 策略中指定资源和条件来限制用户权限的范围。策略中的每个操作都支持资源和条件类型的组合，这些类型根据操作的行为而有所不同。

每条 IAM 策略语句为对一个资源执行的一个操作授予权限。如果操作不对指定资源执行操作，或者您授予对所有资源执行操作的权限，则策略中资源的值为通配符（*）。对于许多操作，可以通过指定资源的 Amazon 资源名称（ARN）或与多个资源匹配的 ARN 模式来限制用户可修改的资源。

按资源类型划分，限制操作范围的一般设计如下：

- 函数 – 对函数进行的操作可以通过函数、版本或别名 ARN 限制到特定函数上。
- 事件源映射 – 可以通过 ARN 将操作限制到特定的事件源映射资源上。事件源映射始终与函数相关联，所以也可以使用 `lambda:FunctionArn` 条件来限制关联函数的操作。
- 层 – 与层使用和权限相关的操作作用于层的版本。
- 代码签名配置 – 可以通过 ARN 将操作限制到特定的代码签名配置资源上。
- 标签 – 使用标准标签条件。有关更多信息，请参阅 the section called “基于属性的访问控制”。

要按资源限制权限，请指定资源的 ARN。

Lambda 资源 ARN 格式

- 函数 – arn:aws:lambda:*us-west-2*:*123456789012*:function:*my-function*
- 函数版本 – arn:aws:lambda:*us-west-2*:*123456789012*:function:*my-function*:*1*
- 函数别名 – arn:aws:lambda:*us-west-2*:*123456789012*:function:*my-function*:*TEST*
- 事件源映射 – arn:aws:lambda:*us-west-2*:*123456789012*:event-source-mapping:*fa123456-14a1-4fd2-9fec-83de64ad683de6d47*
- 层 – arn:aws:lambda:*us-west-2*:*123456789012*:layer:*my-layer*
- 层版本 – arn:aws:lambda:*us-west-2*:*123456789012*:layer:*my-layer*:*1*
- 代码签名配置 – arn:aws:lambda:*us-west-2*:*123456789012*:code-signing-config:*my-csc*

例如，以下策略允许 AWS 账户 123456789012 中的用户调用美国西部（俄勒冈）AWS 区域中名为 my-function 的函数。

Example 调用函数策略

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Invoke",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-function"
        }
    ]
}
```

这是一种特殊情形，其中，操作标识符 (lambda:InvokeFunction) 不同于 API 操作 (Invoke)。对于其他操作，操作标识符为操作名称加上 lambda: 前缀。

Sections

- 了解策略中的“条件”部分

- 在策略的“资源”部分中引用函数
- 支持的 IAM 操作及函数行为

了解策略中的“条件”部分

条件是可选的策略元素，它应用其他逻辑来确定是否允许执行操作。除了所有操作支持的公用条件之外，Lambda 定义了一些条件类型，您可以用来限制某些操作的额外参数的值。

例如，`lambda:Principal` 条件允许您限制用户可以根据函数的基于资源的策略授予调用访问权限的服务或账户。以下策略允许用户授予对 Amazon Simple Notification Service (Amazon SNS) 主题的权限，以调用名为 `test` 的函数。

Example 管理函数策略权限

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ManageFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:test:*",
            "Condition": {
                "StringEquals": {
                    "lambda:Principal": "sns.amazonaws.com"
                }
            }
        }
    ]
}
```

该条件要求委托人是 Amazon SNS 而不是其他服务或账户。资源模式要求函数名称为 `test` 并包含版本号或别名。例如：`test:v1`。

有关 Lambda 和其他 AWS 服务的资源和条件的更多信息，请参阅服务授权参考中的 AWS 服务的操作、资源和条件键。

在策略的“资源”部分中引用函数

您可以使用 Amazon Resource Name（ARN）在策略语句中引用 Lambda 函数。函数 ARN 的格式取决于您是要引用整个函数（无限定）、某个函数版本，还是别名（限定）。

调用 Lambda API 时，用户可以通过在 GetFunction `FunctionName` 参数中传递版本 ARN 或别名 ARN，或者通过在 GetFunction `Qualifier` 参数中设置值，来指定一个版本或别名。Lambda 通过比较 IAM 策略中的资源元素与在 API 调用中传递的 `FunctionName` 和 `Qualifier` 来做出授权决策。如果不匹配，Lambda 会拒绝该请求。

无论您是允许还是拒绝某个函数操作，都必须在策略声明中使用正确的函数 ARN 类型才能获得预期的结果。例如，假设您的策略引用了非限定 ARN，Lambda 会接受引用非限定 ARN 的请求，但拒绝引用限定 ARN 的请求。

> **Note**
>
> 不能使用通配符 (*) 匹配账户 ID。有关接受的语法的更多信息，请参阅《IAM 用户指南》中的 IAM JSON 策略参考。

Example 允许调用非限定 ARN

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction"
        }
    ]
}
```

如果您的策略引用了特定的限定 ARN，Lambda 会接受引用该 ARN 的请求，但拒绝引用非限定 ARN 的请求（例如 `myFunction:2`）。

Example 允许调用特定的限定 ARN

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction:1"
        }
    ]
}
```

如果您的策略引用了任何限定 ARN:*，Lambda 会接受任何限定 ARN，但拒绝引用非限定 ARN 的请求。

Example 允许调用任何限定 ARN

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
        }
    ]
}
```

如果您的策略引用了任何使用 * 的 ARN，Lambda 会接受任何限定或非限定 ARN。

Example 允许调用任何限定或非限定 ARN

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction*"
        }
    ]
}
```

支持的 IAM 操作及函数行为

操作定义了可通过 IAM 策略允许的内容。有关 Lambda 中支持的操作列表，请参阅《Service Authorization Reference》中的 Actions, resources, and condition keys for AWS Lambda。在大多数情况下，如果 IAM 操作允许 Lambda API 操作，则 IAM 操作的名称与 Lambda API 操作的名称相同，但以下情况除外：

API 操作	IAM 操作
Invoke	`lambda:InvokeFunction`
GetLayerVersion GetLayerVersionByArn	`lambda:GetLayerVersion`

除了《Service Authorization Reference》中定义的资源和条件，Lambda 还支持某些操作的以下资源和条件。其中许多资源和条件与策略“资源”部分中的引用函数有关。对函数进行的操作可以通过函数、版本或别名 ARN 限制为特定函数，如下表中所述。

操作	资源	Condition
AddPermission	函数版本	不适用
RemovePermission	函数别名	
Invoke – 权限：`lambda:InvokeFunction`		
UpdateFunctionConfiguration	不适用	`lambda:CodeSigningConfigArn`
CreateFunctionUrlConfig	函数别名	不适用
DeleteFunctionUrlConfig		
GetFunctionUrlConfig		
UpdateFunctionUrlConfig		

AWS Lambda 中的安全性

AWS的云安全性的优先级最高。作为 AWS 客户，您将从专为满足大多数安全敏感型企业的要求而打造的数据中心和网络架构中受益。

安全性是 AWS 和您的共同责任。责任共担模式将其描述为云的安全性和云中的安全性：

- 云的安全性 – AWS负责保护在AWS云中运行AWS服务的基础架构。AWS还向您提供可安全使用的服务。作为 AWS 合规性计划的一部分，第三方审核人员将定期测试和验证安全性的有效性。要了解适用于 AWS Lambda 的合规性计划，请参阅合规性计划范围内的 AWS 服务。
- 云中的安全性：您的责任由您使用的 AWS 服务决定。您还需要对其他因素负责，包括您的数据的敏感性、您公司的要求以及适用的法律法规。

此文档将帮助您了解如何在使用 Lambda 时应用责任共担模型。以下主题说明如何配置 Lambda 以实现您的安全性和合规性目标。您还会了解如何使用其他AWS服务以帮助您监控和保护 Lambda 资源。

有关向 Lambda 应用程序应用安全原则的更多信息，请参阅 Serverless Land 中的 Security。

主题

AWS Lambda 中的数据保护

AWS 责任共担模式适用于 AWS Lambda 中的数据保护。如该模式中所述，AWS 负责保护运行所有 AWS Cloud 的全球基础架构。您负责维护对托管在此基础架构上的内容的控制。您还负责您所使用的 AWS 服务 的安全配置和管理任务。有关数据隐私的更多信息，请参阅数据隐私常见问题。有关欧洲数据保护的信息，请参阅 AWS Security Blog 上的 AWS Shared Responsibility Model and GDPR 博客文章。

出于数据保护目的，我们建议您保护 AWS 账户 凭证并使用 AWS IAM Identity Center 或 AWS Identity and Access Management（IAM）设置单个用户。这样，每个用户只获得履行其工作职责所需的权限。我们还建议您通过以下方式保护数据：

- 对每个账户使用多重身份验证（MFA）。
- 使用 SSL/TLS 与 AWS 资源进行通信。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 使用 AWS CloudTrail 设置 API 和用户活动日记账记录。有关使用 CloudTrail 跟踪来捕获 AWS 活动的信息，请参阅《AWS CloudTrail 用户指南》中的 Working with CloudTrail trails。
- 使用 AWS 加密解决方案以及 AWS 服务 中的所有默认安全控制。
- 使用高级托管安全服务（例如 Amazon Macie），它有助于发现和保护存储在 Amazon S3 中的敏感数据。
- 如果在通过命令行界面或 API 访问 AWS 时需要经过 FIPS 140-3 验证的加密模块，请使用 FIPS 端点。有关可用的 FIPS 端点的更多信息，请参阅美国联邦信息处理标准（FIPS）140-3。

我们强烈建议您切勿将机密信息或敏感信息（如您客户的电子邮件地址）放入标签或自由格式文本字段（如名称字段）。这包括通过控制台、API、AWS CLI 或 AWS 开发工具包使用 Lambda 或其他 AWS 服务 服务时。在用于名称的标签或自由格式文本字段中输入的任何数据都可能会用于计费或诊断日志。如果您向外部服务器提供网址，强烈建议您不要在网址中包含凭证信息来验证对该服务器的请求。

小节目录

传输中加密

Lambda API 端点仅支持基于 HTTPS 的安全连接。使用 AWS Management Console、AWS开发工具包或 Lambda API 管理 Lambda 资源时，所有通信都使用传输层安全性 (TLS) 进行加密。有关 API 终端节点的完整列表，请参阅 AWS 一般参考 中的 AWS区域和终端节点。

当您将函数连接到文件系统时，Lambda 会对所有连接使用传输中加密。有关更多信息，请参阅《Amazon Elastic File System 用户指南》中的 Amazon EFS 中的数据加密。

当您使用环境变量时，您可以启用控制台加密帮助程序，以使用客户端加密来保护传输中的环境变量。有关更多信息，请参阅 保护 Lambda 环境变量。

AWS Lambda 中的静态数据加密

Lambda 始终加密静态环境变量。默认情况下，Lambda 使用它在您的账户中创建的 AWS KMS key，以加密您的环境变量。该 AWS 托管式密钥 名为 `aws/lambda`。

在每个函数的基础上，您可以选择性地将 Lambda 配置为使用客户托管式密钥，而不是使用默认 AWS 托管式密钥 来加密您的环境变量。有关更多信息，请参阅 保护 Lambda 环境变量。

如果函数的事件源映射具有筛选条件对象，则 Lambda 还会默认使用 AWS KMS 密钥对筛选条件进行加密。您可以选择使用客户自主管理型密钥加密筛选条件。

Lambda 始终会对您上载到 Lambda 的文件进行加密，包括部署包和层存档。

默认情况下，Amazon CloudWatch Logs 和 AWS X-Ray 也会对数据进行加密，并可配置为使用客户托管密钥。有关详细信息，请参阅 Encrypt log data in CloudWatch Logs 和 Data protection in AWS X-Ray。

Sections

- 为 Lambda 监控您的加密密钥

为 Lambda 监控您的加密密钥

将 AWS KMS 客户自主管理型密钥与 Lambda 一起使用时，您可以使用 AWS CloudTrail。以下示例是 Lambda 进行的 `Decrypt`、`DescribeKey` 和 `GenerateDataKey` 调用的 CloudTrail 事件，用于访问由您的客户自主管理型密钥加密的数据。

Decrypt

如果您使用 AWS KMS 客户自主管理型密钥对筛选条件对象进行加密，那么当您尝试以纯文本形式访问该对象时（例如，通过 `ListEventSourceMappings` 调用），Lambda 会代表您发送 `Decrypt` 请求。以下示例事件记录了 `Decrypt` 操作：

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
```

```
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:45:23Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2024-05-30T01:05:46Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "requestParameters": {
        "keyId": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "encryptionContext": {
            "aws-crypto-public-key": "ABCD
+7876787678+CDEFGHIJKL/888666888999888555444111555222888333111==",
            "aws:lambda:EventSourceArn": "arn:aws:sqs:eu-west-1:123456789012:sample-
source",
            "aws:lambda:FunctionArn": "arn:aws:lambda:eu-
west-1:123456789012:function:sample-function"
        },
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
```

```
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "sessionCredentialFromConsole": "true"
}
```

DescribeKey

如果您使用 AWS KMS 客户自主管理型密钥对筛选条件对象进行加密，那么当您尝试访问该对象时（例如，通过 GetEventSourceMapping 调用），Lambda 会代表您发送 DescribeKey 请求。以下示例事件记录了 DescribeKey 操作：

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:45:23Z",
                "mfaAuthenticated": "false"
            }
        }
    },
    "eventTime": "2024-05-30T01:09:40Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "DescribeKey",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "54.240.197.238",
    "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36",
```

```
    "requestParameters": {
        "keyId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "tlsDetails": {
        "tlsVersion": "TLSv1.3",
        "cipherSuite": "TLS_AES_256_GCM_SHA384",
        "clientProvidedHostHeader": "kms.eu-west-1.amazonaws.com"
    },
    "sessionCredentialFromConsole": "true"
}
```

GenerateDataKey

使用 AWS KMS 客户自主管理型密钥在 `CreateEventSourceMapping` 或 `UpdateEventSourceMapping` 调用中加密筛选条件对象时，Lambda 会代表您发送 `GenerateDataKey` 请求，要求生成用于加密筛选条件的数据密钥（信封加密）。以下示例事件记录 `GenerateDataKey` 操作：

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
```

```
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:06:07Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2024-05-30T01:04:18Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "requestParameters": {
        "numberOfBytes": 32,
        "keyId": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "encryptionContext": {
            "aws-crypto-public-key": "ABCD
+7876787678+CDEFGHIJKL/888666888999888555444111555222888333111==",
            "aws:lambda:EventSourceArn": "arn:aws:sqs:eu-west-1:123456789012:sample-
source",
            "aws:lambda:FunctionArn": "arn:aws:lambda:eu-
west-1:123456789012:function:sample-function"
        },
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
```

```
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management"
}
```

适用于 AWS Lambda 的 Identity and Access Management

AWS Identity and Access Management（IAM）是一项 AWS 服务，可以帮助管理员安全地控制对 AWS 资源的访问。IAM 管理员控制可以通过身份验证（登录）和授权（具有权限）使用 Lambda 资源的人员。IAM 是一项无需额外费用即可使用的 AWS 服务。

主题

受众

AWS Identity and Access Management (IAM) 的使用方式因您可以在 Lambda 中执行的操作而异。

服务用户 - 如果您使用 Lambda 服务来完成工作，您的管理员会为您提供所需的凭证和权限。当您使用更多 Lambda 功能来完成工作时，您可能需要其他权限。了解如何管理访问权限可帮助您向管理员请求适合的权限。如果您无法访问 Lambda 中的一项功能，请参阅 对 AWS Lambda 身份和访问进行故障排除。

服务管理员 – 如果您在公司负责管理 Lambda 资源，您可能具有 Lambda 的完全访问权限。您有责任确定您的服务用户应访问哪些 Lambda 功能和资源。然后，您必须向 IAM 管理员提交请求以更改服务用户的权限。请查看该页面上的信息以了解 IAM 的基本概念。要了解有关您的公司如何将 IAM 与 Lambda 搭配使用的更多信息，请参阅 AWS Lambda 如何与 IAM 协同工作。

IAM 管理员 – 如果您是 IAM 管理员，您可能希望了解如何编写策略以管理对 Lambda 的访问的详细信息。要查看您可在 IAM 中使用的 Lambda 基于身份的策略示例，请参阅 适用于 AWS Lambda 的基于身份的策略示例。

使用身份进行身份验证

身份验证是您使用身份凭证登录 AWS 的方法。您必须作为 AWS 账户根用户、IAM 用户或通过代入 IAM 角色进行身份验证（登录到 AWS）。

您可以使用通过身份源提供的凭证以联合身份登录到 AWS。AWS IAM Identity Center（IAM Identity Center）用户、您公司的单点登录身份验证以及您的 Google 或 Facebook 凭证都是联合身份的示例。当您以联合身份登录时，您的管理员以前使用 IAM 角色设置了身份联合验证。当您使用联合身份验证访问 AWS 时，您就是在间接代入角色。

根据您的用户类型，您可以登录 AWS Management Console 或 AWS 访问门户。有关登录到 AWS 的更多信息，请参阅《AWS 登录 用户指南》中的如何登录到您的 AWS 账户。

如果您以编程方式访问 AWS，则 AWS 将提供软件开发工具包 (SDK) 和命令行界面 (CLI)，以便使用您的凭证以加密方式签署您的请求。如果您不使用 AWS 工具，则必须自行对请求签名。有关使用建议的方法自行签署请求的更多信息，请参阅《IAM 用户指南》中的适用于 API 请求的 AWS 签名版本 4。

无论使用何种身份验证方法，您可能需要提供其他安全信息。例如，AWS 建议您使用多重身份验证（MFA）来提高账户的安全性。要了解更多信息，请参阅《AWS IAM Identity Center 用户指南》中的 Multi-factor authentication 和《IAM 用户指南》中的 IAM 中的 AWS 多重身份验证。

AWS 账户 根用户

当您创建 AWS 账户 时，最初使用的是一个对账户中所有 AWS 服务 和资源拥有完全访问权限的登录身份。此身份称为 AWS 账户 根用户，使用您创建账户时所用的电子邮件地址和密码登录，即可获得该身份。强烈建议您不要使用根用户执行日常任务。保护好根用户凭证，并使用这些凭证来执行仅根用户可以执行的任务。有关要求您以根用户身份登录的任务的完整列表，请参阅《IAM 用户指南》中的需要根用户凭证的任务。

联合身份

作为最佳实践，要求人类用户（包括需要管理员访问权限的用户）结合使用联合身份验证和身份提供程序，以使用临时凭证来访问 AWS 服务。

联合身份是来自企业用户目录、Web 身份提供程序、AWS Directory Service、Identity Center 目录的用户，或任何使用通过身份源提供的凭证来访问 AWS 服务 的用户。当联合身份访问 AWS 账户 时，他们代入角色，而角色提供临时凭证。

要集中管理访问权限，建议您使用 AWS IAM Identity Center。您可以在 IAM Identity Center 中创建用户和组，也可以连接并同步到您自己的身份源中的一组用户和组以跨所有 AWS 账户 和应用程序使用。有关 IAM Identity Center 的信息，请参阅《AWS IAM Identity Center 用户指南》中的什么是 IAM Identity Center？

IAM 用户和群组

IAM 用户是 AWS 账户 内对某个人员或应用程序具有特定权限的一个身份。在可能的情况下，我们建议使用临时凭证，而不是创建具有长期凭证（如密码和访问密钥）的 IAM 用户。但是，如果您有一些特定的使用场景需要长期凭证以及 IAM 用户，建议您轮换访问密钥。有关更多信息，请参阅《IAM 用户指南》中的 对于需要长期凭证的使用场景定期轮换访问密钥。

IAM 组是一个指定一组 IAM 用户的身份。您不能使用组的身份登录。您可以使用组来一次性为多个用户指定权限。如果有大量用户，使用组可以更轻松地管理用户权限。例如，您可能具有一个名为 IAMAdmins 的组，并为该组授予权限以管理 IAM 资源。

用户与角色不同。用户唯一地与某个人员或应用程序关联，而角色旨在让需要它的任何人代入。用户具有永久的长期凭证，而角色提供临时凭证。要了解更多信息，请参阅《IAM 用户指南》中的 IAM 用户的使用案例。

IAM 角色

IAM 角色是 AWS 账户 中具有特定权限的身份。它类似于 IAM 用户，但与特定人员不关联。要在 AWS Management Console中临时代入 IAM 角色，可以从用户切换到 IAM 角色（控制台）。您可以调用 AWS CLI 或 AWS API 操作或使用自定义网址以担任角色。有关使用角色的方法的更多信息，请参阅《IAM 用户指南》中的担任角色的方法。

具有临时凭证的 IAM 角色在以下情况下很有用：

- 联合用户访问 – 要向联合身份分配权限，请创建角色并为角色定义权限。当联合身份进行身份验证时，该身份将与角色相关联并被授予由此角色定义的权限。有关联合身份验证的角色的信息，请参阅《IAM 用户指南》中的为第三方身份提供商创建角色。如果您使用 IAM Identity Center，则需要配置权限集。为控制您的身份在进行身份验证后可以访问的内容，IAM Identity Center 将权限集与 IAM 中的角色相关联。有关权限集的信息，请参阅《AWS IAM Identity Center 用户指南》中的 权限集。
- 临时 IAM 用户权限 – IAM 用户可代入 IAM 用户或角色，以暂时获得针对特定任务的不同权限。
- 跨账户存取 – 您可以使用 IAM 角色以允许不同账户中的某个人（可信主体）访问您的账户中的资源。角色是授予跨账户访问权限的主要方式。但是，对于某些 AWS 服务，您可以将策略直接附加到资源（而不是使用角色作为代理）。要了解用于跨账户访问的角色和基于资源的策略之间的差别，请参阅《IAM 用户指南》中的 IAM 中的跨账户资源访问。

- 跨服务访问：某些 AWS 服务使用其它 AWS 服务中的特征。例如，当您在某个服务中进行调用时，该服务通常会在 Amazon EC2 中运行应用程序或在 Simple Storage Service (Amazon S3) 中存储对象。服务可能会使用发出调用的主体的权限、使用服务角色或使用服务相关角色来执行此操作。
  - 转发访问会话：当您使用 IAM 用户或角色在 AWS 中执行操作时，您将被视为主体。使用某些服务时，您可能会执行一个操作，然后此操作在其他服务中启动另一个操作。FAS 使用主体调用 AWS 服务 的权限，结合请求的 AWS 服务，向下游服务发出请求。只有在服务收到需要与其他 AWS 服务 或资源交互才能完成的请求时，才会发出 FAS 请求。在这种情况下，您必须具有执行这两个操作的权限。有关发出 FAS 请求时的策略详情，请参阅转发访问会话。
  - 服务角色 - 服务角色是服务代表您在您的账户中执行操作而分派的 IAM 角色。IAM 管理员可以在 IAM 中创建、修改和删除服务角色。有关更多信息，请参阅《IAM 用户指南》中的创建向 AWS 服务 委派权限的角色。
  - 服务相关角色 – 服务相关角色是与 AWS 服务 关联的一种服务角色。服务可以代入代表您执行操作的角色。服务相关角色显示在您的 AWS 账户 中，并由该服务拥有。IAM 管理员可以查看但不能编辑服务相关角色的权限。
- 在 Amazon EC2 上运行的应用程序 – 您可以使用 IAM 角色管理在 EC2 实例上运行并发出 AWS CLI 或 AWS API 请求的应用程序的临时凭证。这优先于在 EC2 实例中存储访问密钥。要将 AWS 角色分配给 EC2 实例并使其对该实例的所有应用程序可用，您可以创建一个附加到实例的实例配置文件。实例配置文件包含角色，并使 EC2 实例上运行的程序能够获得临时凭证。有关更多信息，请参阅《IAM 用户指南》中的使用 IAM 角色为 Amazon EC2 实例上运行的应用程序授予权限。

使用策略管理访问

您将创建策略并将其附加到 AWS 身份或资源，以控制 AWS 中的访问。策略是 AWS 中的对象；在与身份或资源相关联时，策略定义它们的权限。在主体（用户、根用户或角色会话）发出请求时，AWS 将评估这些策略。策略中的权限确定是允许还是拒绝请求。大多数策略在 AWS 中存储为 JSON 文档。有关 JSON 策略文档的结构和内容的更多信息，请参阅 IAM 用户指南中的 JSON 策略概览。

管理员可以使用 AWS JSON 策略来指定谁有权访问什么内容。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

默认情况下，用户和角色没有权限。要授予用户对所需资源执行操作的权限，IAM 管理员可以创建 IAM 策略。管理员随后可以向角色添加 IAM 策略，用户可以代入角色。

IAM 策略定义操作的权限，无关乎您使用哪种方法执行操作。例如，假设您有一个允许 `iam:GetRole` 操作的策略。具有该策略的用户可以从 AWS Management Console、AWS CLI 或 AWS API 获取角色信息。

基于身份的策略

基于身份的策略是可附加到身份（如 IAM 用户、用户组或角色）的 JSON 权限策略文档。这些策略控制用户和角色可在何种条件下对哪些资源执行哪些操作。要了解如何创建基于身份的策略，请参阅《IAM 用户指南》中的使用客户管理型策略定义自定义 IAM 权限。

基于身份的策略可以进一步归类为内联策略或托管策略。内联策略直接嵌入单个用户、组或角色中。托管式策略是可以附加到 AWS 账户 中的多个用户、组和角色的独立策略。托管式策略包括 AWS 托管式策略和客户管理型策略。要了解如何在托管式策略和内联策略之间进行选择，请参阅《IAM 用户指南》中的在托管式策略与内联策略之间进行选择。

基于资源的策略

基于资源的策略是附加到资源的 JSON 策略文档。基于资源的策略的示例包括 IAM 角色信任策略和 Amazon S3 存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。对于在其中附加策略的资源，策略定义指定主体可以对该资源执行哪些操作以及在什么条件下执行。您必须在基于资源的策略中指定主体。主体可以包括账户、用户、角色、联合用户或 AWS 服务。

基于资源的策略是位于该服务中的内联策略。您不能在基于资源的策略中使用来自 IAM 的 AWS 托管策略。

访问控制列表 (ACL)

访问控制列表（ACL）控制哪些主体（账户成员、用户或角色）有权访问资源。ACL 与基于资源的策略类似，尽管它们不使用 JSON 策略文档格式。

Simple Storage Service (Amazon S3)、AWS WAF 和 Amazon VPC 是支持 ACL 的服务示例。要了解有关 ACL 的更多信息，请参阅《Amazon Simple Storage Service 开发人员指南》中的访问控制列表（ACL）概览。

其他策略类型

AWS 支持额外的、不太常用的策略类型。这些策略类型可以设置更常用的策略类型向您授予的最大权限。

- 权限边界：权限边界是一个高级特征，用于设置基于身份的策略可以为 IAM 实体（IAM 用户或角色）授予的最大权限。您可为实体设置权限边界。这些结果权限是实体基于身份的策略及其权限边界的交集。在 `Principal` 中指定用户或角色的基于资源的策略不受权限边界限制。任一项策略中的显式拒绝将覆盖允许。有关权限边界的更多信息，请参阅《IAM 用户指南》中的 IAM 实体的权限边界。

- 服务控制策略 (SCP) – SCP 是 JSON 策略，指定了组织或组织单元 (OU) 在 AWS Organizations 中的最大权限。AWS Organizations 服务可以分组和集中管理您的企业拥有的多个 AWS 账户 账户。如果在组织内启用了所有功能，则可对任意或全部账户应用服务控制策略 (SCP)。SCP 限制成员账户中实体（包括每个 AWS 账户根用户）的权限。有关组织和 SCP 的更多信息，请参阅《AWS Organizations User Guide》中的 Service control policies。
- 会话策略 – 会话策略是当您以编程方式为角色或联合用户创建临时会话时作为参数传递的高级策略。结果会话的权限是用户或角色的基于身份的策略和会话策略的交集。权限也可以来自基于资源的策略。任一项策略中的显式拒绝将覆盖允许。有关更多信息，请参阅《IAM 用户指南》中的会话策略。

多个策略类型

当多个类型的策略应用于一个请求时，生成的权限更加复杂和难以理解。要了解 AWS 如何确定在涉及多种策略类型时是否允许请求，请参阅《IAM 用户指南》中的策略评估逻辑。

AWS Lambda 如何与 IAM 协同工作

在使用 IAM 管理对 Lambda 的访问之前，您应该了解哪些 IAM 功能可用于 Lambda。

IAM 功能	Lambda 支持
基于身份的策略	是
基于资源的策略	是
策略操作	是
策略资源	是
策略条件键（特定于服务）	是
ACL	否
ABAC（策略中的标签）	部分
临时凭证	是
转发访问会话 (FAS)	否

IAM 功能	Lambda 支持
服务角色	是
服务相关角色	部分

要深入了解 Lambda 和其他 AWS 服务如何与大多数 IAM 功能配合使用，请参阅《IAM 用户指南》中的使用 IAM 的 AWS 服务。

Lambda 的基于身份的策略

支持基于身份的策略：是

基于身份的策略是可附加到身份（如 IAM 用户、用户组或角色）的 JSON 权限策略文档。这些策略控制用户和角色可在何种条件下对哪些资源执行哪些操作。要了解如何创建基于身份的策略，请参阅《IAM 用户指南》中的使用客户管理型策略定义自定义 IAM 权限。

通过使用 IAM 基于身份的策略，您可以指定允许或拒绝的操作和资源以及允许或拒绝操作的条件。您无法在基于身份的策略中指定主体，因为它适用于其附加的用户或角色。要了解可在 JSON 策略中使用的所有元素，请参阅《IAM 用户指南》中的 IAM JSON 策略元素引用。

Lambda 基于身份的策略示例

要查看 Lambda 基于身份的策略示例，请参阅适用于 AWS Lambda 的基于身份的策略示例。

Lambda 内基于资源的策略

支持基于资源的策略：是

基于资源的策略是附加到资源的 JSON 策略文档。基于资源的策略的示例包括 IAM 角色信任策略和 Amazon S3 存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。对于在其中附加策略的资源，策略定义指定主体可以对该资源执行哪些操作以及在什么条件下执行。您必须在基于资源的策略中指定主体。主体可以包括账户、用户、角色、联合用户或 AWS 服务。

要启用跨账户存取，您可以将整个账户或其他账户中的 IAM 实体指定为基于资源的策略中的主体。将跨账户主体添加到基于资源的策略只是建立信任关系工作的一半而已。当主体和资源处于不同的 AWS 账户 中时，则信任账户中的 IAM 管理员还必须授予主体实体（用户或角色）对资源的访问权限。他们通过将基于身份的策略附加到实体以授予权限。但是，如果基于资源的策略向同一个账户中的主体授予

访问权限，则不需要额外的基于身份的策略。有关更多信息，请参阅《IAM 用户指南》中的 IAM 中的跨账户资源访问。

您可以将基于资源的策略附加到 Lambda 函数或层。此策略定义了哪些主体可以对函数或层执行操作。

要了解如何将基于资源的策略附加到函数或层，请参阅在 Lambda 中使用基于资源的 IAM 策略。

Lambda 的策略操作

支持策略操作：是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么内容。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

JSON 策略的 `Action` 元素描述可用于在策略中允许或拒绝访问的操作。策略操作通常与关联的 AWS API 操作同名。有一些例外情况，例如没有匹配 API 操作的仅限权限 操作。还有一些操作需要在策略中执行多个操作。这些附加操作称为相关操作。

在策略中包含操作以授予执行关联操作的权限。

要查看 Lambda 操作的列表，请参阅《Service Authorization Reference》中的 Actions defined by AWS Lambda。

Lambda 中的策略操作在操作前使用以下前缀：

```
lambda
```

要在单个语句中指定多项操作，请使用逗号将它们隔开。

```
"Action": [
      "lambda:action1",
      "lambda:action2"
         ]
```

要查看 Lambda 基于身份的策略示例，请参阅适用于 AWS Lambda 的基于身份的策略示例。

Lambda 的策略资源

支持策略资源：是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么内容。也就是说，哪个主体 可以对什么资源执行操作，以及在什么条件下执行。

Resource JSON 策略元素指定要向其应用操作的一个或多个对象。语句必须包含 Resource 或 NotResource 元素。作为最佳实践，请使用其 Amazon 资源名称 (ARN) 指定资源。对于支持特定资源类型（称为资源级权限）的操作，您可以执行此操作。

对于不支持资源级权限的操作（如列出操作），请使用通配符 (*) 指示语句应用于所有资源。

```
"Resource": "*"
```

要查看 Lambda 资源类型及其 ARN 的列表，请参阅《Service Authorization Reference》中 Resource types defined by AWS Lambda。要了解可以在哪些操作中指定每个资源的 ARN，请参阅 AWS Lambda 定义的操作。

要查看 Lambda 基于身份的策略示例，请参阅适用于 AWS Lambda 的基于身份的策略示例。

Lambda 的策略条件键

支持特定于服务的策略条件键：是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么内容。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

在 Condition 元素（或 Condition 块）中，可以指定语句生效的条件。Condition 元素是可选的。您可以创建使用条件运算符（例如，等于或小于）的条件表达式，以使策略中的条件与请求中的值相匹配。

如果您在一个语句中指定多个 Condition 元素，或在单个 Condition 元素中指定多个键，则 AWS 使用逻辑 AND 运算评估它们。如果您为单个条件键指定多个值，则 AWS 使用逻辑 OR 运算来评估条件。在授予语句的权限之前必须满足所有的条件。

在指定条件时，您也可以使用占位符变量。例如，只有在使用 IAM 用户名标记 IAM 用户时，您才能为其授予访问资源的权限。有关更多信息，请参阅《IAM 用户指南》中的 IAM 策略元素：变量和标签。

AWS 支持全局条件键和特定于服务的条件键。要查看所有 AWS 全局条件键，请参阅《IAM 用户指南》中的 AWS 全局条件上下文键。

有关 Lambda 条件键的列表，请参阅《Service Authorization Reference》中的 Condition keys for AWS Lambda。要了解您可以对哪些操作和资源使用条件键，请参阅 AWS Lambda 定义的操作。

要查看 Lambda 基于身份的策略示例，请参阅适用于 AWS Lambda 的基于身份的策略示例。

Lambda 中的 ACL

支持 ACL：否

访问控制列表（ACL）控制哪些主体（账户成员、用户或角色）有权访问资源。ACL 与基于资源的策略类似，尽管它们不使用 JSON 策略文档格式。

ABAC 与 Lambda 配合使用

支持 ABAC（策略中的标签）：部分支持

基于属性的访问控制 (ABAC) 是一种授权策略，该策略基于属性来定义权限。在 AWS 中，这些属性称为标签。您可以将标签附加到 IAM 实体（用户或角色）以及 AWS 资源。标记实体和资源是 ABAC 的第一步。然后设计 ABAC 策略，以在主体的标签与他们尝试访问的资源标签匹配时允许操作。

ABAC 在快速增长的环境中非常有用，并在策略管理变得繁琐的情况下可以提供帮助。

要基于标签控制访问，您需要使用 `aws:ResourceTag/key-name`、`aws:RequestTag/key-name` 或 `aws:TagKeys` 条件键在策略的条件元素中提供标签信息。

如果某个服务对于每种资源类型都支持所有这三个条件键，则对于该服务，该值为是。如果某个服务仅对于部分资源类型支持所有这三个条件键，则该值为部分。

有关 ABAC 的更多信息，请参阅《IAM 用户指南》中的使用 ABAC 授权定义权限。要查看设置 ABAC 步骤的教程，请参阅《IAM 用户指南》中的使用基于属性的访问权限控制（ABAC）。

有关标记 Lambda 资源的更多信息，请参阅在 Lambda 中使用基于属性的访问控制。

将临时凭证与 Lambda 配合使用

支持临时凭证：是

某些 AWS 服务 在您使用临时凭证登录时无法正常工作。有关更多信息，包括 AWS 服务 与临时凭证配合使用，请参阅 IAM 用户指南中的使用 IAM 的 AWS 服务。

如果您不使用用户名和密码而用其它方法登录到 AWS Management Console，则使用临时凭证。例如，当您使用贵公司的单点登录（SSO）链接访问 AWS 时，该过程将自动创建临时凭证。当您以用户身份登录控制台，然后切换角色时，您还会自动创建临时凭证。有关切换角色的更多信息，请参阅《IAM 用户指南》中的从用户切换到 IAM 角色（控制台）。

您可以使用 AWS CLI 或者 AWS API 创建临时凭证。之后，您可以使用这些临时凭证访问 AWS。AWS 建议您动态生成临时凭证，而不是使用长期访问密钥。有关更多信息，请参阅 IAM 中的临时安全凭证。

转发 Lambda 访问会话

支持转发访问会话（FAS）：否

当您使用 IAM 用户或角色在 AWS 中执行操作时，您将被视为主体。使用某些服务时，您可能会执行一个操作，然后此操作在其他服务中启动另一个操作。FAS 使用主体调用 AWS 服务 的权限，结合请求的 AWS 服务，向下游服务发出请求。只有在服务收到需要与其他 AWS 服务 或资源交互才能完成的请求时，才会发出 FAS 请求。在这种情况下，您必须具有执行这两个操作的权限。有关发出 FAS 请求时的策略详情，请参阅转发访问会话。

Lambda 的服务角色

支持服务角色：是

服务角色是由一项服务担任、代表您执行操作的 IAM 角色。IAM 管理员可以在 IAM 中创建、修改和删除服务角色。有关更多信息，请参阅《IAM 用户指南》中的创建向 AWS 服务 委派权限的角色。

在 Lambda 中，服务角色称为执行角色。

> **Warning**
> 更改执行角色的权限可能会破坏 Lambda 功能。

Lambda 的服务相关角色

支持服务相关角色：部分支持

服务相关角色是一种与 AWS 服务 相关的服务角色。服务可以代入代表您执行操作的角色。服务相关角色显示在您的 AWS 账户 中，并由该服务拥有。IAM 管理员可以查看但不能编辑服务相关角色的权限。

Lambda 不具有服务相关角色，但 Lambda@Edge 具有。有关更多信息，请参阅《Amazon CloudFront 开发人员指南》中的 Lambda@Edge 的服务相关角色。

有关创建或管理服务相关角色的详细信息，请参阅能够与 IAM 搭配使用的 AWS 服务。在表中查找服务相关角色列中包含 Yes 的表。选择是链接以查看该服务的服务相关角色文档。

适用于 AWS Lambda 的基于身份的策略示例

默认情况下，用户和角色没有创建或修改 Lambda 资源的权限。他们也无法使用 AWS Management Console、AWS Command Line Interface（AWS CLI）或 AWS API 执行任务。要授予用户对所需资源执行操作的权限，IAM 管理员可以创建 IAM 策略。管理员随后可以向角色添加 IAM 策略，用户可以代入角色。

要了解如何使用这些示例 JSON 策略文档创建基于 IAM 身份的策略，请参阅《IAM 用户指南》中的创建 IAM 策略（控制台）。

有关 Lambda 定义的操作和资源类型的详细信息，包括每种资源类型的 ARN 格式，请参阅《Service Authorization Reference》中的 Actions, resources, and condition keys for AWS Lambda。

主题

策略最佳实践

基于身份的策略确定某个人是否可以创建、访问或删除您账户中的 Lambda 资源。这些操作可能会使 AWS 账户产生成本。创建或编辑基于身份的策略时，请遵循以下指南和建议：

- AWS 托管策略及转向最低权限许可入门 – 要开始向用户和工作负载授予权限，请使用 AWS 托管策略来为许多常见使用场景授予权限。您可以在 AWS 账户 中找到这些策略。我们建议通过定义特定于您的使用场景的 AWS 客户管理型策略来进一步减少权限。有关更多信息，请参阅《IAM 用户指南》中的 AWS 托管策略或工作职能的 AWS 托管策略。
- 应用最低权限 – 在使用 IAM 策略设置权限时，请仅授予执行任务所需的权限。为此，您可以定义在特定条件下可以对特定资源执行的操作，也称为最低权限许可。有关使用 IAM 应用权限的更多信息，请参阅《IAM 用户指南》中的 IAM 中的策略和权限。
- 使用 IAM 策略中的条件进一步限制访问权限 – 您可以向策略添加条件来限制对操作和资源的访问。例如，您可以编写策略条件来指定必须使用 SSL 发送所有请求。如果通过特定（AWS 服务例如 AWS CloudFormation）使用服务操作，您还可以使用条件来授予对服务操作的访问权限。有关更多信息，请参阅《IAM 用户指南》中的 IAM JSON 策略元素：条件。
- 使用 IAM Access Analyzer 验证您的 IAM 策略，以确保权限的安全性和功能性 – IAM Access Analyzer 会验证新策略和现有策略，以确保策略符合 IAM 策略语言 (JSON) 和 IAM 最佳实践。IAM

Access Analyzer 提供 100 多项策略检查和可操作的建议，以帮助您制定安全且功能性强的策略。有关更多信息，请参阅《IAM 用户指南》中的使用 IAM Access Analyzer 验证策略。

- 需要多重身份验证（MFA）– 如果您所处的场景要求您的 AWS 账户 中有 IAM 用户或根用户，请启用 MFA 来提高安全性。若要在调用 API 操作时需要 MFA，请将 MFA 条件添加到您的策略中。有关更多信息，请参阅《IAM 用户指南》中的使用 MFA 保护 API 访问。

有关 IAM 中的最佳实操的更多信息，请参阅《IAM 用户指南》中的 IAM 中的安全最佳实操。

使用 Lambda 控制台

要访问 AWS Lambda控制台，您必须拥有一组最低的权限。这些权限必须允许您列出和查看有关 AWS 账户 中 Lambda 资源的详细信息。如果创建比必需的最低权限更为严格的基于身份的策略，对于附加了该策略的实体（用户或角色），控制台将无法按预期正常运行。

对于只需要调用 AWS CLI 或 AWS API 的用户，无需为其提供最低控制台权限。相反，只允许访问与其尝试执行的 API 操作相匹配的操作。

有关授予函数开发的最小访问权限的示例策略，请参阅授予用户对 Lambda 函数的访问权限。除了 Lambda API 以外，Lambda 控制台还使用其他服务来显示触发器配置，并允许您添加新触发器。如果您的用户将 Lambda 与其他服务结合使用，则他们还需要这些服务的访问权限。有关配置其他服务及 Lambda 的详细信息，请参阅使用来自其 AWS 他服务的事件调用 Lambda。

允许用户查看他们自己的权限

该示例说明了您如何创建策略，以允许 IAM 用户查看附加到其用户身份的内联和托管策略。此策略包括在控制台上完成此操作或者以编程方式使用 AWS CLI 或 AWS API 所需的权限。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
```

```
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

适用于 AWS Lambda 的 AWS 托管式策略

AWS 托管式策略是由 AWS 创建和管理的独立策略。AWS 托管式策略旨在为许多常见用例提供权限，以便您可以开始为用户、组和角色分配权限。

请记住，AWS 托管式策略可能不会为您的特定使用场景授予最低权限，因为它们可供所有 AWS 客户使用。我们建议通过定义特定于您的使用场景的客户托管式策略来进一步减少权限。

您无法更改 AWS 托管式策略中定义的权限。如果 AWS 更新在 AWS 托管式策略中定义的权限，则更新会影响该策略所附加到的所有主体身份（用户、组和角色）。当新的 AWS 服务 启动或新的 API 操作可用于现有服务时，AWS 最有可能更新 AWS 托管式策略。

有关更多信息，请参阅《IAM 用户指南》中的 AWS 托管式策略。

主题

- AWS 托管式策略：AWSLambda_FullAccess
- AWS 托管式策略：AWSLambda_ReadOnlyAccess
- AWS 托管式策略：AWSLambdaBasicExecutionRole
- AWS 托管式策略：AWSLambdaDynamoDBExecutionRole
- AWS 托管式策略：AWSLambdaENIManagementAccess

- AWS 托管式策略：AWSLambdaExecute
- AWS 托管式策略：AWSLambdaInvocation-DynamoDB
- AWS 托管式策略：AWSLambdaKinesisExecutionRole
- AWS 托管式策略：AWSLambdaMSKExecutionRole
- AWS 托管式策略：AWSLambdaRole
- AWS 托管式策略：AWSLambdaSQSQueueExecutionRole
- AWS 托管式策略：AWSLambdaVPCAccessExecutionRole
- Lambda 对 AWS 托管式策略的更新

AWS 托管式策略：AWSLambda_FullAccess

该策略向所有 Lambda 操作授予完全访问权限。它还向用于开发和维护 Lambda 资源的其他 AWS 服务授予权限。

您可以将 `AWSLambda_FullAccess` 策略附加到用户、组和角色。

权限详细信息

该策略包含以下权限：

- `lambda`：允许主体完全访问 Lambda。
- `cloudformation`：允许主体描述 AWS CloudFormation 堆栈并列出这些堆栈中的资源。
- `cloudwatch`：允许主体列出 Amazon CloudWatch 指标并获取指标数据。
- `ec2`：允许主体描述安全组、子网和 VPC。
- `iam`：允许主体获取策略、策略版本、角色、角色策略、附加角色策略和角色列表。此策略还允许主体将角色传递给 Lambda。将执行角色分配给函数时，需要使用 `PassRole` 权限。
- `kms`：允许主体列出别名。
- `logs`：允许主体描述 Amazon CloudWatch Logs 日志组。对于与 Lambda 函数关联的日志组，此策略允许主体描述日志流、获取日志事件和筛选日志事件。
- `states`：允许主体描述和列出 AWS Step Functions 状态机。
- `tag`：允许主体根据其标签获取资源。
- `xray`：允许主体获取 AWS X-Ray 跟踪摘要并检索按 ID 指定的跟踪列表。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambda_FullAccess。

AWS 托管式策略：AWSLambda_ReadOnlyAccess

此策略授予对 Lambda 资源以及用于开发和维护 Lambda 资源的其他 AWS 服务的只读访问权限。

您可以将 `AWSLambda_ReadOnlyAccess` 策略附加到用户、组和角色。

权限详细信息

该策略包含以下权限：

- `lambda`：允许主体获取并列出所有资源。
- `cloudformation`：允许主体描述并列出 AWS CloudFormation 堆栈以及其中的资源。
- `cloudwatch`：允许主体列出 Amazon CloudWatch 指标并获取指标数据。
- `ec2`：允许主体描述安全组、子网和 VPC。
- `iam`：允许主体获取策略、策略版本、角色、角色策略、附加角色策略和角色列表。
- `kms`：允许主体列出别名。
- `logs`：允许主体描述 Amazon CloudWatch Logs 日志组。对于与 Lambda 函数关联的日志组，此策略允许主体描述日志流、获取日志事件和筛选日志事件。
- `states`：允许主体描述和列出 AWS Step Functions 状态机。
- `tag`：允许主体根据其标签获取资源。
- `xray`：允许主体获取 AWS X-Ray 跟踪摘要并检索按 ID 指定的跟踪列表。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambda_ReadOnlyAccess。

AWS 托管式策略：AWSLambdaBasicExecutionRole

此策略授予将日志上传到 CloudWatch Logs 的权限。

您可以将 `AWSLambdaBasicExecutionRole` 策略附加到用户、组和角色。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambdaBasicExecutionRole。

AWS 托管式策略：AWSLambdaDynamoDBExecutionRole

此策略授予读取 Amazon DynamoDB Streams 中的记录并写入 CloudWatch Logs 的权限。

您可以将 `AWSLambdaDynamoDBExecutionRole` 策略附加到用户、组和角色。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambdaDynamoDBExecutionRole。

AWS 托管式策略：AWSLambdaENIManagementAccess

此策略授予创建、描述和删除启用 VPC 的 Lambda 函数所使用的弹性网络接口的权限。

您可以将 `AWSLambdaENIManagementAccess` 策略附加到用户、组和角色。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambdaENIManagementAccess。

AWS 托管式策略：AWSLambdaExecute

此策略授予对 Amazon Simple Storage Service 的 `PUT` 和 `GET` 访问权限，以及对 CloudWatch Logs 的完全访问权限。

您可以将 `AWSLambdaExecute` 策略附加到用户、组和角色。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambdaExecute。

AWS 托管式策略：AWSLambdaInvocation-DynamoDB

此策略授予对 Amazon DynamoDB Streams 的读取权限。

您可以将 `AWSLambdaInvocation-DynamoDB` 策略附加到用户、组和角色。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambdaInvocation-DynamoDB。

AWS 托管式策略：AWSLambdaKinesisExecutionRole

此策略授予读取 Amazon Kinesis Data Streams 中的事件和写入 CloudWatch Logs 的权限。

您可以将 `AWSLambdaKinesisExecutionRole` 策略附加到用户、组和角色。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambdaKinesisExecutionRole。

AWS 托管式策略：AWSLambdaMSKExecutionRole

此策略授予读取和访问 Amazon Managed Streaming for Apache Kafka 集群中的记录、管理弹性网络接口及写入 CloudWatch Logs 的权限。

您可以将 AWSLambdaMSKExecutionRole 策略附加到用户、组和角色。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambdaMSKExecutionRole。

AWS 托管式策略：AWSLambdaRole

此策略授予调用 Lambda 函数的权限。

您可以将 AWSLambdaRole 策略附加到用户、组和角色。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambdaRole。

AWS 托管式策略：AWSLambdaSQSQueueExecutionRole

此策略授予读取和删除 Amazon Simple Queue Service 队列中的消息的权限，以及写入 CloudWatch Logs 的权限。

您可以将 AWSLambdaSQSQueueExecutionRole 策略附加到用户、组和角色。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambdaSQSQueueExecutionRole。

AWS 托管式策略：AWSLambdaVPCAccessExecutionRole

此策略授予管理 Amazon Virtual Private Cloud 中的弹性网络接口和写入 CloudWatch 日志的权限。

您可以将 AWSLambdaVPCAccessExecutionRole 策略附加到用户、组和角色。

有关此策略的更多信息，包括 JSON 策略文档和策略版本，请参阅《AWS Managed Policy Reference Guide》中的 AWSLambdaVPCAccessExecutionRole。

Lambda 对 AWS 托管式策略的更新

更改	描述	日期
AWSLambdaVPCAccessExecutionRole – 变更	Lambda 更新了 AWSLambdaVPCAccessExecutionRole 策略以允许操作 ec2:DescribeSubnets 。	2024 年 1 月 5 日

更改	描述	日期
AWSLambda_ReadOnlyAccess：变更	Lambda 更新了 AWSLambda_ReadOnlyAccess 策略，以允许主体列出 AWS CloudFormation 堆栈。	2023 年 7 月 27 日
AWS Lambda 开启了跟踪更改	AWS Lambda 为其 AWS 托管式策略开启了跟踪更改。	2023 年 7 月 27 日

对 AWS Lambda 身份和访问进行故障排除

可以使用以下信息，以帮助您诊断和修复在使用 Lambda 和 IAM 时可能遇到的常见问题。

主题

- 我无权在 Lambda 中执行操作
- 我无权执行 iam:PassRole
- 我希望允许我 AWS 账户 以外的人访问 Lambda 资源

我无权在 Lambda 中执行操作

如果您收到错误提示，表明您无权执行某个操作，则您必须更新策略以允许执行该操作。

当 mateojackson IAM 用户尝试使用控制台查看有关虚构 *my-example-widget* 资源的详细信息，但不拥有虚构 lambda:*GetWidget* 权限时，会发生以下示例错误。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lambda:GetWidget on resource: my-example-widget
```

在此情况下，必须更新 mateojackson 用户的策略，以允许使用 lambda:*GetWidget* 操作访问 *my-example-widget* 资源。

如果您需要帮助，请联系 AWS 管理员。您的管理员是提供登录凭证的人。

我无权执行 iam:PassRole

如果收到错误，则表明您无权执行 iam:PassRole 操作，则必须更新策略以允许您将角色传递给 Lambda。

有些 AWS 服务 允许将现有角色传递到该服务，而不是创建新服务角色或服务相关角色。为此，您必须具有将角色传递到服务的权限。

当名为 marymajor 的 IAM 用户尝试使用控制台在 Lambda 中执行操作时，会发生以下示例错误。但是，服务必须具有服务角色所授予的权限才可执行此操作。Mary 不具有将角色传递到服务的权限。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

在这种情况下，必须更新 Mary 的策略以允许她执行 iam:PassRole 操作。

如果您需要帮助，请联系 AWS 管理员。您的管理员是提供登录凭证的人。

我希望允许我 AWS 账户 以外的人访问 Lambda 资源

您可以创建一个角色，以便其他账户中的用户或您组织外的人员可以使用该角色来访问您的资源。您可以指定谁值得信赖，可以担任角色。对于支持基于资源的策略或访问控制列表（ACL）的服务，您可以使用这些策略向人员授予对您的资源的访问权。

要了解更多信息，请参阅以下内容：

- 要了解 Lambda 是否支持这些功能，请参阅 AWS Lambda 如何与 IAM 协同工作。
- 要了解如何为您拥有的 AWS 账户中的资源提供访问权限，请参阅《IAM 用户指南》中的为您拥有的另一个 AWS 账户中的 IAM 用户提供访问权限。
- 要了解如何为第三方 AWS 账户 提供您的资源的访问权限，请参阅 IAM 用户指南中的为第三方拥有的 AWS 账户 提供访问权限。
- 要了解如何通过联合身份验证提供访问权限，请参阅《IAM 用户指南》中的为经过外部身份验证的用户（联合身份验证）提供访问权限。
- 要了解使用角色和基于资源的策略进行跨账户访问之间的差别，请参阅《IAM 用户指南》中的 IAM 中的跨账户资源访问。

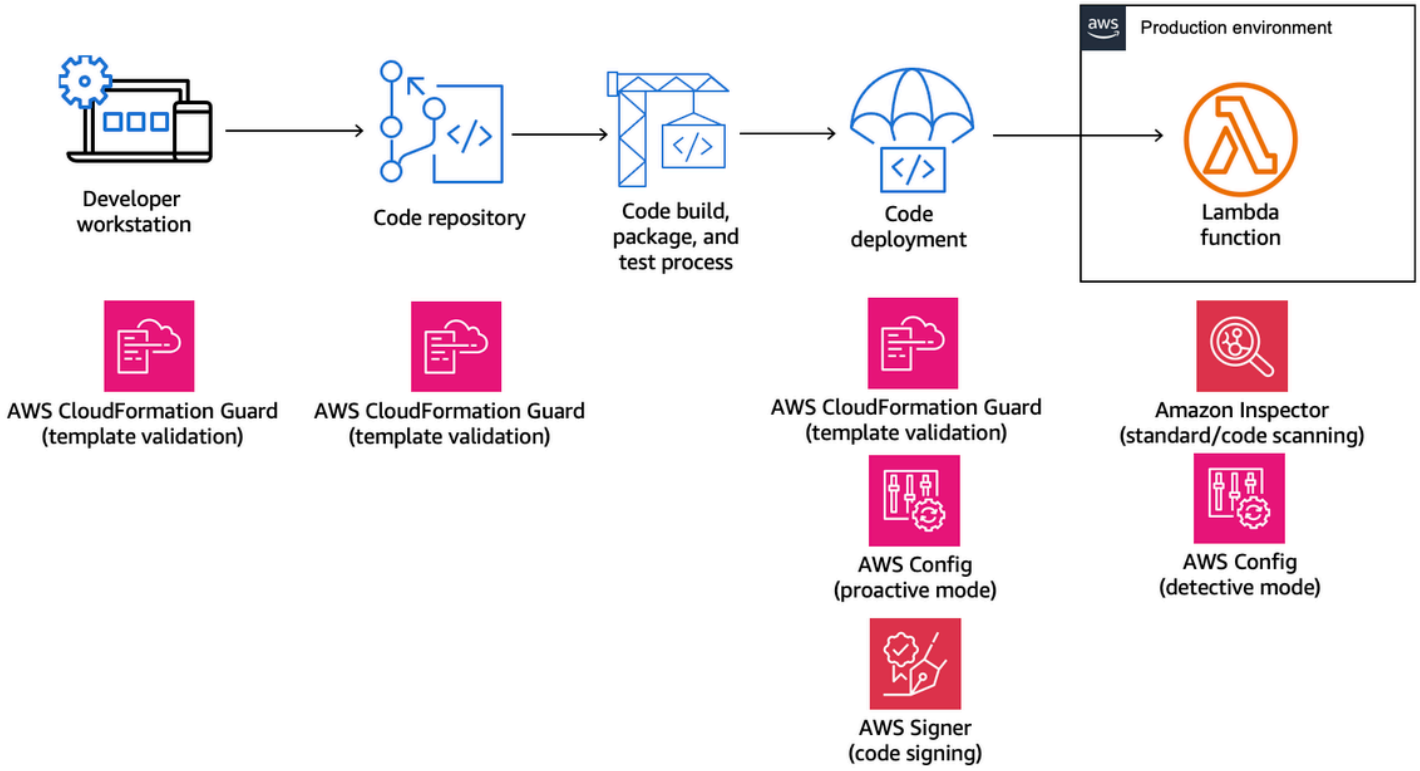
为 Lambda 函数和层创建治理策略

要构建和部署无服务器、云原生应用程序，您必须通过适当的治理和防护机制来实现敏捷性和上市速度。设定业务层面的优先事项，可以强调敏捷性是重中之重，也可以通过治理、防护机制和控制来强调风险规避。实际上，您不会有“非此即彼”的策略，而是使用“和”策略，用于在软件开发生命周期中平衡敏捷性和防护机制。无论这些要求属于贵公司生命周期的哪个阶段，治理能力都可能成为流程和工具链中的一项实施要求。

以下是组织可能为 Lambda 实施的治理控制的几个示例：

- Lambda 函数不可公开访问。
- Lambda 函数必须连接到 VPC。
- Lambda 函数不应使用已弃用的运行时系统。
- Lambda 函数必须使用一组必需的标签进行标记。
- 组织外部不得访问 Lambda 层。
- 带有附加安全组的 Lambda 函数在函数和安全组之间必须具有匹配的标签。
- 带有附加层的 Lambda 函数必须使用经批准的版本
- Lambda 环境变量必须使用客户托管式密钥进行静态加密。

下图是在整个软件开发和部署过程中实施控制和策略的深入治理策略的示例：



以下主题介绍了如何在您的组织中实施开发和部署 Lambda 功能的控制措施，包括针对初创企业和企业的控制措施。您的组织可能已具备相关工具。以下主题采用模块化的方式介绍这些控件，以便您挑选实际需要的组件。

主题

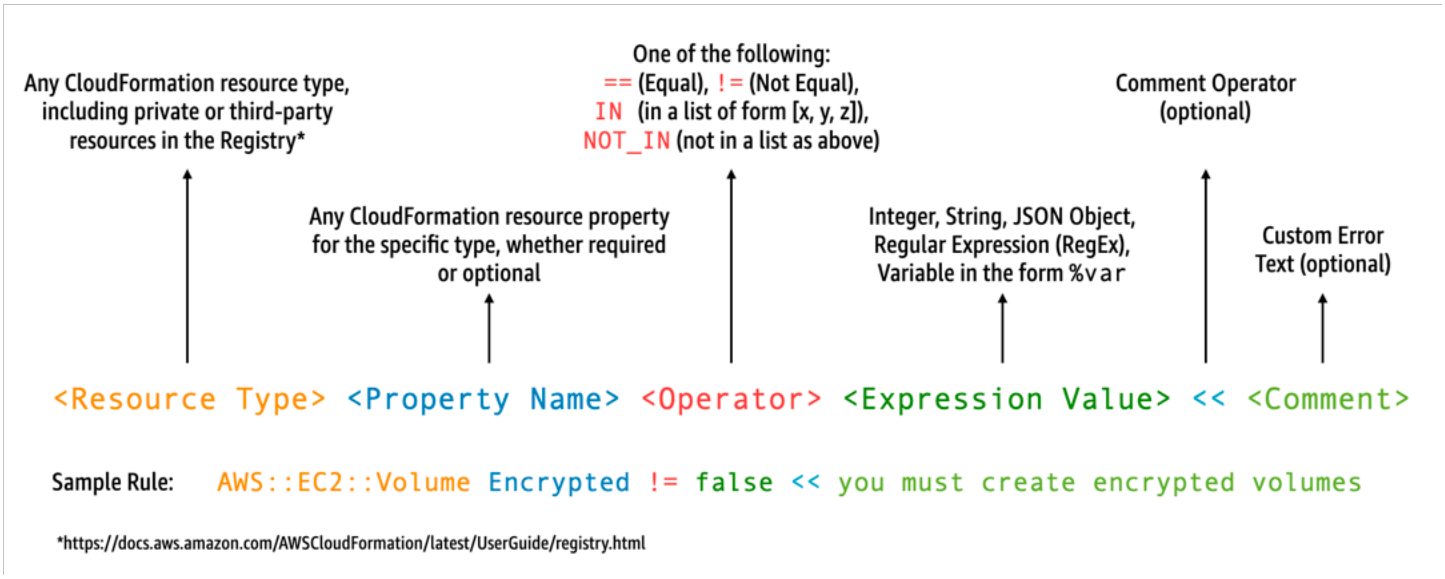
- 使用 AWS CloudFormation Guard 主动控制 Lambda
- 使用 AWS Config 对 Lambda 实施预防性控制
- 使用 AWS Config 检测不合规的 Lambda 部署和配置
- 使用 AWS Signer 的 Lambda 代码签名
- 使用 Amazon Inspector 自动执行 Lambda 的安全评测
- 实施可观测性以实现 Lambda 安全性和合规性

使用 AWS CloudFormation Guard 主动控制 Lambda

AWS CloudFormation Guard 是一款开源、通用、策略即代码评估工具。通过根据策略规则验证基础设施即代码（IaC）模板和服务组合，可用于预防性治理和合规性。这些规则可以根据您的团队或组织的要求进行自定义。对于 Lambda 函数，Guard 规则可用于通过定义创建或更新 Lambda 函数时所需的属性设置来控制资源创建和配置更新。

合规性管理员定义部署和更新 Lambda 函数所需的控制和治理策略列表。平台管理员在 CI/CD 管道中实施控制，将其作为带有代码存储库的预提交验证 Webhook，并为开发人员提供命令行工具，以便在本地工作站上验证模板和代码。开发人员编写代码，使用命令行工具验证模板，然后将代码提交到存储库，在部署到 AWS 环境中之前，会自动通过 CI/CD 管道验证存储库。

Guard 允许您使用特定域的语言编写规则和实施控制，如下所示。



例如，假设要确保开发人员只选择最新的运行时系统。可以指定两种不同的策略，一种用于识别已弃用的运行时系统，另一种用于识别即将弃用的运行时系统。为此，可以编写以下 `etc/rules.guard` 文件：

```
let lambda_functions = Resources.*[
    Type == "AWS::Lambda::Function"
]

rule lambda_already_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
```

```
                Runtime !in ["dotnetcore3.1", "nodejs12.x", "python3.6", "python2.7",
 "dotnet5.0", "dotnetcore2.1", "ruby2.5", "nodejs10.x", "nodejs8.10", "nodejs4.3",
 "nodejs6.10", "dotnetcore1.0", "dotnetcore2.0", "nodejs4.3-edge", "nodejs"] <<Lambda
 function is using a deprecated runtime.>>
            }
        }
    }
}

rule lambda_soon_to_be_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["nodejs16.x", "nodejs14.x", "python3.7", "java8",
 "dotnet7", "go1.x", "ruby2.7", "provided"] <<Lambda function is using a runtime that
 is targeted for deprecation.>>
            }
        }
    }
}
```

现在，假设您编写了以下定义 Lambda 函数的 `iac/lambda.yaml` CloudFormation 模板：

```
  Fn:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.7
      CodeUri: src
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      Layers:
        - arn:aws:lambda:us-east-1:111122223333:layer:LambdaInsightsExtension:35
```

安装 Guard 实用程序后，请验证您的模板：

```
cfn-guard validate --rules etc/rules.guard --data iac/lambda.yaml
```

该输出类似于以下示例：

```
lambda.yaml Status = FAIL
FAILED rules
rules.guard/lambda_soon_to_be_deprecated_runtime
```

```
---
Evaluating data lambda.yaml against rules rules.guard
Number of non-compliant resources 1
Resource = Fn {
  Type      = AWS::Lambda::Function
  Rule = lambda_soon_to_be_deprecated_runtime {
    ALL {
      Check =  Runtime not IN
 ["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]
 {
        ComparisonError {
          Message          = Lambda function is using a runtime that is targeted for
 deprecation.
          Error            = Check was not compliant as property [/Resources/
Fn/Properties/Runtime[L:88,C:15]] was not present in [(resolved, Path=[L:0,C:0]
 Value=["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"])]
        }
          PropertyPath    = /Resources/Fn/Properties/Runtime[L:88,C:15]
          Operator        = NOT IN
          Value           = "python3.7"
          ComparedWith    =
 [["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]]
          Code:
               86.  Fn:
               87.    Type: AWS::Lambda::Function
               88.    Properties:
               89.      Runtime: python3.7
               90.      CodeUri: src
               91.      Handler: fn.handler

      }
    }
  }
}
```

Guard 使开发人员从本地开发人员工作站看到他们需要更新模板才能使用组织允许的运行时系统。这种情况发生在提交到代码存储库以及随后在 CI/CD 管道中检查失败之前。这样一来，开发人员会收到有关如何开发合规模板，以及如何将时间转向编写可带来商业价值的代码的反馈。此控件可以应用于本地开发人员工作站、预提交验证 Webhook 和/或部署前的 CI/CD 管道中。

警告

如果您使用 AWS Serverless Application Model（AWS SAM）模板来定义 Lambda 函数，请注意，您需要更新 Guard 规则以搜索如下所示的 `AWS::Serverless::Function` 资源类型。

```
let lambda_functions = Resources.*[
    Type == "AWS::Serverless::Function"
]
```

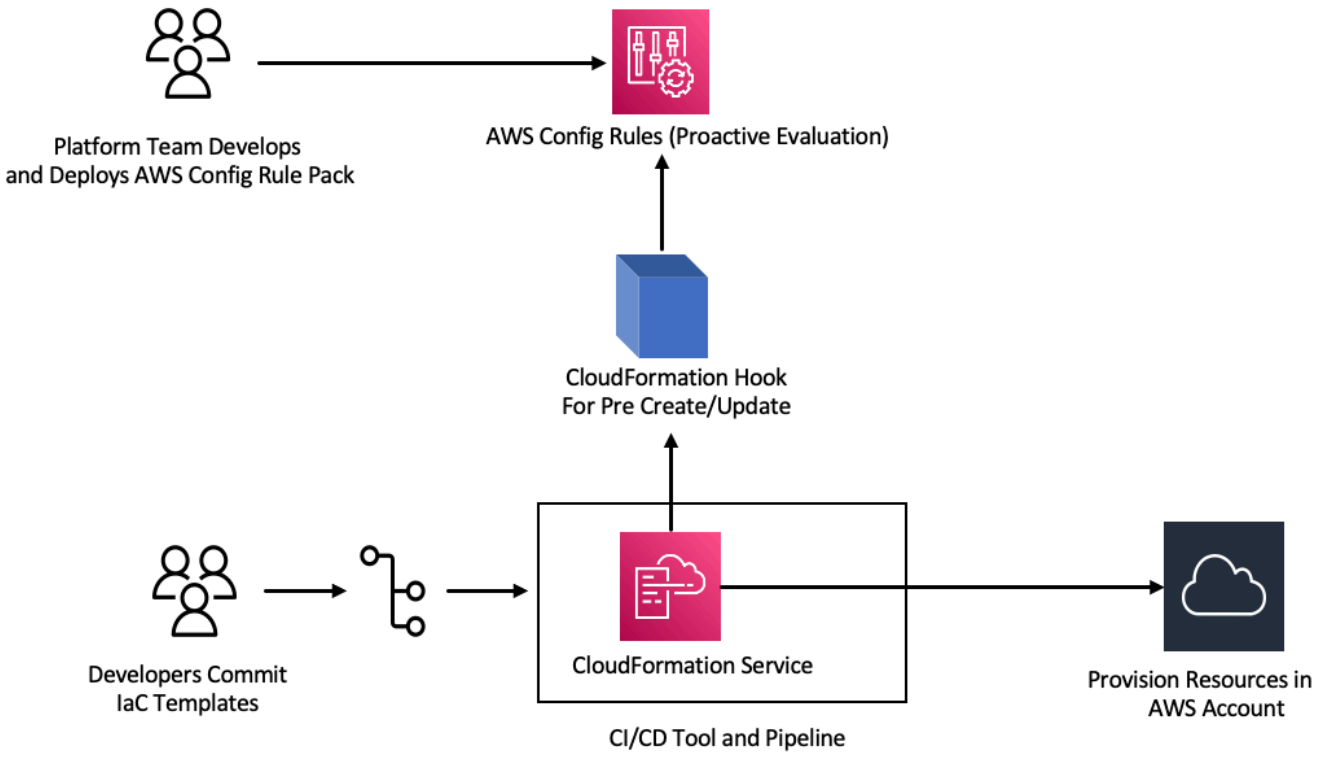
Guard 还希望这些属性包含在资源定义中。同时，AWS SAM 模板使单独的 Globals 部分中可以指定属性。Globals 部分中定义的属性不会通过您的 Guard 规则进行验证。

正如 Guard 故障排除文档中所述，请注意，Guard 不支持 `!GetAtt` 或 `!Sub` 之类的短格式内置函数，而是要求使用扩展形式 `Fn::GetAtt` 和 `Fn::Sub`。（前面的示例没有评估 Role 属性，因此为了简单起见，使用了短格式内置函数。）

使用 AWS Config 对 Lambda 实施预防性控制

在开发过程中尽早确保无服务器应用程序的合规性至关重要。在本主题中，我们将介绍如何使用 AWS Config 实现预防性控制。这样，您就能够在开发过程的早期阶段实施合规性检查，并在 CI/CD 管道中实施相同的控制。这还可以在集中管理的规则存储库中实现标准控制化，以便在 AWS 账户中一致地应用控制。

例如，假设您的合规性管理员定义了一项要求，用于确保所有 Lambda 函数都包含 AWS X-Ray 跟踪。借助 AWS Config 的主动模式，您可以在部署前对 Lambda 函数资源进行合规性检查，从而降低部署配置不当的 Lambda 函数的风险，并通过向开发人员就基础设施即代码模板提供更快的反馈来节省其时间。以下是使用 AWS Config 进行预防性控制的可视化流程：



考虑要求所有 Lambda 函数都必须启用跟踪。对此，平台团队确定需要在所有账户中主动运行的特定 AWS Config 规则。此规则将缺乏已配置的 X-Ray 跟踪配置的任何 Lambda 函数标记为不合规资源。该团队制定规则，将其打包到一致性包中，然后在所有 AWS 账户中部署一致性包，以确保组织中的所有账户统一应用这些控制措施。您可以用 AWS CloudFormation Guard 2.x.x 语法编写规则，其形式如下：

```
rule name when condition { assertion }
```

以下是一个 Guard 规则示例，用于检查以确保 Lambda 函数已启用跟踪：

```
rule lambda_tracing_check {
  when configuration.tracingConfig exists {
      configuration.tracingConfig.mode == "Active"
  }
}
```

平台团队采取进一步行动，要求每个 AWS CloudFormation 部署必须调用预创建/更新钩子。他们承担开发此钩子和配置管道的全部责任，加强对合规性规则的集中控制，并在所有部署中保持应用程序的一致性。要开发、打包和注册钩子，请参阅 CloudFormation Command Line Interface（CFN-CLI）文档中的 Developing AWS CloudFormation Hooks。您可以使用 CloudFormation CLI 来创建钩子项目：

```
cfn init
```

此命令要求您提供有关钩子项目的一些基本信息，并创建一个包含以下文件的项目：

```
README.md
<hook-name>.json
rpdk.log
src/handler.py
template.yml
hook-role.yaml
```

作为钩子开发者，您需要在 <hook-name>.json 配置文件中添加所需的目标资源类型。在下面的配置中，将钩子配置为在使用 CloudFormation 创建任何 Lambda 函数之前执行。您也可以为 preUpdate 和 preDelete 操作添加类似的处理程序。

```
    "handlers": {
        "preCreate": {
            "targetNames": [
                "AWS::Lambda::Function"
            ],
            "permissions": []
        }
    }
```

您还需要确保 CloudFormation 钩子具有调用 AWS Config API 的相应权限。为此，您可以更新名为 hook-role.yaml 的角色定义文件。默认情况下，角色定义文件具有以下信任策略，该策略允许 CloudFormation 代入该角色。

```
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - hooks.cloudformation.amazonaws.com
                - resources.cloudformation.amazonaws.com
```

要允许此钩子调用配置 API，您必须在策略语句中添加以下权限。然后，使用 cfn submit 命令提交钩子项目，CloudFormation 会在其中为您创建一个具有所需权限的角色。

```
      Policies:
        - PolicyName: HookTypePolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - "config:Describe*"
                  - "config:Get*"
                  - "config:List*"
                  - "config:SelectResourceConfig"
                Resource: "*
```

接下来，您需要在 src/handler.py 文件中编写一个 Lambda 函数。在此文件中，您可以找到在启动项目时就已创建的名为 preCreate、preUpdate、和 preDelete 的方法。您的目标是编写一个通用、可重复使用的函数，该函数可以主动模式使用 AWS SDK for Python (Boto3) 调用 AWS Config StartResourceEvaluation API。此 API 调用将资源属性作为输入，并根据规则定义评估资源。

```
def validate_lambda_tracing_config(resource_type, function_properties:
 MutableMapping[str, Any]) -> ProgressEvent:
  LOG.info("Fetching proactive data")
  config_client = boto3.client('config')
  resource_specs = {
      'ResourceId': 'MyFunction',
      'ResourceType': resource_type,
```

```
      'ResourceConfiguration': json.dumps(function_properties),
      'ResourceConfigurationSchemaType': 'CFN_RESOURCE_SCHEMA'
  }
  LOG.info("Resource Specifications:", resource_specs)
  eval_response = config_client.start_resource_evaluation(EvaluationMode='PROACTIVE',
 ResourceDetails=resource_specs, EvaluationTimeout=60)
  ResourceEvaluationId = eval_response.ResourceEvaluationId
  compliance_response =
 config_client.get_compliance_details_by_resource(ResourceEvaluationId=ResourceEvaluationId)
  LOG.info("Compliance Verification:",
 compliance_response.EvaluationResults[0].ComplianceType)
  if "NON_COMPLIANT" == compliance_response.EvaluationResults[0].ComplianceType:
      return ProgressEvent(status=OperationStatus.FAILED, message="Lambda function
 found with no tracing enabled : FAILED", errorCode=HandlerErrorCode.NonCompliant)
  else:
      return ProgressEvent(status=OperationStatus.SUCCESS, message="Lambda function
 found with tracing enabled : PASS.")
```

现在，您可以从预创建钩子的处理程序中调用通用函数。下面是处理程序的示例：

```
@hook.handler(HookInvocationPoint.CREATE_PRE_PROVISION)
def pre_create_handler(
        session: Optional[SessionProxy],
        request: HookHandlerRequest,
        callback_context: MutableMapping[str, Any],
        type_configuration: TypeConfigurationModel
) -> ProgressEvent:
    LOG.info("Starting execution of the hook")
    target_name = request.hookContext.targetName
    LOG.info("Target Name:", target_name)
    if "AWS::Lambda::Function" == target_name:
        return validate_lambda_tracing_config(target_name,
            request.hookContext.targetModel.get("resourceProperties")
        )
    else:
        raise exceptions.InvalidRequest(f"Unknown target type: {target_name}")
```

完成此步骤后，您可以注册钩子并将其配置为侦听所有 AWS Lambda 函数创建事件。

开发人员使用 Lambda 为无服务器微服务准备基础设施即代码（IaC）模板。准备工作包括遵守内部标准，然后进行本地测试并将模板提交到存储库。IaC 模板示例如下：

```
  MyLambdaFunction:
```

```
  Type: 'AWS::Lambda::Function'
  Properties:
    Handler: index.handler
    Role: !GetAtt LambdaExecutionRole.Arn
    FunctionName: MyLambdaFunction
    Code:
      ZipFile: |
        import json

        def handler(event, context):
            return {
                'statusCode': 200,
                'body': json.dumps('Hello World!')
            }
    Runtime: python3.12
    TracingConfig:
        Mode: PassThrough
    MemorySize: 256
    Timeout: 10
```

作为 CI/CD 流程的一部分，在部署 CloudFormation 模板时，CloudFormation 服务会在预置 `AWS::Lambda::Function` 资源类型之前调用预创建/更新钩子。该钩子利用在主动模式下运行的 AWS Config 规则来验证 Lambda 函数配置是否包含规定的跟踪配置。来自钩子的响应决定了下一步行动。如果合规，则钩子会发出成功响应，CloudFormation 将继续预置资源。否则，CloudFormation 堆栈部署将失败，管道会立即停止，系统会记录详细信息以供后续审查。合规性通知将被发送到相关利益方。

您可以在 CloudFormation 控制台中找到钩子成功/失败信息：

Stack info | Events | Resources | Outputs | Parameters | Template | Change sets

Events (19)

Search events

Timestamp	Logical ID	Status	Status reason	Hook invocations
2023-08-29 23:50:23 UTC-0500	HookTestStack	ROLLBACK_COMPLETE	-	-
2023-08-29 23:50:22 UTC-0500	LambdaExecutionRole	DELETE_COMPLETE	-	-
2023-08-29 23:50:21 UTC-0500	MyApi	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	LambdaExecutionRole	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:20 UTC-0500	MyLambdaFunction	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	MyApi	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:18 UTC-0500	HookTestStack	ROLLBACK_IN_PROGRESS	The following resource(s) failed to create: [MyLambdaFunction]. Rollback requested by user.	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_FAILED	The following hook(s) failed: [AWSSamples::LambdaTracingCheck::Hook]	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:16 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:15 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	-
2023-08-29 23:50:14 UTC-0500	LambdaExecutionRole	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:58 UTC-0500	MyApi	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:55 UTC-0500	HookTestStack	CREATE_IN_PROGRESS	User Initiated	-
2023-08-29 23:49:50 UTC-0500	HookTestStack	REVIEW_IN_PROGRESS	User Initiated	-

如果您为 CloudFormation 钩子启用了日志，则可以捕获钩子评估结果。以下是状态为失败的钩子的示例日志，表明 Lambda 函数未启用 X-Ray：

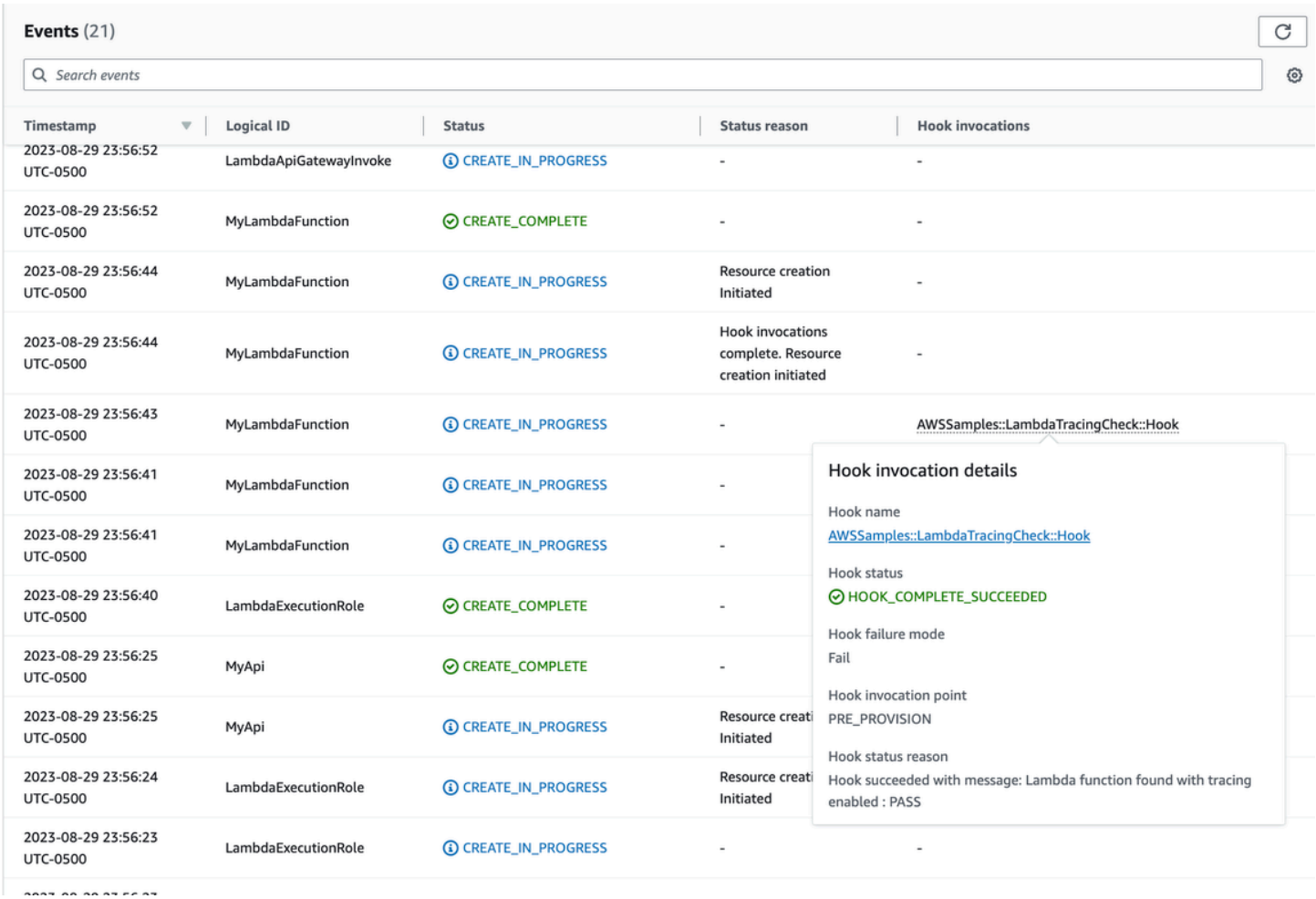
2023-08-29T23:50:17.574-05:00 ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'…

```
ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'>, message='Lambda
function found with no tracing enabled : FAILED', result=None, callbackContext=None, callbackDelaySeconds=0, resourceModel=None,
resourceModels=None, nextToken=None)
```

Copy

No newer events at this moment. *Auto retry paused.* Resume

如果开发者选择更改 IaC，将 `TracingConfig Mode` 值更新为 `Active` 并重新部署，则钩子将成功执行，堆栈会继续创建 Lambda 资源。



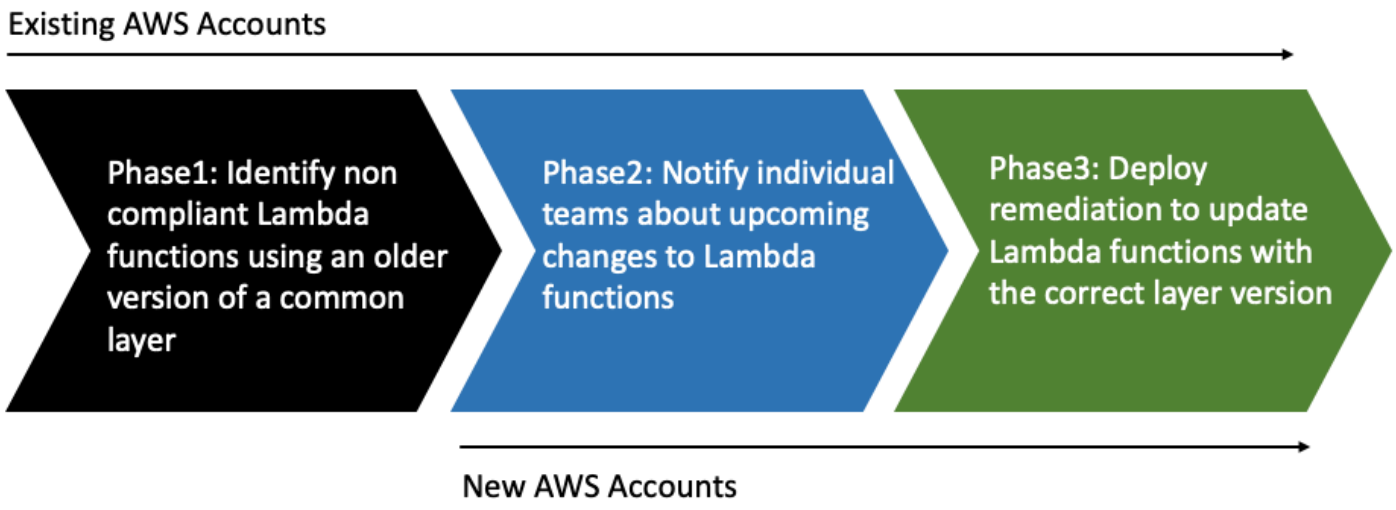
这样，在开发和部署 AWS 账户中的无服务器资源时，您就可以通过 AWS Config 以主动模式实施预防性控制。通过将 AWS Config 规则集成到 CI/CD 管道中，您可以识别并选择阻止不合规的资源部署，例如缺乏活动跟踪配置的 Lambda 函数。这样可以确保只有符合最新治理策略的资源才会部署到您的 AWS 环境中。

使用 AWS Config 检测不合规的 Lambda 部署和配置

除了主动性评估外，AWS Config 还可以被动检测不符合您的治理策略的资源部署和配置。被动侦测非常重要，因为治理策略会随着组织学习和实施新的最佳实践而不断演变。

假设您在部署或更新 Lambda 函数时设置了全新的策略：所有 Lambda 函数都必须始终使用特定的、经批准的 Lambda 层版本。您可以配置 AWS Config 来监控新函数或更新函数的层配置。如果 AWS Config 检测到某函数未使用已批准的层版本，则会将该函数标记为不合规资源。您可以选择配置 AWS Config，通过使用 AWS Systems Manager 自动化文档指定补救操作来自动修复资源。例如，您可以通过 AWS SDK for Python (Boto3) 使用 Python 编写自动化文档，将不合规函数更新为指向已批准的层版本。因此，AWS Config 既可以作为检测性控制，也可以作为纠正性控制，实现了合规性管理的自动化。

让我们将这个过程分解为三个重要的实施阶段：



第 1 阶段：确定访问资源

首先在账户中激活 AWS Config，然后将其配置为记录 AWS Lambda 函数。这样，AWS Config 就可以观测到 Lambda 函数的创建或更新时间。然后，您可以配置使用 AWS CloudFormation Guard 语法的自定义策略规则来检查是否存在特定的策略违规行为。Guard 规则的一般形式如下：

```
rule name when condition { assertion }
```

以下示例规则用于确保层未设置为旧版本：

```
rule desiredlayer when configuration.layers !empty {
```

```
    some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn
}
```

让我们来了解一下规则的语法和结构：

- 规则名称：所提供的示例中的规则名称为 `desiredlayer`。
- 条件：此子句指定了检查规则的条件。在所提供的示例中，条件为 `configuration.layers ! empty`。这意味着只有在配置中的 `layers` 属性不为空时，才会对资源进行评估。
- 断言：在 `when` 子句之后，断言决定了规则检查的内容。该断言 `some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn` 会检查是否有任何 Lambda 层 ARN 与 `OldLayerArn` 值不匹配。如果不匹配，则断言为真，规则通过；否则，规则失败。

`CONFIG_RULE_PARAMETERS` 是使用 AWS Config 规则配置的一组特殊参数。在本例中，`OldLayerArn` 是 `CONFIG_RULE_PARAMETERS` 中的一个参数。这样，用户可以提供其认为已过时或已弃用的特定 ARN 值，然后该规则就会检查是否有任何 Lambda 函数正在使用此旧 ARN。

第 2 阶段：可视化和设计

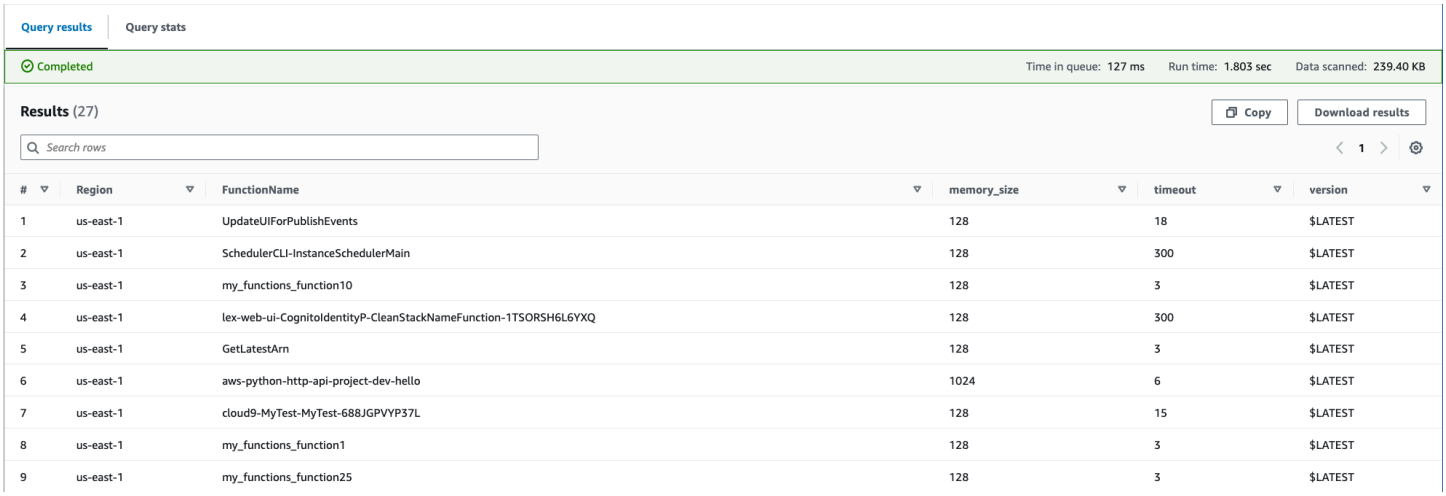
AWS Config 收集配置数据并将这些数据存储在 Amazon Simple Storage Service（Amazon S3）存储桶中。您可以使用 Amazon Athena 直接从 S3 存储桶中查询这些数据。通过 Athena，您可以在组织层面聚合这些数据，从而生成所有账户中资源配置的整体视图。要设置资源配置数据的聚合，请参阅 AWS Cloud Operations and Management Blog 上的 Visualizing AWS Config data using Athena and Amazon QuickSight。

以下是一个 Athena 查询示例，用于确定使用特定层 ARN 的所有 Lambda 函数：

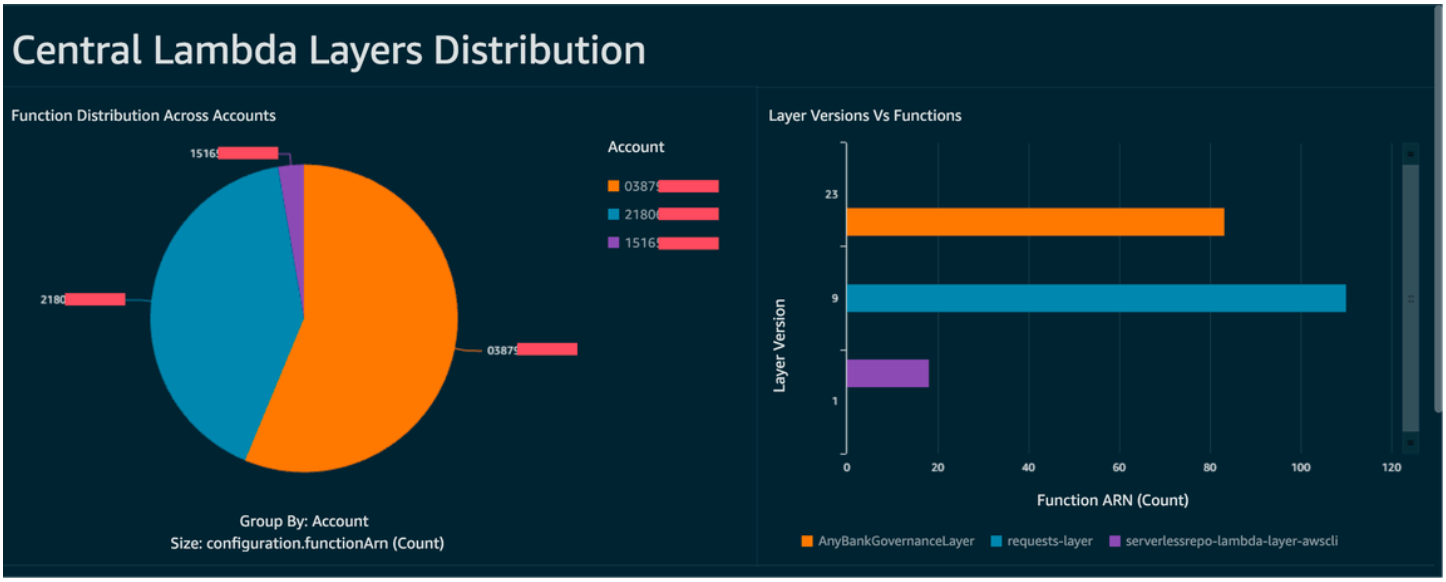
```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
```

```
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.version') AS version
  FROM
    unnested
  WHERE
    lambda_configuration LIKE '%arn:aws:lambda:us-
east-1:111122223333:layer:AnyGovernanceLayer:24%'
```

以下是查询的结果：



聚合整个组织的 AWS Config 数据后，就可以使用 Amazon QuickSight 创建控制面板。通过将 Athena 结果导入 Amazon QuickSight，您可以直观地了解 Lambda 函数在多大程度上遵守了层版本规则。此控制面板可以突出显示合规资源和不合规资源，从而有助于您确定执行策略，如下一节所述。下图是一个示例控制面板，用于报告应用于组织内函数的层版本的分布情况。



第 3 阶段：实施和强制执行

现在，您可以选择通过 Systems Manager 自动化文档将您在第 1 阶段创建的层版本规则与补救操作配对，该文档是您使用 AWS SDK for Python (Boto3) 编写的 Python 脚本。该脚本将调用每个 Lambda 函数的 UpdateFunctionConfiguration API 操作，从而使用新的层 ARN 更新函数配置。您也可以使用脚本向代码存储库提交拉取请求以更新层 ARN。这样，未来的代码部署也将使用正确的层 ARN 进行更新。

使用 AWS Signer 的 Lambda 代码签名

AWS Signer 是一项完全托管的代码签名服务，让您可以根据数字签名验证代码，从而确认代码未被更改且来自可信的发布者。AWS Signer 可以与 AWS Lambda 结合使用，用于验证函数和层在部署到您的 AWS 环境之前未被更改。这可以保护您的组织免受恶意行为者的侵害，这些恶意行为者可能已获得创建新函数或更新现有函数的凭证。

要为 Lambda 函数设置代码签名，请先创建一个启用了版本控制的 S3 存储桶。之后，使用 AWS Signer 创建签名配置文件，将 Lambda 指定为平台，然后指定签名配置文件的有效期限。例如：

```
Signer:
  Type: AWS::Signer::SigningProfile
  Properties:
    PlatformId: AWSLambda-SHA384-ECDSA
    SignatureValidityPeriod:
      Type: DAYS
      Value: !Ref pValidDays
```

然后使用签名配置文件，并通过 Lambda 创建签名配置。当签名配置看到与预期数字签名不匹配的构件时，必须指定要采取的措施：警告（但允许部署）或强制执行（并阻止部署）。以下示例配置为强制执行并阻止部署。

```
SigningConfig:
  Type: AWS::Lambda::CodeSigningConfig
  Properties:
    AllowedPublishers:
      SigningProfileVersionArns:
        - !GetAtt Signer.ProfileVersionArn
    CodeSigningPolicies:
      UntrustedArtifactOnDeployment: Enforce
```

现在，您已将 AWS Signer 与 Lambda 一同配置，以阻止不受信任的部署。假设您已完成功能请求的编码，现在准备好部署该函数。第一步是使用相应的依赖项压缩代码，然后使用您创建的签名配置文件对构件进行签名。为此，您可以将 zip 构件上传到 S3 存储桶，然后启动签名作业。

```
aws signer start-signing-job \
--source 's3={bucketName=your-versioned-bucket,key=your-prefix/your-zip-
artifact.zip,version=QyaJ3c4qa50LXV.9VaZgXHlsGbvCXxpT}' \
--destination 's3={bucketName=your-versioned-bucket,prefix=your-prefix/}' \
--profile-name your-signer-id
```

您将获得如下输出，其中 jobId 是在目标存储桶和前缀中创建的对象，jobOwner 是运行作业的 12 位 AWS 账户 ID。

```
{
    "jobId": "87a3522b-5c0b-4d7d-b4e0-4255a8e05388",
    "jobOwner": "111122223333"
  }
```

现在，您可以使用已签名的 S3 对象和创建的代码签名配置来部署您的函数。

```
  Fn:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://your-versioned-bucket/your-prefix/87a3522b-5c0b-4d7d-
b4e0-4255a8e05388.zip
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      CodeSigningConfigArn: !Ref pSigningConfigArn
```

或者，您也可以使用原始的未签名源 zip 构件来测试函数部署。部署失败时会显示如下消息：。

```
Lambda cannot deploy the function. The function or layer might be signed using a
 signature that the client is not configured to accept. Check the provided signature
 for unsigned.
```

如果您使用 AWS Serverless Application Model（AWS SAM）构建和部署函数，程序包命令会处理将 zip 构件上传到 S3 的作业，还会启动签名作业并获取已签名的构件。您可以使用参数和命令执行该操作：

```
sam package -t your-template.yaml \
--output-template-file your-output.yaml \
--s3-bucket your-versioned-bucket \
--s3-prefix your-prefix \
--signing-profiles your-signer-id
```

AWS Signer 可帮助您验证部署到账户中的 zip 构件是否值得信任，可用于部署。您可以将上述过程包含在 CI/CD 管道中，并要求所有函数都附加使用前面主题中概述的技术的代码签名配置。通过在 Lambda 函数部署中使用代码签名，可以防止恶意行为者在获得创建或更新函数的凭证后在函数中注入恶意代码。

使用 Amazon Inspector 自动执行 Lambda 的安全评测

Amazon Inspector 是一项漏洞管理服务，可持续扫描您的工作负载，查找已知的软件漏洞和意外的网络暴露。Amazon Inspector 会创建调查发现，该调查发现将描述漏洞，确定受影响的资源，对漏洞的严重性进行评级，并提供补救指导。

Amazon Inspector 支持可为 Lambda 函数和层提供持续、自动的安全漏洞评测。Amazon Inspector 提供两种 Lambda 扫描类型：

- Lambda 标准扫描（默认）：扫描 Lambda 函数及其层中的应用程序依赖项，以便查找程序包漏洞。
- Lambda 代码扫描：扫描函数中及其层中的自定义应用程序代码，以便查找代码漏洞。您可以单独激活 Lambda 标准扫描，也可以同时激活 Lambda 标准扫描和 Lambda 代码扫描。

要启用 Amazon Inspector，请导航到 Amazon Inspector 控制台，展开设置部分，然后选择账户管理。在账户选项卡上，选择激活，然后选择其中一个扫描选项。

在设置 Amazon Inspector 时，您可以为多个账户启用 Amazon Inspector，并将该组织的 Amazon Inspector 管理权限委托给特定账户。启用时，您需要通过创建角色 `AWSServiceRoleForAmazonInspector2` 来授予 Amazon Inspector 权限。您可以通过 Amazon Inspector 控制台使用一键式选项创建此角色。

对于 Lambda 标准扫描，Amazon Inspector 会在以下情况下对 Lambda 函数启动漏洞扫描：

- Amazon Inspector 发现现有 Lambda 函数时。
- 部署新的 Lambda 函数时。
- 部署现有 Lambda 函数或其层的应用程序代码或依赖项更新时。
- Amazon Inspector 在其数据库中添加新的常见脆弱性和风险 (CVE) 项目，且该 CVE 与您的函数相关时。

对于 Lambda 代码扫描，Amazon Inspector 会使用自动推理和机器学习来评估 Lambda 函数应用程序代码，分析应用程序代码的总体安全合规性。如果 Amazon Inspector 在 Lambda 函数应用程序代码中检测到漏洞，Amazon Inspector 会生成详细的代码漏洞的调查发现。有关可能的检测的列表，请参阅 Amazon CodeGuru Detector Library。

要查看调查发现，请前往 Amazon Inspector 控制台。在调查发现菜单上，选择按 Lambda 函数，以显示对 Lambda 函数执行的安全扫描结果。

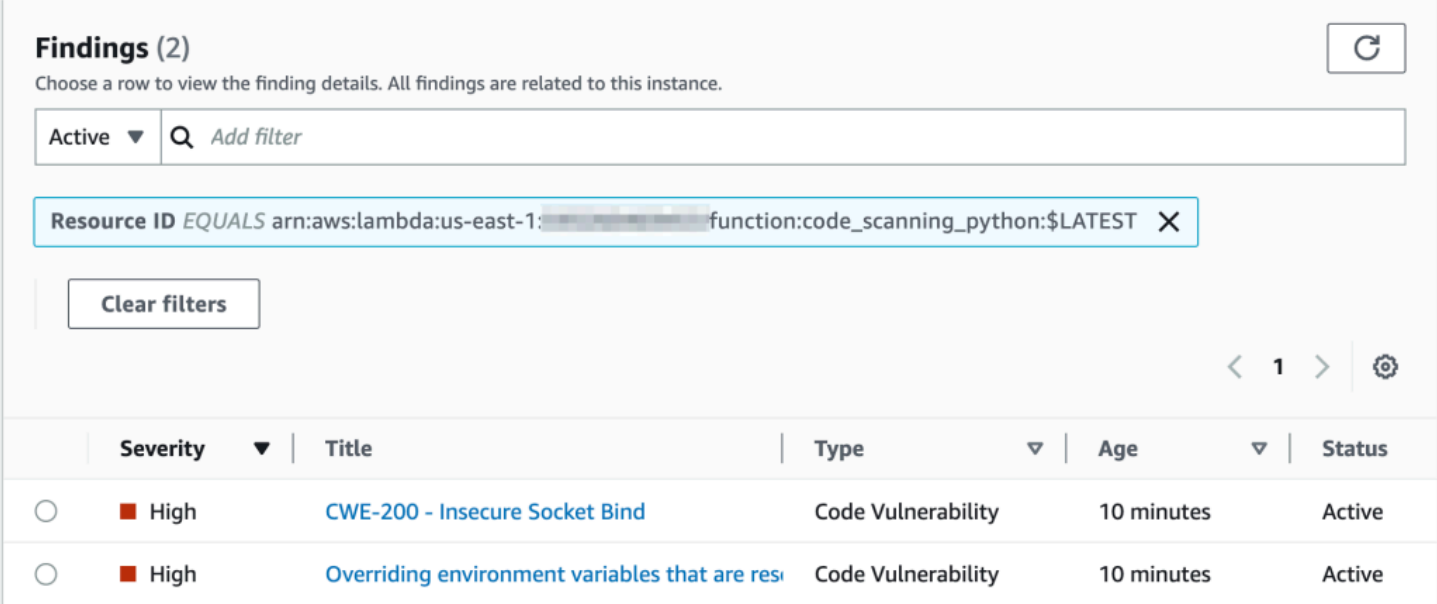
要从标准扫描中排除 Lambda 函数，请使用以下键值对标记函数：

- `Key:InspectorExclusion`
- `Value:LambdaStandardScanning`

要从代码扫描中排除 Lambda 函数，请使用以下键值对标记函数：

- `Key:InspectorCodeExclusion`
- `Value:LambdaCodeScanning`

例如，如下图所示，Amazon Inspector 会自动检测漏洞并将发现的漏洞归类为代码漏洞类型，这表明漏洞存在于函数的代码中，而不是存在于某个依赖于代码的库中。您可以同时查看特定函数或多个函数的这些详细信息。



您可以进一步深入了解这些调查发现，并学习如何补救问题。

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior.

Finding ID: arn:aws:inspector2:us-east-1:[redacted]:finding/[redacted]

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior or failure of the Lambda function.

Finding overview

AWS account ID	[redacted]
Severity	High
Type	Code Vulnerability
Detector name	Override of reserved variable names in a Lambda function
Relevant CWE	--
Rule ID	Rule-434311
Detector tags	#availability, #aws-python-sdk, #aws-lambda, #data-integrity, #maintainability, #security, #security-context, #python
Fix available	Yes
Created at	March 29, 2023 10:08 AM (UTC-04:00)

Vulnerability details

File path	lambda_function.py

Vulnerability location

```
3  import socket
4
5  def lambda_handler(event, context):
6
7      # print("Scenario 1");
8      os.environ['_HANDLER'] = 'hello'
9      # print("Scenario 1 ends")
10
11     # print("Scenario 2");
12     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     s.bind(('',0))
```

Suggested remediation

Your code attempts to override an environment variable that is reserved by the Lambda runtime environment. This can lead to unexpected behavior and might break the execution of your Lambda function.

在使用 Lambda 函数时，请确保遵守 Lambda 函数的命名约定。有关更多信息，请参阅 使用 Lambda 环境变量配置代码中的值。

您应对自己接受的补救建议负责。在接受补救建议之前，请务必仔细审视这些建议。您可能需要对补救建议进行编辑，以确保代码符合您的预期。

实施可观测性以实现 Lambda 安全性和合规性

AWS Config 是查找和修复不合规的 AWS 无服务器资源的有用工具。您对无服务器资源所做的每一次更改都会记录在 AWS Config 中。此外，AWS Config 还允许您将配置快照数据存储在 S3 上。您可以使用 Amazon Athena 和 Amazon QuickSight 制作控制面板并查看 AWS Config 数据。在 使用 AWS Config 检测不合规的 Lambda 部署和配置 中，我们讨论了如何可视化特定配置（如 Lambda 层）。本主题将扩展这些概念。

Lambda 配置的可见性

您可以使用查询来拉取重要的配置，例如 AWS 账户 ID、区域、AWS X-Ray 跟踪配置、VPC 配置、内存大小、运行时系统和标签。您可以通过下面的查询示例来从 Athena 中拉取这些信息：

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    account_id,
    tags,
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.runtime') AS version
    json_extract_scalar(lambda_json, '$.vpcConfig.SubnetIds') AS vpcConfig
    json_extract_scalar(lambda_json, '$.tracingConfig.mode') AS tracingConfig
  FROM
    unnested
```

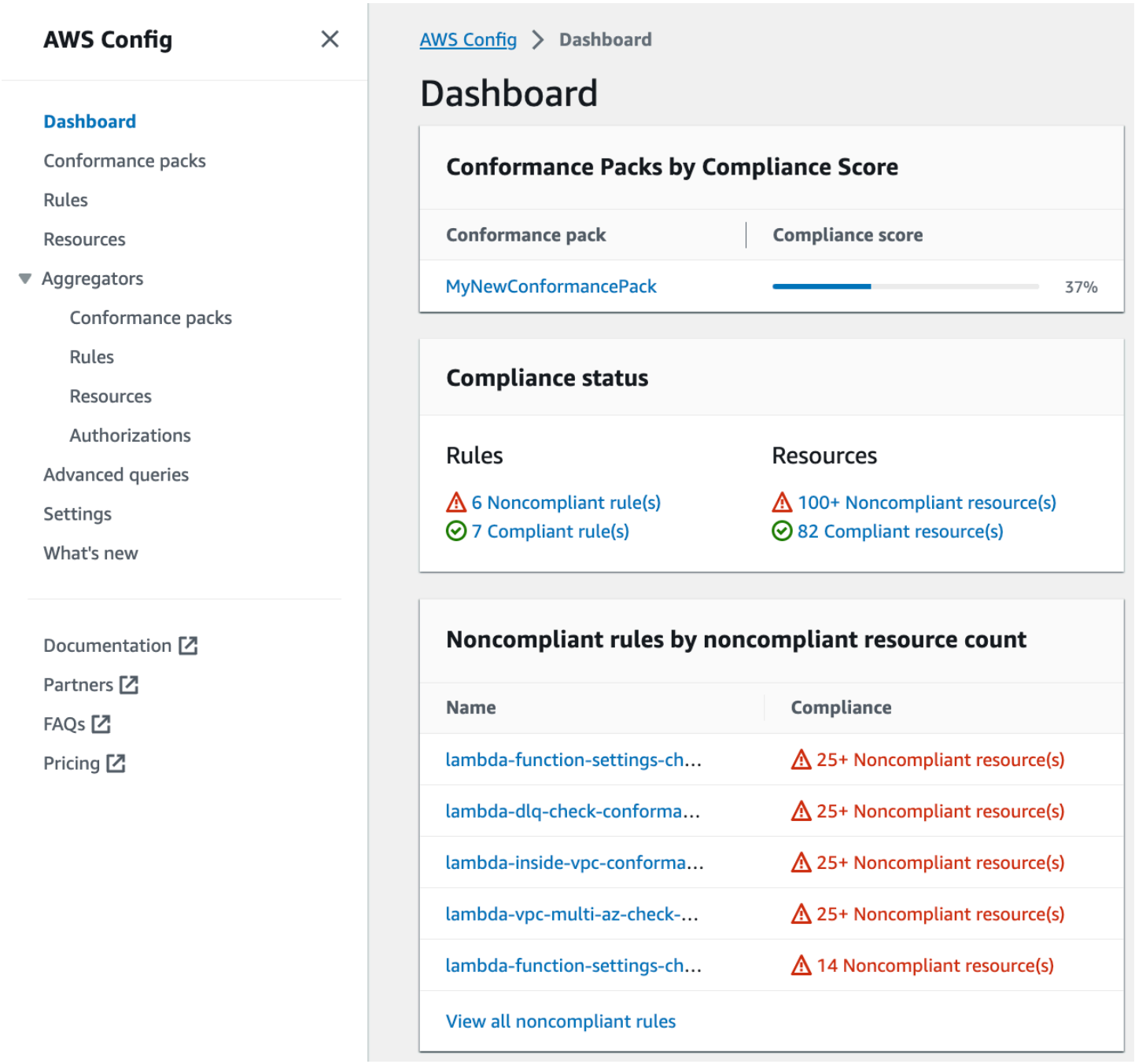
您可以使用查询来构建 Amazon QuickSight 控制面板并可视化数据。要聚合 AWS 资源配置数据，在 Athena 中创建表，以及根据来自 Athena 的数据构建 Amazon QuickSight 控制面板，请参阅 AWS Cloud Operations and Management Blog 上的 Visualizing AWS Config data using Athena and Amazon QuickSight。值得注意的是，此查询还会检索函数的标签信息。这样就可以更深入地了解您的工作负载和环境，尤其是在您使用自定义标签的情况下。

有关您可以执行的操作的更多信息，请参阅本主题后面的 处理可观测性调查发现 部分。

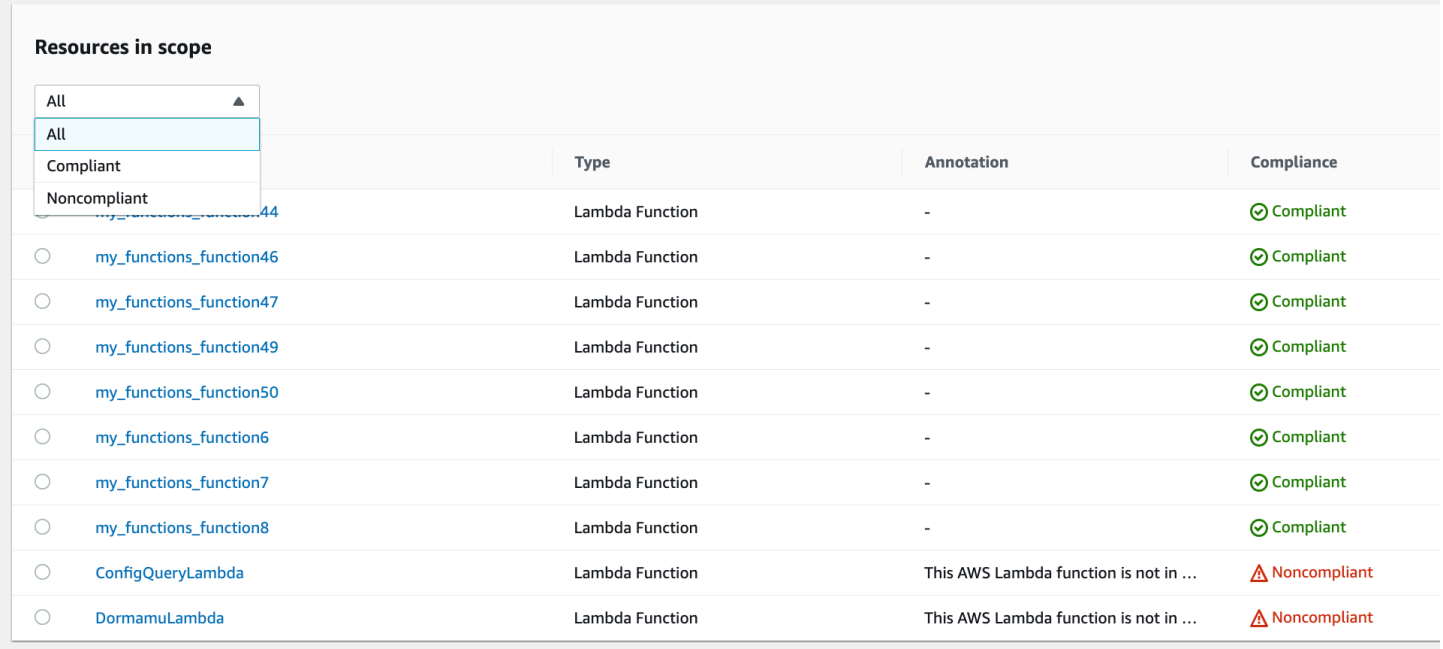
Lambda 合规性的可见性

通过 AWS Config 生成的数据，您可以创建组织级别的控制面板来监控合规性。这样就可以对以下内容实现一致的跟踪和监控：

- 按合规性分数列出的合规包
- 按不合规资源列出的规则
- 合规性状态

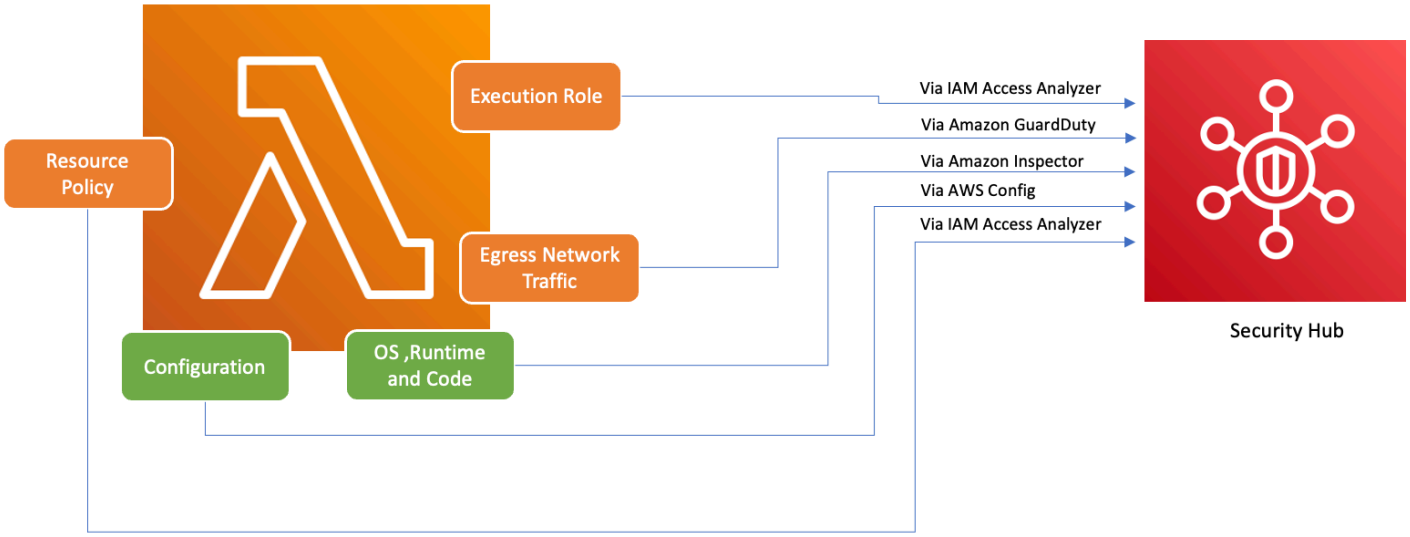


检查每条规则，找出该规则的不合规资源。例如，如果您的组织要求所有 Lambda 函数都必须与 VPC 关联，并且您已经部署了用于识别合规性的 AWS Config 规则，则可以在上面的列表中选择 `lambda-inside-vpc` 规则。



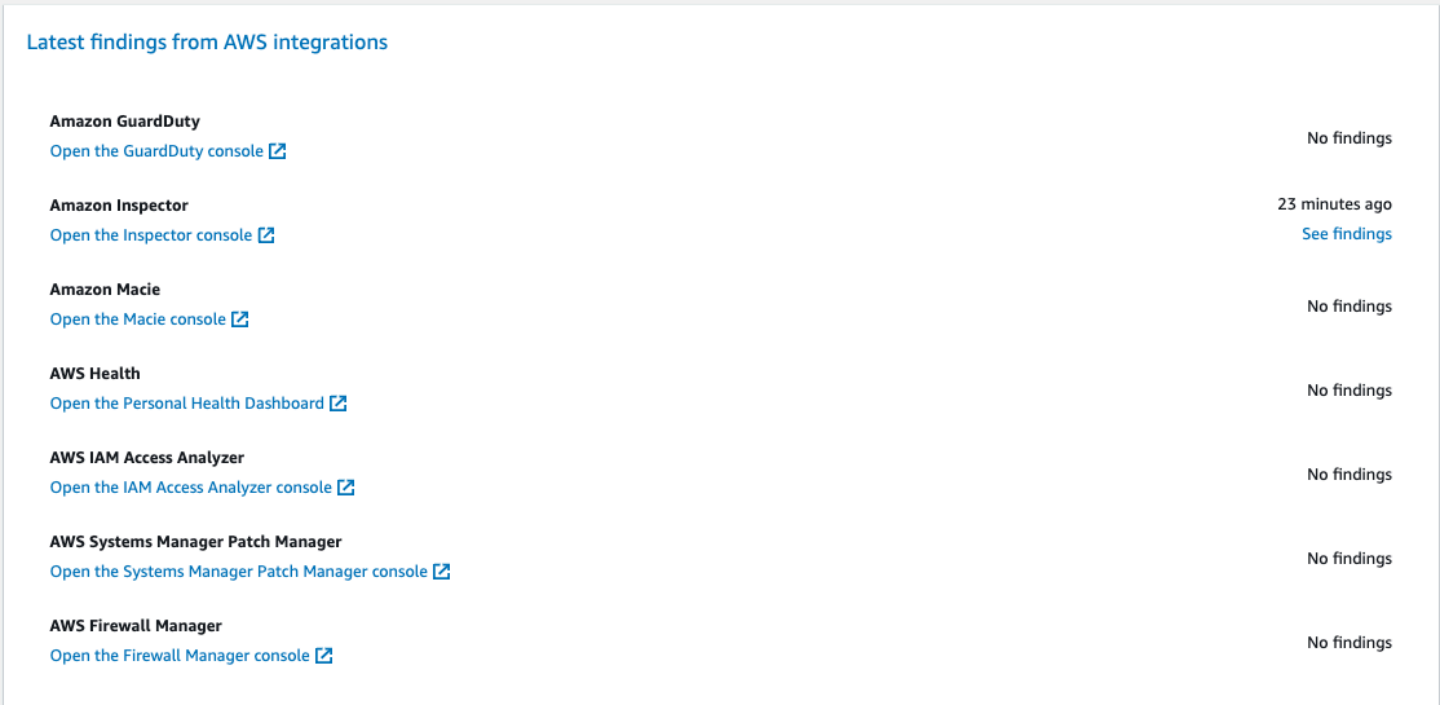
有关您可以执行的操作的更多信息，请下面的 处理可观测性调查发现 部分。

使用 Security Hub 查看 Lambda 函数边界的可见性

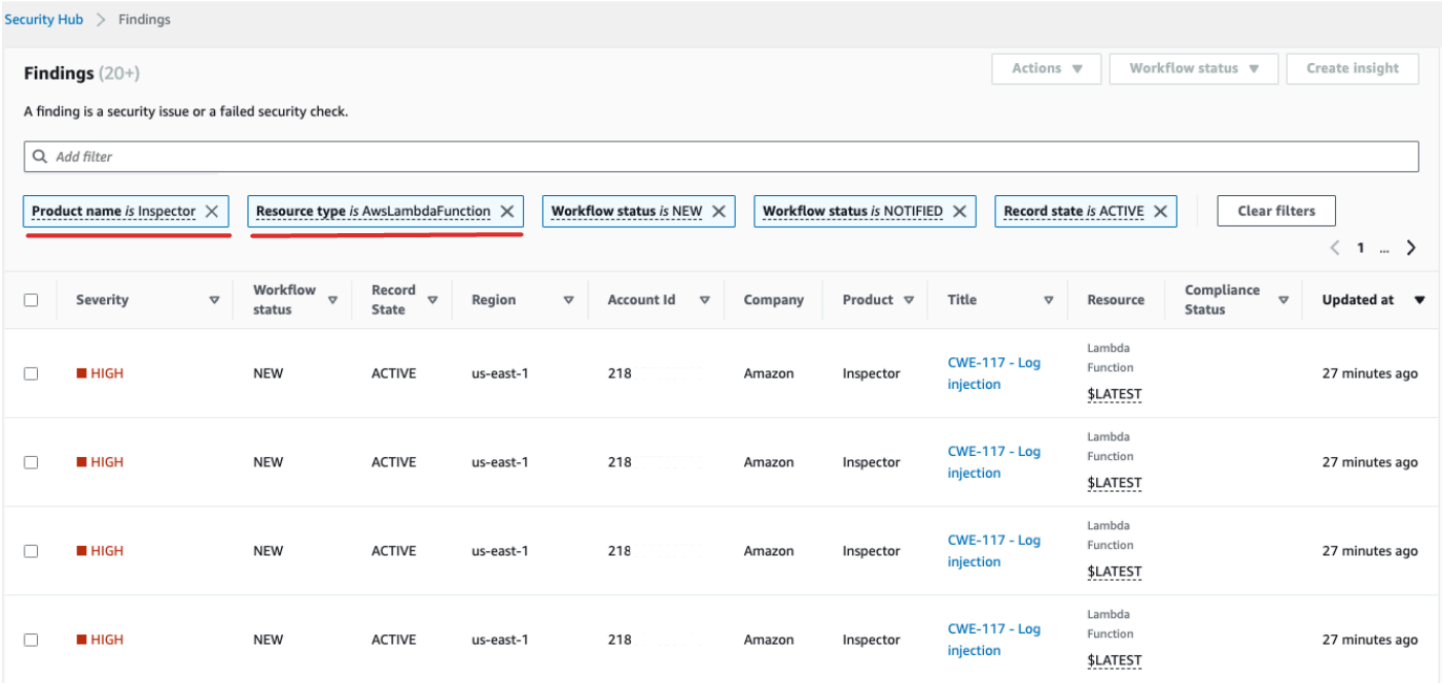


为确保包括 Lambda 在内的 AWS 服务的安全使用，AWS 引入了基础安全最佳实践 v1.0.0。这套最佳实践为保护 AWS 环境中的资源和数据提供了明确的指导原则，强调了保持强大安全态势的重要性。AWS Security Hub 提供了一个统一的安全性和合规性中心，从而对此进行了补充。它汇总、组织并优先处理来自多个 AWS 服务的安全调查发现，如 Amazon Inspector、AWS Identity and Access Management Access Analyzer 和 Amazon GuardDuty。

如果您在 AWS 组织中启用了 Security Hub、Amazon Inspector、IAM Access Analyzer 和 GuardDuty，则 Security Hub 会自动汇总来自这些服务的调查发现。例如，让我们来看一下 Amazon Inspector。通过 Security Hub，您可以有效地识别 Lambda 函数中的代码和程序包漏洞。在 Security Hub 控制台中，导航到底部标有来自 AWS 集成的最新调查发现的部分。在这里，您可以查看和分析来自各种集成的 AWS 服务的调查发现。



要查看详细信息，请选择第二列中的查看调查发现链接。这将显示按产品（例如 Amazon Inspector）筛选的调查发现列表。要将搜索范围限制为 Lambda 函数，请将 `ResourceType` 设置为 `AwsLambdaFunction`。这将显示 Amazon Inspector 与 Lambda 函数相关的调查发现。



对于 GuardDuty，您可以识别可疑的网络流量模式。此类异常情况可能表明您的 Lambda 函数中存在潜在的恶意代码。

通过 IAM Access Analyzer，您可以检查策略，尤其是那些带有向外部实体授予函数访问权限的条件语句的策略。此外，IAM Access Analyzer 还会评估在使用 Lambda API 中 AddPermission 操作时与 `EventSourceToken` 一起设置的权限。

处理可观测性调查发现

鉴于 Lambda 函数的可配置范围很广，而且要求各不相同，因此标准化的自动化补救解决方案可能无法适用于所有情况。此外，在不同的环境中，变更的实施方式也有所不同。如果您遇到任何看似不合规的配置，请考虑以下指导原则：

1. 标记策略

   我们建议实施全面的标记策略。每个 Lambda 函数都应使用关键信息进行标记，例如以下信息：

   - 所有者：负责该函数的人员或团队。
   - 环境：生产、暂存、开发或沙盒。
   - 应用程序：此函数所属的更广泛上下文（如果适用）。

2. 所有者外联活动

与其自动进行重大更改（例如 VPC 配置调整），不如主动联系不合规函数（通过所有者标签标识）的所有者，方便其拥有足够时间执行以下任一操作：

- 调整 Lambda 函数上的不合规配置。
- 提供解释并申请例外情况，或完善合规标准。

3. 维护配置管理数据库（CMDB）

   虽然标签可以提供即时上下文，但通过维护集中式 CMDB，可以提供更深入的见解。CMDB 可以保存有关每个 Lambda 函数、其依赖关系和其他关键元数据的更精细的信息。CMDB 是审计、合规性检查和识别函数所有者的宝贵资源。

随着无服务器基础设施状况的不断演变，采取积极主动的监控态度至关重要。通过 AWS Config、Security Hub 和 Amazon Inspector 等工具，可以快速识别出潜在的异常或不合规配置。但是，仅靠工具并不能确保完全合规或使用最佳配置。将这些工具与有据可查的流程和最佳实践相结合至关重要。

- 反馈环路：确保采取补救措施后会有反馈环路。这意味着要定期重访不合规资源，以确认其是否已更新，或是否在存在相同问题的情况下继续运行。
- 文档：务必记录观察结果、采取的行动以及批准的任何例外情况。适当的文档不仅有助于审计，而且还有助于改进流程，从而在将来提高合规性和安全性。
- 培训和意识：确保所有利益相关者（特别是 Lambda 函数所有者），定期接受培训，并了解最佳实践、组织策略和合规要求。定期举办研讨会、网络研讨会或培训课程，对确保每个人就安全性和合规性保持一致意见大有裨益。

总而言之，尽管工具和技术为检测和标记潜在问题提供了强大功能，但人的因素（理解、沟通、培训和文档）仍然至关重要。它们共同构成了强大的组合，可确保您的 Lambda 函数和更广泛的基础设施保持合规性、安全性并根据您的业务需求进行优化。

AWS Lambda 的合规性验证

作为多个 AWS Lambda 合规性计划的一部分，第三方审计员将评估 AWS 的安全性和合规性。其中包括 SOC、PCI、FedRAMP、HIPAA 及其他。

有关特定合规性计划范围内的 AWS 服务的列表，请参阅合规性计划范围内的 AWS 服务。有关一般信息，请参阅 AWS 合规性计划。

您可以使用 AWS Artifact 下载第三方审计报告。有关更多信息，请参阅下载AWS构件中的报告。

您在使用 Lambda 时的合规性责任由您的数据的敏感性、您公司的合规性目标以及适用的法律法规决定。您可以实施治理控制，以确保贵公司的 Lambda 函数符合合规要求。有关更多信息，请参阅 为 Lambda 函数和层创建治理策略。

AWS Lambda 中的故障恢复能力

AWS全球基础架构围绕AWS区域和可用区构建。AWS区域提供多个在物理上独立且隔离的可用区，这些可用区通过延迟低、吞吐量高且冗余性高的网络连接在一起。利用可用区，您可以设计和操作在可用区之间无中断地自动实现故障转移的应用程序和数据库。与传统的单个或多个数据中心基础架构相比，可用区具有更高的可用性、容错性和可扩展性。

有关 AWS 区域和可用区的更多信息，请参阅 AWS 全球基础设施。

除了AWS全球基础设施之外，Lambda 还提供了多种功能，以帮助支持您的数据故障恢复能力和备份需求。

- 版本控制 – 您可以在 Lambda 中使用版本控制，在开发时保存函数的代码和配置。与别名相配合，您可以使用版本控制来执行蓝/绿和滚动部署。有关详细信息，请参阅管理 Lambda 函数版本。
- 扩展 – 当您的函数在处理之前的请求时收到新的请求，Lambda 会启动另一个函数实例来处理增加的负载。Lambda 会弹性伸缩以处理每个区域 1000 个并发执行，配额可以根据需要增加。有关详细信息，请参阅了解 Lambda 函数扩展。
- 高可用性 – Lambda 会在多个可用区中运行您的函数，确保在单一区域中服务中断时能够处理事件。如果将函数配置为连接到账户中的 Virtual Private Cloud (VPC)，请在多个可用区中指定子网以确保高可用性。有关详细信息，请参阅授予 Lambda 函数访问 Amazon VPC 中资源的权限。
- 预留并发 – 要确保您的函数能够始终扩展以处理更多请求，您可以为其预留并发。为函数设置预留并发可确保其能够扩展（但不超出）指定数量的并发调用。这可确保您不会因为其他函数使用了所有可用的并发而丢失请求。有关详细信息，请参阅为函数配置预留并发。

- 重试 – 对于异步调用和由其他服务触发的调用子集，Lambda 会在遇到错误时自动重试（每次重试有延迟）。同步调用函数的其他客户端和 AWS 服务负责执行重试。有关详细信息，请参阅了解 Lambda 中的重试行为。
- 死信队列 – 对于异步调用，如果所有重试都失败，您可以配置 Lambda 向死信队列发送请求。死信队列是 Amazon SNS 主题或 Amazon SQS 队列，它会接收事件进行故障排除或重新处理。有关详细信息，请参阅添加死信队列。

AWS Lambda 中的基础设施安全性

作为一项托管式服务，AWS Lambda 受 AWS 全球网络安全保护。有关 AWS 安全服务以及 AWS 如何保护基础设施的信息，请参阅 AWS 云安全。要按照基础架构安全最佳实践设计您的 AWS 环境，请参阅《安全性支柱 AWS Well-Architected Framework》中的 基础架构保护。

您可以使用AWS发布的 API 调用，通过网络访问 Lambda。客户端必须支持以下内容：

- 传输层安全性协议 (TLS) 我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 具有完全向前保密 (PFS) 的密码套件，例如 DHE（临时 Diffie-Hellman）或 ECDHE（临时椭圆曲线 Diffie-Hellman）。大多数现代系统（如 Java 7 及更高版本）都支持这些模式。

此外，必须使用访问密钥 ID 和与 IAM 主体关联的秘密访问密钥来对请求进行签名。或者，您可以使用 AWS Security Token Service（AWS STS）生成临时安全凭证来对请求进行签名。

使用代码签名通过 Lambda 验证代码完整性

AWS Lambda 代码签名有助于确保 Lambda 函数中只运行可信代码。为函数启用代码签名时，Lambda 将检查每次代码部署并验证代码包是否由可信来源签名。

> **Note**
>
> 定义为容器映像的函数不支持代码签名。

要验证代码完整性，请使用 AWS Signer 为函数和层创建经数字签名的代码包。当用户尝试部署代码包时，Lambda 会在接受部署之前对代码包执行验证检查。由于代码签名验证检查在部署时运行，因此对函数执行的性能没有影响。

您还可以使用 AWS Signer 创建签名配置文件。您可以使用签名配置文件创建签名代码包。使用 AWS Identity and Access Management (IAM) 可以控制能够对代码包进行签名并创建签名配置文件的对象。有关更多信息，请参阅 AWS Signer 开发人员指南中的身份验证和访问控制。

Lambda 层采用与函数代码包相同的签名代码包格式。将层添加到启用代码签名的函数时，Lambda 会检查该层是否由允许的签名配置文件签名。如果为函数启用代码签名，则添加到函数的所有层还必须由一项允许的签名配置文件进行签名。

您可以配置代码签名以将更改记录到 AWS CloudTrail。成功和受阻的函数部署都将记录到 CloudTrail 并随附签名和验证检查的信息。

使用 AWS Signer 或 AWS Lambda 代码签名不收取任何额外费用。

签名验证

将签名代码包部署到函数时，Lambda 会执行以下验证检查：

1. 完整性 – 验证代码包自签名以来是否尚未修改。Lambda 将包的哈希值与签名的哈希值进行比较。
2. 过期 – 验证代码包的签名是否尚未过期。
3. 不匹配 – 验证代码包是否使用 Lambda 函数允许的一项签名配置文件进行签名。如果签名不存在，则会导致不匹配。
4. 撤销 – 验证代码包的签名是否尚未撤销。

在代码签名配置中定义的签名验证策略确定了在任意一项验证检查失败时，Lambda 应采取以下哪一项操作：

- 警告 – Lambda 允许部署代码包，但会发出警告。Lambda 会发布新的 Amazon CloudWatch 指标，并将警告存储在 CloudTrail 日志中。
- 强制执行 – Lambda 发出警告（与警告操作相同）并阻止代码包的部署。

您可以为过期、不匹配和撤销验证检查配置策略。请注意，您无法为完整性检查配置策略。如果完整性检查失败，Lambda 将阻止部署。

使用 Lambda API 配置代码签名

要使用 AWS CLI 或 AWS 开发工具包管理代码签名配置，请使用以下 API 操作：

- ListCodeSigningConfigs

- CreateCodeSigningConfig
- GetCodeSigningConfig
- UpdateCodeSigningConfig
- DeleteCodeSigningConfig

要管理函数的代码签名配置，请使用以下 API 操作：

- CreateFunction
- GetFunctionCodeSigningConfig
- PutFunctionCodeSigningConfig
- DeleteFunctionCodeSigningConfig
- ListFunctionsByCodeSigningConfig

主题

- 为 Lambda 创建代码签名配置
- 更新代码签名配置
- 为 Lambda 代码签名配置 IAM 策略
- 在代码签名配置上使用标签

为 Lambda 创建代码签名配置

要为函数启用代码签名，您需要创建代码签名配置并将其附加到函数。代码签名配置定义了允许的签名配置文件列表以及在任意一项验证检查失败时要采取的策略操作。

Sections

- 配置先决条件
- 创建代码签名配置
- 为函数启用代码签名

配置先决条件

为 Lambda 函数配置代码签名之前，请使用 AWS Signer 执行以下操作：

- 创建一个或多个签名配置文件。

- 可以使用签名配置文件为函数创建签名代码包。

有关更多信息，请参阅 AWS Signer 开发人员指南中的创建签名配置文件（控制台）。

创建代码签名配置

代码签名配置定义允许的签名配置文件列表和签名验证策略。

创建代码签名配置（控制台）

1. 打开 Lambda 控制台的 Code signing configurations（代码签名配置）页面。
2. 选择 Create configuration（创建配置）。
3. 对于 Description（描述），输入一个描述性的配置名称。
4. 在 Signing profiles（签名配置文件）下，最多可以在配置中添加 20 个签名配置文件。
   a. 对于 Signing profile version ARN（签名配置文件版本 ARN），选择配置文件版本的 Amazon 资源名称 (ARN) 或输入 ARN。
   b. 要添加其他签名配置文件，请选择 Add signing profiles（添加签名配置文件）。
5. 在 Signature validation policy（签名验证策略）下，选择 Warn（警告）或 Enforce（强制执行）。
6. 选择 Create configuration（创建配置）。

为函数启用代码签名

要为函数启用代码签名，您需要将代码签名配置与函数关联。

将代码签名配置与函数关联（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择要启用代码签名的函数。
3. 打开 Configuration（配置）选项卡。
4. 向下滚动并选择代码签名。
5. 选择编辑。
6. 在 Edit code signing（编辑代码签名）中，为此函数选择代码签名配置。
7. 选择保存。

更新代码签名配置

更新代码签名配置时，这些更改将会在未来的部署中影响附加代码签名配置的函数。

更新代码签名配置（控制台）

1. 打开 Lambda 控制台的 Code signing configurations（代码签名配置）页面。
2. 选择要更新的代码签名配置，然后选择 Edit（编辑）。
3. 对于 Description（描述），输入一个描述性的配置名称。
4. 在 Signing profiles（签名配置文件）下，最多可以在配置中添加 20 个签名配置文件。
   a. 对于 Signing profile version ARN（签名配置文件版本 ARN），选择配置文件版本的 Amazon 资源名称 (ARN) 或输入 ARN。
   b. 要添加其他签名配置文件，请选择 Add signing profiles（添加签名配置文件）。
5. 在 Signature validation policy（签名验证策略）下，选择 Warn（警告）或 Enforce（强制执行）。
6. 选择 Save changes（保存更改）。

为 Lambda 代码签名配置 IAM 策略

要授予用户访问代码签名 API 操作的权限，请将一个或多个策略声明附加到用户策略。有关用户策略的详细信息，请参阅Lambda 的基于身份的 IAM policy。

以下示例策略声明将授予创建、更新和检索代码签名配置的权限。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
          "lambda:CreateCodeSigningConfig",
          "lambda:UpdateCodeSigningConfig",
          "lambda:GetCodeSigningConfig"
        ],
      "Resource": "*"
    }
  ]
```

```
}
```

管理员可以使用 CodeSigningConfigArn 条件键指定开发人员创建或更新函数时必须使用的代码签名配置。

以下示例策略声明将授予创建函数的权限。策略声明包括指定允许的代码签名配置的 lambda:CodeSigningConfigArn 条件。如果 CodeSigningConfigArn 参数缺失或与条件中的值不匹配，Lambda 会阻止任何 CreateFunction API 请求。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowReferencingCodeSigningConfig",
      "Effect": "Allow",
      "Action": [
          "lambda:CreateFunction",
        ],
      "Resource": "*",
      "Condition": {
          "StringEquals": {
              "lambda:CodeSigningConfigArn":
                  “arn:aws:lambda:us-west-2:123456789012:code-signing-
config:csc-0d4518bd353a0a7c6”
          }
      }
    }
  ]
}
```

在代码签名配置上使用标签

对代码签名配置进行标签可以组织和管理资源。标签是与资源关联的自由格式键值对，在 AWS 服务中受支持。有关标签用例的更多信息，请参阅《Tagging AWS Resources and Tag Editor Guide》中的 Common tagging strategies。

可以使用 AWS Lambda API 来查看和更新标签。在 Lambda 控制台中管理特定的代码签名配置时，还可以查看和更新标签。

Sections

- 使用标签所需的权限

- 通过 Lambda 控制台使用标签
- 通过 AWS CLI 使用标签

使用标签所需的权限

要允许 AWS Identity and Access Management（IAM）身份（用户、组或角色）读取资源或为其设置标签，请授予该身份相应的权限：

- lambda:ListTags – 当资源有标签时，将此权限授予需要在其上调用 `ListTags` 的任何人。对于带标签的函数，`GetFunction` 也需要此权限。
- lambda:TagResource – 将此权限授予需要调用 `TagResource` 或执行在创建时授予标记的操作的任何人。

有关更多信息，请参阅 Lambda 的基于身份的 IAM policy。

通过 Lambda 控制台使用标签

可以使用 Lambda 控制台创建带标签的代码签名配置、向现有代码签名配置添加标签、按标签筛选代码签名配置。

在创建代码签名配置时添加标签

1. 在 Lambda 控制台中打开代码签名配置。
2. 从内容窗格的标题中选择创建配置。
3. 在内容窗格的顶部区域中设置代码签名配置。有关设定代码签名配置的更多信息，请参阅the section called “代码签名”。
4. 在标签部分，选择添加新标签。
5. 在键字段中输入标签键。有关标记限制的信息，请参阅《Tagging AWS Resources and Tag Editor Guide》中的 Tag naming limits and requirements。
6. 选择创建配置。

向现有代码签名配置添加标签

1. 在 Lambda 控制台中打开代码签名配置。
2. 选择代码签名配置的名称。
3. 在详细信息窗格下方的选项卡中选择标签。

4. 选择管理标签。
5. 选择 Add new tag（添加新标签）。
6. 在键字段中输入标签键。有关标记限制的信息，请参阅《Tagging AWS Resources and Tag Editor Guide》中的 Tag naming limits and requirements。
7. 选择保存。

按标签筛选代码签名配置

1. 在 Lambda 控制台中打开代码签名配置。
2. 选择搜索框。
3. 在下拉列表中，从标签副标题下方选择标签。
4. 选择使用：“tag-name”查看所有使用此键标记的代码签名配置，或者选择一个运算符进一步按值筛选。
5. 选择标签值以按标签键和值的组合进行筛选。

搜索框还支持搜索标签键。输入键名称，即可在列表中查找该键。

通过 AWS CLI 使用标签

可以使用 Lambda API 在现有 Lambda 资源（包括代码签名配置）上添加和删除标签。还可以在创建代码签名配置时添加标签，这样就可以在资源的整个生命周期中对其进行标记。

使用 Lambda 标签 API 更新标签

可以通过 TagResource 和 UntagResource API 操作，添加和删除受支持 Lambda 资源的标签。

可以使用 AWS CLI 调用这些操作。要向现有资源添加标签，请使用 `tag-resource` 命令。此示例添加了两个标签，一个带有键 *`Department`*，另一个带有键 *`CostCenter`*。

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

要删除标签，请使用 `untag-resource` 命令。此示例删除了键为 *`Department`* 的标签。

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
```

```
--tag-keys Department
```

在创建代码签名配置时添加标签

若要使用标签创建新的 Lambda 代码签名配置，请使用 CreateCodeSigningConfig API 操作。指定 Tags 参数。可以使用 create-code-signing-config AWS CLI 命令和 --tags 选项调用此操作。有关 CLI 命令的更多信息，请参阅《AWS CLI Command Reference》中的 create-code-signing-config。

在将 Tags 参数与 CreateCodeSigningConfig 一起使用之前，请确保角色拥有标记资源的权限以及此操作所需的常规权限。有关标记权限的更多信息，请参阅 the section called “使用标签所需的权限”。

使用 Lambda 标签 API 查看标签

要查看应用于特定 Lambda 资源的标签，请使用 ListTags API 操作。有关更多信息，请参阅 ListTags。

可以提供 ARN（Amazon 资源名称），以使用 list-tags AWS CLI 命令调用此操作。

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

按标签筛选资源

您可以使用 AWS Resource Groups Tagging API GetResources API 操作按标签筛选资源。GetResources 操作最多可接收 10 个筛选条件，每个筛选条件包含一个标签键和最多 10 个标签值。提供具有 ResourceType 的 GetResources，可按特定资源类型进行筛选。

可以使用 get-resources AWS CLI 命令调用此操作。有关使用 get-resources 的示例，请参阅《AWS CLI Command Reference》中的 get-resources。

监控 Lambda 函数并进行故障排除

AWS Lambda 与其他AWS服务集成，以帮助您监控 Lambda 函数并对其进行故障排查。Lambda 会自动通过 Amazon CloudWatch 代表您监控 Lambda 函数报告指标。为帮助您在代码运行时监控代码，Lambda 会自动跟踪请求数、每个请求的调用持续时间和产生错误的请求数。

您可以使用其他AWS服务对 Lambda 函数进行问题排查。本节将介绍如何使用这些AWS服务来监控、跟踪、调试和排查 Lambda 功能和应用程序。有关每个运行时中的函数日志记录和错误的详细信息，请参阅单个运行时部分。

有关监控 Lambda 应用程序的更多信息，请参阅 Serverless Land 中的 Monitoring and observability。

Sections

定价

CloudWatch 控提供了永久性的免费套餐。若超出免费套餐阈值，CloudWatch 将收取指标、控制面板、告警、日志和洞察的费用。有关更多信息，请参阅 Amazon CloudWatch 定价。

查看 Lambda 函数的指标

当您的 AWS Lambda 函数完成对事件的处理时，Lambda 会将有关调用的指标发送到 Amazon CloudWatch。这些指标无需支付费用。

在 CloudWatch 控制台上，您可以使用这些指标构建图形和控制面板。您可以设置告警以响应利用率、性能或错误率的更改。Lambda 每隔 1 分钟向 CloudWatch 发送一次指标数据。如果您想获得更多对 Lambda 函数的即时洞察，可以按照 Serverless Land 中的说明创建高分辨率自定义指标。自定义指标和 CloudWatch 告警需支付费用。有关更多信息，请参阅 Amazon CloudWatch 定价。

本页面介绍 CloudWatch 控制台上提供的 Lambda 函数调用、性能和并发指标。

Sections

在 CloudWatch 控制台上查看指标

您可以使用 CloudWatch 控制台，按函数名称、别名或版本对函数指标进行筛选和排序。

要在 CloudWatch 控制台上查看指标

1. 打开 CloudWatch 控制台的 Metrics page（指标页面）（AWS/Lambda 命名空间）。
2. 在浏览选项卡的指标下，选择以下任一维度：
   - 按函数名称 (FunctionName) – 查看函数的所有版本和别名的聚合指标。
   - 按资源 (Resource) – 查看函数的版本或别名的指标。
   - 按已执行版本 (ExecutedVersion) – 查看别名和版本组合的指标。使用 ExecutedVersion 维度可以比较函数的两个版本的错误率，这两个版本都是加权别名的目标。
   - 跨所有函数（无）– 查看当前 AWS 区域 区域中所有函数的聚合指标。
3. 选择一个指标，然后选择添加到图表或其他图表选项。

默认情况下，图表将使用所有指标的 Sum 统计数据。要选择其他统计数据并自定义图表，请使用 Graphed metrics (图表化指标) 选项卡上的选项。

> **Note**
>
> 指标上的时间戳反映函数被调用的时间。根据调用的持续时间，这可能是发出指标前的几分钟。例如，如果您函数的超时值为 10 分钟，请查看 10 分钟之前的情况，以获取准确的指标。

有关 CloudWatch 的更多信息，请参阅 Amazon CloudWatch 用户指南。

指标类型

以下小节介绍 CloudWatch 控制台上提供的 Lambda 指标的类型。

调用指标

调用指标是 Lambda 函数调用结果的二进制指示器。例如，如果函数返回一个错误，则 Lambda 会发送值为 1 的 `Errors` 指标。要获取每分钟发生的函数错误计数，请查看一分钟时段内的 `Errors` 指标的 `Sum`。

> **Note**
>
> 使用 `Sum` 统计数据查看以下调用指标。

- `Invocations` – 函数代码的调用次数，包括成功调用和导致函数错误的调用。如果调用请求受到限制或导致调用错误，则不会记录调用。`Invocations` 的值等于计费的请求数。
- `Errors` – 导致出现函数错误的调用的次数。函数错误包括您的代码所引发的异常以及 Lambda 运行时所引发的异常。运行时返回因超时和配置错误等问题导致的错误。要计算错误率，请将 `Errors` 的值除以 `Invocations` 的值。请注意，错误指标上的时间戳反映的是调用函数的时间，而非错误发生的时间。
- `DeadLetterErrors` – 对于异步调用，Lambda 尝试将事件发送到死信队列（DLQ）但失败的次数。由于资源或大小限制设置不正确，可能会发生死信错误。
- `DestinationDeliveryFailures` – 对于异步调用和支持的事件源映射，Lambda 尝试将事件发送到目标但失败的次数。对于事件源映射，Lambda 支持流源（DynamoDB 和 Kinesis）的目标。由于权限错误、资源配置不正确或大小限制，可能会发生传输错误。如果您配置的目标是不支持的目标类型，例如 Amazon SQS FIFO 队列或 Amazon SNS FIFO 主题，则可能会发生这种错误。
- `Throttles` – 受限制的调用请求数。当所有函数实例都在处理请求并且没有可用于纵向扩展的并发时，Lambda 将拒绝其他请求，并出现 `TooManyRequestsException` 错误。受限制的请求和其他调用错误不会计为 `Invocations` 或 `Errors`。

- OversizedRecordCount：对于 Amazon DocumentDB 事件源，函数从变更流接收到的大小超过 6 MB 的事件数。Lambda 会丢弃该消息并发出此指标。
- ProvisionedConcurrencyInvocations – 使用预置并发调用函数代码的次数。
- ProvisionedConcurrencySpilloverInvocations – 当所有预置并发均处于使用状态时，使用标准并发调用函数代码的次数。
- RecursiveInvocationsDropped – Lambda 因检测到您的函数属于无限递归循环而停止调用函数的次数。递归循环检测通过跟踪受支持的 AWS SDK 添加的元数据，来监控函数作为请求链的一部分被调用的次数。默认情况下，如果您的函数作为请求链的一部分被调用的次数接近 16 次，Lambda 会中断下一次调用。如果禁用递归循环检测，则不会发出此指标。有关此特征的更多信息，请参阅 使用 Lambda 递归循环检测来防止无限循环。

性能指标

性能指标提供了有关单个函数调用的性能详细信息。例如，Duration 指标指示函数处理事件所花费的时间量（以毫秒为单位）。要了解函数处理事件的速度，请使用 Average 或 Max 统计数据查看这些指标。

- Duration – 函数代码处理事件所花费的时间量。调用的计费持续时间是已舍入到最近的毫秒的 Duration 值。Duration 不包括冷启动时间。
- PostRuntimeExtensionsDuration – 函数代码完成后，运行时为扩展运行代码所花费的累积时间。
- IteratorAge：对于 DynamoDB、Kinesis 和 Amazon DocumentDB 事件源，事件中最后一条记录的期限。该指标测量流接收记录的时间到事件源映射将事件发送到函数的时间之间的时间量。
- OffsetLag – 对于自行管理的 Apache Kafka 和 Amazon Managed Streaming for Apache Kafka（Amazon MSK）事件源，写入到主题的最后一条记录与函数的使用者组处理的最后一条记录之间的偏移量差值。尽管 Kafka 主题可以包含多个分区，但此指标仍可在主题级别衡量偏移延迟。

Duration 还支持百分位数 (p) 统计数据。使用百分位数可排除偏离 Average 和 Maximum 数据的异常值。例如，p95 统计数据显示 95% 的调用的最长持续时间，并排除最慢的 5% 的调用。有关更多信息，请参阅《Amazon CloudWatch 用户指南》中的百分位数。

并发指标

Lambda 将并发指标报告为跨函数、版本、别名或 AWS 区域 处理事件的实例数的总计数。要查看接近并发限制的程度，请使用 Max 统计数据查看这些指标。

- ConcurrentExecutions – 正在处理事件的函数实例的数目。如果此数目达到区域的并发执行配额或您在函数上配置的预留并发限制，则 Lambda 将会限制其他调用请求。
- ProvisionedConcurrentExecutions – 使用预置并发处理事件的函数实例的数目。对于具有预置并发性的别名或版本的每次调用，Lambda 都会发出当前计数。
- ProvisionedConcurrencyUtilization – 对于版本或别名，为将 ProvisionedConcurrentExecutions 值除以配置的预置并发总数。例如，为函数配置的预置并发数为 10，而 ProvisionedConcurrentExecutions 为 7，则 ProvisionedConcurrencyUtilization 为 0.7。
- UnreservedConcurrentExecutions – 对于区域，由不具有预留并发的函数处理的事件数。
- ClaimedAccountConcurrency – 针对某个区域，不可用于按需调用的并发数。ClaimedAccountConcurrency 等于 UnreservedConcurrentExecutions 加上分配的并发数（即总预留并发数加上总预置并发数）。有关更多信息，请参阅 使用 ClaimedAccountConcurrency 指标。

异步调用指标

异步调用指标提供有关来自事件源的异步调用和直接调用的详细信息。您可以设置阈值和警报以通知您某些变化。例如，当排队等待处理的事件数量意外增加时 (AsyncEventsReceived)。或者，当一个事件等待了很长时间才完成处理时 (AsyncEventAge)。

- AsyncEventsReceived – Lambda 成功排队等待处理的事件数。此指标可让您深入了解 Lambda 函数接收的事件数量。监控此指标并设置阈值警报以检查是否存在问题。例如，用于检测出发送到 Lambda 的事件数量是否异常，还能用于快速诊断出因触发错误或函数配置不正确而导致的问题等等。AsyncEventsReceived 和 Invocations 之间的不匹配可能表明处理过程存在差异、事件被丢弃或潜在的队列积压。
- AsyncEventAge – Lambda 成功将事件排队到调用该函数之间的时间。当由于调用失败或节流而重试事件时，此指标的值会增加。监控此指标，并在出现队列积聚时针对不同统计信息的阈值设置警报。要解决该指标增加的问题，请查看 Errors 指标以识别函数错误，并查看 Throttles 指标以确定并发问题。
- AsyncEventsDropped – 在未成功执行函数的情况下丢弃的事件数。如果您配置了死信队列（DLQ）或 OnFailure 目标，则事件会在丢弃之前发送到那里。事件因各种原因被丢弃。例如，事件可能超过最大事件期限或耗尽最大重试次数，或者预留并发可能设置为 0。要解决该指标被丢弃的问题，请查看 Errors 指标以识别函数错误，并查看 Throttles 指标以确定并发问题。

将 CloudWatch Logs 日志与 Lambda 结合使用

AWS Lambda 将代表您自动监控 Lambda 函数以帮助您解决函数故障。只要您的函数的执行角色具有必要的权限，Lambda 就会捕获您的函数处理的所有请求的日志，并将其发送到 Amazon CloudWatch Logs。

您可以将录入语句插入到您的代码中，以帮助您验证代码是否按预期工作。Lambda 自动与 CloudWatch Logs 集成，并将您的代码的所有日志发送到与 Lambda 函数关联的 CloudWatch Logs 组。

默认情况下，Lambda 向名为 `/aws/lambda/<function name>` 的日志组发送日志。如果您希望函数向另一个群组发送日志，则可以使用 Lambda 控制台、AWS Command Line Interface（AWS CLI）或 Lambda API 进行配置。请参阅the section called “配置 CloudWatch 日志组”，了解更多信息。

您可以借助 Lambda 控制台、CloudWatch 控制台、AWS Command Line Interface (AWS CLI) 或 CloudWatch API 查看 Lambda 函数的日志。

> **Note**
>
> 函数调用后，日志可能需要 5 到 10 分钟才能显示。

所需的 IAM 权限

执行角色必须具备以下权限，才能将日志上传 CloudWatch Logs：

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

要了解更多信息，请参阅《Amazon CloudWatch User Guide》中的 Using identity-based policies (IAM policies) for CloudWatch Logs。

您可以使用 Lambda 提供的 `AWSLambdaBasicExecutionRole` AWS 托管式策略添加这些 CloudWatch Logs 权限。要将此策略添加到您的角色，请运行以下命令：

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

有关更多信息，请参阅 the section called “AWS 托管策略”。

定价

使用 Lambda 日志没有额外的费用；不过，会收取标准 CloudWatch Logs 费用。有关更多信息，请参阅 CloudWatch 定价。

为 Lambda 函数配置高级日志记录控件

为了让您可以更好地控制如何捕获、处理和使用函数日志，Lambda 提供了以下日志配置选项：

- 日志格式 - 为函数日志选择纯文本或结构化的 JSON 格式
- 日志级别 - 对于 JSON 结构化日志，选择 Lambda 发送到 CloudWatch 的日志的详细信息级别，例如 ERROR、DEBUG 或 INFO
- 日志组 - 选择您的函数发送日志的目标 CloudWatch 日志组

要了解有关配置高级日志记录控件的更多信息，请参阅以下各节：

主题

- 配置 JSON 和纯文本日志格式
- 日志级别筛选
- 配置 CloudWatch 日志组

配置 JSON 和纯文本日志格式

将日志输出捕获为 JSON 键值对，可以更轻松地在调试函数时进行搜索和筛选。使用 JSON 格式的日志，您还可以在日志中添加标签和上下文信息。这可以帮助您对大量日志数据执行自动分析。除非您的开发工作流程依赖于使用纯文本格式 Lambda 日志的现有工具，否则我们建议您为日志格式选择 JSON。

对于所有 Lambda 托管的运行时系统，您可以选择将函数的系统日志以非结构化纯文本格式还是 JSON 格式发送到 CloudWatch Logs。系统日志是由 Lambda 生成的日志，有时也被称为平台事件日志。

对于支持的运行时系统，当您使用支持的内置日志记录方法之一时，Lambda 还能以结构化的 JSON 格式输出函数的应用程序日志（您的函数代码生成的日志）。当您为这些运行时系统配置函数的日志格式时，您选择的配置将同时应用于系统日志和应用程序日志。

对于支持的运行时系统，如果您的函数使用支持的日志库或方法，则无需对 Lambda 的现有代码进行任何更改，即可在结构化的 JSON 中捕获日志。

> **Note**
>
> 使用 JSON 日志格式可以添加额外的元数据，并将日志消息编码为包含一系列键值对的 JSON 对象。因此，函数日志消息的大小可能会增加。

支持的运行时系统和日志记录方法

Lambda 目前支持为以下运行时系统输出 JSON 结构化应用程序日志的选项。

运行时	支持的版本
Java	Amazon Linux 1 上除 Java 8 之外的所有 Java 运行时系统
Node.js	Node.js 16 及更高版本
Python	Python 3.8 及更高版本

为了让 Lambda 以结构化的 JSON 格式将函数的应用程序日志发送到 CloudWatch，您的函数必须使用以下内置日志记录工具来输出日志：

- Java - `LambdaLogger` 记录器或 Log4j2。
- Node.js - 控制台方法 `console.trace`、`console.debug`、`console.log`、`console.info`、`console.error` 和 `console.warn`
- Python - 标准 Python `logging` 库

有关在支持的运行时系统中使用高级日志记录控件的更多信息，请参阅 the section called “日志记录”、the section called “日志记录” 和 the section called “Python Lambda 函数日志记录和监控”。

对于其他托管 Lambda 运行时系统，Lambda 本身目前仅支持以结构化的 JSON 格式捕获系统日志。但是，您仍然可以在任何运行时系统中使用输出 JSON 格式日志输出的日志记录工具（例如 Powertools for AWS Lambda），以结构化的 JSON 格式捕获应用程序日志。

默认日志格式

目前，所有 Lambda 运行时系统的默认日志格式均为纯文本。

如果您已经在使用像 Powertools for AWS Lambda 这样的日志库来生成 JSON 结构化格式的函数日志，则在选择 JSON 日志格式后无需更改代码。Lambda 不会对任何已采用 JSON 编码的日志进行双重编码，因此仍将像以前一样捕获函数的应用程序日志。

系统日志的 JSON 格式

当您将函数的日志格式配置为 JSON 时，每个系统日志项（平台事件）都将被捕获为一个 JSON 对象，并且包含带有以下键的键值对：

- `"time"` - 生成日志消息的时间
- `"type"` - 正在记录的事件类型
- `"record"` - 日志输出的内容

`"record"` 值的格式因所记录的事件类型而异。有关更多信息，请参阅the section called “遥测 API Event 对象类型”。有关分配给系统日志事件的日志级别的更多信息，请参阅 the section called “系统日志级别的事件映射”。

为了进行比较，以下两个示例以纯文本和结构化的 JSON 格式显示了相同的日志输出。请注意，在大多数情况下，系统日志事件以 JSON 格式输出时所含的信息要多于以纯文本格式输出时所含的信息。

Example 纯文本：

```
2024-03-13 18:56:24.046000 fbe8c1   INIT_START  Runtime Version:
 python:3.12.v18  Runtime Version ARN: arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0
```

Example 结构化的 JSON：

```
{
  "time": "2024-03-13T18:56:24.046Z",
  "type": "platform.initStart",
  "record": {
    "initializationType": "on-demand",
    "phase": "init",
    "runtimeVersion": "python:3.12.v18",
    "runtimeVersionArn": "arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0"
```

```
  }
}
```

> **Note**
>
> the section called “遥测 API” 始终以 JSON 格式发出平台事件，例如 START 和 REPORT。配置 Lambda 发送到 CloudWatch 的系统日志的格式不会影响 Lambda 遥测 API 的行为。

应用程序日志的 JSON 格式

当您将函数的日志格式配置为 JSON 时，使用支持的日志记录库和方法编写的应用程序日志输出将被捕获为 JSON 对象，其中包含带有以下键的键值对。

- `"timestamp"` - 生成日志消息的时间
- `"level"` - 分配给消息的日志级别
- `"message"` - 日志消息的内容
- `"requestId"`（Python 和 Node.js）或 `"AWSrequestId"`（Java）- 函数调用的唯一请求 ID

根据函数使用的运行时系统和日志记录方法，此 JSON 对象还可能包含其他密钥对。例如，在 Node.js 中，如果函数通过 `console` 方法使用多个参数记录错误对象，则 JSON 对象将包含带有键 `errorMessage`、`errorType`、和 `stackTrace` 的额外键值对。要了解有关不同 Lambda 运行时系统中的 JSON 格式日志的更多信息，请参阅 the section called “Python Lambda 函数日志记录和监控”、the section called “日志记录” 和 the section called “日志记录”。

> **Note**
>
> 对于系统日志和应用程序日志，Lambda 用于时间戳值的键不同。对于系统日志，Lambda 使用密钥 `"time"` 来与遥测 API 保持一致。对于应用程序日志，Lambda 遵循支持的运行时系统并使用 `"timestamp"` 的惯例。

为了进行比较，以下两个示例以纯文本和结构化的 JSON 格式显示了相同的日志输出。

Example 纯文本：

```
2024-10-27T19:17:45.586Z 79b4f56e-95b1-4643-9700-2807f4e68189 INFO some log message
```

Example 结构化的 JSON：

```
{
    "timestamp":"2024-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"some log message",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

设置函数的日志格式

要配置函数的日志格式，您可以使用 Lambda 控制台或 AWS Command Line Interface（AWS CLI）。您还可以使用 CreateFunction 和 UpdateFunctionConfiguration Lambda API 命令、AWS Serverless Application Model（AWS SAM）AWS::Serverless::Function 资源以及 AWS CloudFormation AWS::Lambda::Function 资源来配置函数的日志格式。

更改函数的日志格式不会影响 CloudWatch Logs 中存储的现有日志。只有新的日志才会使用更新的格式。

如果您将函数的日志格式更改为 JSON 且未设置日志级别，则 Lambda 会自动将函数的应用程序日志级别和系统日志级别设置为 INFO。这意味着 Lambda 仅向 CloudWatch Logs 发送 INFO 及以下级别的日志输出。要了解有关应用程序和系统日志级别筛选的更多信息，请参阅 the section called “日志级别筛选”

> **Note**
>
> 对于 Python 运行时系统，当函数的日志格式设置为纯文本时，默认的日志级别设置为 WARN。这意味着 Lambda 仅向 CloudWatch Logs 发送 WARN 及以下级别的日志输出。将函数的日志格式更改为 JSON 可更改此默认行为。要了解有关 Python 中的日志记录的更多信息，请参阅 the section called “Python Lambda 函数日志记录和监控”。

对于发出嵌入式指标格式（EMF）日志的 Node.js 函数，将函数的日志格式更改为 JSON 可能会导致 CloudWatch 无法识别您的指标。

> **Important**
>
> 如果您的函数使用 Powertools for AWS Lambda（TypeScript）或开源的 EMF 客户端库来发出 EMF 日志，请将 Powertools 和 EMF 库更新到最新版本，以确保 CloudWatch 能够继续正

确解析您的日志。如果您切换到 JSON 日志格式，我们还建议您进行测试以确保与函数的嵌入式指标兼容。有关发出 EMF 日志的 node.js 函数的更多建议，请参阅 the section called “使用带有结构化的 JSON 日志的嵌入式指标格式（EMF）客户端库”。

配置函数的日志格式（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择函数
3. 在函数配置页面上，选择监控和操作工具。
4. 在日志记录配置窗格中，选择编辑。
5. 在日志内容下，为日志格式选择文本或 JSON。
6. 选择保存。

要更改现有函数的日志格式（AWS CLI）

- 要更改现有函数的日志格式，请使用 update-function-configuration 命令。将 `LoggingConfig` 中的 `LogFormat` 选项设置为 `JSON` 或 `Text`。

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON
```

要在创建函数时设置日志格式（AWS CLI）

- 要在创建新函数时配置日志格式，请使用 create-function 命令中的 `--logging-config` 选项。将 `LogFormat` 设置为 `JSON` 或 `Text`。以下示例命令创建 Node.js 函数，该函数以结构化 JSON 格式输出日志。

  如果您在创建函数时未指定日志格式，Lambda 将使用您选择的运行时系统版本的默认日志格式。有关默认日志记录格式的信息，请参阅 the section called “默认日志格式”。

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs20.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
```

```
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogFormat=JSON
```

日志级别筛选

Lambda 可以筛选函数的日志，以便仅将不超过特定详细信息级别的日志发送到 CloudWatch Logs。您可以分别为函数的系统日志（Lambda 生成的日志）和应用程序日志（您的函数代码生成的日志）单独配置日志级别筛选。

对于 the section called “支持的运行时系统和日志记录方法”，要筛选函数的应用程序日志，无需对 Lambda 的函数代码进行任何更改。

对于所有其他运行时系统和日志记录方法，您的函数代码必须将日志事件作为 JSON 格式的对象输出到 `stdout` 或 `stderr`，并在其中包含键 `"level"` 的键值对。例如，Lambda 将以下到 `stdout` 的输出解释为调试级别的日志。

```
print('{"level": "debug", "msg": "my debug log", "timestamp":
 "2024-11-02T16:51:31.587199Z"}')
```

如果 `"level"` 值字段无效或缺失，Lambda 将为日志输出分配 INFO 级别。要让 Lambda 使用时间戳字段，您必须以有效的 RFC 3339 时间戳格式指定时间。如果您未提供有效的时间戳，Lambda 将为日志分配 INFO 级别并为您添加时间戳。

命名时间戳键时，请遵循所用运行时系统的惯例。Lambda 支持托管运行时系统使用的大多数常用命名惯例。例如，在使用.NET 运行时系统的函数中，Lambda 会识别键 `"Timestamp"`。

> **Note**
>
> 要使用日志级别筛选，必须将您的函数配置为使用 JSON 日志格式。目前，所有 Lambda 托管运行时系统的默认日志格式均为纯文本。要了解如何将函数的日志格式配置为 JSON，请参阅 the section called “设置函数的日志格式”。

对于应用程序日志（由函数代码生成的日志），您可以从以下日志级别中进行选择。

日志级别	标准使用情况
TRACE（最详细）	用于跟踪代码执行路径的最精细信息

日志级别	标准使用情况
调试	系统调试的详细信息
信息	记录函数正常运行情况的消息
警告	有关潜在错误的消息，如果不加以解决，这些错误可能会导致意外行为
错误	有关会阻碍代码按预期执行的问题的消息
FATAL（最简略）	有关导致应用程序停止运行的严重错误的消息

当您选择日志级别后，Lambda 会将该级别及更低级别的日志发送到 CloudWatch Logs。例如，如果您将函数的应用程序日志级别设置为 WARN，则 Lambda 不会发送 INFO 和 DEBUG 级别的日志输出。日志筛选的默认应用程序日志级别为 INFO。

当 Lambda 筛选函数的应用程序日志时，将为没有级别的日志消息分配日志级别 INFO。

对于系统日志（由 Lambda 服务生成的日志），您可以在以下日志级别之间进行选择。

日志级别	使用量
DEBUG（最详细）	系统调试的详细信息
信息	记录函数正常运行情况的消息
WARN（最简略）	有关潜在错误的消息，如果不加以解决，这些错误可能会导致意外行为

当您选择日志级别后，Lambda 会发送该级别及更低级别的日志。例如，如果您将函数的系统日志级别设置为 INFO，则 Lambda 不会发送 DEBUG 级别的日志输出。

默认情况下，Lambda 会将系统日志级别设置为 INFO。使用此设置，Lambda 会自动向 CloudWatch 发送 `"start"` 和 `"report"` 日志消息。要接收不同详细程度的系统日志，请将日志级别更改为 DEBUG 或 WARN。要查看 Lambda 将不同系统日志事件映射到的日志级别列表，请参阅 the section called “系统日志级别的事件映射”。

配置日志级别筛选

要为您的函数配置应用程序和系统日志级别的筛选，您可以使用 Lambda 控制台或 AWS Command Line Interface（AWS CLI）。您还可以使用 CreateFunction 和 UpdateFunctionConfiguration Lambda API 命令、AWS Serverless Application Model（AWS SAM）AWS::Serverless::Function 资源以及 AWS CloudFormation AWS::Lambda::Function 资源来配置函数的日志级别。

请注意，如果在代码中设置函数的日志级别，则此设置将优先于您配置的任何其他日志级别设置。例如，如果您通过 Python `logging setLevel()` 方法将函数的日志级别设置为 INFO，则此设置将优先于您通过 Lambda 控制台配置的 WARN 设置。

配置现有函数的应用程序或系统日志级别（控制台）

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 在函数配置页面上，选择监控和操作工具。
4. 在日志记录配置窗格中，选择编辑。
5. 在日志内容下，确保为日志格式选择 JSON。
6. 使用单选按钮，为您的函数选择所需的应用程序日志级别和系统日志级别。
7. 选择保存。

配置现有函数的应用程序或系统日志级别（AWS CLI）

- 要更改现有函数的应用程序或系统日志级别，请使用 update-function-configuration 命令。使用 `--logging-config` 将 `SystemLogLevel` 设为 `DEBUG`、`INFO` 或 `WARN`。将 `ApplicationLogLevel` 设置为 `DEBUG`、`INFO`、`WARN`、`ERROR` 或 `FATAL` 之一。

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

在创建函数时配置日志级别筛选

- 要在创建新函数时配置日志级别筛选，请在 create-function 命令中使用 `--logging-config` 来设置 `SystemLogLevel` 和 `ApplicationLogLevel` 键。将 `SystemLogLevel` 设置为 `DEBUG`、`INFO` 或 `WARN` 之一。将 `ApplicationLogLevel` 设置为 `DEBUG`、`INFO`、`WARN`、`ERROR` 或 `FATAL` 之一。

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs20.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

系统日志级别的事件映射

对于 Lambda 生成的系统级日志事件，下表定义了分配给每个事件的日志级别。要了解有关表中所列事件的更多信息，请参阅 the section called “Event 架构参考”

事件名称	状况	分配的日志级别
initStart	已设置 runtimeVersion	信息
initStart	未设置 runtimeVersion	调试
initRuntimeDone	status=success	调试
initRuntimeDone	status!=success	警告
initReport	initializationType=snapstart	信息
initReport	initializationType!=snapstart	调试
initReport	status!=success	警告
restoreStart	已设置 runtimeVersion	信息
restoreStart	未设置 runtimeVersion	调试
restoreRuntimeDone	status=success	调试
restoreRuntimeDone	status!=success	警告
restoreReport	status=success	信息
restoreReport	status!=success	警告

事件名称	状况	分配的日志级别
开启	-	信息
runtimeDone	status=success	调试
runtimeDone	status!=success	警告
报告	status=success	信息
报告	status!=success	警告
extension	state=success	信息
extension	state!=success	警告
logSubscription	-	信息
telemetrySubscription	-	信息
logsDropped	-	警告

> **Note**
>
> the section called “遥测 API” 始终会发出一整套平台事件。配置 Lambda 发送到 CloudWatch 的系统日志的级别不会影响 Lambda 遥测 API 的行为。

使用自定义运行时系统进行应用程序日志级别筛选

当您为函数配置应用程序日志级别筛选时，Lambda 会在后台使用 `AWS_LAMBDA_LOG_LEVEL` 环境变量在运行时系统中设置应用程序日志级别。Lambda 还会使用 `AWS_LAMBDA_LOG_FORMAT` 环境变量设置函数的日志格式。您可以使用这些变量将 Lambda 高级日志记录控件集成到自定义运行时系统中。

要使用自定义运行时系统通过 Lambda 控制台、AWS CLI 和 Lambda API 为函数配置日志记录设置，请配置您的自定义运行时系统以检查这些环境变量的值。然后，您可以根据所选的日志格式和日志级别配置运行时系统的记录器。

配置 CloudWatch 日志组

默认情况下，CloudWatch 会在首次调用您的函数时为其自动创建一个名为 `/aws/lambda/<function name>` 的日志组。要将您的函数配置为向现有日志组发送日志，或者为函数创建新的日志组，可以使用 Lambda 控制台或 AWS CLI。您还可以使用 CreateFunction 和 UpdateFunctionConfiguration Lambda API 命令以及 AWS Serverless Application Model（AWS SAM）AWS::Serverless::Function 资源配置自定义日志组。

您可以配置多个 Lambda 函数向同一 CloudWatch 日志组发送日志。例如，您可以使用单个日志组来存储构成特定应用程序的所有 Lambda 函数的日志。当您为 Lambda 函数使用自定义日志组时，Lambda 创建的日志流包括函数名称和函数版本。这可以确保即使将同一个日志组用于多个函数，也能保留日志消息与函数之间的映射。

自定义日志组的日志流命名格式遵循以下惯例：

```
YYYY/MM/DD/<function_name>[<function_version>][<execution_environment_GUID>]
```

请注意，配置自定义日志组时，您为日志组选择的名称必须遵守 CloudWatch Logs 命名规则。此外，自定义日志组名称不得以字符串 `aws/` 开头。如果以 `aws/` 为开头创建自定义日志组，则 Lambda 将无法创建该日志组。这将导致函数的日志不会发送到 CloudWatch。

更改函数的日志组（控制台）

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 在函数配置页面上，选择监控和操作工具。
4. 在日志记录配置窗格中，选择编辑。
5. 在日志记录组窗格中，为 CloudWatch 日志组选择自定义。
6. 在自定义日志组下，输入您希望函数发送日志的目标 CloudWatch 日志组的名称。如果您输入现有日志组的名称，则函数将使用该组。如果不存在具有所输入名称的日志组，则 Lambda 将使用该名称为您的函数创建一个新的日志组。

更改函数的日志组（AWS CLI）

- 要更改现有函数的日志组，请使用 update-function-configuration 命令。

```
aws lambda update-function-configuration \
  --function-name myFunction \
```

```
  --logging-config LogGroup=myLogGroup
```

在创建函数时指定自定义日志组（AWS CLI）

- 要在使用 AWS CLI 创建新 Lambda 函数时指定自定义日志组，请使用 `--logging-config` 选项。以下示例命令可创建 Node.js Lambda 函数，用于将日志发送到名为 `myLogGroup` 的日志组。

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs20.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogGroup=myLogGroup
```

执行角色权限

您的函数必须拥有 logs:PutLogEvents 权限，才能向 CloudWatch Logs 发送日志。当您使用 Lambda 控制台配置函数的日志组时，如果您的函数没有此权限，Lambda 会默认将其添加到函数的执行角色。当 Lambda 添加此权限时，它会向该函数授予向任何 CloudWatch Logs 日志组发送日志的权限。

要防止 Lambda 自动更新函数的执行角色并改为手动对其进行编辑，请展开权限并取消选中添加所需权限。

当您使用 AWS CLI 配置函数的日志组时，Lambda 不会自动添加 `logs:PutLogEvents` 权限。如果函数的执行角色还没有该权限，请添加。此权限包含在 AWSLambdaBasicExecutionRole 托管策略中。

查看 Lambda 函数的 CloudWatch 日志

您可以使用 Lambda 控制台、CloudWatch 控制台或 AWS Command Line Interface（AWS CLI）查看 Lambda 函数的 Amazon CloudWatch Logs。按照以下各节中的说明访问函数的日志。

使用 CloudWatch Logs Live Tail 流式处理函数日志

Amazon CloudWatch Logs Live Tail 通过直接在 Lambda 控制台中显示新日志事件的流式列表，来帮助您快速对函数进行问题排查。您可以通过 Lambda 函数实时地查看和筛选提取的日志，帮助您快速检测并解决问题。

> **Note**
>
> Live Tail 会话按会话使用时间每分钟产生费用。有关定价的更多信息，请参阅 Amazon CloudWatch 定价。

比较 Live Tail 和 --log-type Tail

CloudWatch Logs Live Tail 和 Lambda API 中的 LogType: Tail 选项（AWS CLI 中的 `--log-type Tail`）之间有多项差异：

- `--log-type Tail` 仅返回调用日志的前 4 KB。Live Tail 不具备此限制，并且每秒最多可以接收 500 个日志事件。
- `--log-type Tail` 捕获并发送带有响应的日志，这可能会影响函数的响应延迟。Live Tail 不会影响函数响应延迟。
- `--log-type Tail` 仅支持同步调用。Live Tail 适用于同步和异步调用。

权限

启动和停止 CloudWatch Logs Live Tail 会话需要以下权限：

- `logs:DescribeLogGroups`
- `logs:StartLiveTail`
- `logs:StopLiveTail`

在 Lambda 控制台中启动 Live Tail 会话

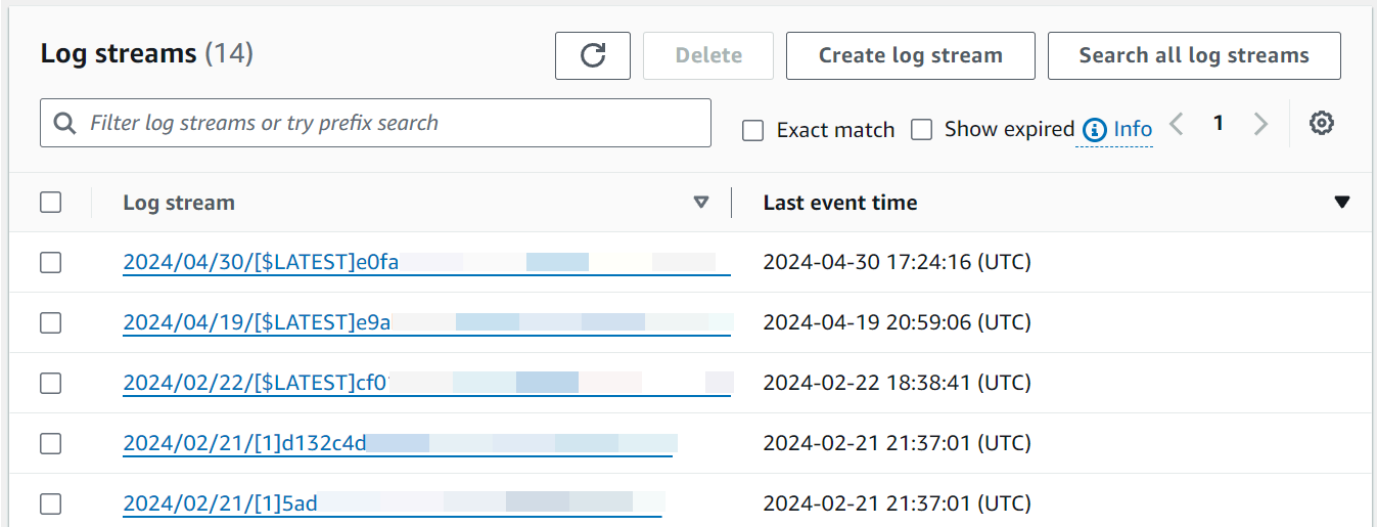
1. 打开 Lamba 控制台的函数页面。
2. 选择此函数的名称。
3. 选择测试选项卡。
4. 在测试事件窗格中，选择 CloudWatch Logs Live Tail。
5. 对于选择日志组，默认情况下会选择该函数的日志组。一次最多可选择 5 个日志组。
6. （可选）要仅显示包含某些单词或其他字符串的日志事件，请在添加筛选模式框中输入词或字符串。筛选字段不区分大小写。您可以在此字段中包含多个术语和模式运算符，包含正则表达式（regex）。有关更多信息，请参阅《Amazon CloudWatch Logs User Guide》中的 Filter pattern syntax。

7. 选择启动。匹配的日志事件开始出现在窗口中。
8. 要停止 Live Tail 会话，请选择停止。

> **Note**
>
> Live Tail 会话会在处于非活动状态 15 分钟后或 Lambda 控制台会话超时后自动停止。

使用控制台访问函数日志

1. 打开 Lamba 控制台的函数页面。
2. 选择一个函数。
3. 选择监控选项卡。
4. 选择查看 CloudWatch 日志打开 CloudWatch 控制台。
5. 向下滚动，选择要查看的函数调用的日志流。



使用 AWS CLI 访问日志

AWS CLI 是一种开源工具，让您能够在命令行 Shell 中使用命令与 AWS 服务进行交互。要完成本节中的步骤，您必须拥有 AWS CLI 版本 2。

您可以通过 AWS CLI，使用 `--log-type` 命令选项检索调用的日志。响应包含一个 `LogResult` 字段，其中包含多达 4KB 来自调用的 base64 编码日志。

Example 检索日志 ID

以下示例说明如何从 LogResult 字段中检索名为 my-function 的函数的日志 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example 解码日志

在同一命令提示符下，使用 base64 实用程序解码日志。以下示例说明如何为 my-function 检索 base64 编码的日志。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

您应看到以下输出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64 实用程序在 Linux、macOS 和 Ubuntu on Windows 上可用。macOS 用户可能需要使用 base64 -D。

Example get-logs.sh 脚本

在同一命令提示符下，使用以下脚本下载最后五个日志事件。此脚本使用 sed 从输出文件中删除引号，并休眠 15 秒以等待日志可用。输出包括来自 Lambda 的响应，以及来自 get-log-events 命令的输出。

复制以下代码示例的内容并将其作为 get-logs.sh 保存在 Lambda 项目目录中。

如果使用 cli-binary-format 版本 2，则 AWS CLI 选项是必需的。要将其设为默认设置，请运行 aws configure set cli-binary-format raw-in-base64-out。有关更多信息，请参阅版本 2 的 AWS Command Line Interface 用户指南中的 AWS CLI 支持的全局命令行选项。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux（仅限）

在同一命令提示符下，macOS 和 Linux 用户可能需要运行以下命令以确保脚本可执行。

```
chmod -R 755 get-logs.sh
```

Example 检索最后五个日志事件

在同一命令提示符下，运行以下脚本以获取最后五个日志事件。

```
./get-logs.sh
```

您应看到以下输出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
```

```
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

使用 AWS CloudTrail 记录 AWS Lambda API 调用

AWS Lambda 与 AWS CloudTrail 集成，后者是提供用户、角色或 AWS 服务 所执行操作记录的服务。CloudTrail 将 Lambda 的 API 调用捕获为事件。捕获的调用包括来自 Lambda 控制台的调用和对 Lambda API 操作的代码调用。借助通过 CloudTrail 收集的信息，您可以确定向 Lambda 发出哪些请求、发出请求的 IP 地址、请求的发出时间以及其他详细信息。

每个事件或日记账条目都包含有关生成请求的人员信息。身份信息有助于您确定以下内容：

- 请求是使用根用户凭证还是用户凭证发出的。
- 请求是否代表 IAM Identity Center 用户发出。
- 请求是使用角色还是联合用户的临时安全凭证发出的。
- 请求是否由其他 AWS 服务 发出。

当您创建账户并可以自动访问 CloudTrail 事件历史记录时，CloudTrail 在您的 AWS 账户 中处于活动状态。CloudTrail 事件历史记录提供对 AWS 区域 中过去 90 天的已记录管理事件的可查看、可搜索、可下载和不可变记录。有关更多信息，请参见《AWS CloudTrail 用户指南》的 使用 CloudTrail 事件历史记录。查看事件历史记录不会收取 CloudTrail 费用。

要持续记录您的 AWS 账户 过去 90 天的事件，请创建跟踪或 CloudTrail Lake 事件数据存储。

CloudTrail 跟踪

通过跟踪记录，CloudTrail 可将日志文件传送至 Simple Storage Service (Amazon S3) 存储桶。使用 AWS Management Console 创建的所有跟踪均具有多区域属性。您可以通过使用 AWS CLI 创建单区域或多区域跟踪。建议创建多区域跟踪，因为您可记录您账户中的所有 AWS 区域 的活动。如果您创建单区域跟踪，则只能查看跟踪的 AWS 区域 中记录的事件。有关跟踪的更多信息，请参阅《AWS CloudTrail 用户指南》中的为您的 AWS 账户 创建跟踪和为组织创建跟踪。

通过创建跟踪，您可以从 CloudTrail 免费向您的 Amazon S3 存储桶传送一份正在进行的管理事件的副本，但会收取 Amazon S3 存储费用。有关 CloudTrail 定价的更多信息，请参阅 AWS CloudTrail 定价。有关 Amazon S3 定价的信息，请参阅 Amazon S3 定价。

CloudTrail Lake 事件数据存储

CloudTrail Lake 允许您对事件运行基于 SQL 的查询。CloudTrail Lake 可将基于行的 JSON 格式的现有事件转换为 Apache ORC 格式。ORC 是一种针对快速检索数据进行优化的列式存储格式。事件将被聚合到事件数据存储中，它是基于您通过应用高级事件选择器选择的条件的不可变的事件集

合。应用于事件数据存储的选择器用于控制哪些事件持续存在并可供您查询。有关 CloudTrail Lake 的更多信息，请参阅《AWS CloudTrail 用户指南》中的使用 AWS CloudTrail Lake。

CloudTrail Lake 事件数据存储和查询会产生费用。创建事件数据存储时，您可以选择要用于事件数据存储的定价选项。定价选项决定了摄取和存储事件的成本，以及事件数据存储的默认和最长保留期。有关 CloudTrail 定价的更多信息，请参阅 AWS CloudTrail 定价。

CloudTrail 中的 Lambda 数据事件

数据事件可提供对资源或在资源中所执行资源操作（例如，读取或写入 Amazon S3 对象）的相关信息。这些也称为数据层面操作。数据事件通常是高容量活动。默认情况下，CloudTrail 不记录大多数数据事件，CloudTrail 事件历史记录也不会记录这些事件。

默认情况下，为支持的服务记录的一个 CloudTrail 数据事件是 `LambdaESMDisabled`。要详细了解如何使用此事件来帮助排查 Lambda 事件源映射的问题，请参阅 the section called “使用 CloudTrail 排查已禁用的 Lambda 事件源的问题”。

记录数据事件将收取额外费用。有关 CloudTrail 定价的更多信息，请参阅 AWS CloudTrail 定价。

您可以使用 CloudTrail 控制台、AWS CLI 或 CloudTrail API 操作记录 `AWS::Lambda::Function` 资源类型的数据事件。有关如何记录数据事件的更多信息，请参阅《AWS CloudTrail 用户指南》中的使用 AWS Management Console 记录数据事件和使用 AWS Command Line Interface 记录数据事件。

下表列出了您可以为其记录数据事件的 Lambda 资源类型。数据事件类型（控制台）列显示可从 CloudTrail 控制台上的数据事件类型列表中选择的值。resources.type 值列显示了您在使用 AWS CLI 或 CloudTrail API 配置高级事件选择器时需要指定的 `resources.type` 值。记录到 CloudTrail 的数据 API 列显示了针对该资源类型记录到 CloudTrail 的 API 调用。

数据事件类型（控制台）	resources.type 值	记录至 CloudTrail 的数据 API
Lambda	`AWS::Lambda::Function`	Invoke

您可以将高级事件选择器配置为在 `eventName`、`readOnly` 和 `resources.ARN` 字段上进行筛选，从而仅记录那些对您很重要的事件。以下示例是数据事件配置的 JSON 视图，该视图仅记录特定函数的事件。有关这些字段的更多信息，请参阅《AWS CloudTrail API 参考》中的 AdvancedFieldSelector。

```
[
  {
    "name": "function-invokes",
    "fieldSelectors": [
      {
        "field": "eventCategory",
        "equals": [
          "Data"
        ]
      },
      {
        "field": "resources.type",
        "equals": [
          "AWS::Lambda::Function"
        ]
      },
      {
        "field": "resources.ARN",
        "equals": [
          "arn:aws:lambda:us-east-1:111122223333:function:hello-world"
        ]
      }
    ]
  }
]
```

CloudTrail 中的 Lambda 管理事件

管理事件提供对您 AWS 账户 内的资源所执行管理操作的相关信息。这些也称为控制层面操作。原定设置情况下，CloudTrail 会记录管理事件。

Lambda 支持将以下操作记录为 CloudTrail 日志文件中的管理事件。

> **Note**
>
> 在 CloudTrail 日志文件中，`eventName` 可能包括日期和版本信息，但它仍在引用同一公有 API 操作。例如，`GetFunction` 操作显示为 `GetFunction20150331v2`。以下列表指定事件名称与 API 操作名称不同的时间。

- AddLayerVersionPermission

- AddPermission（事件名称：AddPermission20150331v2）
- CreateAlias（事件名称：CreateAlias20150331）
- CreateEventSourceMapping（事件名称：CreateEventSourceMapping20150331）
- CreateFunction（事件名称：CreateFunction20150331）

  （CreateFunction 的 CloudTrail 日志省略了 Environment 和 ZipFile 参数。）
- CreateFunctionUrlConfig
- DeleteAlias（事件名称：DeleteAlias20150331）
- DeleteCodeSigningConfig
- DeleteEventSourceMapping（事件名称：DeleteEventSourceMapping20150331）
- DeleteFunction（事件名称：DeleteFunction20150331）
- DeleteFunctionConcurrency（事件名称：DeleteFunctionConcurrency20171031）
- DeleteFunctionUrlConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias（事件名称：GetAlias20150331）
- GetEventSourceMapping
- GetFunction
- GetFunctionUrlConfig
- GetFunctionConfiguration
- GetLayerVersionPolicy
- GetPolicy
- ListEventSourceMappings
- ListFunctions
- ListFunctionUrlConfigs
- PublishLayerVersion（事件名称：PublishLayerVersion20181031）

  （PublishLayerVersion 的 CloudTrail 日志省略了 ZipFile 参数。）
- PublishVersion（事件名称：PublishVersion20150331）
- PutFunctionConcurrency（事件名称：PutFunctionConcurrency20171031）
- PutFunctionCodeSigningConfig
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig

- PutRuntimeManagementConfig
- RemovePermission（事件名称：RemovePermission20150331v2）
- TagResource（事件名称：TagResource20170331v2）
- UntagResource（事件名称：UntagResource20170331v2）
- UpdateAlias（事件名称：UpdateAlias20150331）
- UpdateCodeSigningConfig
- UpdateEventSourceMapping（事件名称：UpdateEventSourceMapping20150331）
- UpdateFunctionCode（事件名称：UpdateFunctionCode20150331v2）

  （UpdateFunctionCode 的 CloudTrail 日志省略了 ZipFile 参数。）
- UpdateFunctionConfiguration（事件名称：UpdateFunctionConfiguration20150331v2）

  （UpdateFunctionConfiguration 的 CloudTrail 日志省略了 Environment 参数。）
- UpdateFunctionEventInvokeConfig
- UpdateFunctionUrlConfig

使用 CloudTrail 排查已禁用的 Lambda 事件源的问题

当您使用 UpdateEventSourceMapping API 操作更改事件源映射的状态时，API 调用将作为管理事件记录在 CloudTrail 中。事件源映射还可能由于错误而直接过渡到 Disabled 状态。

对于以下服务，当您的事件源转换为“已禁用”状态时，Lambda 会将 LambdaESMDisabled 数据事件发布到 CloudTrail：

- Amazon Simple Queue Service(Amazon SQS)
- Amazon DynamoDB
- Amazon Kinesis

Lambda 不支持任何其他事件源映射类型的此事件。

要在支持服务的事件源映射转换到 Disabled 状态时收到警报，请使用 LambdaESMDisabled CloudTrail 事件在 Amazon CloudWatch 中设置告警。要了解有关设置 CloudWatch 告警的更多信息，请参阅为 CloudTrail 事件创建 CloudWatch 告警：示例。

LambdaESMDisabled 事件消息中的 serviceEventDetails 实体包含以下错误代码之一。

RESOURCE_NOT_FOUND

请求中指定的资源不存在。

FUNCTION_NOT_FOUND

附加到事件源的函数不存在。

REGION_NAME_NOT_VALID

提供给事件源或函数的区域名称无效。

AUTHORIZATION_ERROR

尚未设置权限或权限配置错误。

FUNCTION_IN_FAILED_STATE

函数代码无法编译或遇到了无法恢复的异常，或者发生了部署错误。

Lambda 事件示例

一个事件表示一个来自任何源的请求，包括有关所请求的 API 操作、操作的日期和时间、请求参数等方面的信息。CloudTrail 日志文件不是公用 API 调用的有序堆栈跟踪，因此事件不会按任何特定顺序显示。

以下示例显示了 `GetFunction` 和 `DeleteFunction` 操作的 CloudTrail 日志条目。

> **Note**
>
> `eventName` 可能包括日期和版本信息（如 `"GetFunction20150331"`），但它仍在引用同一公有 API。

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
```

```
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:03:36Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "GetFunction",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "errorCode": "AccessDenied",
      "errorMessage": "User: arn:aws:iam::111122223333:user/myUserName is not
 authorized to perform: lambda:GetFunction on resource: arn:aws:lambda:us-
west-2:111122223333:function:other-acct-function",
      "requestParameters": null,
      "responseElements": null,
      "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    },
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:04:42Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "DeleteFunction20150331",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "requestParameters": {
        "functionName": "basic-node-task"
      },
      "responseElements": null,
      "requestID": "a2198ecc-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "20b84ce5-730f-482e-b2b2-e8fcc87ceb22",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    }
```

```
    ]
}
```

有关 CloudTrail 记录内容的信息，请参阅《AWS CloudTrail 用户指南》中的 CloudTrail 记录内容。

使用 AWS X-Ray 可视化 Lambda 函数调用

您可以使用 AWS X-Ray 来可视化应用程序的组件、确定性能瓶颈以及对导致错误的请求进行故障排除。您的 Lambda 函数会将跟踪数据发送到 X-Ray，X-Ray 将处理这些数据以生成服务映射和可搜索的跟踪摘要。

如果您在调用函数的服务中启用了 X-Ray 跟踪，则 Lambda 会自动将跟踪发送到 X-Ray。上游服务（例如 Amazon API Gateway）或通过 X-Ray 开发工具包检测的托管于 Amazon EC2 上的应用程序将对传入请求进行采样，并添加跟踪头来指示 Lambda 是否发送跟踪。上游消息创建者（例如 Amazon SQS）的跟踪会自动链接到下游 Lambda 函数的跟踪，从而创建整个应用程序的端到端视图。有关更多信息，请参阅《AWS X-Ray 开发人员指南》中的 Tracing event-driven applications（跟踪事件驱动型应用程序）。

> **Note**
>
> 目前，使用 Amazon Managed Streaming for Apache Kafka（Amazon MSK）、自行管理的 Apache Kafka、具有 ActiveMQ 及 RabbitMQ 或 Amazon DocumentDB 事件源映射的 Amazon MQ 的 Lambda 函数，不支持 X-Ray 跟踪。

要使用控制台切换 Lambda 函数的活动跟踪，请按照以下步骤操作：

打开活跃跟踪

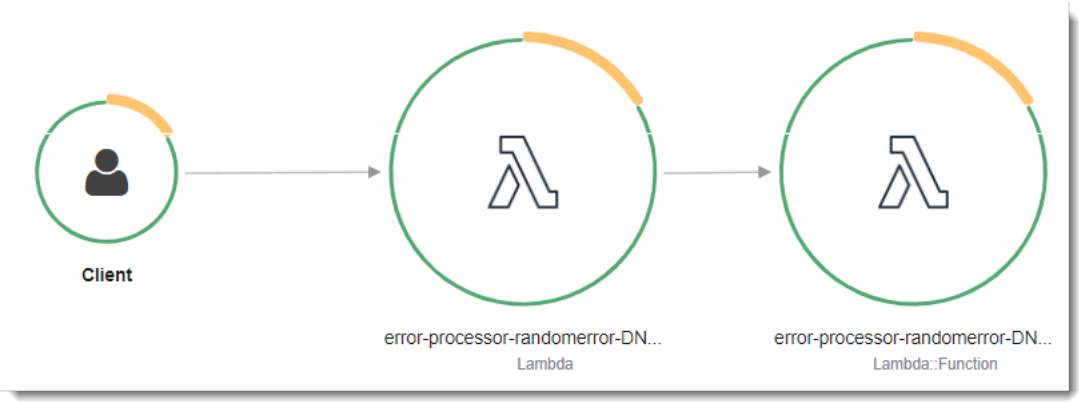
1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Configuration（配置），然后选择 Monitoring and operations tools（监控和操作工具）。
4. 选择编辑。
5. 在 X-Ray 下方，开启 Active tracing（活动跟踪）。
6. 选择保存。

您的函数需要权限才能将跟踪数据上载到 X-Ray。在 Lambda 控制台中激活跟踪后，Lambda 会将所需权限添加到函数的执行角色。如果没有，请将 AWSXRayDaemonWriteAccess 策略添加到执行角色。

X-Ray 无法跟踪对应用程序的所有请求。X-Ray 将应用采样算法确保跟踪有效，同时仍会提供所有请求的一个代表性样本。采样率是每秒 1 个请求和 5% 的其他请求。您无法为函数配置此 X-Ray 采样率。

了解 X-Ray 跟踪

在 X-Ray 中，跟踪记录有关由一个或多个服务处理的请求的信息。Lambda 会每个跟踪记录 2 个分段，这些分段将在服务图上创建两个节点。下图突出显示了这两个节点：



位于左侧的第一个节点表示接收调用请求的 Lambda 服务。第二个节点表示特定的 Lambda 函数。

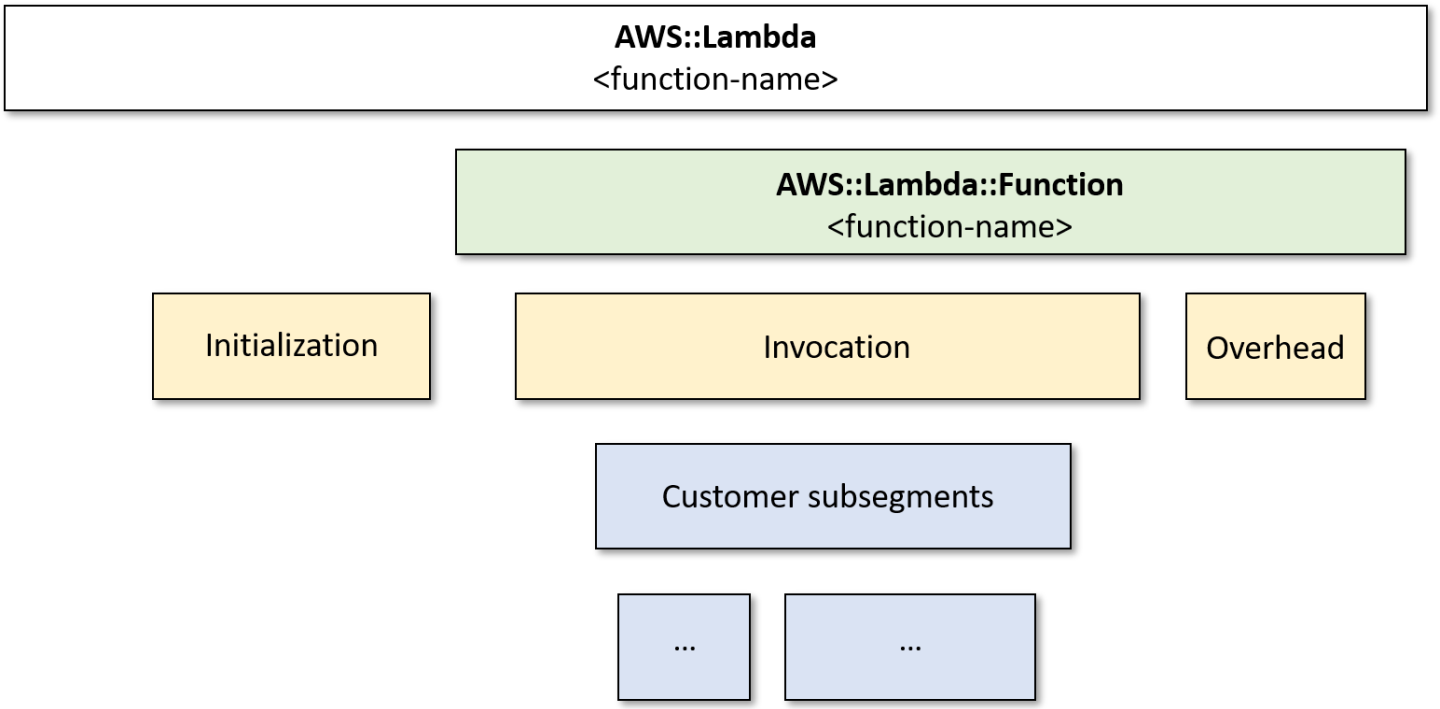
针对 Lambda 服务记录的分段 `AWS::Lambda` 涵盖了准备 Lambda 执行环境所需的所有步骤。这包括调度 MicroVM、使用已配置的资源创建或解冻执行环境、下载函数代码和所有层。

`AWS::Lambda::Function` 分段表示函数完成的工作。

> **Note**
>
> AWS 目前正在实施对 Lambda 服务的更改。由于这些更改，您可能会看到 AWS 账户 中不同 Lambda 函数发出的系统日志消息和跟踪分段的结构和内容之间存在细微差异。
> 此更改会影响函数段的子分段。以下段落介绍了这些子分段的新旧格式。
> 这些更改将在未来几周内实施，除中国和 GovCloud 区域外，所有 AWS 区域 的函数都将过渡到使用新格式的日志消息和跟踪分段。

旧式 AWS X-Ray Lambda 分段结构

`AWS::Lambda` 分段的旧式 X-Ray 结构如下所示：



在此格式中，函数分段包含 `Initialization`、`Invocation` 和 `Overhead` 的子分段。（仅限 Lambda SnapStart）还有一个 `Restore` 子分段（此示意图中未显示）。

该 `Initialization` 子分段表示 Lambda 执行环境生命周期的初始化阶段。在此阶段，Lambda 将初始化扩展、初始化运行时并运行函数的初始化代码。

`Invocation` 子段表示 Lambda 调用函数处理程序的调用阶段。这从运行时和扩展注册开始，在运行时准备好发送响应时结束。

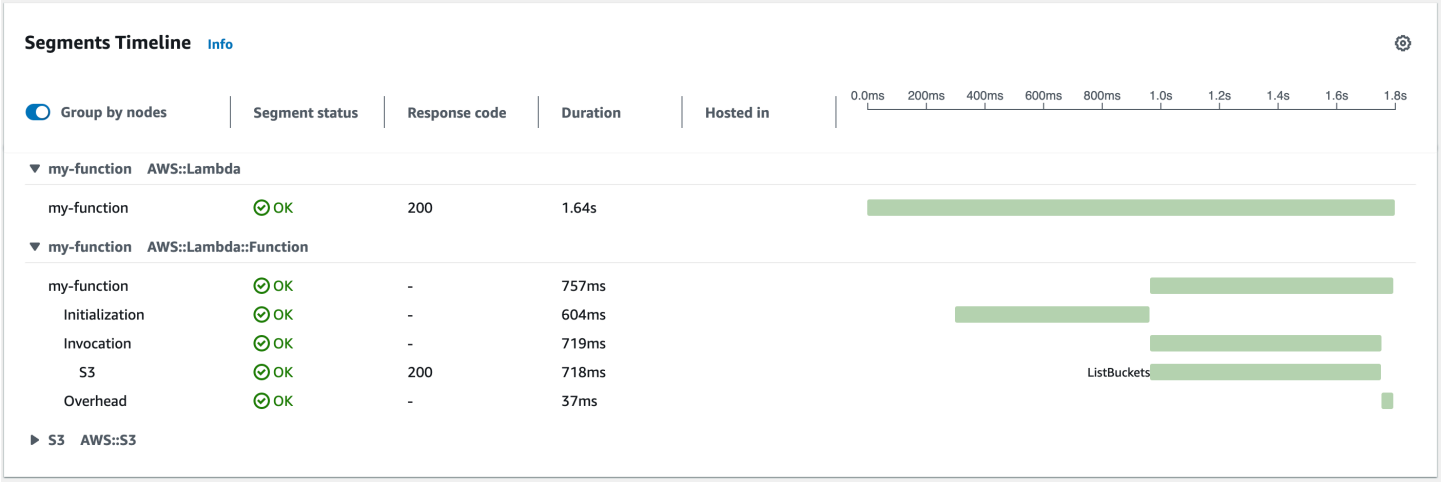
（仅限 Lambda SnapStart）`Restore` 子分段会显示 Lambda 还原快照、加载运行时（JVM）和运行任何 `afterRestore` 运行时钩子所用的时间。恢复快照的过程可能包含在 MicroVM 之外的活动上花费的时间。该时间在 `Restore` 子分段中报告。您无需为在 microVM 之外还原快照所花费的时间付费。

`Overhead` 子段表示在运行时发送响应和下次调用信号之间出现的阶段。在此期间，运行时会完成与调用相关的所有任务，并准备冻结沙盒。

> **Important**
>
> 您可以使用 X-Ray SDK 扩展 `Invocation` 子分段以及进行下游调用、注释和元数据的额外子分段。您无法直接访问函数分段，也无法记录在处理程序调用范围之外完成的工作。

有关 Lambda 执行环境阶段的更多信息，请参阅 执行环境。

该示意图显示了使用旧式 X-Ray 结构的示例跟踪。

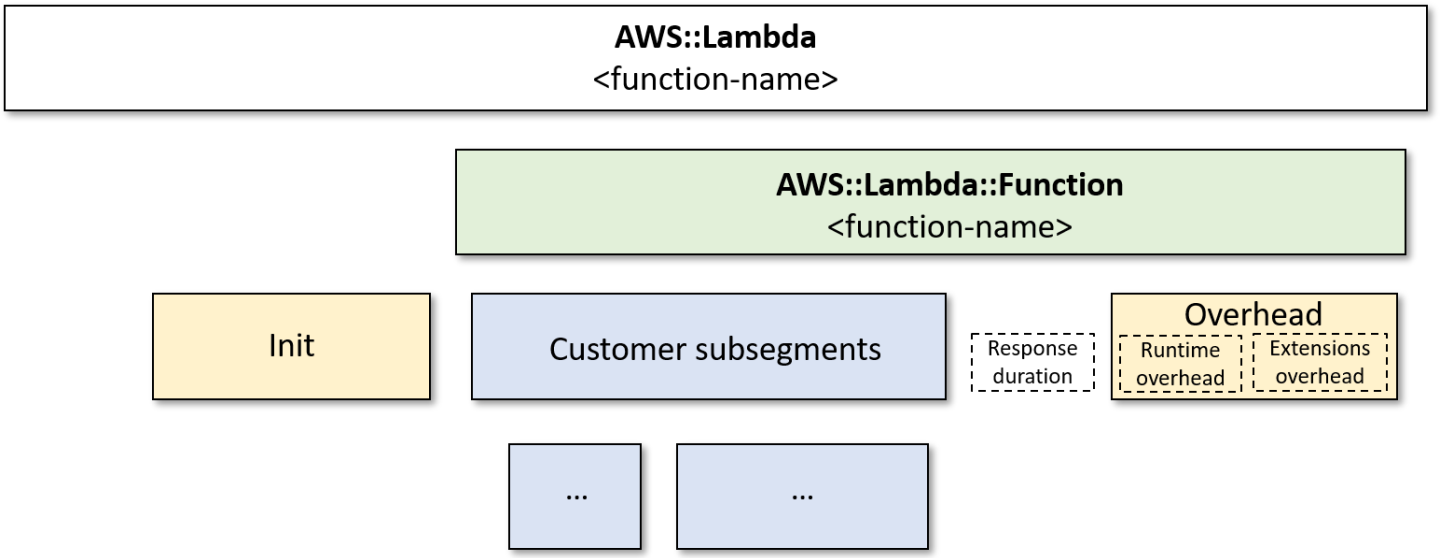


注意示例中的两个分段。两者都命名为 my-function，但其中一个函数具有 AWS::Lambda 源，另一个则具有 AWS::Lambda::Function 源。如果 AWS::Lambda 分段显示错误，则表示 Lambda 服务存在问题。如果 AWS::Lambda::Function 分段显示错误，则说明函数存在问题。

> **Note**
>
> 有时，您可能会注意到 X-Ray 跟踪中的函数初始化阶段与调用阶段之间存在较大间隔。对于使用预置并发的函数，这是因为 Lambda 会在调用之前尽早初始化函数实例。对于使用非预留（按需）并发的函数，即使没有调用，Lambda 也可能会主动初始化函数实例。直观上看，这两种情况都表现为初始化阶段与调用阶段之间的时间间隔。

新式 AWS X-Ray Lambda 分段结构

AWS::Lambda 分段的新式 X-Ray 结构如下所示：

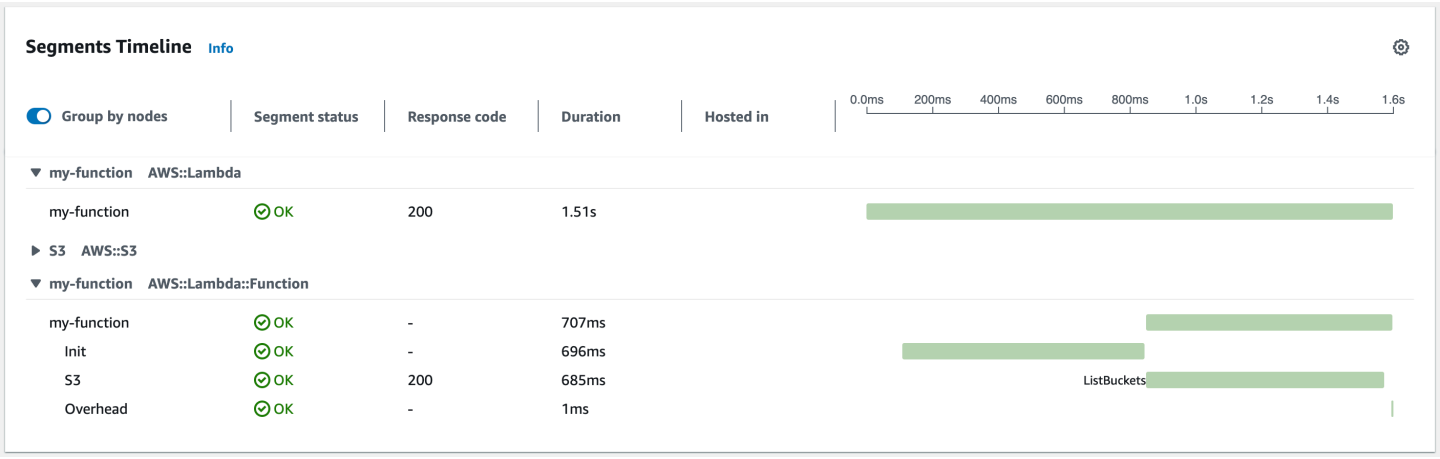


在此新格式中，`Init` 子分段像以前一样表示 Lambda 执行环境生命周期的初始化阶段。

新格式中没有调用分段。而是将客户子分段直接附加到 `AWS::Lambda::Function` 分段 此分段包含以下指标作为注释：

- `aws.responseLatency`：函数运行所用的时间
- `aws.responseDuration`：将响应传递给客户所用的时间
- `aws.runtimeOverhead`：运行时完成所需的额外时间
- `aws.extensionOverhead`：扩展完成所需的额外时间

该示意图显示了使用新式 X-Ray 结构的示例跟踪。



注意示例中的两个分段。两者都命名为 my-function，但其中一个函数具有 AWS::Lambda 源，另一个则具有 AWS::Lambda::Function 源。如果 AWS::Lambda 分段显示错误，则表示 Lambda 服务存在问题。如果 AWS::Lambda::Function 分段显示错误，则说明函数存在问题。

有关 Lambda 中特定于语言的跟踪简介，请参阅以下主题：

- 在 AWS Lambda 中检测 Node.js 代码
- 在 AWS Lambda 中检测 Python 代码
- 在 AWS Lambda 中检测 Ruby 代码
- 在 AWS Lambda 中检测 Java 代码
- 在 AWS Lambda 中检测 Go 代码
- 在 AWS Lambda 中检测 C# 代码

有关支持活动检测的服务的完整列表，请参阅 AWS X-Ray 开发人员指南中的受支持的AWS服务。

执行角色权限

Lambda 需要以下权限才能将跟踪数据发送到 X-Ray。将这些权限添加到您的函数的执行角色中。

- xray:PutTraceSegments
- xray:PutTelemetryRecords

这些权限包含在 AWSXRayDaemonWriteAccess 托管策略中。

AWS X-Ray 进程守护程序

X-Ray 开发工具包使用进程守护程序，而不是直接向 X-Ray API 发送跟踪数据。AWS X-Ray 进程守护程序是在 Lambda 环境中运行，并侦听包含分段和子分段的 UDP 流量的应用程序。它缓冲传入的数据并将其分批写入 X-Ray，从而减少跟踪调用所需的处理和内存开销。

Lambda 运行时允许进程守护程序最多占用函数配置内存的 3% 或 16 MB（以较大者为准）。如果您的函数在调用期间内存不足，则运行时首先终止进程守护程序以释放内存。

进程守护程序完全由 Lambda 管理，用户无法配置。通过调用函数生成的所有分段都记录在与 Lambda 函数相同的账户中。无法将进程守护程序配置为将这些分段重定向到任何其他账户。

有关更多信息，请参阅 X-Ray 开发人员指南中的 X-Ray 进程守护程序。

使用 Lambda API 启用活动跟踪

要使用 AWS CLI 或 AWS 开发工具包管理跟踪配置，请使用以下 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下示例 AWS CLI 命令对名为 my-function 的函数启用活跃跟踪。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

跟踪模式是发布函数版本时版本特定配置的一部分。您无法更改已发布版本上的跟踪模式。

使用 AWS CloudFormation 启用主动跟踪

要对 AWS CloudFormation 模板中的 `AWS::Lambda::Function` 资源激活跟踪，请使用 `TracingConfig` 属性。

Example function-inline.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

对于 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 资源，请使用 `Tracing` 属性。

Example template.yml – 跟踪配置

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
```

```
      Tracing: Active
      ...
```

使用 Amazon CloudWatch Lambda 洞察监控函数性能

Amazon CloudWatch Lambda Insights 收集和聚合无服务器应用程序的 Lambda 函数运行时性能指标和日志。本页将介绍如何启用和使用 Lambda Insights 来诊断 Lambda 函数的问题。

小节目录

Lambda Insights 如何监视无服务器应用程序

CloudWatch Lambda Insights 是针对在 AWS Lambda 上运行的无服务器应用程序的监控和故障排除解决方案。该解决方案收集、聚合和汇总系统级指标，包括 CPU 时间、内存、磁盘和网络使用情况。它还收集、聚合和汇总诊断信息（如冷启动和 Lambda 工件关闭），以帮助您隔离 Lambda 函数的问题并快速解决这些问题。

Lambda Insights 使用作为 Lambda 层提供的全新 CloudWatch Lambda Insights 扩展。在 Lambda 函数上为支持的运行时启用此扩展时，它收集系统级指标，并为每次调用该 Lambda 函数发出一个性能日志事件。CloudWatch 使用嵌入式指标格式从日志事件中提取指标。有关更多信息，请参阅使用 AWS Lambda 扩展。

Lambda Insights 层扩展 `/aws/lambda-insights/` 日志组的 `CreateLogStream` 和 `PutLogEvents`。

定价

当您为 Lambda 函数启用 Lambda Insights 时，Lambda Insights 会为每个函数报告 8 个指标，每个函数调用都会向 CloudWatch 发送约 1KB 的日志数据。您只需为 Lambda Insights 报告的函数指标和日

志付费。无最低费用或强制性服务使用政策。如果未调用该函数，则不需要为 Lambda Insights 付费。有关定价示例，请参阅 Amazon CloudWatch 定价。

支持的运行时

您可以将 Lambda Insights 与支持 Lambda 扩展的任何运行时一起使用。

在 Lambda 控制台中启用 Lambda Insights

您可以对新函数和现有 Lambda 函数启用 Lambda Insights 增强监控功能。在 Lambda 控制台中为支持的运行时针对函数启用 Lambda Insights 时，Lambda 将 Lambda Insights 扩展作为层添加到函数中，并进行验证或尝试以将 `CloudWatchLambdaInsightsExecutionRolePolicy` 策略附加到函数的执行角色。

在 Lambda 控制台中启用 Lambda Insights

1. 打开 Lamba 控制台的函数页面。
2. 选择您的函数。
3. 选择 Configuration 选项卡。
4. 在左侧菜单中，选择监控和操作工具。
5. 在其他监控工具窗格中，选择编辑。
6. 在 CloudWatch Lambda Insights 下，启用增强监控。
7. 选择保存。

以编程方式启用 Lambda Insights

还可以使用 AWS Command Line Interface (AWS CLI)、AWS Serverless Application Model (SAM) CLI、AWS CloudFormation 或 AWS Cloud Development Kit (AWS CDK) 启用 Lambda Insights。以编程方式在函数上为支持的运行时启用 Lambda Insights 时，CloudWatch 将 `CloudWatchLambdaInsightsExecutionRolePolicy` 策略附加到函数的 执行角色。

有关更多信息，请参阅 Amazon CloudWatch 用户指南中的 Lambda Insights 入门。

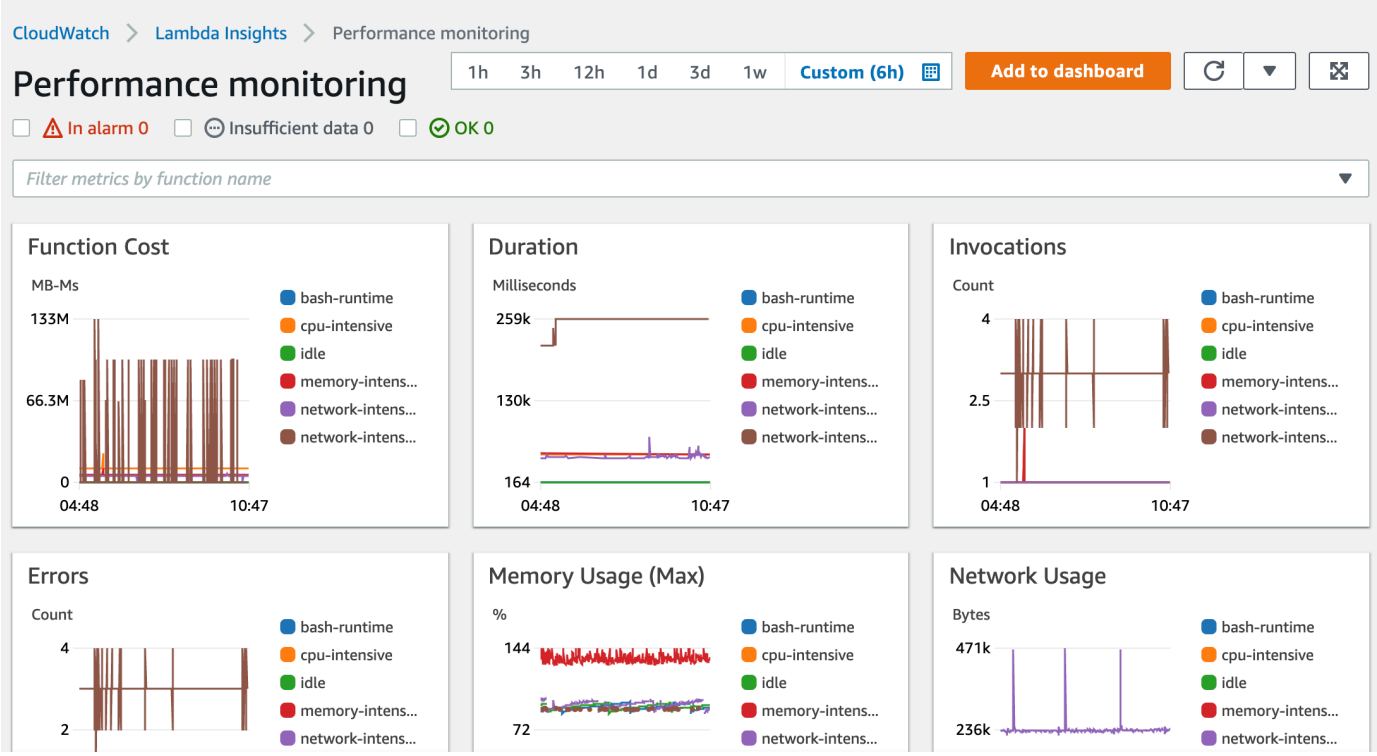
使用 Lambda Insights 控制面板

在 CloudWatch 控制台中，Lambda Insights 控制面板有两个视图：多函数概览和单函数视图。多函数概览聚合当前AWS账户和区域中 Lambda 函数的运行时指标。单函数视图显示单个 Lambda 函数的可用运行时指标。

您可以使用 CloudWatch 控制台中的 Lambda Insights 控制面板多函数概览来识别过度利用和未充分利用的 Lambda 函数。您可以使用 CloudWatch 控制台中的 Lambda Insights 控制面板单函数视图对单个请求进行故障排除。

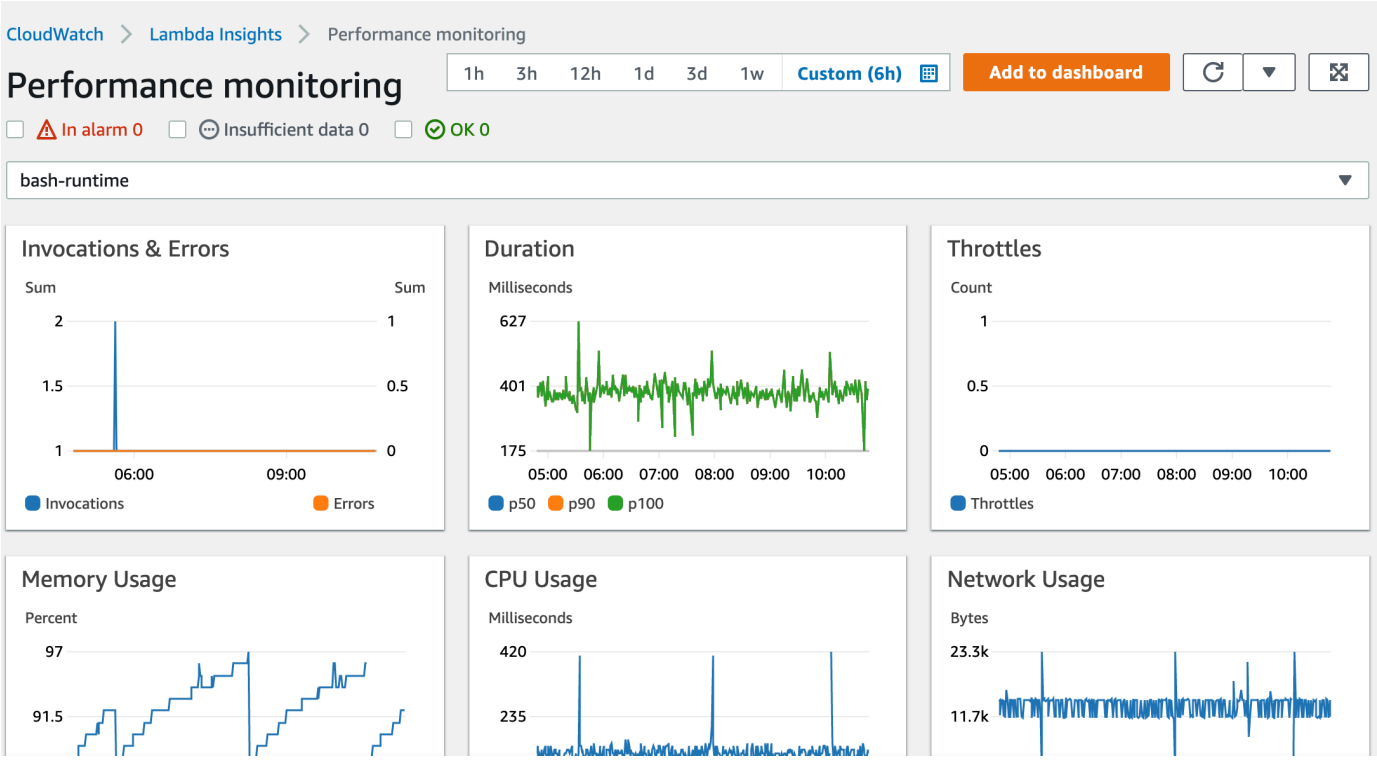
查看所有函数的运行时指标

1. 在 CloudWatch 控制台中打开多函数页面。
2. 从预定义的时间范围中进行选择，或选择自定义时间范围。
3. （可选）选择 Add to dashboard（添加到控制面板）以将小组件添加到 CloudWatch 控制面板。



查看单个函数的运行时指标

1. 在 CloudWatch 控制台中打开单函数页面。
2. 从预定义的时间范围中进行选择，或选择自定义时间范围。
3. （可选）选择 Add to dashboard（添加到控制面板）以将小组件添加到 CloudWatch 控制面板。



有关更多信息，请参阅在 CloudWatch 控制面板上创建和使用小组件。

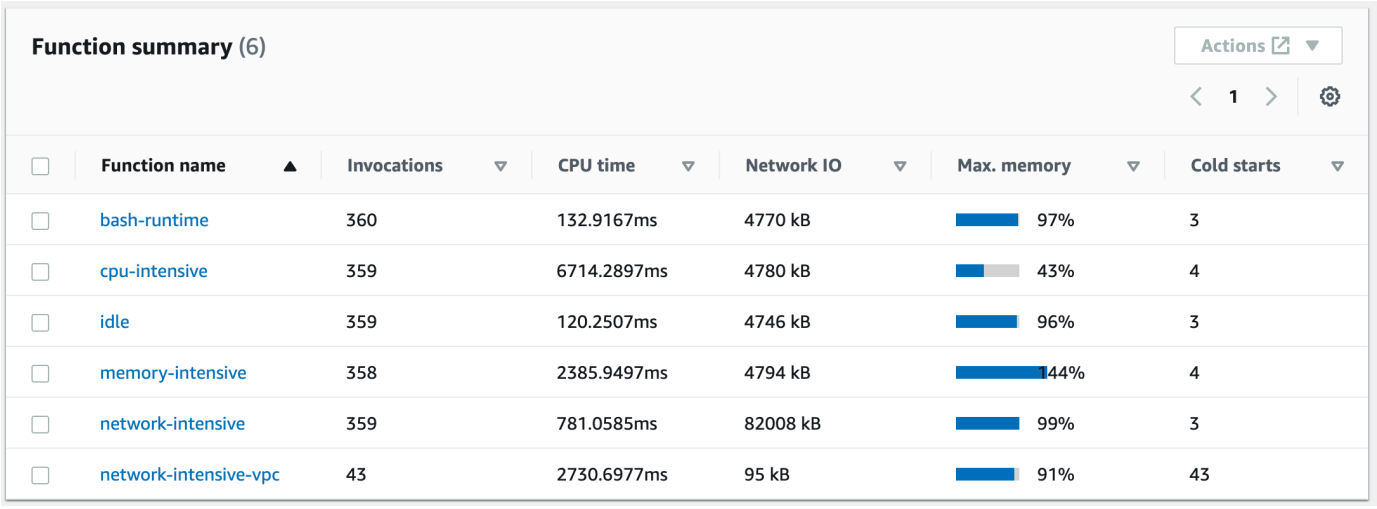
检测函数异常的工作流程示例

您可以使用 Lambda Insights 控制面板上的多函数概览来识别和检测函数的计算内存异常。例如，如果多函数概述指示函数正在使用大量内存，则可以在内存使用情况窗格中查看详细的内存利用率指标。然后，您可以转到“指标”控制面板以启用异常检测或创建警报。

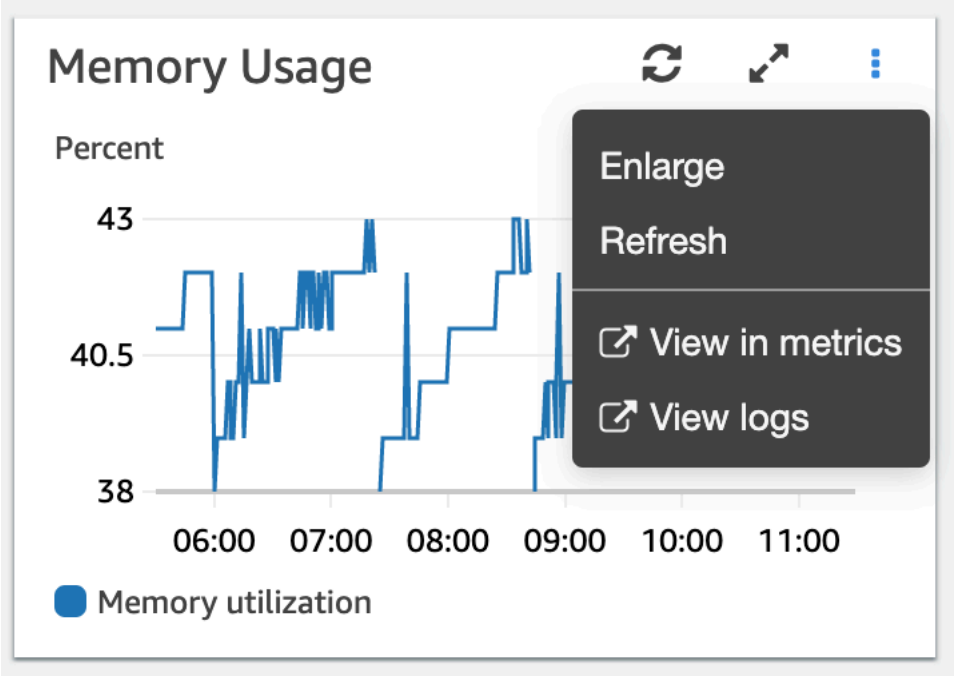
为函数启用异常检测

1. 在 CloudWatch 控制台中打开多函数页面。
2. 在函数摘要下，选择函数的名称。

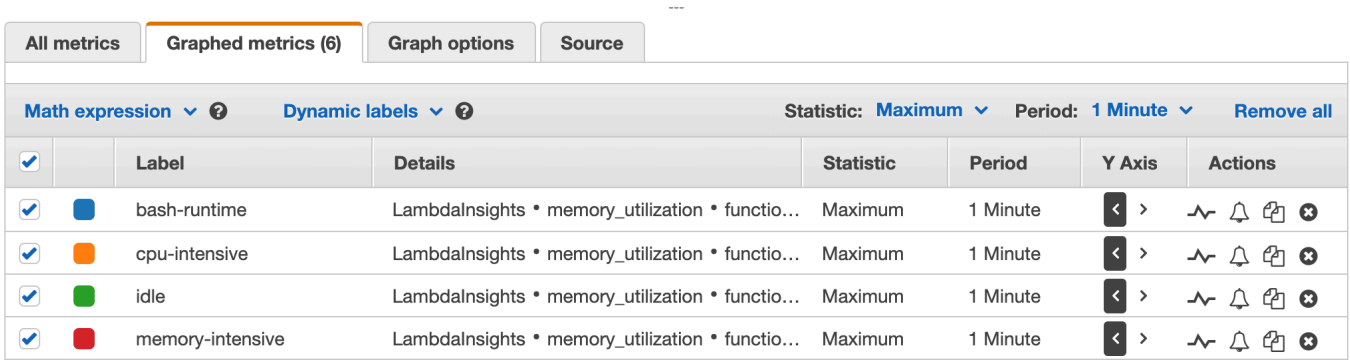
   单函数视图随即打开，其中包含函数运行时指标。



3. 在内存使用情况窗格中，选择三个竖直的点，然后选择在指标中查看，以打开指标控制面板。



4. 在绘成图表的指标选项卡的操作列中，选择第一个图标以便为函数启用异常检测。



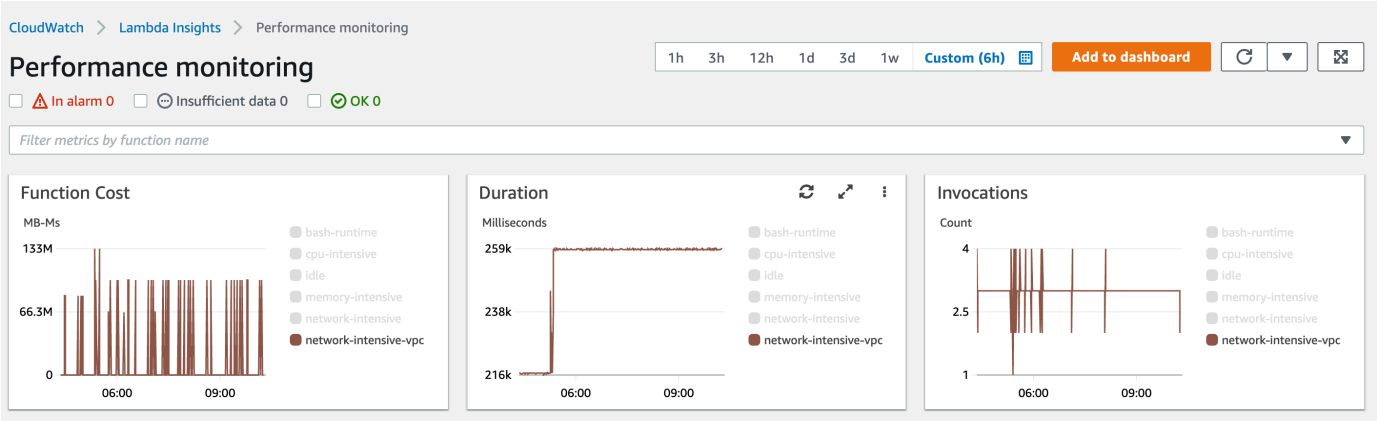
有关更多信息，请参阅使用 CloudWatch 异常检测。

使用查询排除函数故障的示例工作流程

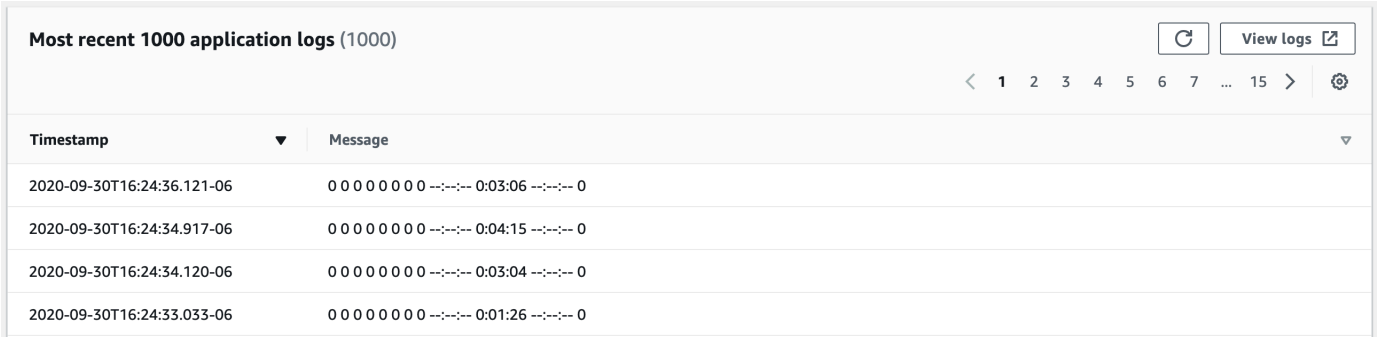
您可以使用 Lambda Insights 控制面板上的单函数视图来识别函数持续时间峰值的根本原因。例如，如果多函数概述指示函数持续时间大幅增加，则可以暂停或选择持续时间窗格中的每个函数，以确定导致增加的函数。然后，您可以转到单函数视图并查看应用程序日志以确定根本原因。

对函数运行查询

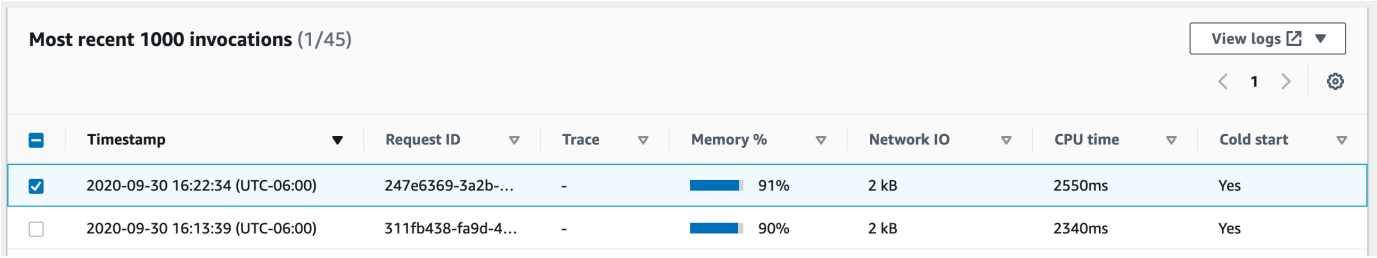
1. 在 CloudWatch 控制台中打开多函数页面。
2. 在持续时间窗格中，选择要筛选持续时间指标的函数。



3. 打开单函数页。
4. 选择按函数名称筛选指标下拉列表，然后选择您的函数。
5. 要查看最近 1000 个应用程序日志，请选择应用程序日志选项卡。
6. 查看时间戳和消息，以确定要进行故障排除的调用请求。



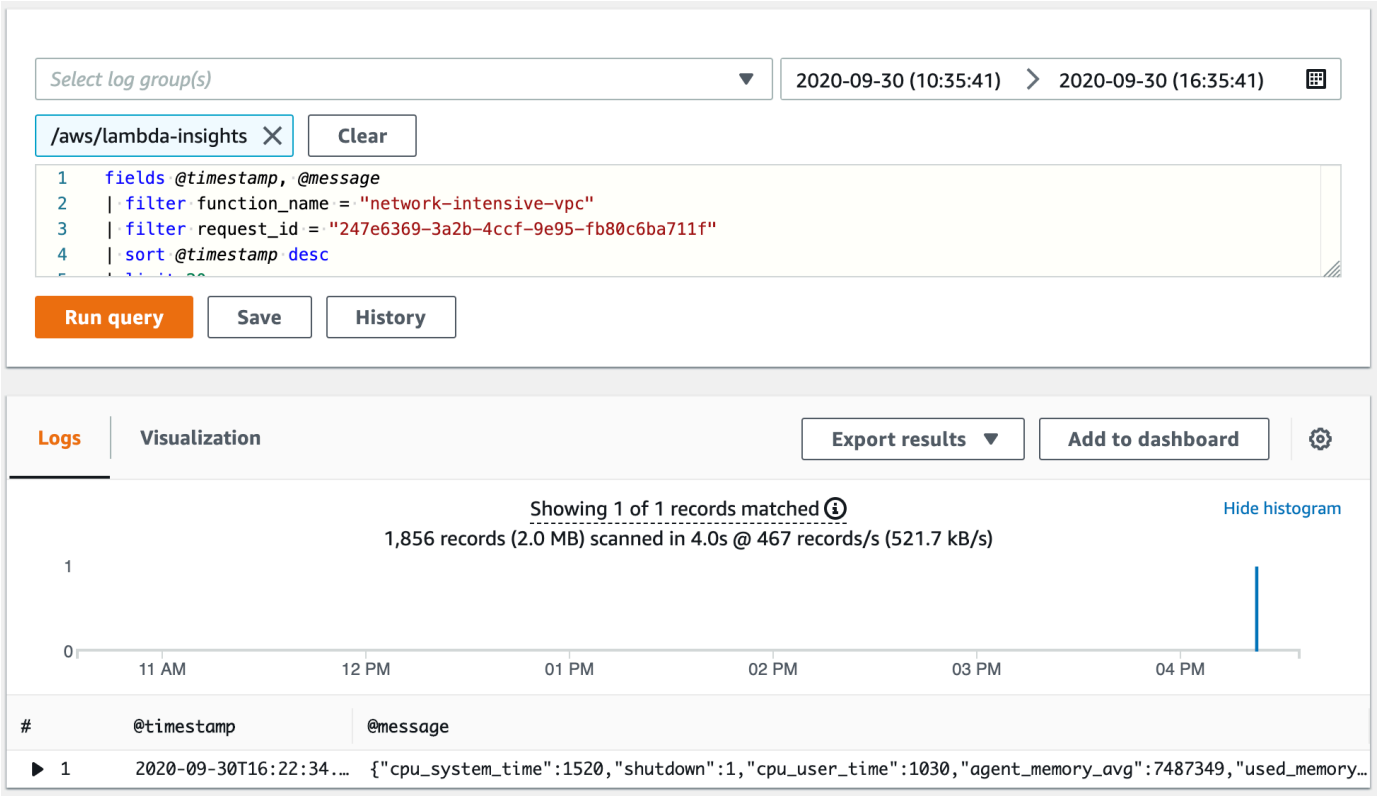
7. 要显示最近 1000 次调用，请选择调用选项卡。
8. 对于要排除故障的调用请求选择时间戳或消息。



9. 选择查看日志下拉列表，然后选择查看性能日志。

   将在 Logs Insights 控制面板中打开为您的函数自动生成的查询。

10. 选择运行查询以便为调用请求生成日志消息。



接下来做什么？

- 参阅 Amazon CloudWatch 用户指南中的创建控制面板，了解如何创建 CloudWatch Logs 控制面板。
- 参阅 Amazon CloudWatch 用户指南中的添加查询到控制面板或导出查询结果，了解如何添加查询到 CloudWatch Logs。

使用层管理 Lambda 依赖项

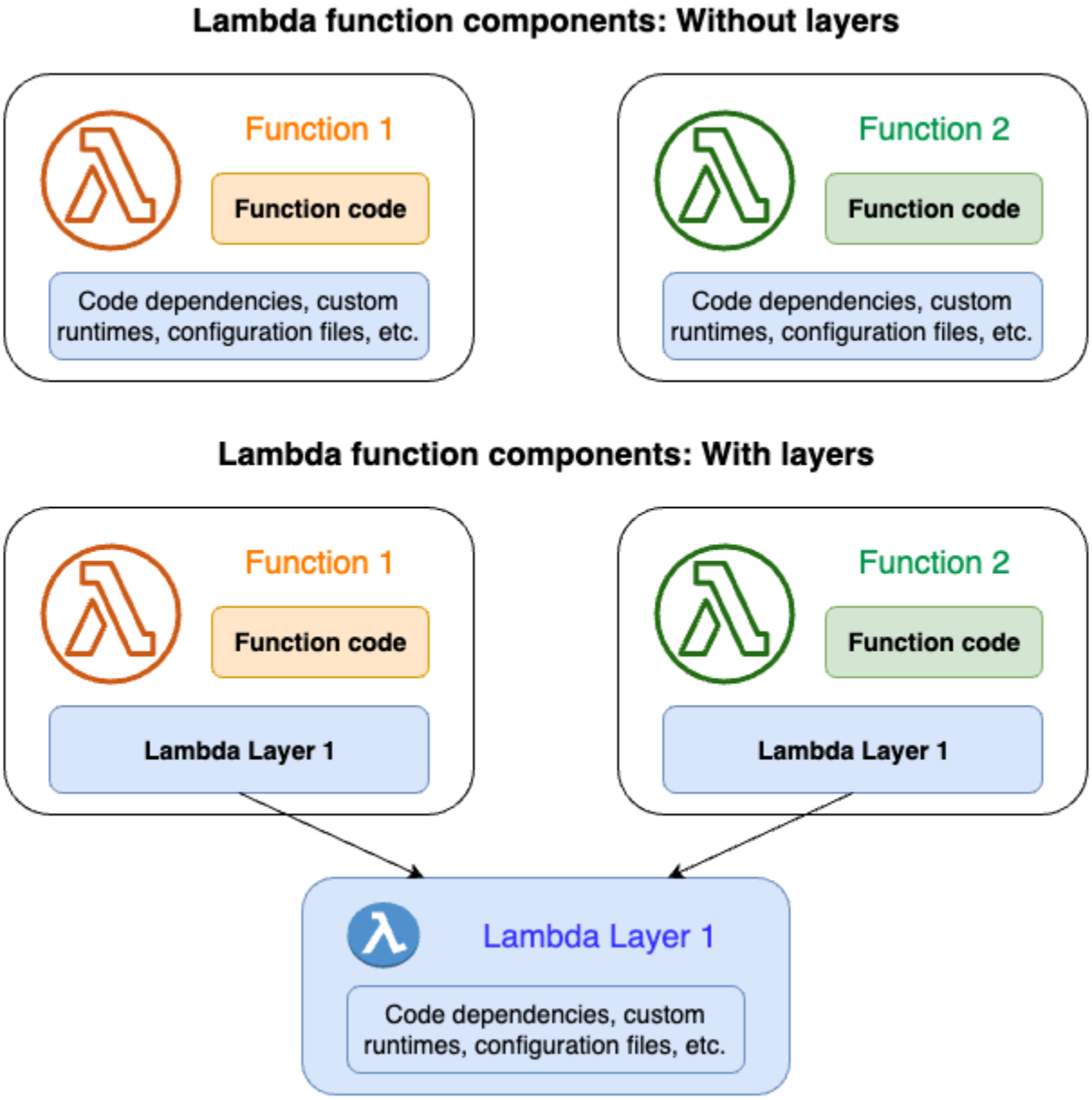
Lambda 层是包含补充代码或数据的 .zip 文件存档。层通常包含库依赖项、自定义运行时系统或配置文件。

您可能会基于很多原因考虑使用层：

- 减小部署包的大小。与其将所有函数依赖项和函数代码都包含在部署包中，不如将其放在一个层中。这样可以减小部署包的大小并对其进行组织。
- 分离核心函数逻辑与依赖项。借助层，您无需使用函数代码即可更新函数依赖项，反之亦然。这有助于将二者分离，并帮助您专注于函数逻辑。
- 在多个函数之间共享依赖项。创建层后，您可以将其应用到账户中任意数量的函数。如果没有层，则需要在每个单独的部署包中包含相同的依赖项。
- 使用 Lambda 控制台代码编辑器。代码编辑器是快速测试次要功能代码更新的得力工具。但是，如果部署包过大，则无法使用编辑器。使用层可以减小部署包的大小，并解锁代码编辑器的用法。

如果您在 Go 或 Rust 中使用 Lambda 函数，我们建议不要使用层。对于 Go 和 Rust 函数，您可以将函数代码作为可执行文件提供，其中包括编译后的函数代码及其所有依赖项。将依赖项放在层中会迫使函数在初始化阶段手动加载其他程序集，而这可能会增加冷启动时间。为了让 Go 和 Rust 函数获得最佳性能，请在部署包中加入依赖项。

下图说明了共享依赖项的两个函数之间的高级架构差异。一个使用 Lambda 层，另一个不使用。



当您向函数添加层时，Lambda 会将层内容提取到函数执行环境中的 /opt 目录中。所有原生支持的 Lambda 运行时系统都包含 /opt 目录中特定目录的路径。因而函数可以访问层内容。有关这些特定路径以及如何正确打包层的更多信息，请参阅 the section called “打包层”。

每个函数最多可以包含五个层。此外，您只能在将 Lambda 函数部署为 .zip 文件存档的情况下使用层。对于定义为容器映像的函数，您可以在创建容器映像时打包首选运行时系统和所有代码依赖项。有关更多信息，请参阅AWS计算博客上的在容器镜像中使用 Lambda 层和扩展。

主题

如何使用层

要创建层，请将依赖项打包到.zip 文件中，类似于创建常规部署包的方式。更具体地说，创建和使用层的一般过程包括以下三个步骤：

- 首先，打包层内容。这需要创建 .zip 文件存档。有关更多信息，请参阅 the section called “打包层”。
- 接下来，在 Lambda 中创建层。有关更多信息，请参阅 the section called “创建和删除层”。
- 向函数添加层。有关更多信息，请参阅 the section called “添加层”。

层和层版本

层版本是层的特定版本的不可变快照。当您创建新层时，Lambda 会创建一个版本号为 1 的新的层版本。每次向层发布更新时，Lambda 都会递增版本号并创建新的层版本。

每个层版本均由一个 Amazon 资源名称（ARN）进行唯一标识。向函数添加层时，必须指定要使用的确切的层版本。

打包层内容

Lambda 层是包含补充代码或数据的 .zip 文件存档。层通常包含库依赖项、自定义运行时系统或配置文件。

本部分介绍如何正确打包层内容。有关层的更多概念性信息以及您可能会考虑使用层的原因，请参阅 Lambda 层。

创建层的第一步是将所有层内容捆绑到.zip 文件存档中。由于 Lambda 函数在 Amazon Linux 上运行，因此层内容必须能够在 Linux 环境中编译和构建。

为确保层内容在 Linux 环境中正常运行，推荐使用 Docker 或 AWS Cloud9 等工具创建层内容。AWS Cloud9 是一个基于云的集成式开发环境（IDE），提供对 Linux 服务器的内置访问权限，用于运行和测试代码。有关更多信息，请参阅 AWS 计算博客上的 Using Lambda layers to simplify your development process。

主题

每个 Lambda 运行时的层路径

当您向函数添加层时，Lambda 会将层内容加载到该执行环境的 /opt 目录中。对于每个 Lambda 运行时系统，PATH 变量都包括 /opt 目录中的特定文件夹路径。为确保 PATH 变量能够获取层内容，层 .zip 文件应在以下文件夹路径中具有依赖项：

运行时	路径
Node.js	nodejs/node_modules
	nodejs/node16/node_modules (NODE_PATH)
	nodejs/node18/node_modules (NODE_PATH)
	nodejs/node20/node_modules (NODE_PATH)
Python	python
	python/lib/ *python3.x* /site-packages （站点目录）

运行时	路径
Java	java/lib (CLASSPATH)
Ruby	ruby/gems/3.2.0 (GEM_PATH)
	ruby/lib (RUBYLIB)
所有运行时	bin (PATH)
	lib (LD_LIBRARY_PATH)

以下示例显示了如何构建层 .zip 存档中的文件夹架构。

Node.js

Example 适用于 Node.js 的 AWS X-Ray 软件开发工具包的文件结构

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Python

Example 请求库的文件结构

```
layer_content.zip
# python
    # lib
        # python3.11
            # site-packages
                # requests
                # <other_dependencies> (i.e. dependencies of the requests package)
                # ...
```

Ruby

Example JSON gem 的文件结构

```
json.zip
# ruby/gems/2.7.0/
```

```
                | build_info
                | cache
                | doc
                | extensions
                | gems
                | # json-2.1.0
                # specifications
                  # json-2.1.0.gemspec
```

Java

Example Jackson JAR 文件的文件结构

```
layer_content.zip
# java
    # lib
        # jackson-core-2.17.0.jar
        # <other potential dependencies>
        # ...
```

All

Example JQ 库的文件结构

```
jq.zip
# bin/jq
```

有关打包、创建和添加层的特定语言说明，请参阅以下页面：

- Python – the section called “图层”
- Java - the section called “图层”

我们建议不要对以下语言使用层。链接的页面包含更多信息。

- Go – the section called “图层”
- Rust

在 Lambda 中创建和删除层

Lambda 层是包含补充代码或数据的 .zip 文件存档。层通常包含库依赖项、自定义运行时系统或配置文件。

本部分介绍如何在 Lambda 中创建和删除层。有关层的更多概念性信息以及您可能会考虑使用层的原因，请参阅 Lambda 层。

打包层内容完成后，下一步是在 Lambda 中创建层。本部分演示如何仅使用 Lambda 控制台或 Lambda API 创建和删除层。要使用 AWS CloudFormation 创建层，请参阅 the section called “层与 AWS CloudFormation”。要使用 AWS Serverless Application Model（AWS SAM）创建层，请参阅 the section called “层与 AWS SAM”。

主题

创建层

您可以从本地计算机或 Amazon Simple Storage Service（Amazon S3）上传 .zip 文件存档。设置函数的执行环境时，Lambda 将图层内容提取到 `/opt` 目录。

层可以有一个或多个层版本。创建层时，Lambda 将层版本设置为版本 1。您可以随时更改现有层版本的权限。但是，要更新代码或进行其他配置更改，必须创建该层的新版本。

创建层（控制台）

1. 打开 Lambda 控制台的 Layers page（层页面）。
2. 选择 Create layer（创建层）。
3. 在 Layer configuration（层配置）下，在 Name（名称）中，输入层的名称。
4. （可选）对于 Description（描述），输入对层的描述。
5. 要上载层代码，请执行以下操作之一：
   - 要从电脑上传 .zip 文件，请选择 Upload a .zip file（上传 .zip 文件）。然后，选择 Upload（上载）以选择本地 .zip 文件。
   - 要从 Simple Storage Service（Amazon S3）上传文件，请选择 Upload a file from Amazon S3 [从 Simple Storage Service（Amazon S3）上传文件]。然后，对于 Amazon S3 link URL（Simple Storage Service (Amazon S3) 链接 URL），输入文件的链接。

6. （可选）对于兼容架构，选择一个值或两个值。有关更多信息，请参阅 the section called “指令集（ARM/x86）”。
7. （可选）对于兼容的运行时系统，选择与层兼容的运行时系统。
8. （可选）对于 License（许可证），输入任何必要的许可证信息。
9. 选择 Create（创建）。

或者，您也可以使用 PublishLayerVersion API 创建层。例如，您可以使用 `publish-layer-version` AWS Command Line Interface（CLI）命令并指定名称、描述和 .zip 文件存档。许可证信息、兼容的运行时系统和兼容的架构参数都是可选的。

```
aws lambda publish-layer-version --layer-name my-layer \
    --description "My layer" \
    --license-info "MIT" \
    --zip-file fileb://layer.zip \
    --compatible-runtimes python3.10 python3.11 \
    --compatible-architectures "arm64" "x86_64"
```

您应该可以看到类似于如下所示的输出内容：

```
{
    "Content": {
        "Location": "https://awslambda-us-east-2-layers.s3.us-east-2.amazonaws.com/
snapshots/123456789012/my-layer-4aaa2fbb-ff77-4b0a-ad92-5b78a716a96a?
versionId=27iWyA73cCAYqyH...",
        "CodeSha256": "tv9jJO+rPbXUUXuRKi7CwHzKtLDkDRJLB3cC3Z/ouXo=",
        "CodeSize": 169
    },
    "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
    "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer:1",
    "Description": "My layer",
    "CreatedDate": "2023-11-14T23:03:52.894+0000",
    "Version": 1,
    "CompatibleArchitectures": [
        "arm64",
        "x86_64"
     ],
    "LicenseInfo": "MIT",
    "CompatibleRuntimes": [
        "python3.10",
        "python3.11"
```

```
    ]
}
```

每次调用 `publish-layer-version` 时，都会创建该层的新版本。

删除层版本

要删除层版本，请使用 DeleteLayerVersion API。例如，您可以使用 `delete-layer-version` CLI 命令并指定层名称和层版本。

```
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

删除一个层版本后，您无法再将 Lambda 函数配置为使用该层版本。但是，已使用此版本的任何函数仍能访问它。此外，Lambda 绝不会重用版本号作为层名称。

在计算配额时，删除层版本意味着它不再计入默认的 75 GB 函数和层存储配额中。但是，对于消耗已删除层版本的函数，层内容仍计入函数的部署包大小配额（即 .zip 文件存档为 250 MB）。

向函数添加层

Lambda 层是包含补充代码或数据的 .zip 文件存档。层通常包含库依赖项、自定义运行时系统或配置文件。

本部分介绍如何向 Lambda 函数添加层。有关层的更多概念性信息以及您可能会考虑使用层的原因，请参阅 Lambda 层。

在将 Lambda 函数配置为使用层之前，您必须：

- 打包层内容
- 在 Lambda 中创建层
- 确保您拥有对层版本调用 GetLayerVersion API 的权限。对于 AWS 账户 中的函数，您必须在用户策略中拥有此权限。要在另一个账户中使用层，该账户的拥有者必须在基于资源的策略中授予您的账户权限。有关示例，请参阅 the section called “其他账户的层访问”。

您最多可以在 Lambda 函数中添加五个层。函数和所有层的总解压缩大小不能超出 250 MB 的解压缩部署程序包大小配额。有关更多信息，请参阅 Lambda 限额。

即使在删除该层版本或撤消该层的访问权限后，函数也可以继续使用已添加的任何层版本。但是，您无法创建使用已删除层版本的新函数。

> **Note**
>
> 确保添加到函数的层与函数的运行时系统和指令集架构兼容。

向函数添加层（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择要配置的函数。
3. 在层下，选择添加层。
4. 在选择层下，选择一个层源：
   a. 对于 AWS 层或自定义层层源，请从下拉菜单中选择一个层。在版本下，从下拉菜单中选择层版本。
   b. 对于指定 ARN 层源，请在文本框中输入 ARN 并选择验证。然后，选择添加。

添加层的顺序就是 Lambda 将层内容合并到执行环境中的顺序。您可以使用控制台更改层合并顺序。

更新函数的层合并顺序（控制台）

1. 打开 Lamba 控制台的函数页面。
2. 选择要配置的函数。
3. 在层下，选择编辑。
4. 选择其中一个图层。
5. 选择提前合并或者稍后合并以调整层的顺序。
6. 选择保存。

层受版本控制。每个层版本的内容都是不可变的。层所有者可以发布新的层版本以提供更新内容。您可以使用控制台更新附加到函数的层版本。

更新函数的层版本（控制台）

1. 打开 Lambda 控制台的 Layers page（层页面）。
2. 选择要为其更新版本的层。
3. 选择使用此版本的函数选项卡。
4. 选择要修改的函数，然后选择编辑。
5. 从层版本中，选择要更改的目标层版本。
6. 选择 Update functions（更新函数）。

不能跨 AWS 账户更新函数的层版本。

主题

- 通过函数访问层内容
- 查找层信息

通过函数访问层内容

如果 Lambda 函数包含层，Lambda 会将层的内容提取到函数执行环境中的 `/opt` 目录。Lambda 按函数列出的顺序（从低到高）提取层。Lambda 会合并同名文件夹。如果同一文件出现在多个层中，则该函数将使用上次提取的层中的版本。

每个 Lambda 运行时系统都会将特定的 /opt 目录文件夹添加到 PATH 变量。函数代码无需指定路径即可访问层内容。有关 Lambda 执行环境中路径设置的更多信息，请参阅 the section called “定义运行时环境变量”：

请参阅 the section called “每个 Lambda 运行时的层路径” 以了解创建层时在何处纳入库。

如果您使用的是 Node.js 或 Python 运行时系统，则可使用 Lambda 控制台中的内置代码编辑器。您能够导入作为层添加到当前函数的任何库。

查找层信息

要查找账户中与函数运行时系统兼容的层，请使用 ListLayers API。例如，您可以使用以下 `list-layers` AWS Command Line Interface（CLI）命令：

```
aws lambda list-layers --compatible-runtime python3.9
```

您应该可以看到类似于如下所示的输出内容：

```
{
    "Layers": [
        {
            "LayerName": "my-layer",
            "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
            "LatestMatchingVersion": {
                "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
                "Version": 2,
                "Description": "My layer",
                "CreatedDate": "2023-11-15T00:37:46.592+0000",
                "CompatibleRuntimes": [
                    "python3.9",
                    "python3.10",
                    "python3.11",
                ]
            }
        }
    ]
}
```

要列出账户中的所有层，您可以省略 `--compatible-runtime` 选项。响应详细信息显示了每层的最新版本。

您还可以使用 ListLayerVersions API 获取层的最新版本。例如，您可以使用以下 `list-layer-versions` CLI 命令：

```
aws lambda list-layer-versions --layer-name my-layer
```

您应该可以看到类似于如下所示的输出内容：

```
{
    "LayerVersions": [
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
            "Version": 2,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:37:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        },
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:1",
            "Version": 1,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:27:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        }
    ]
}
```

与层结合使用 AWS CloudFormation

您可以使用 AWS CloudFormation 创建层并将层与 Lambda 函数关联起来。以下示例模板会创建一个名为 `my-lambda-layer` 的层，并使用层属性将该层附加到 Lambda 函数。

```
---
Description: CloudFormation Template for Lambda Function with Lambda Layer
Resources:
  MyLambdaLayer:
    Type: AWS::Lambda::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      Content:
        S3Bucket: amzn-s3-demo-bucket
        S3Key: my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-lambda-function
      Runtime: python3.9
      Handler: index.handler
      Timeout: 10
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
      Layers:
        - !Ref MyLambdaLayer
```

与层结合使用 AWS SAM

您可以使用 AWS Serverless Application Model（AWS SAM）自动在应用程序中创建层。`AWS::Serverless::LayerVersion` 资源类型创建一个层版本，您可以从自己的 Lambda 函数配置中引用该版本。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: AWS SAM Template for Lambda Function with Lambda Layer

Resources:
  MyLambdaLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      ContentUri: s3://amzn-s3-demo-bucket/my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: MyLambdaFunction
      Runtime: python3.9
      Handler: app.handler
      CodeUri: s3://amzn-s3-demo-bucket/my-function
      Layers:
        - !Ref MyLambdaLayer
```

使用 Lambda 扩展增强 Lambda 函数

您可以使用 Lambda 扩展来增强您的 Lambda 函数。例如，使用 Lambda 扩展将函数与您首选的监控、可观察性、安全性和监管工具集成。您可以从 AWS Lambda 合作伙伴提供的一系列工具中进行选择，也可以创建自己的 Lambda 扩展。

Lambda 支持内部和外部扩展。外部扩展作为独立进程在执行环境中运行，并在完全处理函数调用后继续运行。由于扩展作为单独的进程运行，因此您可以使用不同于函数的语言来编写它们。所有 Lambda 运行时 均支持扩展。

内部扩展作为运行时进程的一部分运行。您的函数通过使用包装脚本或进程内机制（如 `JAVA_TOOL_OPTIONS`）访问内部扩展。有关更多信息，请参阅修改运行时环境。

您可以使用 Lambda 控制台、AWS Command Line Interface (AWS CLI) 或基础设施即代码 (IaC) 服务和工具（如 AWS CloudFormation、AWS Serverless Application Model (AWS SAM) 和 Terraform）向函数添加扩展。

您需要按扩展所占用的执行时间（以 1 毫秒为增量）付费。安装自己的扩展没有任何费用。有关扩展的定价的更多信息，请参阅 AWS Lambda 定价 有关合作伙伴扩展的定价信息，请参阅这些合作伙伴的网站。有关官方合作伙伴扩展列表，请参阅 the section called “扩展合作伙伴”。

有关扩展以及如何将扩展与 Lambda 函数一起使用的教程，请参阅 AWS Lambda 扩展研讨会。

主题

执行环境

Lambda 在执行环境中调用您的函数，该环境提供一个安全和隔离的运行时环境。执行环境管理运行函数所需的资源，并为函数的运行时和扩展提供生命周期支持。

执行环境的生命周期包括以下阶段：

- **Init**：在此阶段，Lambda 会使用配置的资源创建或取消冻结执行环境、下载函数代码和所有层、初始化任何扩展、初始化运行时，然后运行函数的初始化代码（主处理程序外的代码）。Init 阶段发生在第一次调用期间，或者在函数调用之前（如果您已启用预配置并发）。

  Init 阶段分为三个子阶段：Extension init、Runtime init 和 Function init。这些子阶段可确保所有扩展和运行时在函数代码运行之前完成其设置任务。

  激活 Lambda SnapStart 后，在您发布一个函数版本时会发生 Init 阶段。Lambda 保存初始化的执行环境的内存和磁盘状态的快照，永久保存加密快照并对其进行缓存以实现低延迟访问。如果您具有 beforeCheckpoint 运行时挂钩，则该代码将在 Init 阶段结束时运行。
- **Restore**（仅限 SnapStart）：当您首次调用 SnapStart 函数时，随着该函数的纵向扩展，Lambda 会从永久保存的快照中恢复新的执行环境，而不是从头开始初始化函数。如果您有 afterRestore() 运行时挂钩，则代码将在 Restore 阶段结束时运行。afterRestore() 运行时挂钩执行期间将产生费用。必须加载运行时（JVM），并且 afterRestore() 运行时挂钩必须在超时限制（10 秒）内完成。否则，您将收到 SnapStartTimeoutException。Restore 阶段完成后，Lambda 将调用函数处理程序（调用阶段）。
- **Invoke**：在此阶段，Lambda 调用函数处理程序。在函数运行完成后，Lambda 准备处理另一次函数调用。
- **Shutdown**：如果 Lambda 函数在一段时间内没有收到任何调用，则触发此阶段。在 Shutdown 阶段，Lambda 会关闭运行时，提醒扩展让其完全停止，然后删除环境。Lambda 向每个扩展发送一个 Shutdown 事件，通知扩展环境即将关闭。

在 Init 阶段，Lambda 将包含扩展的层提取到执行环境中的 /opt 目录中。Lambda 在 /opt/extensions/ 目录中查找扩展，将每个文件解释为启动扩展的可执行引导程序，然后并行启动所有扩展。

对性能和资源的影响

函数的扩展的大小将计入部署包大小限制。对于 .zip 文件存档，函数和所有扩展解压后的总大小不得超过解压后的部署程序包大小限制，即 250 MB。

扩展可能会影响函数的性能，因为它们共享函数资源，如 CPU、内存和存储。例如，如果扩展执行计算密集型操作，那么您可能会发现函数的执行时间增加。

每个扩展都必须在 Lambda 调用函数之前完成其初始化。因此，占用大量初始化时间的扩展可能会增加函数调用的延迟。

要测量扩展在函数执行后所花的额外时间，可以使用PostRuntimeExtensionsDuration 函数指标。要衡量所使用的内存增加量，可以使用 MaxMemoryUsed 指标。要了解特定扩展的影响，您可以并行运行不同版本的函数。

权限

扩展可以访问与函数相同的资源。由于扩展是在与函数相同的环境中执行的，因此在函数和扩展之间共享权限。

对于 .zip 文件归档，您可以创建一个 AWS CloudFormation 模板，以简化将相同扩展配置（包括 AWS Identity and Access Management (IAM) 权限）附加到多个函数的任务。

配置 Lambda 扩展

配置扩展（.zip 文件存档）

您可以将扩展作为 Lambda 层添加到函数中。使用层可让您在整个组织或整个 Lambda 开发人员社区中共享扩展。您可以向层添加一个或多个扩展。您最多可以为一个函数注册 10 个扩展。

您可以使用与任何层相同的方法将扩展添加到函数中。有关更多信息，请参阅 Lambda 层。

将扩展添加到您的函数（控制台）

1. 打开 Lamba 控制台的 Functions（函数）页面。
2. 选择函数。
3. 选择 Code（代码）（如果尚未选择）。
4. 在 Layers（层）下，选择 Edit（编辑）。
5. 对于选择层，选择指定 ARN。
6. 对于指定 ARN，输入扩展层的 Amazon Resource Name (ARN)。
7. 选择添加。

在容器映像中使用扩展

您可以向容器映像添加扩展。ENTRYPOINT 容器映像设置指定函数的主进程。在 Dockerfile 中配置 ENTRYPOINT 设置，或者覆盖函数配置。

您可以在一个容器内运行多个进程。Lambda 管理主进程和任何其他进程的生命周期。Lambda 使用扩展 API 管理扩展生命周期。

示例：添加外部扩展

外部扩展在 Lambda 函数之外的单独进程中运行。Lambda 为 `/opt/extensions/` 目录中的每个扩展启动一个进程。Lambda 使用扩展 API 管理扩展生命周期。函数运行完成后，Lambda 向每个外部扩展发送一个 `Shutdown` 事件。

Example 向 Python 基本映像添加外部扩展

```
FROM public.ecr.aws/lambda/python:3.11

# Copy and install the app
```

```
COPY /app /app
WORKDIR /app
RUN pip install -r requirements.txt

# Add an extension from the local directory into /opt
ADD my-extension.zip /opt
CMD python ./my-function.py
```

后续步骤

要了解有关扩展的更多信息，我们建议使用以下资源：

- 有关基本的工作示例，请参阅 AWS Lambda 计算博客上的构建 AWS 扩展。
- 有关 AWS Lambda 合作伙伴提供的扩展的信息，请参阅 AWS Lambda 计算博客上的AWS 扩展简介。
- 要查看可用的示例扩展和包装脚本，请参阅 AWS Lambda Sample GitHub 存储库上的 AWS 扩展。

AWS Lambda 扩展合作伙伴

AWS Lambda 已与多个第三方实体结成合作伙伴，提供多种扩展以与您的 Lambda 函数集成。以下列表详细列出了可供您随时使用的第三方扩展。

- AppDynamics – 提供 Node.js 或 Python Lambda 函数的自动分析，从而提供有关函数性能的可见性和提示。
- Axiom – 提供用于监控 Lambda 函数性能和汇总系统级指标的控制面板。
- Check Point CloudGuard – 一种基于扩展的运行时解决方案，可为无服务器应用程序提供完整的生命周期安全性。
- Datadog – 通过使用指标、跟踪和日志，为无服务器应用程序提供全面、实时的可见性。
- Dynatrace – 提供跟踪和指标的可见性，并利用 AI 在整个应用程序堆栈中进行自动错误检测和根本原因分析。
- Elastic – 提供应用程序性能监控（APM），以便使用相关的跟踪、指标和日志来识别和解决根本原因问题。
- Epsagon – 侦听调用事件、存储跟踪，并将它们并行发送到 Lambda 函数执行。
- Fastly– 保护您的 Lambda 函数免受可疑活动的侵害，例如注入式攻击、通过凭证填充获取账户、恶意机器人和 API 滥用。
- HashiCorp Vault – 管理秘密并使开发人员可以在函数代码中使用这些秘密，而无需通知函数文件库。
- Honeycomb – 用于调试应用程序堆栈的可观察性工具。
- Lumigo – 分析 Lambda 函数调用情况并收集用于在无服务器和微服务环境中排除问题的指标。
- New Relic – 与 Lambda 函数一起运行，自动收集、增强遥测数据，并将这些数据传输到 New Relic 的统一可观察性平台。
- Sedai – 由 AI/ML 提供支持的自主云管理平台，可为云运营团队提供持续优化，从而最大限度地节省云成本，提高大规模性能和可用性。
- Sentry – 诊断、修复和优化 Lambda 函数的性能。
- Site24x7 – 实现对 Lambda 环境的实时可观察性
- Splunk – 收集高分辨率、低延迟的指标，以便高效且有效地监控 Lambda 函数。
- Sumo Logic – 提供对无服务器应用程序的运行状况和性能的可见性。
- Thundra – 提供异步遥测报告，例如跟踪、指标和日志。
- Salt Security – 通过自动设置和对不同运行时的支持，简化 Lambda 函数的 API 状态治理和 API 安全。

AWS 托管式扩展

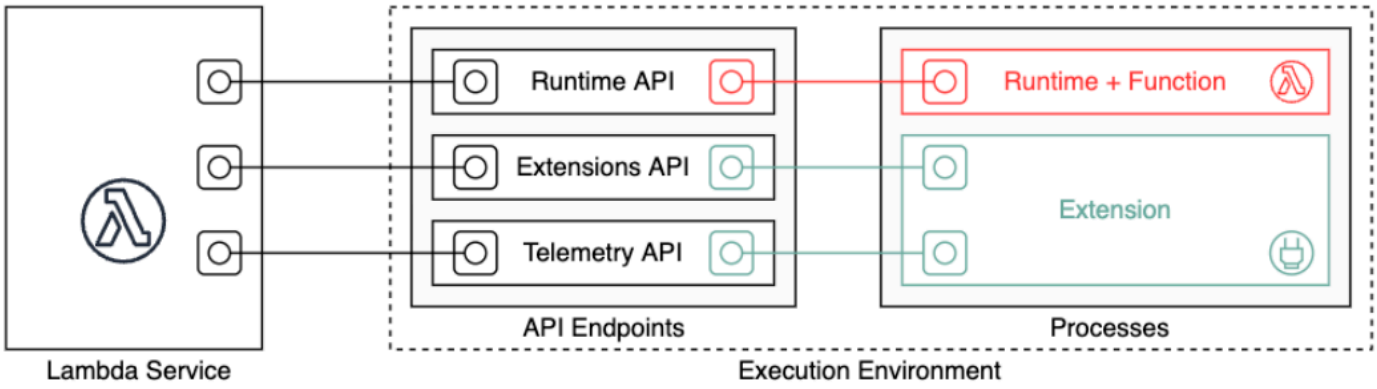
AWS 提供了其自己的托管式扩展，包括：

- AWS AppConfig - 使用功能标志和动态数据更新 Lambda 函数。您还可以使用此扩展来更新其他动态配置，例如运维节流和优化。
- Amazon CodeGuru Profiler - 通过查明应用程序最宝贵的代码行并提供改进代码的建议，来提高应用程序性能并降低成本。
- CloudWatch Lambda Insights - 通过自动化控制面板监控、故障排除和优化 Lambda 函数的性能。
- AWS Distro for OpenTelemetry（ADOT） – 启用函数将跟踪数据发送到 AWS 监控服务（例如 AWS X-Ray），以及支持 OpenTelemetry 的目的地（例如 Honeycomb 和 Lightstep）。
- AWS 参数和密钥 – 可让客户安全地从 AWS Systems Manager Parameter Store 检索参数和从 AWS Secrets Manager 检索密钥。

有关其他扩展示例和演示项目，请参阅 AWS Lambda 扩展。

使用 Lambda 扩展 API 创建扩展

Lambda 函数作者使用扩展将 Lambda 与他们首选的监控、可观察性、安全性和监管工具集成。函数作者可以使用来自 AWS、AWS 合作伙伴和开源项目的扩展。有关使用扩展的更多信息，请参阅 AWS Lambda 计算博客上的 AWS 扩展简介。本节介绍如何使用 Lambda 扩展 API、Lambda 执行环境生命周期以及 Lambda 扩展 API 参考。



作为扩展作者，您可以使用 Lambda 扩展 API 深入集成到 Lambda 执行环境中。您的扩展可以注册函数和执行环境生命周期事件。为响应这些事件，您可以启动新进程、运行逻辑、控制并参与 Lambda 生命周期的所有阶段：初始化、调用和关闭。此外，您可以使用运行时日志 API 接收日志流。

扩展作为独立进程在执行环境中运行，并可以在完全处理函数调用后继续运行。由于扩展作为进程运行，因此您可以使用不同于函数的语言来编写它们。我们建议您使用已编译的语言实现扩展。在这种情况下，扩展是一个独立的二进制文件，与支持的运行时兼容。所有 Lambda 运行时 均支持扩展。如果使用非编译的语言，请确保在扩展中包含兼容的运行时。

Lambda 还支持内部扩展。内部扩展作为运行时进程的独立线程运行。运行时启动并停止内部扩展。与 Lambda 环境集成的另一种方法是使用特定于语言的环境变量和包装脚本。您可以使用这些来配置运行时环境并修改运行时进程的启动行为。

您可以通过两种方式将扩展添加到函数。对于部署为 .zip 文件存档的函数，您可以将扩展部署为层。对于定义为容器映像的函数，您可以将扩展添加到容器映像中。

> **Note**
>
> 有关示例扩展和包装脚本的信息，请参阅 AWS Lambda Sample GitHub 存储库上的 AWS 扩展。

主题

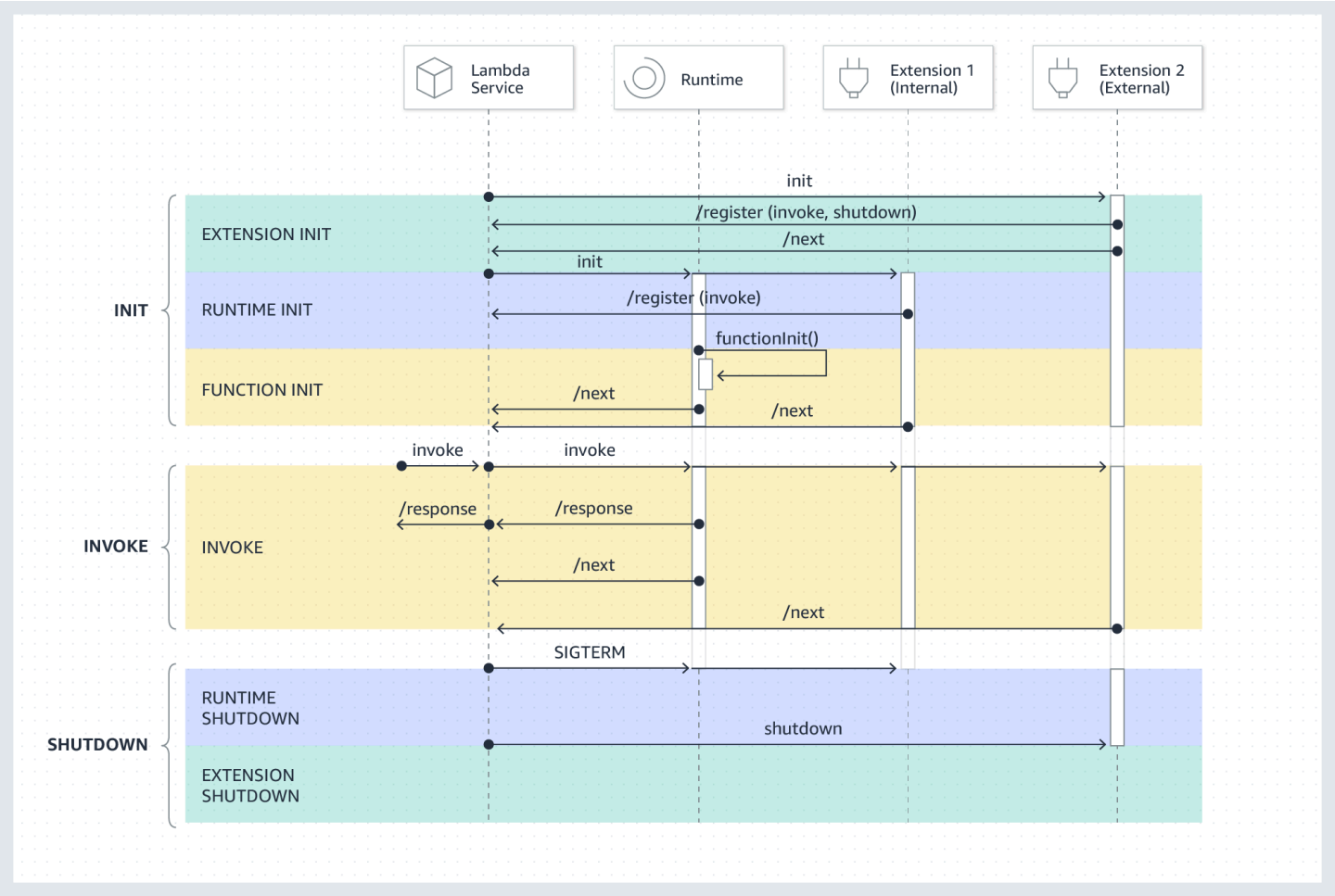
Lambda 执行环境生命周期

执行环境的生命周期包括以下阶段：

- **Init**：在此阶段，Lambda 会使用配置的资源创建或取消冻结执行环境、下载函数代码和所有层、初始化任何扩展、初始化运行时，然后运行函数的初始化代码（主处理程序外的代码）。Init 阶段发生在第一次调用期间，或者在函数调用之前（如果您已启用预配置并发）。

  Init 阶段分为三个子阶段：Extension init、Runtime init 和 Function init。这些子阶段可确保所有扩展和运行时在函数代码运行之前完成其设置任务。
- **Invoke**：在此阶段，Lambda 调用函数处理程序。在函数运行完成后，Lambda 准备处理另一次函数调用。
- **Shutdown**：如果 Lambda 函数在一段时间内没有收到任何调用，则触发此阶段。在 Shutdown 阶段，Lambda 会关闭运行时，提醒扩展让其完全停止，然后删除环境。Lambda 向每个扩展发送一个 Shutdown 事件，通知扩展环境即将关闭。

每个阶段都以一个从 Lambda 到运行时和所有已注册扩展的事件开始。运行时和每个扩展信号通过发送 Next API 请求来表示完成。Lambda 在每个进程结束并且没有挂起的事件时会冻结执行环境。



主题

Init 阶段

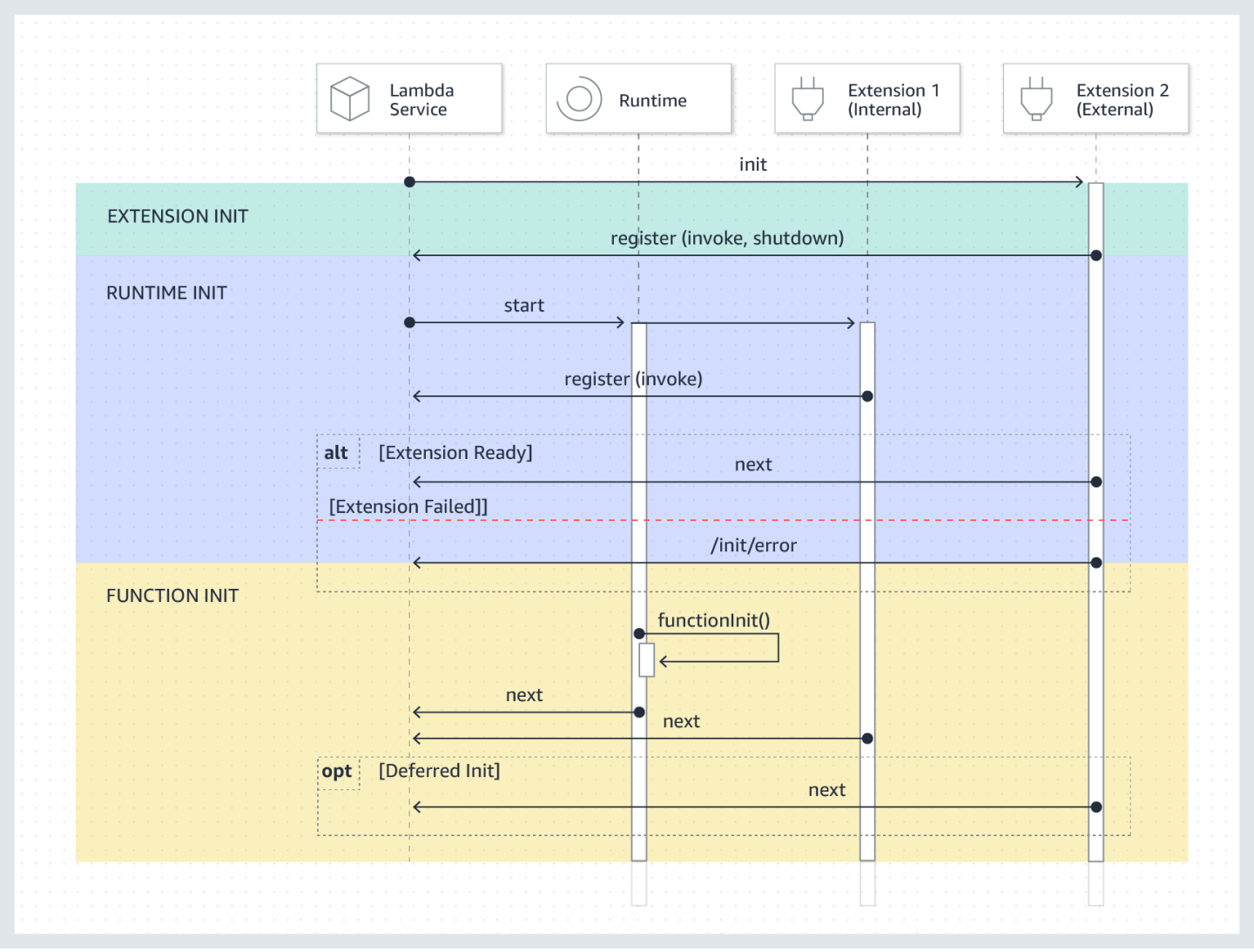
在 `Extension init` 阶段，每个扩展都需要在 Lambda 注册才能接收事件。Lambda 会使用扩展的完整文件名来验证扩展是否已完成引导启动序列。因此，每个 `Register` API 调用都必须包含 `Lambda-Extension-Name` 标头以及扩展的完整文件名。

您最多可以为一个函数注册 10 个扩展。此限制通过 Register API 调用强制执行。

在每个扩展注册后，Lambda 会启动 Runtime init 阶段。运行时进程调用 functionInit，启动 Function init 阶段。

Init 阶段在运行时之后结束，每个注册的扩展通过发送 Next API 请求指示完成。

> **Note**
>
> 扩展可以在 Init 阶段的任何时间点完成其初始化。



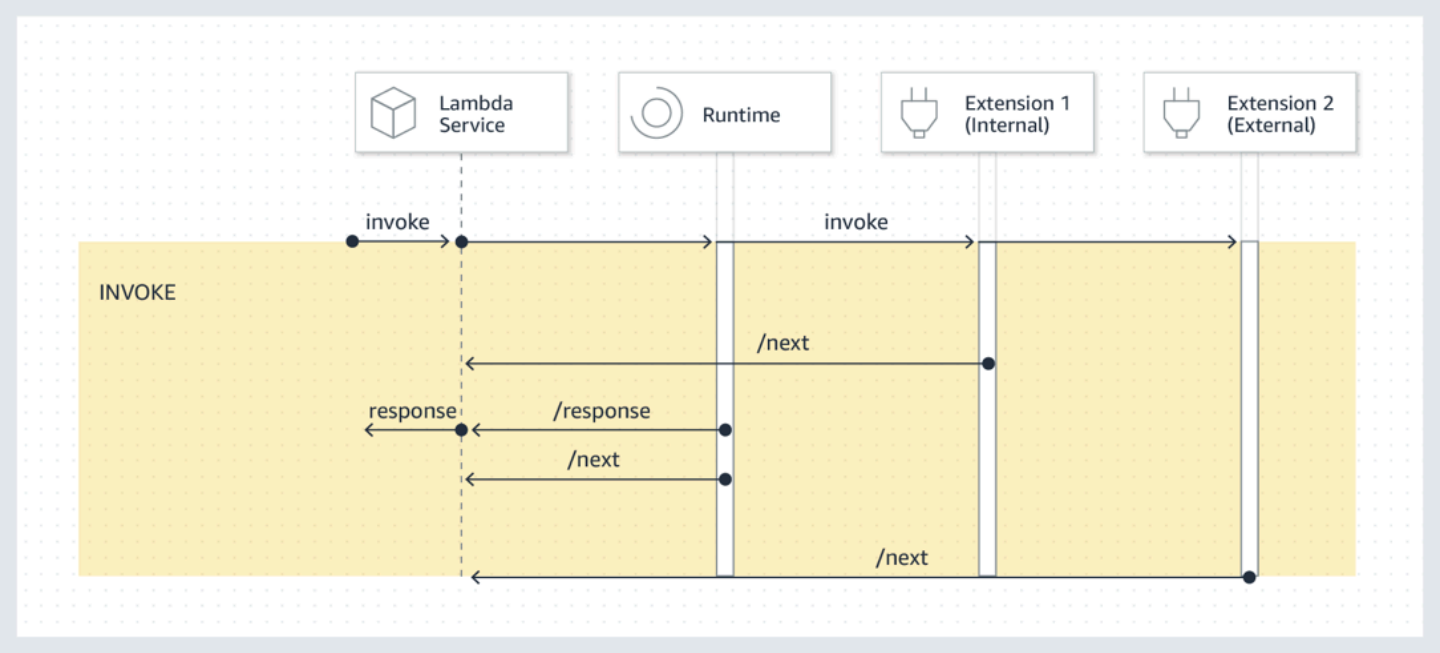
调用阶段

当调用 Lambda 函数来响应 Next API 请求时，Lambda 会向运行时和为 Invoke 事件注册的每个扩展发送一个 Invoke 事件。

在调用过程中，外部扩展与函数并行运行。在函数完成后，它们也会继续运行。这使您能够捕获诊断信息，或将日志、指标和跟踪发送到您选择的位置。

从运行时接收函数响应后，即使扩展仍在运行，Lambda 也会将响应返回给客户端。

Invoke 阶段在运行时之后结束，所有扩展都通过发送 Next API 请求来指示它们已完成。



事件负载：发送到运行时（和 Lambda 函数）的事件包含整个请求、标头（如 RequestId）和负载。发送到每个扩展的事件包含描述事件内容的元数据。此生命周期事件包括事件的类型、函数超时的时间 (deadlineMs)、requestId、被调用函数的 Amazon 资源名称（ARN）和跟踪标头。

希望访问函数事件主体的扩展可以使用与扩展进行通信的运行时开发工具包。在调用函数时，函数开发人员使用运行时开发工具包将负载发送到扩展。

以下是一个示例负载：

```
{
    "eventType": "INVOKE",
    "deadlineMs": 676051,
    "requestId": "3da1f2dc-3222-475e-9205-e2e6c6318895",
```

```
    "invokedFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:ExtensionTest",
    "tracing": {
        "type": "X-Amzn-Trace-Id",
        "value":
 "Root=1-5f35ae12-0c0fec141ab77a00bc047aa2;Parent=2be948a625588e32;Sampled=1"
    }
 }
```

持续时间限制：函数的超时设置会限制整个 Invoke 阶段的持续时间。例如，如果将函数超时设置为 360 秒，则该函数和所有扩展都需要在 360 秒内完成。请注意，没有独立的调用后阶段。持续时间是运行时系统和所有扩展调用的时间总和，直到函数和所有扩展完成运行之后才计算。

性能影响和扩展开销：扩展可能会影响函数性能。作为扩展作者，您可以控制扩展对性能的影响。例如，如果扩展执行计算密集型操作，那么函数的持续时间会增加，这是因为扩展和函数代码共用相同的 CPU 资源。此外，如果扩展在函数调用完成后执行大量操作，则函数持续时间会增加，因为 Invoke 阶段会持续，直到所有扩展指示它们都已完成。

> **Note**
>
> Lambda 以与函数的内存设置成正比的方式分配 CPU 处理能力。在较低的内存设置下，您可能会看到执行和初始化持续时间增加，这是因为函数和扩展进程正在争用相同的 CPU 资源。要缩短执行和初始化持续时间，请尝试增加内存设置。

为了帮助识别 Invoke 阶段上由扩展引起的性能影响，Lambda 会输出 PostRuntimeExtensionsDuration 指标。此指标衡量在运行时 Next API 请求与上次扩展 Next API 请求之间所花的累积时间。要衡量所使用的内存增加量，请使用 MaxMemoryUsed 指标。有关函数指标的更多信息，请参阅查看 Lambda 函数的指标。

函数开发人员可以并行运行不同版本的函数，以了解特定扩展的影响。我们建议扩展作者发布预期的资源消耗，以便函数开发人员更容易选择合适的扩展。

关闭阶段

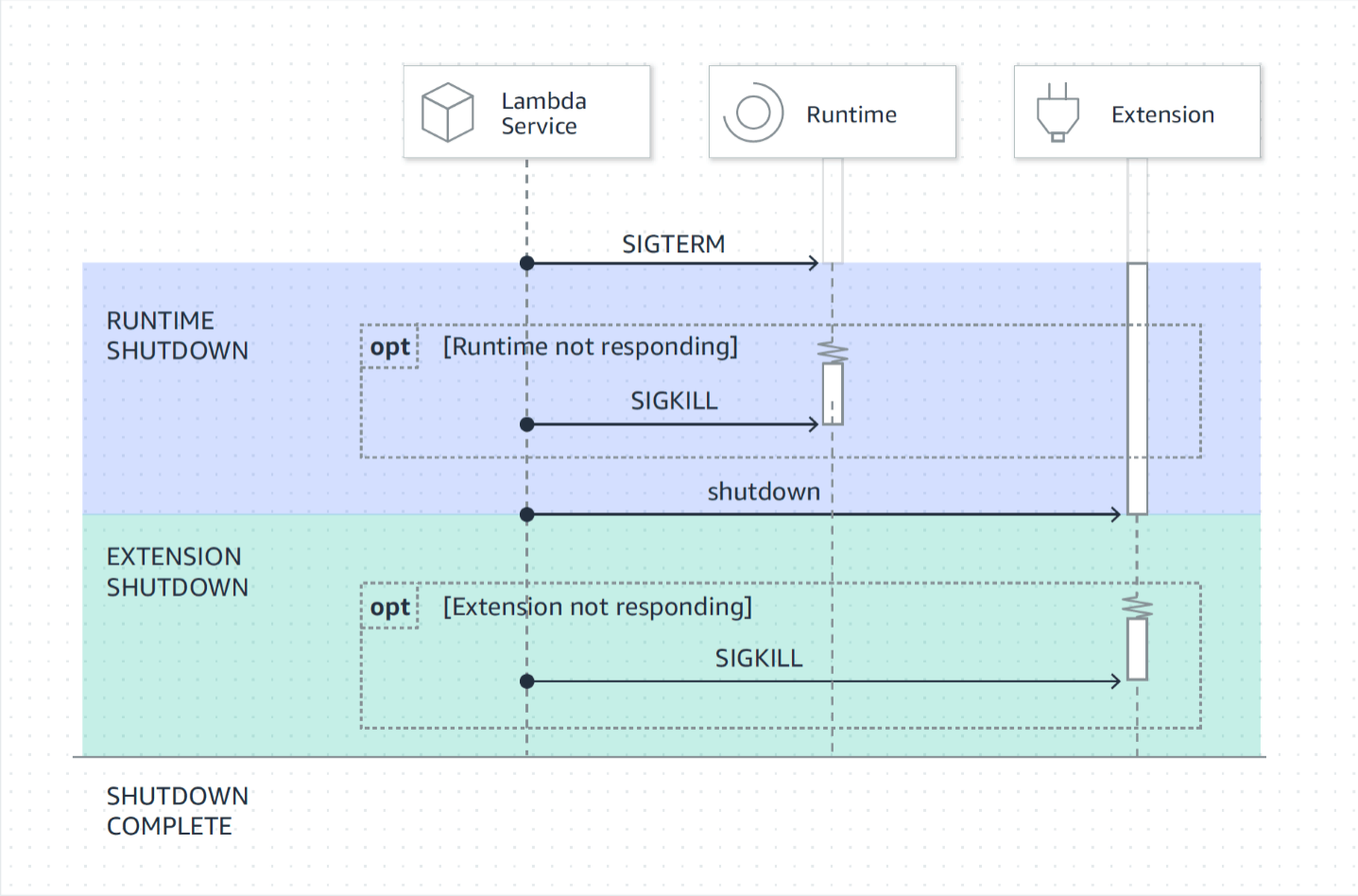
若 Lambda 即将关闭运行时，它会向每个已注册的外部扩展发送一个 Shutdown。扩展可以使用此时间执行最终清理任务。Shutdown 事件是为了响应 Next API 请求而发送的。

持续时间限制：Shutdown 阶段的最长持续时间取决于已注册扩展的配置：

- 0 毫秒 – 无已注册扩展的函数
- 500 毫秒 – 带有注册的内部扩展的函数
- 2000 毫秒 – 具有一个或多个注册的外部扩展的函数

对于具有外部扩展的函数，Lambda 为运行时进程保留最多 300 毫秒（对于具有内部扩展的运行时，则为 500 毫秒），以执行正常关闭。Lambda 将 2000 毫秒限制的剩余部分分配给要关闭的外部扩展。

如果运行时或扩展没有在限制范围内响应 Shutdown 事件，则 Lambda 会使用 SIGKILL 信号结束该进程。



事件负载：Shutdown 事件包含关闭原因和剩余时间（以毫秒为单位）。

shutdownReason 包括以下值：

- 降速 – 正常关机
- 超时 – 持续时间限制超时
- 失败 – 错误情况，例如 out-of-memory 事件

```
{
  "eventType": "SHUTDOWN",
  "shutdownReason": "reason for shutdown",
  "deadlineMs": "the time and date that the function times out in Unix time
 milliseconds"
}
```

权限和配置

在与 Lambda 函数相同的执行环境中运行扩展。扩展还与函数共享资源，例如 CPU、内存和 /tmp 磁盘存储。此外，扩展会使用与函数相同的 AWS Identity and Access Management (IAM) 角色和安全上下文。

文件系统和网络访问权限：扩展名在与函数运行时相同的文件系统和网络名称命名空间中运行。这意味着扩展需要与关联的操作系统兼容。如果扩展需要任何其他出站网络流量规则，您必须将这些规则应用于函数配置。

> **Note**
>
> 因为函数代码目录是只读的，所以扩展无法修改函数代码。

环境变量：扩展可以访问函数的环境变量，但以下特定于运行时进程的变量除外：

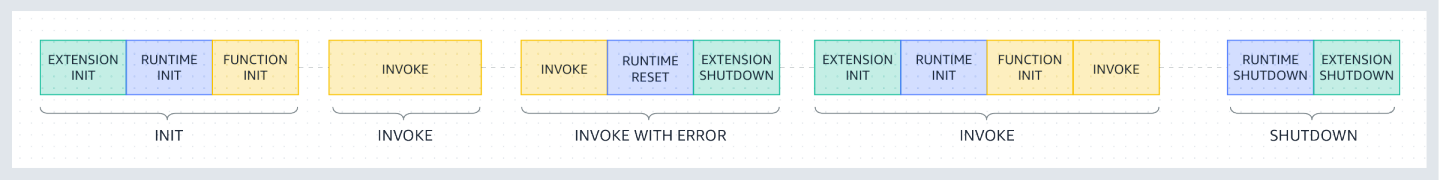
- AWS_EXECUTION_ENV
- AWS_LAMBDA_LOG_GROUP_NAME
- AWS_LAMBDA_LOG_STREAM_NAME
- AWS_XRAY_CONTEXT_MISSING
- AWS_XRAY_DAEMON_ADDRESS
- LAMBDA_RUNTIME_DIR
- LAMBDA_TASK_ROOT
- _AWS_XRAY_DAEMON_ADDRESS
- _AWS_XRAY_DAEMON_PORT
- _HANDLER

故障处理

初始化失败：如果扩展失败，Lambda 会重新启动执行环境以强制执行一致的行为，并利于扩展快速失败。此外，对于某些客户来说，扩展必须满足日志记录、安全性、监管和遥测收集等关键任务需求。

调用失败（例如内存不足、函数超时）：由于扩展与运行时共享资源，因此内存耗尽会影响到它们。当运行时失败时，所有扩展和运行时本身都会参与 Shutdown 阶段。此外，运行时可以作为当前调用的一部分自动重新启动，也可以通过延迟的重新初始化机制重新启动。

如果在 Invoke 期间出现故障（例如函数超时或运行时错误），Lambda 服务会执行重置。重置的行为类似于 Shutdown 事件。首先，Lambda 会关闭运行时，然后向每个注册的外部扩展发送一个 Shutdown 事件。该事件包括关闭的原因。如果此环境用于新调用，则扩展和运行时将作为下一次调用的一部分重新初始化。



有关上图的更详细说明，请参见 在调用阶段失败。

扩展日志：Lambda 将扩展的日志输出发送到 CloudWatch Logs。Lambda 还会在 Init 期间为每个扩展生成一个其他的日志事件。日志事件记录成功时的名称和注册首选项（事件、配置）或失败时的失败原因。

扩展故障排除

- 如果 Register 请求失败，请确保 Lambda-Extension-Name API 调用中的 Register 标头包含扩展的完整文件名。
- 如果 Register 请求对于内部扩展失败，请确保请求未注册 Shutdown 事件。

扩展 API 参考

扩展 API 版本 2020-01-01 的 OpenAPI 规范在此处提供：extensions-api.zip

您可以从 AWS_LAMBDA_RUNTIME_API 环境变量中检索 API 终端节点的值。要发送 Register 请求，请在每个 API 路径之前使用前缀 2020-01-01/。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

API 方法

- 注册
- 下一步
- Init 错误
- 退出错误

注册

在 Extension init 期间，所有扩展都需要在 Lambda 注册才能接收事件。Lambda 会使用扩展的完整文件名来验证扩展是否已完成引导启动序列。因此，每个 Register API 调用都必须包含 Lambda-Extension-Name 标头以及扩展的完整文件名。

内部扩展由运行时进程启动和停止，因此不允许它们注册 Shutdown 事件。

路径 – /extension/register

方法 – POST

Request headers（请求标头）

- Lambda-Extension-Name – 扩展名的完整文件名。必需：是。类型：字符串。
- Lambda-Extension-Accept-Feature – 在注册期间使用此功能以指定可选的注册功能。必需：否 类型：以逗号分隔的字符串。可使用此设置指定的功能：
  - accountId – 如果指定，扩展注册响应将包含账户 ID，该 ID 与您为其注册扩展的 Lambda 函数相关联。

请求正文参数

- events – 要注册的事件数组。必需：否 类型：字符串数组。有效字符串：INVOKE、SHUTDOWN。

响应标头

- Lambda-Extension-Identifier – 生成所有后续请求所需的唯一代理标识符（UUID 字符串）。

响应代码

- 200 – 响应正文包含函数名称、函数版本和处理程序名称。

- 400 – 错误请求
- 403 – 禁止访问
- 500 – 容器错误。不可恢复状态。扩展应立即退出。

Example 示例请求正文

```
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Example 示例响应正文

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler"
}
```

Example 具有可选 accountId 功能的响应正文示例

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler",
    "accountId": "123456789012"
}
```

下一步

扩展发送 `Next` API 请求以接收下一个事件，此事件可以是 `Invoke` 事件或 `Shutdown` 事件。响应正文包含负载，该负载是包含事件数据的 JSON 文档。

扩展发送 `Next` API 请求，以表示它已准备好接收新事件。这是一个阻止性调用。

不要对 GET 调用设置超时，因为扩展可能暂停一段时间，直到存在要返回的事件。

路径 – `/extension/event/next`

方法 – GET

Request headers（请求标头）

- `Lambda-Extension-Identifier` – 扩展的唯一标识符（UUID 字符串）。必需：是。类型：UUID 字符串。

响应标头

- `Lambda-Extension-Event-Identifier` – 事件的唯一标识符（UUID 字符串）。

响应代码

- 200 – 响应包含有关下一个事件（`EventInvoke` 或 `EventShutdown`）的信息。
- 403 – 禁止访问
- 500 – 容器错误。不可恢复状态。扩展应立即退出。

Init 错误

扩展使用此方法向 Lambda 报告初始化错误。当扩展在注册后无法初始化时调用它。Lambda 接收到错误后，进一步的 API 调用不会成功。扩展应在收到 Lambda 的响应后退出。

路径 – `/extension/init/error`

方法 – POST

Request headers（请求标头）

- `Lambda-Extension-Identifier` – 扩展名的唯一标识符。必需：是。类型：UUID 字符串。
- `Lambda-Extension-Function-Error-Type` – 扩展遇到的错误类型。必需：是。此标头由字符串值组成。Lambda 接受任何字符串，但建议您使用 <category.reason> 格式。例如：
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

请求正文参数

- `ErrorRequest` – 有关错误的其他信息。必需：否

此字段是具有以下结构的 JSON 对象：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

请注意，Lambda 接受任何 `errorType` 值。

以下示例显示了 Lambda 函数错误消息，其中函数无法解析调用中提供的事件数据。

Example 函数错误

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

响应代码

- 202 – 已接受
- 400 – 错误请求
- 403 – 禁止访问
- 500 – 容器错误。不可恢复状态。扩展应立即退出。

退出错误

扩展使用此方法在退出前向 Lambda 报告错误。当您遇到意外故障时调用它。Lambda 接收到错误后，进一步的 API 调用不会成功。扩展应在收到 Lambda 的响应后退出。

路径 – `/extension/exit/error`

方法 – POST

Request headers（请求标头）

- `Lambda-Extension-Identifier` – 扩展名的唯一标识符。必需：是。类型：UUID 字符串。

- `Lambda-Extension-Function-Error-Type` – 扩展遇到的错误类型。必需：是。此标头由字符串值组成。Lambda 接受任何字符串，但建议您使用 <category.reason> 格式。例如：
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

请求正文参数

- `ErrorRequest` – 有关错误的其他信息。必需：否

此字段是具有以下结构的 JSON 对象：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

请注意，Lambda 接受任何 `errorType` 值。

以下示例显示了 Lambda 函数错误消息，其中函数无法解析调用中提供的事件数据。

Example 函数错误

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

响应代码

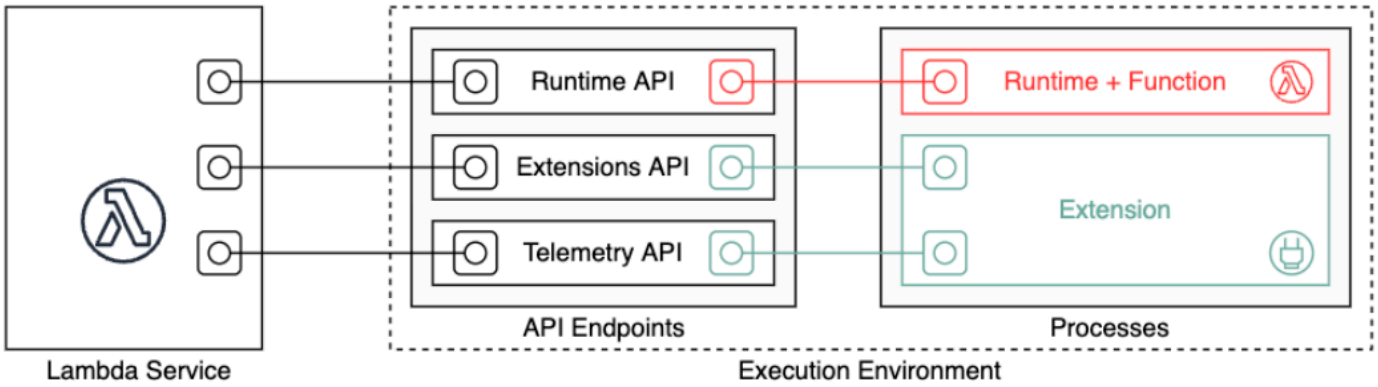
- 202 – 已接受
- 400 – 错误请求
- 403 – 禁止访问
- 500 – 容器错误。不可恢复状态。扩展应立即退出。

使用遥测 API 访问扩展的实时遥测数据

通过遥测 API，扩展可以直接从 Lambda 接收遥测数据。在函数初始化和调用期间，Lambda 会自动捕获遥测数据，其中包括日志、平台指标和平台跟踪。通过遥测 API，扩展可以近乎实时地直接从 Lambda 获取该遥测数据。

在 Lambda 执行环境中，您可以为 Lambda 扩展订阅遥测流。订阅后，Lambda 会自动将所有遥测数据流式发送到您的扩展。然后，您可以灵活处理、筛选这些数据，并将其分派到首选目标，例如 Amazon Simple Storage Service（Amazon S3）存储桶或第三方可观测性工具提供商。

下图显示了扩展 API 和遥测 API 如何从执行环境中将扩展连接到 Lambda。此外，运行时 API 还将您的运行时系统和函数连接到 Lambda。



> **Important**
>
> Lambda 遥测 API 取代 Lambda Logs API。尽管 Logs API 仍然功能完备，但我们建议您今后仅使用遥测 API。您可以使用遥测 API 或 Logs API 为扩展订阅遥测流。使用其中一个 API 进行订阅后，任何使用其他 API 进行订阅的尝试都会返回错误。

扩展可以使用遥测 API 订阅三种不同的遥测流：

- 平台遥测 – 日志、指标和跟踪，描述与执行环境运行时生命周期、扩展生命周期和函数调用相关的事件和错误。
- 函数日志 – Lambda 函数代码生成的自定义日志。
- 扩展日志 – Lambda 扩展代码生成的自定义日志。

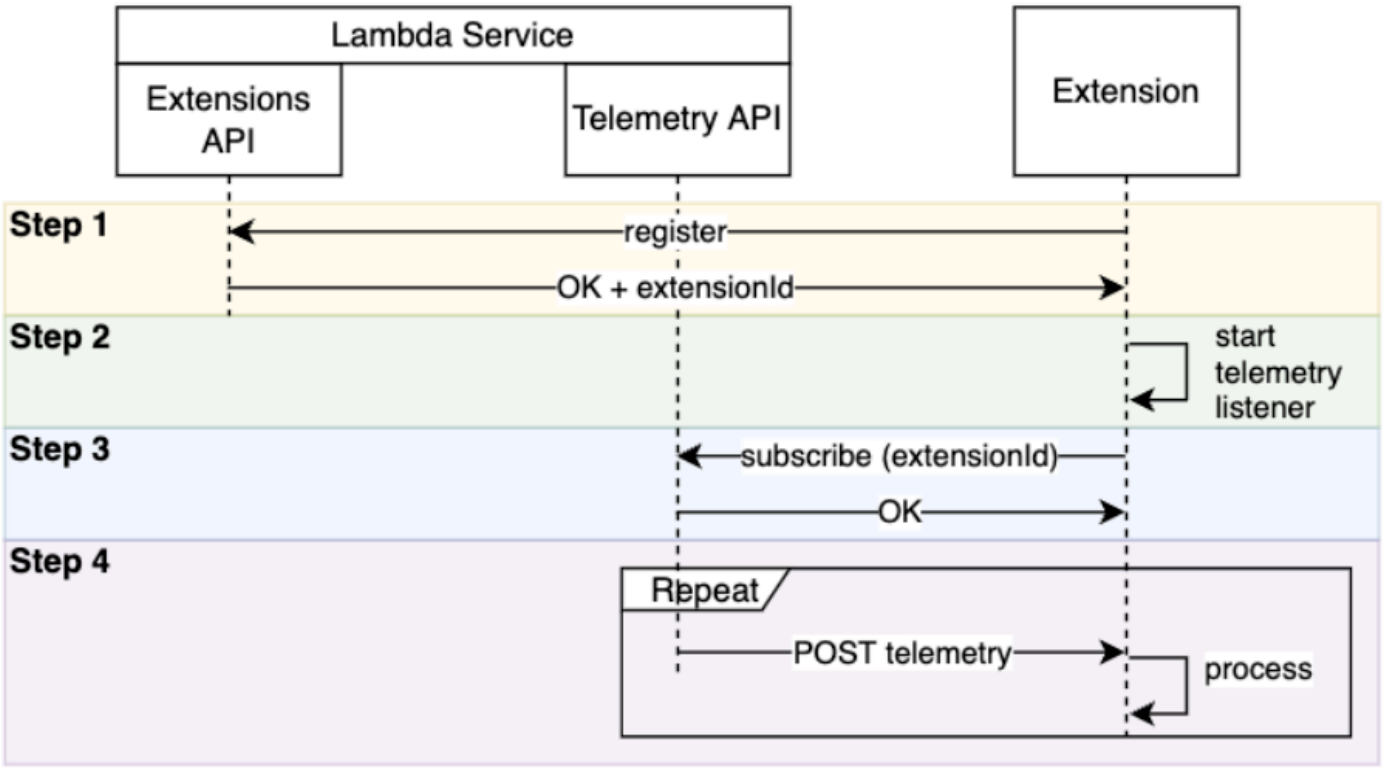
> **Note**
>
> 即使扩展订阅了遥测流，Lambda 也会将日志和指标发送到 CloudWatch，并将跟踪发送到 X-Ray（如果您已激活跟踪）。

Sections

使用遥测 API 创建扩展

Lambda 扩展在执行环境中作为独立进程运行。函数调用完成后，扩展可以继续运行。由于扩展作为单独的进程，因此您可以使用与函数代码不同的语言来编写这些扩展。我们建议使已编译的语言（如 Golang 或 Rust）来编写扩展。通过这种方式，扩展为独立的二进制文件，与任何支持的运行时兼容。

下图说明了创建使用遥测 API 接收和处理遥测数据的扩展的四步过程。



以下是每个步骤的详细内容：

1. 使用 the section called “扩展 API” 注册扩展。这步骤为您提供了 `Lambda-Extension-Identifier`，您需要在以下步骤中使用。有关如何注册扩展的更多信息，请参阅 the section called “注册扩展”。
2. 创建遥测侦听器。这可以是基本的 HTTP 或 TCP 服务器。Lambda 使用遥测侦听器的 URI 向扩展发送遥测数据。有关更多信息，请参阅 the section called “创建遥测侦听器”。
3. 使用遥测 API 中的订阅 API，为您的扩展订阅所需的遥测流。在此步骤中，您需要遥测侦听器的 URI。有关更多信息，请参阅 the section called “向遥测 API 发送订阅请求”。
4. 通过遥测侦听器从 Lambda 获取遥测数据。您可以对这些数据进行任何的自定义处理，例如将数据分派到 Amazon S3 或外部可观测性服务。

> **Note**
>
> Lambda 函数的执行环境可以作为其生命周期的一部分多次启动和停止。通常，您的扩展代码在函数调用期间运行，在关闭阶段也最多运行 2 秒。我们建议在遥测数据到达侦听器时对其进行批处理。然后，使用 `Invoke` 和 `Shutdown` 生命周期事件将每个批次发送到所需目的地。

注册扩展

在订阅遥测数据之前，您必须注册 Lambda 扩展。注册发生在扩展初始化阶段。以下示例显示了注册扩展的 HTTP 请求。

```
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
 Lambda-Extension-Name: lambda_extension_name
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

如果请求成功，订阅者将收到一条 HTTP 200 成功响应。响应标头包含 `Lambda-Extension-Identifier`。响应正文包含函数的其他属性。

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
    "functionName": "lambda_function",
    "functionVersion": "$LATEST",
    "handler": "lambda_handler",
    "accountId": "123456789012"
}
```

有关更多信息，请参阅 the section called “扩展 API 参考”。

创建遥测侦听器

Lambda 扩展必须有一个侦听器来处理来自遥测 API 的传入请求。以下代码显示了 Golang 中的示例遥测侦听器实施：

```
// Starts the server in a goroutine where the log events will be sent
func (s *TelemetryApiListener) Start() (string, error) {
 address := listenOnAddress()
 l.Info("[listener:Start] Starting on address", address)
 s.httpServer = &http.Server{Addr: address}
 http.HandleFunc("/", s.http_handler)
 go func() {
  err := s.httpServer.ListenAndServe()
  if err != http.ErrServerClosed {
   l.Error("[listener:goroutine] Unexpected stop on Http Server:", err)
```

```
   s.Shutdown()
  } else {
   l.Info("[listener:goroutine] Http Server closed:", err)
  }
 }()
 return fmt.Sprintf("http://%s/", address), nil
}

// http_handler handles the requests coming from the Telemetry API.
// Everytime Telemetry API sends log events, this function will read them from the
 response body
// and put into a synchronous queue to be dispatched later.
// Logging or printing besides the error cases below is not recommended if you have
 subscribed to
// receive extension logs. Otherwise, logging here will cause Telemetry API to send new
 logs for
// the printed lines which may create an infinite loop.
func (s *TelemetryApiListener) http_handler(w http.ResponseWriter, r *http.Request) {
 body, err := ioutil.ReadAll(r.Body)
 if err != nil {
  l.Error("[listener:http_handler] Error reading body:", err)
  return
 }

 // Parse and put the log messages into the queue
 var slice []interface{}
 _ = json.Unmarshal(body, &slice)

 for _, el := range slice {
  s.LogEventsQueue.Put(el)
 }

 l.Info("[listener:http_handler] logEvents received:", len(slice), " LogEventsQueue
 length:", s.LogEventsQueue.Len())
 slice = nil
}
```

指定目标协议

使用 Telemetry API 订阅接收遥测数据时，除了目标 URI 之外，您还可以指定目标协议：

```
{
    "destination": {
```

```
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda 接受两种接收遥测数据的协议：

- HTTP（推荐）– Lambda 会将遥测数据作为 JSON 格式的记录数组传送到本地 HTTP 端点（`http://sandbox.localdomain:${PORT}/${PATH}`）。$PATH 参数是可选的。Lambda 仅支持 HTTP，不支持 HTTPS。Lambda 通过 POST 请求传送遥测数据。
- TCP – Lambda 会将遥测数据以换行符分隔的 JSON（NDJSON）格式传送到 TCP 端口。

> **Note**
>
> 我们强烈建议使用 HTTP 而不是 TCP。使用 TCP 时，Lambda 平台无法在其将遥测数据传送到应用层时进行确认。因此，如果扩展崩溃，遥测数据可能会丢失。HTTP 则无此限制。

订阅接收遥测数据前，建立本地 HTTP 侦听器或 TCP 端口。在安装期间，请注意以下事项：

- Lambda 只会将遥测发送到执行环境内的目标。
- 如果没有侦听器，或者 POST 请求遇到错误，Lambda 会（使用退避）重试发送遥测数据。如果遥测侦听器崩溃，侦听器会在 Lambda 重新启动执行环境后恢复接收遥测数据。
- Lambda 会预留端口 9001。没有其他端口号限制或建议。

配置内存使用量和缓冲

执行环境的内存使用量随着订阅者数量的增加而线性增加。订阅会消耗内存资源，因为每个订阅都会打开一个新的内存缓冲区来存储遥测数据。缓冲区内存使用量计入执行环境中的总内存消耗量。

通过遥测 API 订阅接收遥测数据时，您可以缓冲遥测数据并将其批量传送给订阅者。要优化内存使用量，您可以指定缓冲配置：

```
{
    "buffering": {
        "maxBytes": 256*1024,
        "maxItems": 1000,
        "timeoutMs": 100
```

```
    }
}
```

参数	描述	默认值和限制
maxBytes	内存中缓冲的最大遥测量（以字节为单位）。	默认值：262144 最小值：262144 最大值：1048576。
maxItems	内存中缓冲的最大事件数。	默认值：10000 最小值：1000 最大值：10000
timeoutMs	缓冲批次的最长时间（以毫秒为单位）。	默认值：1000 最小值：25 最大值：30000

设置缓冲时，请记住以下几点：

- 如果任何输入流关闭，Lambda 会刷新日志。例如，如果运行时系统崩溃，就会发生这种情况。
- 在订阅请求中，每个订阅用户都可以自定义其缓冲配置。
- 在确定用于读取数据的缓冲区大小时，预计接收的有效负载大小为 `2 * maxBytes + metadataBytes`，其中 `maxBytes` 为缓冲设置的组成部分。要衡量需要考虑的 `metadataBytes` 数量，请查看以下元数据。Lambda 会将与此类似的元数据附加到每条记录：

  ```
  {
     "time": "2022-08-20T12:31:32.123Z",
     "type": "function",
     "record": "Hello World"
  }
  ```

- 如果订阅用户处理传入遥测数据的速度不够快，或者如果您的函数代码生成非常高的日志卷，Lambda 可能会删除记录以保持内存利用率限制。发生这种情况时，Lambda 会发送 `platform.logsDropped` 事件。

向遥测 API 发送订阅请求

Lambda 扩展可以通过向遥测 API 发送订阅请求来订阅接收遥测数据。订阅请求应包含有关您希望扩展订阅的事件类型的信息。此外，请求可以包含传送目标信息和缓冲配置。

在发送订阅请求之前，您必须拥有扩展 ID（`Lambda-Extension-Identifier`）。使用扩展 API 注册扩展时，您会从 API 响应中获得扩展 ID。

订阅发生在扩展初始化阶段。以下示例显示了订阅所有三个遥测流的 HTTP 请求：平台遥测、函数日志和扩展日志。

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2022-12-13",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

如果请求成功，订阅者将收到 HTTP 200 成功响应。

```
HTTP/1.1 200 OK
"OK"
```

入站遥测 API 消息

使用遥测 API 订阅后，扩展会自动开始通过 POST 请求接收来自 Lambda 的遥测数据。每个 POST 请求正文都包含一个 `Event` 对象数组。每个 `Event` 的架构如下：

```
{
```

```
    time: String,
    type: String,
    record: Object
}
```

- `time` 属性定义 Lambda 平台何时生成事件。这与事件的实际发生时间不同。`time` 的字符串值是 ISO 8601 格式的时间戳。
- `type` 属性定义事件类型。下表介绍了所有可能的值。
- `record` 属性定义包含遥测数据的 JSON 对象。此 JSON 对象的架构取决于 `type`。

下表汇总了所有类型的 `Event` 对象，并链接到每种事件类型的遥测 API Event 架构参考。

类别	事件类型	说明	事件记录架构
平台事件	`platform.initStart`	函数初始化已启动。	the section called “`platform.initStart`” 架构
平台事件	`platform.initRuntimeDone`	函数初始化已完成。	the section called “`platform.initRuntimeDone`” 架构
平台事件	`platform.initReport`	函数初始化的报告。	the section called “`platform.initReport`” 架构
平台事件	`platform.start`	函数调用已启动。	the section called “`platform.start`” 架构
平台事件	`platform.runtimeDone`	运行时通过成功或失败的方式完成对事件的处理。	the section called “`platform.runtimeDone`” 架构

类别	事件类型	说明	事件记录架构
平台事件	platform.report	函数调用的报告。	the section called “platform.report”架构
平台事件	platform.restoreStart	运行时恢复已开始。	the section called “platform.restoreStart”架构
平台事件	platform.restoreRuntimeDone	运行时恢复已完成。	the section called “platform.restoreRuntimeDone”架构
平台事件	platform.restoreReport	运行时恢复报告。	the section called “platform.restoreReport”架构
平台事件	platform.telemetrySubscription	扩展订阅了遥测 API。	the section called “platform.telemetrySubscription”架构
平台事件	platform.logsDropped	Lambda 删除了日志条目。	the section called “platform.logsDropped”架构
函数日志	function	来自函数代码的日志行。	the section called “function”架构
扩展日志	extension	来自扩展代码的日志行。	the section called “extension”架构

Lambda 遥测 API 参考

使用 Lambda 遥测 API 端点订阅遥测流的扩展。您可以从 AWS_LAMBDA_RUNTIME_API 环境变量中检索遥测 API 端点。若要发送 API 请求，请附加 API 版本（2022-07-01/）和 telemetry/。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

有关订阅响应版本 2022-12-13 的 OpenAPI 规范（OAS）定义，请参阅以下内容：

- HTTP – telemetry-api-http-schema.zip
- TCP – telemetry-api-tcp-schema.zip

API 操作

- 订阅

订阅

若要订阅遥测流，Lambda 扩展可以发送订阅 API 请求。

- 路径 – /telemetry
- 方法 – PUT
- 标头
  - Content-Type: application/json
- 请求正文参数
  - schemaVersion
    - 必需：是
    - 类型：字符串
    - 有效值："2022-12-13" 或 "2022-07-01"
  - 目标 – 定义遥测事件目标和事件传输协议的配置设置。
    - 必需：是
    - 类型：对象

```
{
    "protocol": "HTTP",
```

```
    "URI": "http://sandbox.localdomain:8080"
}
```

- 协议 – Lambda 用于发送遥测数据的协议。
  - 必需：是
  - 类型：字符串
  - 有效值："HTTP"|"TCP"
- URI – 要向其发送遥测数据的 URI。
  - 必需：是
  - 类型：字符串
- 有关更多信息，请参阅 the section called “指定目标协议”。

- 类型 – 您希望扩展订阅的遥测类型。
  - 必需：是
  - 类型：字符串数组
  - 有效值："platform"|"function"|"extension"
- 缓冲 – 事件缓冲的配置设置。
  - 必需：否
  - 类型：对象

```
{
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   }
}
```

  - maxItems – 在内存中缓冲的最大事件数。
    - 必需：否
    - 类型：整数
    - 默认值：1000
    - 最小值：1000
    - 最大值：10000

    - 必需：否
    - 类型：整数
    - 默认值：262144
    - 最小值：262144
    - 最大值：1048576。
- timeoutMs – 缓冲批的最长时间（以毫秒为单位）。
    - 必需：否
    - 类型：整数
    - 默认值：1000
    - 最小值：25
    - 最大值：30000
- 有关更多信息，请参阅 the section called “配置内存使用量和缓冲”。

订阅 API 请求示例

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2022-12-13",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

如果订阅请求成功，扩展将收到 HTTP 200 成功响应：

```
HTTP/1.1 200 OK
```

```
"OK"
```

如果订阅请求失败，扩展将收到错误响应。例如：

```
HTTP/1.1 400 OK
{
    "errorType": "ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

以下是扩展可以收到的一些其他响应代码：

- 200 – 已成功完成请求
- 202 – 已接受请求。本地测试环境中的订阅请求响应
- 400 – 错误请求
- 500 – 服务错误

Lambda 遥测 API Event 架构参考

使用 Lambda 遥测 API 端点订阅遥测流的扩展。您可以从 AWS_LAMBDA_RUNTIME_API 环境变量中检索遥测 API 端点。若要发送 API 请求，请附加 API 版本（2022-07-01/）和 telemetry/。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

有关订阅响应版本 2022-12-13 的 OpenAPI 规范（OAS）定义，请参阅以下内容：

- HTTP – telemetry-api-http-schema.zip
- TCP – telemetry-api-tcp-schema.zip

下表汇总了遥测 API 支持的所有 Event 对象类型。

类别	事件类型	说明	事件记录架构
平台事件	platform.initStart	函数初始化已启动。	the section called “platform.initStart ” 架构
平台事件	platform.initRuntimeDone	函数初始化已完成。	the section called “platform.initRuntimeDone ” 架构
平台事件	platform.initReport	函数初始化的报告。	the section called “platform.initReport ” 架构
平台事件	platform.start	函数调用已启动。	the section called “platform.start ” 架构
平台事件	platform.runtimeDone	运行时通过成功或失败的方式完成对事件的处理。	the section called “platform.runtimeDone ” 架构

类别	事件类型	说明	事件记录架构
平台事件	platform.report	函数调用的报告。	the section called “platform.report”架构
平台事件	platform.restoreStart	运行时恢复已开始。	the section called “platform.restoreStart”架构
平台事件	platform.restoreRuntimeDone	运行时恢复已完成。	the section called “platform.restoreRuntimeDone”架构
平台事件	platform.restoreReport	运行时恢复报告。	the section called “platform.restoreReport”架构
平台事件	platform.telemetrySubscription	扩展订阅了遥测 API。	the section called “platform.telemetrySubscription”架构
平台事件	platform.logsDropped	Lambda 删除了日志条目。	the section called “platform.logsDropped”架构
函数日志	function	来自函数代码的日志行。	the section called “function”架构
扩展日志	extension	来自扩展代码的日志行。	the section called “extension”架构

目录

遥测 API **Event** 对象类型

本节详细介绍 Lambda 遥测 API 支持的 Event 对象类型。在事件描述中，问号（？）表示该属性可能不存在于对象中。

platform.initStart

platform.initStart 事件表示函数初始化阶段已开始。platform.initStart Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

PlatformInitStart 对象具有以下属性：

- functionName – String
- functionVersion – String
- initializationType – the section called “InitType” 对象
- instanceId? – String
- instanceMaxMemory? – Integer
- phase – the section called “InitPhase” 对象
- runtimeVersion? – String
- runtimeVersionArn? – String

以下是 platform.initStart 类型的示例 Event：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.initStart",
    "record": {
        "initializationType": "on-demand",
        "phase": "init",
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
```

```
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

platform.initRuntimeDone

platform.initRuntimeDone 事件表示函数初始化阶段已完成。platform.initRuntimeDone Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = platform.initRuntimeDone
- record: PlatformInitRuntimeDone
```

PlatformInitRuntimeDone 对象具有以下属性：

- initializationType – the section called “InitType” 对象
- phase – the section called “InitPhase” 对象
- status – the section called “Status” 对象
- spans? – the section called “Span” 对象的列表。

以下是 platform.initRuntimeDone 类型的示例 Event：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initRuntimeDone",
    "record": {
        "initializationType": "on-demand"
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 70.5
            }
        ]
    }
}
```

platform.initReport

platform.initReport 事件包含函数初始化阶段的总体报告。platform.initReport Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = platform.initReport
- record: PlatformInitReport
```

PlatformInitReport 对象具有以下属性：

- errorType? – 字符串
- initializationType – the section called “InitType” 对象
- phase – the section called “InitPhase” 对象
- metrics – the section called “InitReportMetrics” 对象
- spans? – the section called “Span” 对象的列表。
- status – the section called “Status” 对象

以下是 platform.initReport 类型的示例 Event：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initReport",
    "record": {
        "initializationType": "on-demand",
        "status": "success",
        "phase": "init",
        "metrics": {
            "durationMs": 125.33
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 90.1
            }
        ]
    }
```

```
}
```

platform.start

platform.start 事件表示函数调用阶段已开始。platform.start Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

PlatformStart 对象具有以下属性：

- requestId – String
- version? – String
- tracing? – the section called “TraceContext”

以下是 platform.start 类型的示例 Event：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.start",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "version": "$LATEST",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        }
    }
}
```

platform.runtimeDone

platform.runtimeDone 事件表示函数调用阶段已完成。platform.runtimeDone Event 对象具有以下形状：

```
Event: Object
```

```
- time: String
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone
```

PlatformRuntimeDone 对象具有以下属性：

- errorType? – String
- metrics? – the section called “RuntimeDoneMetrics” 对象
- requestId – String
- status – the section called “Status” 对象
- spans? – the section called “Span” 对象的列表。
- tracing? – the section called “TraceContext” 对象

以下是 platform.runtimeDone 类型的示例 Event：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.runtimeDone",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "status": "success",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ],
        "metrics": {
            "durationMs": 140.0,
            "producedBytes": 16
        }
    }
}
```

platform.report

platform.report 事件包含函数调用阶段的总体报告。platform.report Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = platform.report
- record: PlatformReport
```

PlatformReport 对象具有以下属性：

- metrics – the section called “ReportMetrics” 对象
- requestId – String
- spans? – the section called “Span” 对象的列表。
- status – the section called “Status” 对象
- tracing? – the section called “TraceContext” 对象

以下是 platform.report 类型的示例 Event：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.report",
    "record": {
        "metrics": {
            "billedDurationMs": 694,
            "durationMs": 693.92,
            "initDurationMs": 397.68,
            "maxMemoryUsedMB": 84,
            "memorySizeMB": 128
        },
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
    }
}
```

platform.restoreStart

platform.restoreStart 事件表示函数环境还原事件已启动。在环境还原事件中，Lambda 会从缓存的快照创建环境，而不是从头开始初始化该环境。有关更多信息，请参阅 Lambda SnapStart。platform.restoreStart Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

PlatformRestoreStart 对象具有以下属性：

- functionName – String
- functionVersion – String
- instanceId? – String
- instanceMaxMemory? – String
- runtimeVersion? – String
- runtimeVersionArn? – String

以下是 platform.restoreStart 类型的示例 Event：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreStart",
    "record": {
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

platform.restoreRuntimeDone

platform.restoreRuntimeDone 事件表示函数环境还原事件已完成。在环境还原事件中，Lambda 会从缓存的快照创建环境，而不是从头开始初始化该环境。有关更多信息，请参阅 Lambda SnapStart。platform.restoreRuntimeDone Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
```

```
- record: PlatformRestoreRuntimeDone
```

PlatformRestoreRuntimeDone 对象具有以下属性：

- errorType? – String
- spans? – the section called “Span” 对象的列表。
- status – the section called “Status” 对象

以下是 platform.restoreRuntimeDone 类型的示例 Event：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreRuntimeDone",
    "record": {
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ]
    }
}
```

platform.restoreReport

platform.restoreReport 事件包含函数还原事件的总体报告。platform.restoreReport Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = platform.restoreReport
- record: PlatformRestoreReport
```

PlatformRestoreReport 对象具有以下属性：

- errorType? – 字符串
- metrics? – the section called “RestoreReportMetrics” 对象
- spans? – the section called “Span” 对象的列表。

- status – the section called “Status” 对象

以下是 platform.restoreReport 类型的示例 Event：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreReport",
    "record": {
        "status": "success",
        "metrics": {
            "durationMs": 15.19
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 30.0
            }
        ]
    }
}
```

platform.extension

extension 事件包含来自扩展代码的日志。extension Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

PlatformExtension 对象具有以下属性：

- events – String 列表
- name – String
- state – String

以下是 platform.extension 类型的示例 Event：

```
{
```

```
    "time": "2022-10-12T00:02:15.000Z",
    "type": "platform.extension",
    "record": {
        "events": [ "INVOKE", "SHUTDOWN" ],
        "name": "my-telemetry-extension",
        "state": "Ready"
    }
}
```

platform.telemetrySubscription

platform.telemetrySubscription 事件包含有关扩展订阅的信息。platform.telemetrySubscription Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
- record: PlatformTelemetrySubscription
```

PlatformTelemetrySubscription 对象具有以下属性：

- name – String
- state – String
- types – String 列表

以下是 platform.telemetrySubscription 类型的示例 Event：

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

platform.logsDropped

platform.logsDropped 事件包含有关已丢弃事件的信息。函数输出日志的速度过快，超过 Lambda 的处理能力时，Lambda 会发出 platform.logsDropped 事件。当 Lambda 无法按照函数

生成日志的速度将日志发送给 CloudWatch 或订阅到遥测 API 的扩展时，它会丢弃日志以防止函数的执行速度变慢。platform.logsDropped Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = platform.logsDropped
- record: PlatformLogsDropped
```

PlatformLogsDropped 对象具有以下属性：

- droppedBytes – Integer
- droppedRecords – Integer
- reason – String

以下是 platform.logsDropped 类型的示例 Event：

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.logsDropped",
    "record": {
        "droppedBytes": 12345,
        "droppedRecords": 123,
        "reason": "Some logs were dropped because the downstream consumer is slower
 than the logs production rate"
    }
}
```

function

function 事件包含来自函数代码的日志。function Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = function
- record: {}
```

record 字段的格式取决于函数的日志格式为纯文本还是 JSON 格式。要了解有关日志格式配置选项的更多信息，请参阅 the section called “配置 JSON 和纯文本日志格式”

以下是日志格式为纯文本的 function 类型 Event 示例：

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": "[INFO] Hello world, I am a function!"
}
```

以下是日志格式为 JSON 的 function 类型 Event 示例：

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": {
        "timestamp": "2022-10-12T00:03:50.000Z",
        "level": "INFO",
        "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
        "message": "Hello world, I am a function!"
    }
}
```

> **Note**
>
> 如果您使用的架构版本早于 2022-12-13 版本，则即使将函数的日志格式配置为 JSON，也会始终将 "record" 呈现为字符串。

extension

extension 事件包含来自扩展代码的日志。extension Event 对象具有以下形状：

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

record 字段的格式取决于函数的日志格式为纯文本还是 JSON 格式。要了解有关日志格式配置选项的更多信息，请参阅 the section called “配置 JSON 和纯文本日志格式”

以下是日志格式为纯文本的 extension 类型 Event 示例：

```
{
```

```
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
}
```

以下是日志格式为 JSON 的 `extension` 类型 `Event` 示例：

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": {
       "timestamp": "2022-10-12T00:03:50.000Z",
       "level": "INFO",
       "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
       "message": "Hello world, I am an extension!"
    }
}
```

> **Note**
>
> 如果您使用的架构版本早于 2022-12-13 版本，则即使将函数的日志格式配置为 JSON，也会始终将 `"record"` 呈现为字符串。

共享对象类型

本节详细介绍 Lambda 遥测 API 支持的共享对象类型。

`InitPhase`

描述初始化步骤发生时的阶段的字符串枚举。在大多数情况下，Lambda 在 `init` 阶段期间运行函数初始化代码。但是，在某些发生错误的情况下，Lambda 可能会在 `invoke` 阶段期间重新运行函数初始化代码。（此情况称为抑制初始化。）

- 类型 – `String`
- 有效值 – `init|invoke|snap-start`

`InitReportMetrics`

包含有关初始化阶段指标的对象。

- 类型 – Object

InitReportMetrics 对象具有以下形状：

```
InitReportMetrics: Object
- durationMs: Double
```

以下是 InitReportMetrics 对象的示例：

```
{
    "durationMs": 247.88
}
```

InitType

描述 Lambda 如何初始化环境的字符串枚举。

- 类型 – String
- 有效值 – on-demand|provisioned-concurrency

ReportMetrics

包含有关已完成阶段指标的对象。

- 类型 – Object

ReportMetrics 对象具有以下形状：

```
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

以下是 ReportMetrics 对象的示例：

```
{
```

```
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
    "maxMemoryUsedMB": 84,
    "memorySizeMB": 128
}
```

RestoreReportMetrics

包含有关已完成还原阶段指标的对象。

- 类型 – Object

RestoreReportMetrics 对象具有以下形状：

```
RestoreReportMetrics: Object
- durationMs: Double
```

以下是 RestoreReportMetrics 对象的示例：

```
{
    "durationMs": 15.19
}
```

RuntimeDoneMetrics

包含有关调用阶段指标的对象。

- 类型 – Object

RuntimeDoneMetrics 对象具有以下形状：

```
RuntimeDoneMetrics: Object
- durationMs: Double
- producedBytes?: Integer
```

以下是 RuntimeDoneMetrics 对象的示例：

```
{
    "durationMs": 200.0,
```

```
    "producedBytes": 15
}
```

Span

包含有关跨度详细信息的对象。跨度表示某个跟踪中的一个工作或操作单位。有关跨度的更多信息，请参阅 OpenTelemetry Docs 网站 Tracing API（跟踪 API）页面上的 Span（跨度）。

Lambda 对 `platform.RuntimeDone` 事件支持以下跨度：

- `responseLatency` 跨度描述了 Lambda 函数开始发送响应所需的时间。
- `responseDuration` 跨度描述 Lambda 函数完成发送整个响应所需的时间。
- `runtimeOverhead` 跨度描述了 Lambda 运行时系统发出信号表明其已准备好处理下一个函数调用所需的时间。这是返回函数响应后，运行时系统调用下一个调用 API 以获取下一个事件所需的时间。

以下是 `responseLatency` 跨度对象的示例：

```
{
        "name": "responseLatency",
        "start": "2022-08-02T12:01:23.521Z",
        "durationMs": 23.02
      }
```

Status

描述初始化或调用阶段状态的对象。如果状态为 `failure` 或 `error`，则 `Status` 对象还会包含一个 `errorType` 字段，用来描述错误。

- 类型 – `Object`
- 有效状态值 – `success|failure|error|timeout`

TraceContext

描述跟踪属性的对象。

- 类型 – `Object`

`TraceContext` 对象具有以下形状：

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
- value: String
```

以下是 TraceContext 对象的示例：

```
{
    "spanId": "073a49012f3c312e",
    "type": "X-Amzn-Trace-Id",
    "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
}
```

TracingType

描述 the section called “TraceContext” 对象中跟踪类型的字符串枚举。

- 类型 – String
- 有效值 – X-Amzn-Trace-Id

将 Lambda 遥测 API **Event** 对象转换为 OpenTelemetry 跨度

AWS Lambda 遥测 API 架构在语义上与 OpenTelemetry（OTel）兼容。这意味着您可以将 AWS Lambda 遥测 API `Event` 对象转换为 OpenTelemetry（OTel）跨度。转换时，不应将单个 `Event` 对象映射到单个 OTel 跨度。相反，您应该在单个 OTel 跨度中呈现与生命周期阶段相关的所有三个事件。例如，`start`、`runtimeDone` 和 `runtimeReport` 事件代表单个函数调用。将所有这三项事件作为单个 OTel 跨度呈现。

您可以使用跨度事件或子（嵌套）跨度来转换事件。此页面上的表格描述了两种方法的遥测 API 架构属性和 OTel 跨度属性之间的映射。有关 OTel 跨度的更多信息，请参阅 OpenTelemetry Docs 网站 Tracing API（跟踪 API）页面上的 Span（跨度）。

Sections

使用 Span 事件映射到 OTel 跨度

在下表中，e 代表来自遥测源的事件。

映射 *Start 事件

OpenTelemetry	Lambda 遥测 API 架构
`Span.Name`	扩展根据 `type` 字段生成此值。
`Span.StartTime`	使用 `e.time`。
`Span.EndTime`	不适用，因为事件尚未完成。
`Span.Kind`	设置为 `Server`。
`Span.Status`	设置为 `Unset`。
`Span.TraceId`	解析 `e.tracing.value` 中找到的 AWS X-Ray 标头，然后使用 `TraceId` 值。
`Span.ParentId`	解析 `e.tracing.value` 中找到的 X-Ray 标头，然后使用 `Parent` 值。

OpenTelemetry	Lambda 遥测 API 架构
Span.SpanId	如果可用，请使用 e.tracing.spanId 。否则，生成一个新的 SpanId。
Span.SpanContext.TraceState	X-Ray 跟踪上下文不适用。
Span.SpanContext.TraceFlags	解析 e.tracing.value 中找到的 X-Ray 标头，然后使用 Sampled 值。
Span.Attributes	扩展可以在此处添加任何自定义值。

映射 *RuntimeDone 事件

OpenTelemetry	Lambda 遥测 API 架构
Span.Name	扩展根据 type 字段生成此值。
Span.StartTime	使用匹配 *Start 事件中的 e.time。 或者，请使用 e.time - e.metrics.durationMs 。
Span.EndTime	不适用，因为事件尚未完成。
Span.Kind	设置为 Server。
Span.Status	如果 e.status 不等于 success，则设置为 Error。 否则，设置为 Ok。
Span.Events[]	使用 e.spans[] 。
Span.Events[i].Name	使用 e.spans[i].name 。
Span.Events[i].Time	使用 e.spans[i].start 。

OpenTelemetry	Lambda 遥测 API 架构
Span.TraceId	解析 e.tracing.value 中找到的 AWS X-Ray 标头，然后使用 TraceId 值。
Span.ParentId	解析 e.tracing.value 中找到的 X-Ray 标头，然后使用 Parent 值。
Span.SpanId	使用 *Start 事件中的相同 SpanId。如果不可用，则使用 e.tracing.spanId 或生成一个新的 SpanId。
Span.SpanContext.TraceState	X-Ray 跟踪上下文不适用。
Span.SpanContext.TraceFlags	解析 e.tracing.value 中找到的 X-Ray 标头，然后使用 Sampled 值。
Span.Attributes	扩展可以在此处添加任何自定义值。

映射 *Report 事件

OpenTelemetry	Lambda 遥测 API 架构
Span.Name	扩展根据 type 字段生成此值。
Span.StartTime	使用匹配 *Start 事件中的 e.time。 或者，请使用 e.time - e.metrics.durationMs 。
Span.EndTime	使用 e.time。
Span.Kind	设置为 Server。
Span.Status	使用与 *RuntimeDone 事件相同的值。
Span.TraceId	解析 e.tracing.value 中找到的 AWS X-Ray 标头，然后使用 TraceId 值。

OpenTelemetry	Lambda 遥测 API 架构
Span.ParentId	解析 e.tracing.value 中找到的 X-Ray 标头，然后使用 Parent 值。
Span.SpanId	使用 *Start 事件中的相同 SpanId。如果不可用，则使用 e.tracing.spanId 或生成一个新的 SpanId。
Span.SpanContext.TraceState	X-Ray 跟踪上下文不适用。
Span.SpanContext.TraceFlags	解析 e.tracing.value 中找到的 X-Ray 标头，然后使用 Sampled 值。
Span.Attributes	扩展可以在此处添加任何自定义值。

使用子跨度映射到 OTel 跨度

下表描述了如何将 Lambda 遥测 API 事件转换为带 *RuntimeDone 跨度子（嵌套）跨度的 OTel 跨度。对于 *Start 和 *Report 映射，请参阅 the section called “使用 Span 事件映射到 OTel 跨度” 中的表，因为它们与子跨度相同。在此表中，e 代表来自遥测源的事件。

映射 *RuntimeDone 事件

OpenTelemetry	Lambda 遥测 API 架构
Span.Name	扩展根据 type 字段生成此值。
Span.StartTime	使用匹配 *Start 事件中的 e.time。 或者，请使用 e.time - e.metrics.durationMs 。
Span.EndTime	不适用，因为事件尚未完成。
Span.Kind	设置为 Server。
Span.Status	如果 e.status 不等于 success，则设置为 Error。

OpenTelemetry	Lambda 遥测 API 架构
	否则，设置为 Ok。
Span.TraceId	解析 e.tracing.value 中找到的 AWS X-Ray 标头，然后使用 TraceId 值。
Span.ParentId	解析 e.tracing.value 中找到的 X-Ray 标头，然后使用 Parent 值。
Span.SpanId	使用 *Start 事件中的相同 SpanId。如果不可用，则使用 e.tracing.spanId 或生成一个新的 SpanId。
Span.SpanContext.TraceState	X-Ray 跟踪上下文不适用。
Span.SpanContext.TraceFlags	解析 e.tracing.value 中找到的 X-Ray 标头，然后使用 Sampled 值。
Span.Attributes	扩展可以在此处添加任何自定义值。
ChildSpan[i].Name	使用 e.spans[i].name 。
ChildSpan[i].StartTime	使用 e.spans[i].start 。
ChildSpan[i].EndTime	使用 e.spans[i].start + e.spans[i].durations 。
ChildSpan[i].Kind	与父 Span.Kind 一样。
ChildSpan[i].Status	与父 Span.Status 一样。
ChildSpan[i].TraceId	与父 Span.TraceId 一样。
ChildSpan[i].ParentId	使用父 Span.SpanId 。
ChildSpan[i].SpanId	生成新的 SpanId。
ChildSpan[i].SpanContext.TraceState	X-Ray 跟踪上下文不适用。

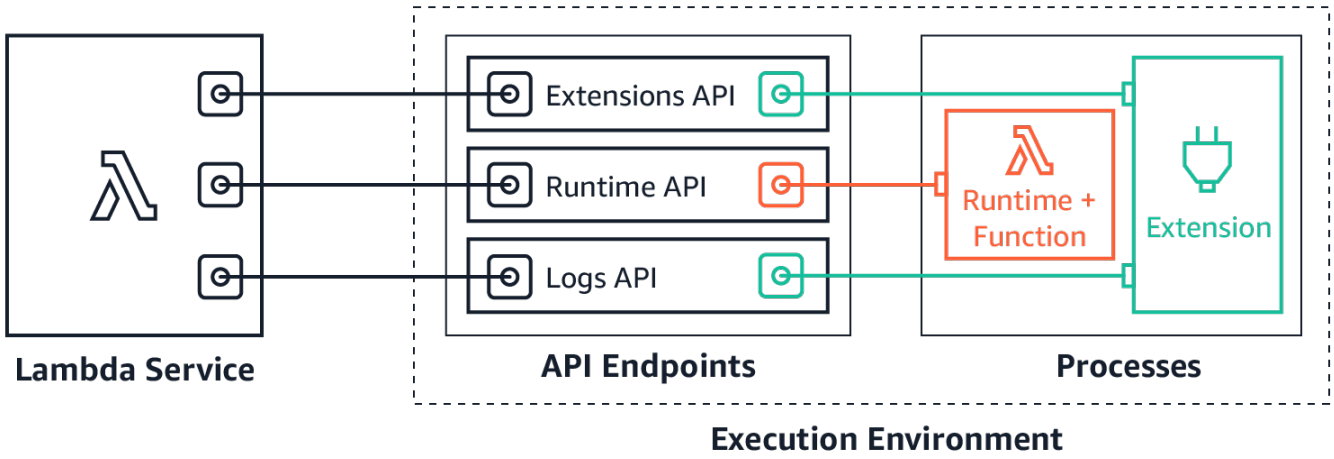
OpenTelemetry	Lambda 遥测 API 架构
`ChildSpan[i].SpanContext.TraceFlags`	与父 `Span.SpanContext.TraceFlags` 一样。

使用 Lambda 日志 API

> **Important**
>
> Lambda 遥测 API 取代 Lambda Logs API。尽管 Logs API 仍然功能完备，但我们建议您今后仅使用遥测 API。您可以使用遥测 API 或 Logs API 为扩展订阅遥测流。使用其中一个 API 进行订阅后，任何使用其他 API 进行订阅的尝试都会返回错误。

Lambda 会自动捕获运行时日志并将其流式传输到 Amazon CloudWatch。此日志流包含函数代码和扩展生成的日志，以及 Lambda 作为函数调用的一部分生成的日志。

Lambda 扩展可以使用 Lambda Runtime Logs API 从 Lambda 执行环境中直接订阅日志流。Lambda 将日志流式传输到扩展，便于扩展处理、筛选日志并将其发送到首选目标。



Logs API 允许扩展订阅三种不同的日志流：

- Lambda 函数生成并写入 `stdout` 或 `stderr` 的函数日志。
- 扩展代码生成的扩展日志。
- Lambda 平台日志，记录与调用和扩展相关的事件和错误。

> **Note**
>
> Lambda 会将所有日志发送到 CloudWatch，即使扩展订阅了一个或多个日志流也是如此。

主题

订阅接收日志

Lambda 扩展可以通过向 Logs API 发送订阅请求来订阅接收日志。

要订阅接收日志，则需要扩展标识符 (`Lambda-Extension-Identifier`)。首先，注册扩展以接收扩展标识符。然后，在初始化期间订阅 Logs API。初始化阶段完成后，Lambda 不会处理订阅请求。

> **Note**
>
> Logs API 订阅是幂等的。重复的订阅请求不会导致重复订阅。

内存使用量

内存使用量随着订阅者数量的增加而线性增加。订阅会消耗内存资源，因为每个订阅都会打开一个新的内存缓冲区来存储日志。为了帮助优化内存使用量，您可以调整缓冲配置。缓冲区内存使用量计入执行环境中的总内存消耗量。

目标协议

您可以选择以下协议之一来接收日志：

1. HTTP（推荐）– Lambda 会将日志作为 JSON 格式的记录数组传送到本地 HTTP 端点 (`http://sandbox.localdomain:${PORT}/${PATH}`)。`$PATH` 参数是可选的。请注意，只支持 HTTP，不支持 HTTPS。您可以选择通过 PUT 或 POST 来接收日志。
2. TCP – Lambda 会将日志以换行符分隔的 JSON (NDJSON) 格式传送到 TCP 端口。

我们建议使用 HTTP 而不是 TCP。使用 TCP 时，Lambda 平台无法在其将日志传送到应用层时进行确认。因此，如果扩展崩溃，日志可能会丢失。HTTP 则无此限制。

我们还建议在订阅接收日志之前设置本地 HTTP 侦听器或 TCP 端口。在安装期间，请注意以下事项：

- Lambda 只会将日志发送到执行环境内的目标。
- 如果没有侦听器，或者 POST 或 PUT 请求导致错误，Lambda 会尝试重试发送日志（退避尝试）。如果日志订阅者崩溃，它会在 Lambda 重新启动执行环境后继续接收日志。
- Lambda 会预留端口 9001。没有其他端口号限制或建议。

缓冲配置

Lambda 可以缓冲日志并将其发送给订阅者。您可以通过指定以下可选字段在订阅请求中配置此行为。请注意，Lambda 对未指定的字段使用默认值。

- timeoutMs – 缓冲批的最长时间（以毫秒为单位）。默认值：1,000。最低：25。最大值：30,000。
- maxBytes – 在内存中缓冲的最大大小（以字节为单位）。默认值：262,144。最小值：262,144。最大值：1,048,576。
- maxItems – 在内存中缓冲的最大事件数。默认值：10,000。最小值：1,000。最大值：10,000。

在缓冲配置期间，请注意以下几点：

- 如果任何输入流关闭（例如，运行时崩溃），Lambda 会刷新日志。
- 在订阅请求中，每个订阅者都可以指定不同的缓冲配置。
- 考虑读取数据所需的缓冲区大小。预计最多可接收 `2*maxBytes+metadata` 的有效负载，其中，`maxBytes` 在订阅请求中配置。例如，Lambda 会将以下元数据字节添加到每条记录：

```
{
"time": "2020-08-20T12:31:32.123Z",
"type": "function",
"record": "Hello World"
}
```

- 如果订阅者处理传入日志的速度不够快，Lambda 可能会删除日志以保持内存利用率限制。为指示丢弃记录数，Lambda 会发送一个 `platform.logsDropped` 日志。有关更多信息，请参阅 the section called “Lambda：并非所有我的函数的日志都会出现”。

示例订阅

以下示例显示订阅平台和函数日志的请求。

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs HTTP/1.1
{ "schemaVersion": "2020-08-15",
  "types": [
      "platform",
      "function"
    ],
  "buffering": {
      "maxItems": 1000,
      "maxBytes": 262144,
      "timeoutMs": 100
    },
  "destination": {
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080/lambda_logs"
  }
}
```

如果请求成功，订阅者将收到一条 HTTP 200 成功响应。

```
HTTP/1.1 200 OK
"OK"
```

Logs API 的示例代码

有关显示如何将日志发送到自定义目标的示例代码，请参阅 AWS Lambda 计算博客上的使用 AWS 扩展将日志发送到自定义目标。

有关显示如何开发基本 Lambda 扩展和订阅 Logs API 的 Python 和 Go 代码示例，请参阅AWS示例 GitHub 存储库上的 AWS Lambda扩展。有关构建 Lambda 扩展的更多信息，请参阅 the section called “扩展 API”。

Logs API 参考

您可以从 `AWS_LAMBDA_RUNTIME_API` 环境变量中检索 Logs API 终端节点。要发送 API 请求，请在 API 路径前使用前缀 `2020-08-15/`。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs
```

Logs API 版本 2020-08-15 的 OpenAPI 规范可从此处获得：logs-api-request.zip

订阅

要订阅一个或多个在 Lambda 执行环境中可用的日志流，扩展会发送 Subscribe API 请求。

路径 – `/logs`

方法 – PUT

主体参数

`destination`请参阅 the section called “目标协议”。必需：是。类型：字符串。

`buffering`请参阅 the section called “缓冲配置”。必需：否 类型：字符串。

`types` – 要接收的日志类型的数组。必需：是。类型：字符串数组。有效值："platform"、"function"、"extension"。

`schemaVersion` – 必需：否。默认值：“2020-08-15”。设置为“2021-03-18”以便扩展接收 `platform.runtimeDone` 消息。

响应参数

提供适用于 HTTP 和 TCP 协议的订阅响应版本 2020-08-15 的 OpenAPI 规范：

- HTTP：logs-api-http-response.zip
- TCP：logs-api-tcp-response.zip

响应代码

- 200 – 已成功完成请求
- 202 – 已接受请求。在本地测试期间响应订阅请求。
- 4XX – 错误请求
- 500 – 服务错误

如果请求成功，订阅者将收到一条 HTTP 200 成功响应。

```
HTTP/1.1 200 OK
"OK"
```

如果请求失败，订阅者将收到一条错误响应。例如：

```
HTTP/1.1 400 OK
{
    "errorType": "Logs.ValidationError",
    "errorMessage": URI port is not provided; types should not be empty"
}
```

日志消息

Logs API 允许扩展订阅三种不同的日志流：

- 函数 – Lambda 函数生成并写入 `stdout` 或 `stderr` 的日志。
- 扩展 – 扩展代码生成的日志。
- 平台 – 运行时平台生成的平台日志，用于记录与调用和扩展相关的事件和错误。

主题

函数日志

Lambda 函数和内部扩展会生成函数日志并将其写入 `stdout` 或 `stderr`。

以下示例显示了函数日志消息的格式。{ "time": "2020-08-20T12:31:32.123Z", "type": "function", "record": Stack trace:\n\my-function (line 10)\n" }

扩展日志

扩展可以生成扩展日志。日志格式与函数日志的格式相同。

平台日志

Lambda 会为平台事件（例如 `platform.start`、`platform.end` 和 `platform.fault`）生成日志消息。

或者，您可以订阅 2021-03-18 版本的 Logs API 架构，其中包括 `platform.runtimeDone` 日志消息。

示例平台日志消息

以下示例显示了平台开始日志和平台结束日志。这些日志指定了 requestId 指定的调用的调用开始时间和调用结束时间。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.start",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.end",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
```

platform.initRuntimeDone 日志消息显示 Runtime init 子阶段的状态，该子阶段是初始化生命周期阶段的一部分。Runtime init 成功后，运行时会发送 /next 运行时 API 请求（针对 on-demand 和 provisioned-concurrency 初始化类型）或 restore/next（针对 snap-start 初始化类型）。以下示例显示适用于 snap-start 初始化类型的成功 platform.initRuntimeDone 日志消息。

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initRuntimeDone",
  "record":{
      "initializationType":"snap-start",
      "status":"success"
  }
}
```

platform.initReport 日志消息显示该 Init 阶段的持续时间以及该阶段已计费的毫秒数。当初始化类型为 provisioned-concurrency 时，Lambda 会在调用期间发送此消息。当初始化类型为 snap-start 时，Lambda 会在还原快照后发送此消息。以下示例显示适用于 snap-start 初始化类型的 platform.initReport 日志消息。

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initReport",
  "record":{
      "initializationType":"snap-start",
      "metrics":{
```

```
          "durationMs":731.79,
          "billedDurationMs":732
          }
  }
}
```

平台报告日志包含有关 requestId 指定的调用的指标。仅当调用包括冷启动时，initDurationMs 字段才会包含在日志中。如果 AWS X-Ray 跟踪处于活动状态，日志会包含 X-Ray 元数据。以下示例显示了包括冷启动的调用的平台报告日志。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.report",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56",
        "metrics": {"durationMs": 101.51,
            "billedDurationMs": 300,
            "memorySizeMB": 512,
            "maxMemoryUsedMB": 33,
            "initDurationMs": 116.67
        }
    }
}
```

平台故障日志捕获运行时或执行环境错误。以下示例显示了平台故障日志消息。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.fault",
    "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before
 completing request"
}
```

> **Note**
>
> AWS 目前正在实施对 Lambda 服务的更改。由于这些更改，您可能会看到 AWS 账户 中不同 Lambda 函数发出的系统日志消息和跟踪分段的结构和内容之间存在细微差异。
> 受此更改影响的日志输出之一是平台故障日志 "record" 字段。以下示例显示了新旧格式的说明性 "record" 字段。新样式的故障日志包含更简洁的消息
> 这些更改将在未来几周内实施，除中国和 GovCloud 区域外，所有 AWS 区域 的函数都将过渡到使用新格式的日志消息和跟踪分段。

Example 平台故障日志记录（旧样式）

```
"record":"RequestId: ...\tError: Runtime exited with error: exit status
 255\nRuntime.ExitError"
```

Example 平台故障日志记录（新样式）

```
"record":"RequestId: ... Status: error\tErrorType: Runtime.ExitError"
```

在扩展利用扩展 API 注册时，Lambda 会生成平台扩展日志。以下示例显示了平台扩展消息。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.extension",
    "record": {"name": "Foo.bar",
        "state": "Ready",
        "events": ["INVOKE", "SHUTDOWN"]
     }
}
```

在扩展订阅日志 API 时，Lambda 会生成平台日志订阅日志。以下示例显示了日志订阅消息。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
    "record": {"name": "Foo.bar",
        "state": "Subscribed",
        "types": ["function", "platform"],
    }
}
```

当扩展无法处理其接收的日志数量时，Lambda 会生成平台日志丢弃日志。以下示例显示了 `platform.logsDropped` 日志消息。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsDropped",
    "record": {"reason": "Consumer seems to have fallen behind as it has not
 acknowledged receipt of logs.",
        "droppedRecords": 123,
```

```
        "droppedBytes" 12345
    }
}
```

platform.restoreStart 日志消息显示 Restore 阶段的开始时间（仅限 snap-start 初始化类型）。例如：

```
{
  "time":"2022-07-17T18:43:44.782Z",
  "type":"platform.restoreStart",
  "record":{}
}
```

platform.restoreReport 日志消息显示 Restore 阶段的持续时间以及该阶段已计费的毫秒数（仅限 snap-start 初始化类型）。例如：

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreReport",
  "record":{
      "metrics":{
          "durationMs":70.87,
          "billedDurationMs":13
      }
  }
}
```

平台 **runtimeDone** 消息

如果在订阅请求中将架构版本设置为“2021-03-18”，则在函数调用成功完成或出现错误后，Lambda 会发送一条 platform.runtimeDone 消息。扩展可以使用此消息来停止此函数调用的所有遥测收集。

架构版本 2021-03-18 中日志事件类型的 OpenAPI 规范可从此处获得：schema-2021-03-18.zip

在运行时发送 Next 或 Error 运行时 API 请求时，Lambda 会生成 platform.runtimeDone 日志消息。platform.runtimeDone 日志会通知 Logs API 的使用者函数调用已完成。扩展可以使用此信息来决定何时发送在调用期间收集的所有遥测数据。

示例

在函数调用完成时，Lambda 会在运行时发送 NEXT 请求后发送 platform.runtimeDone 消息。以下示例显示了每个状态值的消息：成功、失败和超时。

Example 成功消息示例

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
       "requestId":"6f7f0961f83442118a7af6fe80b88",
       "status": "success"
    }
}
```

Example 失败消息示例

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "failure"
   }
}
```

Example 超时消息示例

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "timeout"
  }
}
```

Example platform.restoreRuntimeDone 消息示例（仅限 **`snap-start`** 初始化类型）

platform.restoreRuntimeDone 日志消息显示 `Restore` 阶段是否成功。当运行时发送 `restore/next` 运行时 API 请求时，Lambda 会发送此消息。存在三种可能的状态：成功、失败和超时。以下示例显示了成功的 platform.restoreRuntimeDone 日志消息。

```
{
  "time":"2022-07-17T18:43:45.936Z",
```

```
  "type":"platform.restoreRuntimeDone",
  "record":{
      "status":"success"
  }
}
```

排查 Lambda 中的问题

以下主题为您在使用 Lambda API、控制台或工具时可能遇到的错误和问题提供故障排除建议。如果您发现某个问题未在此处列出，可以使用此页上的 Feedback 按钮来报告。

有关更多故障排除建议和常见支持问题的答案，请访问AWS知识中心。

有关对 Lambda 应用程序进行调试和故障排除的更多信息，请参阅 Serverless Land 中的 Debugging。

主题

Lambda 中的部署问题疑难解答

更新函数时，Lambda 使用更新的代码或设置来启动函数的新实例，从而部署更改。部署错误会阻止新版本的使用，并可能由于部署程序包、代码、权限或工具中的问题而引起。

当您使用 Lambda API 或客户端（如 AWS CLI）将更新直接部署到函数时，您可以直接在输出中查看 Lambda 中的错误。如果您使用 AWS CloudFormation、AWS CodeDeploy 或 AWS CodePipeline 等服务，请在该服务的日志或事件流中查找来自 Lambda 的响应。

以下主题为您在使用 Lambda API、控制台或工具时可能遇到的错误和问题提供故障排除建议。如果您发现某个问题未在此处列出，可以使用此页上的 Feedback 按钮来报告。

有关更多故障排除建议和常见支持问题的答案，请访问AWS知识中心。

有关对 Lambda 应用程序进行调试和故障排除的更多信息，请参阅 Serverless Land 中的 Debugging。

主题

- Amazon S3：错误代码 PermanentRedirect。
- 常规：找不到、无法加载、无法导入、找不到类、没有此类文件或目录
- 常规：未定义的方法处理程序
- Lambda：图层转换失败
- Lambda：InvalidParameterValueException or RequestEntityTooLargeException
- Lambda：InvalidParameterValueException
- Lambda：并发和内存限额

常规：权限被拒绝/无法加载此类文件

错误：EACCES: permission denied, open '/var/task/index.js'（EACCES: 权限被拒绝，打开“/var/task/index.js”）

错误：cannot load such file -- function（无法加载此文件 – 函数）

错误：[Errno 13] Permission denied: '/var/task/function.py'（[Errno 13] 权限被拒绝:“/var/task/function.py”）

Lambda 运行时需要权限才能读取部署包中的文件。在 Linux 权限八进制表示法中，Lambda 对于不可执行文件（rw-r--r--）需要 644 个权限，对于目录和可执行文件需要 755 个权限（rwxr-xr-x）。

在 Linux 和 MacOS 中，使用 chmod 命令更改部署包中文件和目录的文件权限。例如，要为可执行文件提供正确的权限，请运行以下命令。

```
chmod 755 <filepath>
```

要在 Windows 中更改文件权限，请参阅 Microsoft Windows 文档中的 Set, View, Change, or Remove Permissions on an Object。

常规：调用 UpdateFunctionCode 时出错

错误：An error occurred (RequestEntityTooLargeException) when calling the UpdateFunctionCode operation（在调用 UpdateFunctionCode 操作时出错 (RequestEntityTooLargeException)）

在将部署包或层归档直接上载到 Lambda 时，ZIP 文件的大小最多为 50 MB。要上载一个较大的文件，请将此文件存储在 Amazon S3 中并使用 S3Bucket 和 S3Key 参数。

> **Note**
>
> 当您直接使用 AWS CLI、AWS 开发工具包或通过其他方式上传文件时，二进制 ZIP 文件将转换为 base64，其大小将增加约 30%。为了支持这一点以及请求中其他参数的大小，Lambda 应用的实际请求大小限制会更大。因此，50 MB 的限制是近似值。

Amazon S3：错误代码 PermanentRedirect。

错误：Error occurred while GetObject.（GetObject 时发生错误。）S3 错误代码：PermanentRedirect。S3 错误消息：存储桶位于此区域中：us-east-2。请使用此区域重试请求

当您从 Amazon S3 存储桶上载函数的部署包时，存储桶必须与函数位于同一区域。当您在 UpdateFunctionCode 调用中指定 Amazon S3 对象或在 AWS CLI 或 AWS SAM CLI 中使用程序包并部署命令时，可能会出现此问题。为您在其中开发应用程序的每个区域创建部署构件存储桶。

常规：找不到、无法加载、无法导入、找不到类、没有此类文件或目录

错误：找不到模块“function”

错误：cannot load such file -- function（无法加载此文件 – 函数）

错误：无法导入模块“function”

错误：找不到类：function.Handler

错误：fork/exec/var/task/function：没有这样的文件或目录

错误：无法从程序集“Function”加载类型“Function.Handler”。

函数处理程序配置中文件或类的名称与您的代码不匹配。有关更多信息，请参阅下文的小节。

常规：未定义的方法处理程序

错误：index.handler 未定义或未导出

错误：模块“function”上缺少处理程序“handler”

错误：undefined method `handler' for #<LambdaHandler:0x000055b76ccebf98>（#<LambdaHandler:0x000055b76ccebf98> 的未定义的方法“handler”）

错误：在类 function.Handler 上没有找到具有正确方法签名的名为 handleRequest 的公共方法。

错误：Unable to find method 'handleRequest' in type 'Function.Handler' from assembly 'Function'（无法从程序集“Function”的类型“Function.Handler”中找到方法“handleRequest”）

函数处理程序配置中的处理程序方法的名称与您的代码不匹配。每个运行时为处理程序定义一个命名约定，例如 *filename.methodname*。处理程序是函数代码中的方法，在调用函数时由运行时运行该方法。

对于某些语言，Lambda 提供了包含接口的库，该接口需要具有特定名称的处理程序方法。有关每种语言的处理程序命名的详细信息，请参阅以下主题。

- 使用 Node.js 构建 Lambda 函数
- 使用 Python 构建 Lambda 函数
- 使用 Ruby 构建 Lambda 函数
- 使用 Java 构建 Lambda 函数
- 使用 Go 构建 Lambda 函数
- 使用 C# 构建 Lambda 函数
- 使用 PowerShell 构建 Lambda 函数

Lambda：图层转换失败

错误：Lambda 图层转换失败。有关解决此问题的建议，请参阅《Lambda 用户指南》中的“排查 Lambda 中的部署问题”页面。

当您使用图层配置 Lambda 函数时，Lambda 会将该层与您的函数代码合并。如果此过程无法完成，Lambda 会返回此错误。如果遭遇此错误，请执行以下步骤：

- 从图层中删除所有未使用的文件
- 删除图层中的所有符号链接
- 重命名任何与函数层中目录同名的文件

Lambda：InvalidParameterValueException or RequestEntityTooLargeException

错误：InvalidParameterValueException: Lambda was unable to configure your environment variables because the environment variables you have provided exceeded the 4KB limit.

（InvalidParameterValueException：Lambda 无法配置您的环境变量，因为您提供的环境变量超过了 4KB 限制。）测量的字符串：{"A1":"uSFeY5cyPiPn7AtnX5BsM...

错误：RequestEntityTooLargeException：对 UpdateFunctionConfiguration 操作的请求必须小于 5120 字节

存储在函数配置中的变量对象的最大体积不得超过 4096 个字节。这包括密钥名称、值、引号、逗号和括号。HTTP 请求正文的总大小也受到限制。

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    ...
}
```

在此示例中，对象为 39 个字符，并且它在存储（不含空格）为字符串 `{"BUCKET":"amzn-s3-demo-bucket","KEY":"file.txt"}` 时占用 39 个字节。环境变量值中的每个标准 ASCII 字符使用一个字节。每个扩展 ASCII 和 Unicode 字符可以使用 2 到 4 个字节。

Lambda：InvalidParameterValueException

错误：InvalidParameterValueException: Lambda 无法配置您的环境变量，因为您提供的环境变量包含目前不支持修改的保留键。

Lambda 会预留一些环境变量键供内部使用。例如，运行时使用 `AWS_REGION` 来确定当前区域且它不能被覆盖。运行时使用其他变量（如 `PATH`），但这些变量可以在函数配置中扩展。有关完整列表，请参阅定义运行时环境变量。

Lambda：并发和内存限额

Error: Specified ConcurrentExecutions for function decreases account's UnreservedConcurrentExecution below its minimum value（错误：为函数指定的 ConcurrentExecutions 会导致账户的 UnreservedConcurrentExecution 降至其最小值以下）

Error: 'MemorySize' value failed to satisfy constraint: Member must have value less than or equal to 3008（错误：“内存大小”值不满足约束条件：成员的值必须小于或等于 3008）

超出您账户的并发或内存限额时，就会出现这些错误。新的 AWS 账户减少了并发和内存配额。若要解决与并发相关的错误，您可以请求增加配额。您无法请求增加配额。

- 并发：如果您尝试使用预留或预配置并发创建函数，或者如果您的每函数并发请求（PutFunctionConcurrency）超出您账户的并发限额，则可能会收到错误。
- 内存：如果分配给函数的内存量超过您账户的内存限额，则会出现错误。

Lambda 中的调用问题疑难解答

当您调用 Lambda 函数时，Lambda 在将事件发送到函数或（对于异步调用）发送到事件队列之前，先验证请求并检查扩展容量。调用错误可能是由请求参数、事件结构、函数设置、用户权限、资源权限或限制中的问题引起的。

如果您直接调用函数，则会在 Lambda 的响应中看到所有调用错误。如果您使用事件源映射或通过其他服务异步调用函数，则可能会在日志、死信队列或失败事件目标中找到错误。错误处理选项和重试行为因调用函数的方式和错误类型而异。

有关 Invoke 操作可以返回的错误类型的列表，请参阅 Invoke（调用）。

Lambda：函数在初始化阶段超时 (Sandbox.Timedout)

错误：任务在 3.00 秒后超时

如果 Init 阶段超时，Lambda 会在下一个调用请求到达时重新运行 Init 阶段，以便再次初始化执行环境。此情况称为隐藏初始化。不过，如果函数配置了较短的超时时间（通常在 3 秒左右），被抑制的初始化可能无法在分配的超时时间内完成，导致 Init 阶段再次超时。另一种情况是，被抑制的初始化已完成，但没有留出足够时间让 Invoke 阶段完成，导致 Invoke 阶段超时。

要减少超时错误，请使用下列一项或多项策略：

- 延长函数超时时间：延长超时时间，让 Init 和Invoke 阶段有时间成功完成。
- 增加函数内存分配：内存越多意味着 CPU 分配也会成比例增加，而这可以加快 Init 和 Invoke 阶段的完成速度。
- 优化函数初始化代码：缩短初始化所需的时间，确保 Init 和 Invoke 阶段可以在配置的超时时间内完成。

- 添加错误处理：在函数代码中添加合适的处理错误，可以防止在 Lambda 执行环境失败后触发重复的初始化尝试。

IAM：lambda:InvokeFunction 未授权

错误：User: arn:aws:iam::123456789012:user/developer is not authorized to perform: lambda:InvokeFunction on resource: my-function（用户: arn:aws:iam::123456789012:user/developer 未获得授权，无法对资源执行 lambda:InvokeFunction: my-function）

您的用户或您代入的角色必须具有调用函数的权限。此要求还适用于 Lambda 函数以及其他调用函数的计算资源。将 AWS 托管式策略 AWSLambdaRole 添加到用户或添加允许对目标函数执行 `lambda:InvokeFunction` 操作的自定义策略。

> **Note**
>
> IAM 操作的名称 (`lambda:InvokeFunction`) 指的是 `Invoke` Lambda API 操作。

有关更多信息，请参阅在 AWS Lambda 中管理权限。

Lambda：无法找到有效的引导程序 (Runtime.InvalidEntrypoint)

错误：无法找到有效的引导程序：[/var/task/bootstrap /opt/bootstrap]

当部署包的根目录不包含名为 `bootstrap` 的可执行文件时，通常会发生此错误。例如，如果使用 .zip 文件部署函数 `provided.al2023`，则 `bootstrap` 文件必须位于 .zip 文件的根目录下，而不是目录中。

Lambda：无法执行操作 ResourceConflictException

错误：ResourceConflictException: The operation cannot be performed at this time. （ResourceConflictException：此时无法执行该操作。）该函数目前处于以下状态：待定

如果在创建时将函数连接到 Virtual Private Cloud (VPC)，则该函数会在 Lambda 创建弹性网络接口时进入 `Pending` 状态。在此期间，您无法调用或修改函数。如果您在创建后将函数连接到 VPC，则可以在更新处于待处理状态时调用该函数，但无法修改其代码或配置。

有关更多信息，请参阅Lambda 函数状态。

Lambda：函数卡在待处理状态

错误：A function is stuck in the *Pending* state for several minutes.（一个函数卡在待处理状态达几分钟。）

如果某个函数卡在 Pending 状态超过六分钟，请调用以下 API 操作之一来取消阻止它。

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda 取消处于暂停状态的操作并将函数置于 Failed 状态。然后，您可以尝试另一次更新。

Lambda：某个函数正在使用所有并发

问题：一个函数使用了所有可用的并发，导致其他函数被限制。

要将 AWS 区域中 AWS 账户的可用并发划分为池，请使用预留并发。预留并发可确保函数始终可以扩展到向其分配的并发，并且不会扩展到向其分配的并发之外。

常规：无法使用其他账户或服务调用函数

问题：您可以直接调用您的函数，但是当另一个服务或账户调用它时，它不会运行。

您向其他服务和账户授予权限以在函数的基于资源的策略中调用函数。如果调用方是另一个账户，则该用户还必须具有函数调用权限。

常规：函数调用正在循环

问题：函数在循环中被连续调用。

当您的函数在触发它的同一 AWS 服务中管理资源时，通常会发生这种情况。例如，可以创建一个函数来将对象存储在 Amazon Simple Storage Service (Amazon S3) 存储桶中，该存储桶配置了一个再次调用函数的通知。要将函数停止运行，请将可用的并发减少到零，以限制未来发生的所有调用。然后，确定导致递归调用的代码路径或配置错误。Lambda 会自动检测并停止某些 AWS 服务和 SDK 的递归循环。有关更多信息，请参阅 the section called “递归循环检测”。

Lambda：预置并发的别名路由

问题：在别名路由期间预配置的并发溢出调用。

Lambda 使用简单的概率模型在两个函数版本之间分发流量。低流量级别情况下，您可能会看到每个版本上配置的流量百分比与实际流量百分比之间的差异很大。如果您的函数使用预置并发，则可以避免溢出调用，方法是在别名路由处于活动状态期间配置更多的预置并发实例。

Lambda：预置并发导致的冷启动

问题：启用预置并发后，会发生冷启动。

函数上的并发执行次数少于或等于配置的预置并发级别时，应该不会发生冷启动。要帮助您确认预置并发是否正常运行，请执行以下操作：

- 检查是否在函数版本或别名上启用了预置并发。

> **Note**
>
> 未发布版本的函数（$LATEST）无法配置预置并发。

- 确保触发器调用了正确的函数版本或别名。例如，如果您正在使用 Amazon API Gateway，检查确认是 API Gateway 在使用预置并发调用函数版本或别名，而非 $LATEST。要确认是否使用了预置并发，您可以检查 ProvisionedConcurrencyInvocations Amazon CloudWatch 指标。非零值表示函数正在初始化的执行环境中处理调用。
- 通过检查 ProvisionedConcurrencySpilloverInvocations Amazon CloudWatch 指标来确定您的函数并发是否超出了配置的预置并发级别。非零值表示所有预置并发都处于使用状态，且在冷启动时发生了部分调用。
- 检查调用频率（每秒请求数）。具有预置并发的函数每个预置并发的最大速率为每秒 10 个请求。例如，配置了 100 个预置并发的函数可以每秒处理 1000 个请求。如果调用速率超过每秒 1000 个请求，则可能会发生部分冷启动。

Lambda：新版本的冷启动问题

问题：在部署新版本的函数后，会发生冷启动。

更新函数别名后，Lambda 会根据别名上配置的权重自动将预置并发转移到新版本。

错误：KMSDisabledException: Lambda was unable to decrypt the environment variables because the KMS key used is disabled.（KMSDisabledException：Lambda 无法解密环境变量，因为使用的 KMS 密钥已被禁用。）请检查功能的 KMS 密钥设置。

如果您的 AWS Key Management Service (AWS KMS) 密钥被禁用或者撤销了允许 Lambda 使用该密钥的授权，则可能会发生此错误。如果缺少授权，请将函数配置为使用其他密钥。然后，重新分配自定义密钥以重新创建授权。

EFS：函数无法挂载 EFS 文件系统

错误：EFSMountFailureException：函数无法使用访问点 arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd 挂载 EFS 文件系统。

对函数文件系统 的挂载请求被拒绝。检查函数的权限，并确认其文件系统和访问点存在且可供使用。

EFS：函数无法连接到 EFS 文件系统

错误：EFSMountConnectivityException: The function couldn't connect to the Amazon EFS file system with access point arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd.（EFSMountConnectivityException：函数无法使用访问点 arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd 连接到 Amazon EFS 文件系统。）检查您的网络配置，然后重试。

函数无法使用 NFS 协议（TCP 端口 2049）与函数的文件系统建立连接。检查 VPC 子网的安全组和路由配置。

如果您在更新函数的 VPC 配置设置后出现这些错误，请尝试卸载并重新挂载文件系统。

EFS：由于超时，函数无法挂载 EFS 文件系统

错误：EFSMountTimeoutException：由于挂载超时，函数无法使用访问点 {arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd} 挂载 EFS 文件系统。

函数能够连接到函数的文件系统，但挂载操作超时。短时间后再试，并考虑限制函数的并发以减少文件系统的负载。

Lambda：Lambda 检测到某一 IO 进程耗时过长

EFSIOException：此函数实例已停止，因为 Lambda 检测到某一 IO 进程花费的时间过长。

先前的调用超时，且 Lambda 无法终止函数处理程序。当附加的文件系统用完了突发信用并且基准吞吐量不足时，可能会出现此问题。要提高吞吐量，可以增加文件系统的大小或使用预置吞吐量。

Lambda 中的执行问题疑难解答

当 Lambda 运行时运行函数代码时，可能会在已经处理事件一段时间的函数实例上处理事件，或者可能需要初始化一个新实例。在函数初始化期间、处理程序代码处理事件时或者当函数返回（或无法返回）响应，可能会发生错误。

函数执行错误可能是由您的代码、函数配置、下游资源或权限中的问题引起。如果您直接调用函数，则会在 Lambda 的响应中看到函数错误。如果您使用事件源映射或通过其他服务异步调用函数，则可能会在日志、死信队列或失败时的目标中找到错误。错误处理选项和重试行为因调用函数的方式和错误类型而异。

当您的函数代码或 Lambda 运行时返回错误时，来自 Lambda 的响应中的状态代码为“200 OK”。响应中是否存在错误由名为 `X-Amz-Function-Error` 的标头指示。400 和 500 系列状态代码保留用于调用错误。

Lambda：执行时间过长

问题：函数执行时间太长。

如果您的代码在 Lambda 中运行所花费的时间长于在本地计算机上运行所花费的时间，则可能受到对该函数可用的内存或处理能力的限制。使用额外内存配置函数以增加内存和 CPU。

Lambda：未显示日志或跟踪

问题：日志未显示在 CloudWatch Logs 中。

问题：跟踪未显示在 AWS X-Ray 中。

您的函数需要权限才能调用 CloudWatch Logs 和 X-Ray。更新函数的执行角色以向其授予权限。添加以下托管策略以启用日志和跟踪。

- AWSLambdaBasicExecutionRole
- AWSXRayDaemonWriteAccess

向函数添加权限时，请同时简单更新其代码或配置。这会强制停止并替换函数的具有过时凭证的运行实例。

> **Note**
>
> 函数调用后，日志可能需要 5 到 10 分钟才能显示。

Lambda：并非所有我的函数的日志都会出现

问题：尽管我拥有正确权限，但 CloudWatch Logs 中缺少函数日志

如果您的 AWS 账户 达到了其 CloudWatch Logs 限额限制，则 CloudWatch 就会对函数日志记录进行节流。发生这种情况时，您函数输出的某些日志可能不会出现在 CloudWatch Logs 中。

如果函数输出日志的速度过快，Lambda 无法对其进行处理，也会导致日志输出无法显示在 CloudWatch Logs 中。当 Lambda 无法按照函数生成日志的速度向 CloudWatch 发送日志时，它会丢弃日志以防止函数的执行速度减慢。单个日志流的日志吞吐量超过 2 MB/s 时，预计会持续观察到删除的日志。

如果函数配置为使用 JSON 格式的日志，则 Lambda 会在丢弃日志时尝试向 CloudWatch Logs 发送 `logsDropped` 事件。但是，当 CloudWatch 对函数的日志记录进行节流时，此事件可能无法到达 CloudWatch Logs，因此 Lambda 丢弃日志时您并不始终会看到记录。

要检查 AWS 账户 是否已达到其 CloudWatch Logs 限额限制，请执行以下操作：

1. 打开服务限额控制台。
2. 在导航窗格中，选择 AWS 服务。
3. 从 AWS 服务列表中，搜索 Amazon CloudWatch Logs。
4. 在服务限额列表中，选择 `CreateLogGroup throttle limit in transactions per second`、`CreateLogStream throttle limit in transactions per second` 和 `PutLogEvents throttle limit in transactions per second` 限额以查看利用率。

您还可以设置 CloudWatch 警报，以便在账户利用率超过您为这些限额指定的限制时提醒您。请参阅根据静态阈值创建 CloudWatch 告警以了解更多信息。

如果 CloudWatch Logs 的默认限额限制不足以满足您的用例，则可以请求增加限额。

Lambda：函数在执行完成之前返回

问题：(Node.js) 函数在代码完成执行之前返回

许多库（包括 AWS 开发工具包）都是异步操作。当您进行网络调用或执行其他需要等待响应的操作时，库返回一个名为 promise 的对象，该对象在后台跟踪操作的进度。

要等待 promise 解析为响应，请使用 `await` 关键字。这会阻止您的处理程序代码执行，直到将 promise 解析为包含响应的对象。如果您不需要在代码中使用来自响应的数据，则可以直接将 promise 返回到运行时。

一些库不返回 promise，但可以包装到返回 promise 的代码中。有关更多信息，请参阅 定义采用 Node.js 的 Lambda 函数处理程序。

AWS 开发工具包：版本和更新

问题：运行时包含的 AWS 开发工具包不是最新版本

问题：运行时中包含的 AWS 开发工具包自动更新

脚本语言的运行时包括 AWS 开发工具包，并定期更新到最新版本。每个运行时的当前版本在运行时页面上列出。要使用较新版本的AWS开发工具包，或将函数锁定为特定版本，您可以将库与函数代码捆绑起来，或创建 Lambda 层。有关创建具有依赖项的部署程序包的详细信息，请参阅以下主题：

Node.js

- 使用 .zip 文件归档部署 Node.js Lambda 函数

Python

- 将 .zip 文件归档用于 Python Lambda 函数

Ruby

- 使用 .zip 文件归档部署 Ruby Lambda 函数

Java

- 使用 .zip 或 JAR 文件归档部署 Java Lambda 函数

Go

- 使用 .zip 文件归档部署 Go Lambda 函数

C#

- 使用 .zip 文件归档构建和部署 C# Lambda 函数

PowerShell

- 使用 .zip 文件归档部署 PowerShell Lambda 函数

Python：库未正确加载

问题：(Python) 无法从部署程序包中正确加载某些库

具有使用 C 或 C ++ 编写的扩展模块的库，必须在与 Lambda (Amazon Linux) 具有相同处理器架构的环境中进行编译。有关更多信息，请参阅 将 .zip 文件归档用于 Python Lambda 函数。

Java：从 Java 11 更新到 Java 17 后，函数处理事件所需的时间更长

问题：（Java）从 Java 11 更新到 Java 17 后，函数处理事件所需的时间更长

使用 JAVA_TOOL_OPTIONS 参数调整编译器。Java 17 及更高 Java 版本的 Lambda 运行时会更改默认的编译器选项。虽然此更改缩短了寿命较短的函数的冷启动时间，但以前的行为更适合计算密集型、运行时间较长的函数。将 JAVA_TOOL_OPTIONS 设置为 -XX:-TieredCompilation 可恢复到 Java 11 的行为。有关 JAVA_TOOL_OPTIONS 参数的更多信息，请参阅 the section called “了解 JAVA_TOOL_OPTIONS 环境变量”。

Lambda 中的联网问题疑难解答

默认情况下，Lambda 在与AWS服务和互联网连接的内部 Virtual Private Cloud (VPC) 中运行您的函数。要访问本地网络资源，您可以将函数配置为连接到您账户中的 VPC。使用此功能时，您可以管理函数的互联网访问以及与 Amazon Virtual Private Cloud（Amazon VPC）资源的网络连接。

网络连接错误可能是由于 VPC 的路由配置、安全组规则、AWS Identity and Access Management（IAM）角色权限、网络地址转换（NAT）或资源可用性（例如 IP 地址或网络接口）中的问题造成。根据具体问题的不同，如果请求无法到达目标，则可能会看到特定错误或超时。

VPC：函数无法访问互联网或超时

问题：连接到 VPC 后，您的 Lambda 函数失去互联网访问权限。

错误：错误：连接 ETIMEDOUT 176.32.98.189:443

错误：错误：任务在 10.00 秒后超时

错误：ReadTimeoutError: Read timed out. (read timeout=15)

当您将函数连接到某个 VPC 时，所有出站请求都会通过该 VPC。要连接到互联网，请将 VPC 配置为将出站流量从函数的子网发送到公有子网中的 NAT 网关。有关更多信息和示例 VPC 配置，请参阅 the section called “VPC 函数的互联网访问权限”。

如果您的某些 TCP 连接超时，这可能是数据包碎片造成的。Lambda 函数无法处理传入的碎片 TCP 请求，因为 Lambda 不支持 TCP 或 ICMP 的 IP 碎片化。

VPC：函数需要在不使用互联网的情况下访问AWS服务

问题：您的 Lambda 函数需要在不使用互联网的情况下访问AWS服务

要在不能访问互联网的情况下从私有子网将函数连接到AWS服务，请使用 VPC 终端节点。

VPC：已达到弹性网络接口限制

错误：ENILimitReachedException：函数的 VPC 已达到弹性网络接口限制。

当您将一个 Lambda 函数连接到某个 VPC 时，Lambda 将为该函数所附加的每个子网和安全组组合创建一个弹性网络接口。原定设置服务配额为每个 VPC 250 个网络接口。要请求增加配额，您可以使用 Service Quotas 控制台。

EC2：类型为“lambda”的弹性网络接口

错误代码：Client.OperationNotPermitted

错误消息：无法修改此类接口的安全组

如果您尝试修改由 Lambda 管理的弹性网络接口（ENI），则会收到此错误。`ModifyNetworkInterfaceAttribute` 不包含在用于更新由 Lambda 创建的弹性网络接口上的操作的 Lambda API 中。

DNS：UNKNOWNHOSTEXCEPTION，无法连接到主机

错误消息：UNKNOWNHOSTEXCEPTION

Lambda 函数最多支持 20 个并发 TCP 连接进行 DNS 解析。函数可能正要耗尽此限制。大多数常见的 DNS 请求都是通过 UDP 完成的。如果函数只建立 UDP DNS 连接，那么此问题与您无关。该错误通常是由配置错误或基础设施降级引发的，因此在深入检查 DNS 流量之前，请确认 DNS 基础设施是否配置正确且运行正常，并且 Lambda 函数是否引用 DNS 中指定的主机。

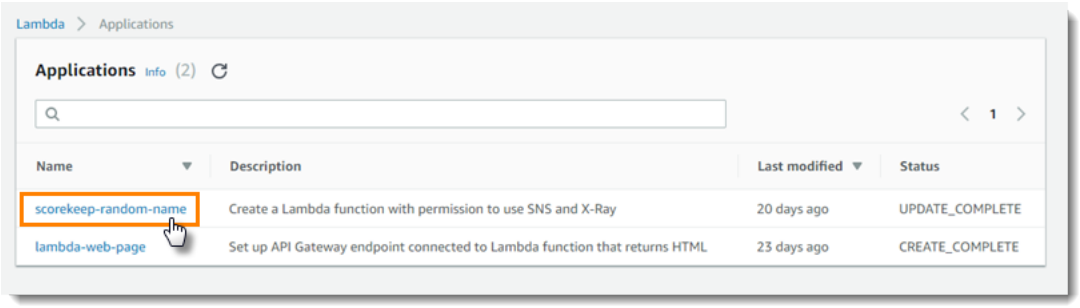
请注意，如果诊断出问题与 TCP 连接最大值有关，您不能请求提高此限制。如果 Lambda 函数因 DNS 有效负载较大而回退到 TCP DNS，请确认解决方案是否正在使用支持 EDNS 的库。有关 EDNS 的更多信息，请参阅 RFC 6891 standard。如果 DNS 有效负载持续超过 EDNS 最大限制，则解决方案仍可能会耗尽 TCP DNS 限制。

在 AWS Lambda 控制台中查看应用程序

AWS Lambda 控制台有助于您监控和管理 Lambda 应用程序。应用程序部分列出了包含 Lambda 函数的 AWS CloudFormation 堆栈。该菜单包含使用 AWS CloudFormation 控制台、AWS CloudFormation、AWS Serverless Application Repository 或 AWS CLI CLI 在 AWS SAM 中启动的堆栈。

查看 Lambda 应用程序

1. 打开 Lambda 控制台的 Applications（应用程序）页面。
2. 选择应用程序。



概述显示有关应用程序的以下信息。

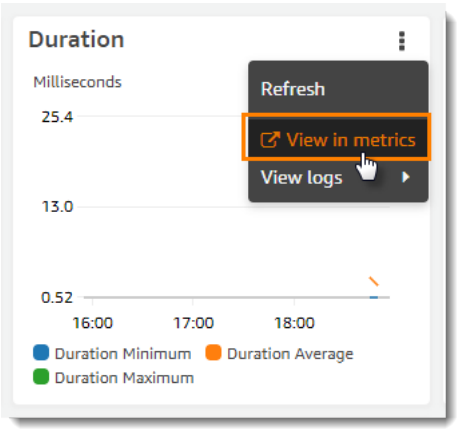
- AWS CloudFormation 模板 或 SAM 模板 – 定义应用程序的模板。
- Resources – 应用程序模板中定义的 AWS 资源。要管理应用程序的 Lambda 函数，请从列表中选择一个函数名称。

监控 Lambda 应用程序

Lambda 控制台的应用程序部分包含一个监控选项卡，您可以在其中查看 Amazon CloudWatch 控制面板，其中包含应用程序中资源的聚合指标。

监控 Lambda 应用程序

1. 打开 Lambda 控制台的 Applications（应用程序）页面。
2. 选择 Monitoring（监控）。
3. 要查看任意图表中有关指标的详细信息，请从下拉菜单中选择查看指标。



该图表显示在新标签中，图表下方列出相关的指标。您可以自定义此图表的视图，同时更改显示的指标和资源、统计数据、时间段以及其他因素，以更好地了解当前情况。

默认情况下，Lambda 控制台会显示基本控制面板。您可以使用 AWS::CloudWatch::Dashboard 资源类型，将一个或多个 Amazon CloudWatch 控制面板添加到您的应用程序模板，来自定义此页面。当您的模板包含一个或多个控制面板时，此页面会显示您的控制面板，而不是默认控制面板。您可以使用页面右上角的下拉菜单切换控制面板。以下示例创建了包含单个小部件的控制面板，该小部件显示了名为 `my-function` 的函数的调用次数。

Example 函数控制面板模板

```
Resources:
  MyDashboard:
    Type: AWS::CloudWatch::Dashboard
    Properties:
      DashboardName: my-dashboard
      DashboardBody: |
        {
```

```
            "widgets": [
                {
                    "type": "metric",
                    "width": 12,
                    "height": 6,
                    "properties": {
                        "metrics": [
                            [
                                "AWS/Lambda",
                                "Invocations",
                                "FunctionName",
                                "my-function",
                                {
                                    "stat": "Sum",
                                    "label": "MyFunction"
                                }
                            ],
                            [
                                {
                                    "expression": "SUM(METRICS())",
                                    "label": "Total Invocations"
                                }
                            ]
                        ],
                        "region": "us-east-1",
                        "title": "Invocations",
                        "view": "timeSeries",
                        "stacked": false
                    }
                }
            ]
        }
```

有关编辑 CloudWatch 控制面板和小组件的详细信息，请参阅 Amazon CloudWatch API 参考中的仪表板正文结构和语法。

创建 Lambda 函数的滚动部署

使用滚动部署来控制推出 Lambda 函数新版本相关风险。在滚动部署中，系统会自动部署函数的新版本，并逐步向新版本发送数量不断增加的流量。您可以配置的参数包括流量和增长率。

您可使用 AWS CodeDeploy 和 AWS SAM 配置滚动部署。CodeDeploy 是一项可自动将应用程序部署到 Amazon EC2 和 AWS Lambda 的服务。有关更多信息，请参阅什么是 CodeDeploy？。通过使用 CodeDeploy 部署您的 Lambda 函数，您可以轻松地监控部署状态，并在检测到任何问题时启动回滚。

AWS SAM 是一个开源框架，用于构建无服务器应用程序。您可以创建 AWS SAM 模板（以 YAML 格式），以便指定滚动部署所需的组件的配置。AWS SAM 使用模板来创建和配置组件。有关更多信息，请参阅什么是 AWS SAM？。

在滚动部署中，AWS SAM 会执行以下任务：

- 它会配置您的 Lambda 函数并创建别名。

  别名路由配置是实施滚动部署的基础功能。
- 它会创建一个 CodeDeploy 应用程序和一个部署组。

  部署组可管理滚动部署和回滚（如果需要）。
- 它会检测您何时创建 Lambda 函数的新版本。
- 它会触发 CodeDeploy 启动部署新版本。

示例 AWS SAM Lambda 模板

下面的示例演示了用于简单滚动部署的 AWS SAM 模板。

```
AWSTemplateFormatVersion : '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A sample SAM template for deploying Lambda functions.

Resources:
# Details about the myDateTimeFunction Lambda function
  myDateTimeFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: myDateTimeFunction.handler
```

```
      Runtime: nodejs18.x
# Creates an alias named "live" for the function, and automatically publishes when you
 update the function.
      AutoPublishAlias: live
      DeploymentPreference:
# Specifies the deployment configuration
          Type: Linear10PercentEvery2Minutes
```

此模板定义一个名为 myDateTimeFunction 的 Lambda 函数，其中包含以下属性。

AutoPublishAlias

AutoPublishAlias 属性创建一个名为 live 的别名。此外，AWS SAM 框架会自动检测您何时为函数保存新代码。然后，框架发布新的函数版本并更新 live 别名以便指向新版本。

DeploymentPreference

DeploymentPreference 属性决定 CodeDeploy 应用程序将流量从 Lambda 函数的原始版本转移到新版本的速率。Linear10PercentEvery2Minutes 值每两分钟将额外 10% 的流量转移到新版本。

有关预定义部署配置的列表，请参阅部署配置。

有关如何将 CodeDeploy 与 Lambda 函数结合使用的详细教程，请参阅使用 CodeDeploy 部署更新的 Lambda 函数。

将 Lambda 与 Kubernetes 结合使用

您可以使用 AWS Controllers for Kubernetes（ACK） 或 Crossplane 通过 Kubernetes API 部署和管理 Lambda 函数。

AWS Controllers for Kubernetes（ACK）

您可以使用 ACK 部署和管理来自 Kubernetes API 的 AWS 资源。通过 ACK，AWS 为 AWS 服务 [例如 Lambda、Amazon Elastic Container Registry（Amazon ECR）、Amazon Simple Storage Service（Amazon S3）和 Amazon SageMaker] 提供开源自定义控制器。每个支持的 AWS 服务有自己的自定义控制器。在您的 Kubernetes 集群中，为每个您要使用的 AWS 服务安装控制器。然后，创建自定义资源定义（CRD）来定义 AWS 资源。

我们建议您使用 Helm 3.8 或更高版本安装 ACK 控制器。每个 ACK 控制器都有自己的 Helm 图表，用于安装控制器、CRD 和 Kubernetes RBAC 规则。有关更多信息，请参阅 ACK 文档中的 Install an ACK Controller。

创建 ACK 自定义资源后，您可以像使用任何其他内置 Kubernetes 对象一样使用此资源。例如，您可以使用首选的 Kubernetes 工具链（包括 kubectl）部署和管理 Lambda 函数。

以下是通过 ACK 预置 Lambda 函数的示例用例：

- 您的组织使用基于角色的访问控制（RBAC）和服务账户的 IAM 角色来创建权限边界。借助 ACK，您可以为 Lambda 重用此安全模型，而无需创建新用户和策略。
- 您的组织有一个 DevOps 流程，可以使用 Kubernetes 清单将资源部署到 Amazon Elastic Kubernetes Service（Amazon EKS）集群中。借助 ACK，您可以使用清单来配置 Lambda 函数，而无需创建单独的基础架构作为代码模板。

有关使用 ACK 的更多信息，请参阅 ACK 文档中的 Lambda 教程。

Crossplane

Crossplane 是一个开源云原生计算基金会（CNCF）项目，其使用 Kubernetes 来管理云基础设施资源。借助 Crossplane，开发人员可以请求基础设施，而无需了解其复杂性。平台团队保留对基础设施的预置和管理方式的控制。

借助 Crossplane，您可以使用首选的 Kubernetes 工具链（例如 kubectl）部署和管理 Lambda 函数，以及任何可以将清单部署到 Kubernetes 的 CI/CD 管道。以下是通过 Crossplane 预置 Lambda 函数的示例用例：

- 您的组织想要通过确保 Lambda 函数具有正确的标签来强制实施合规性。平台团队可以使用 Crossplane Compositions 通过 API 抽象来定义此策略。然后，开发人员可以使用这些抽象来部署带标签的 Lambda 函数。
- 您的项目使用将 GitOps 和 Kubernetes 结合使用。在此模型中，Kubernetes 不断将 git 存储库（所需状态）与集群内运行的资源（当前状态）进行协调。如果存在差异，GitOps 流程会自动对集群进行更改。您可以将 GitOps 与 Kubernetes 结合使用，以便借助 CRD 和控制器等熟悉的 Kubernetes 工具和概念，通过 Crossplane 部署和管理 Lambda 函数。

要了解将 Crossplane 与 Lambda 结合使用的更多信息，请参阅以下内容：

- AWS Blueprints for Crossplane：此存储库包含如何使用 Crossplane 部署 AWS 资源（包括 Lambda 函数）的示例。

  > **Note**
  >
  > AWS Blueprints for Crossplane 正在积极开发中，不应用于生产。

- 使用 Amazon EKS 和 Crossplane 部署 Lambda：此视频演示了使用 Crossplane 部署 AWS 无服务器架构的高级示例，从开发人员和平台两方的角度探索设计。

Lambda 示例应用程序

本指南的 GitHub 存储库包括演示如何使用各种语言和AWS服务的示例应用程序。每个示例应用程序都包含用于轻松部署和清理的脚本以及支持资源。

Node.js

Node.js 中的示例 Lambda 应用程序

- blank-nodejs – 此 Node.js 函数用于显示日志记录、环境变量、AWS X-Ray 跟踪、层、单元测试以及AWS开发工具包的使用情况。
- nodejs-apig – 一个带有公有 API 端点的函数，此函数可处理来自 API Gateway 的事件并返回 HTTP 响应。
- efs-nodejs – 此函数可在 Amazon VPC 中使用 Amazon EFS 文件系统。此示例包括配置为与 Lambda 一起使用的 VPC、文件系统、挂载目标和访问点。

Python

Python 中的 Lambda 应用程序示例

- blank-python – 一个 Python 函数，用于显示日志记录、环境变量、AWS X-Ray 跟踪、层、单元测试和AWS开发工具包的使用情况。

Ruby

Ruby 中的示例 Lambda 应用程序

- blank-ruby – 一个 Ruby 函数，显示日志记录、环境变量、AWS X-Ray 跟踪、层、单元测试和AWS开发工具包的使用情况。
- 适用于 AWS Lambda 的 Ruby 代码示例 – 在 Ruby 中编写的代码示例，演示了如何与 AWS Lambda 互动。

Java

Java 中的 Lambda 应用程序示例

- java17-examples：这是一种 Java 函数，演示如何使用 Java 记录来表示输入事件数据对象。

- java-basic – 具有单元测试和变量日志记录配置的最小 Java 函数的集合。
- java-events – Java 函数的集合，其中包含用于处理来自 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis 等各种服务的事件的框架代码。这些函数使用最新版本的 aws-lambda-events 库（3.0.0 及更新版本）。这些示例不需要 AWS 开发工具包作为依赖项。
- s3-java – 此 Java 函数可处理来自 Amazon S3 的通知事件，并使用 Java 类库（JCL）从上传的图像文件创建缩略图。
- 自定义序列化 –如何使用 fastJson、Gson、Moshi 和 jackson-jr 等常用库实现自定义序列化的示例。
- 使用 API Gateway 调用 Lambda 函数 – Java 函数，用于扫描包含员工信息的 Amazon DynamoDB 表。然后，该函数使用 Amazon Simple Notification Service 向员工发送短信，祝贺他们工作周年纪念日快乐。此示例使用 API Gateway 调用函数。

在 Lambda 上运行常见 Java 框架

- spring-cloud-function-samples：此示例来自 Spring，展示了如何使用 Spring Cloud Function 框架创建 AWS Lambda 函数。
- 无服务器 Spring Boot 应用程序演示：该示例展示了如何在带有 SnapStart 和不带有 SnapStart 的托管式 Java 运行时系统中设置典型的 Spring Boot 应用程序，或者如何使用自定义运行时系统设置为 GraalVM 本机映像。
- 无服务器 Micronaut 应用程序演示：该示例展示了如何在带有 SnapStart 和不带有 SnapStart 的托管式 Java 运行时系统中使用 Micronaut，或者如何使用自定义运行时系统设置为 GraalVM 本机映像。在《Micronaut/Lambda 指南》中了解更多信息。
- 无服务器 Quarkus 应用程序演示：该示例展示了如何在带有 SnapStart 和不带有 SnapStart 的托管式 Java 运行时系统中使用 Quarkus，或者如何使用自定义运行时系统设置为 GraalVM 本机映像。在《Quarkus/Lambda 指南》和《Quarkus/SnapStart 指南》中了解更多信息。

Go

Lambda 为 Go 运行时提供了以下示例应用程序：

Go 中的 Lambda 应用程序示例

- go-al2：返回公有 IP 地址的 hello world 函数。此应用程序使用 `provided.al2` 自定义运行时系统。
- blank-go – 此 Go 函数显示 Lambda 的 Go 库、日志记录、环境变量和 AWS SDK 的使用情况。此应用程序使用 `go1.x` 运行时系统。

C#

C# 中的 Lambda 应用程序示例

- blank-csharp – 此 C# 函数可显示 Lambda 的 .NET 库、日志记录、环境变量、AWS X-Ray 跟踪、单元测试和AWS开发工具包的使用情况。
- blank-csharp-with-layer：一个 C# 函数，通过使用 .NET CLI 创建打包函数依赖项的层。
- ec2-spot – 此函数可在 Amazon EC2 中管理竞价型实例请求。

PowerShell

Lambda 为 PowerShell 提供了以下示例应用程序：

- blank-powershell – 此 PowerShell 函数可显示日志记录、环境变量和AWS开发工具包的使用情况。

要部署示例应用程序，请按照 README 文件中的说明操作。

将 Lambda 与 AWS SDK 配合使用

AWS 软件开发工具包 (SDK) 适用于许多常用编程语言。每个软件开发工具包都提供 API、代码示例和文档，使开发人员能够更轻松地以其首选语言构建应用程序。

SDK 文档	代码示例
AWS SDK for C++	AWS SDK for C++ 代码示例
AWS CLI	AWS CLI 代码示例
AWS SDK for Go	AWS SDK for Go 代码示例
AWS SDK for Java	AWS SDK for Java 代码示例
AWS SDK for JavaScript	AWS SDK for JavaScript 代码示例
AWS SDK for Kotlin	AWS SDK for Kotlin 代码示例
AWS SDK for .NET	AWS SDK for .NET 代码示例
AWS SDK for PHP	AWS SDK for PHP 代码示例
AWS Tools for PowerShell	Tools for PowerShell 代码示例
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) 代码示例
AWS SDK for Ruby	AWS SDK for Ruby 代码示例
AWS SDK for Rust	AWS SDK for Rust 代码示例
适用于 SAP ABAP 的 AWS SDK	适用于 SAP ABAP 的 AWS SDK 代码示例
AWS SDK for Swift	AWS SDK for Swift 代码示例

有关特定于 Lambda 的示例，请参阅 适用于使用 AWS SDK 的 Lambda 的代码示例。

示例可用性

找不到所需的内容？ 通过使用此页面底部的提供反馈链接请求代码示例。

适用于使用 AWS SDK 的 Lambda 的代码示例

以下代码示例显示如何将 Lambda 与 AWS 软件开发工具包（SDK）一起使用。

基础知识是向您展示如何在服务中执行基本操作的代码示例。

操作是大型程序的代码摘录，必须在上下文中运行。您可以通过操作了解如何调用单个服务函数，还可以通过函数相关场景的上下文查看操作。

场景是向您展示如何通过在一个服务中调用多个函数或与其他 AWS 服务 结合来完成特定任务的代码示例。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

开始使用

开始使用 Lambda

以下代码示例显示如何开始使用 Lambda。

.NET

AWS SDK for .NET

> **Note**
>
> 在 GitHub 上查看更多内容。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
namespace LambdaActions;

using Amazon.Lambda;

public class HelloLambda
{
    static async Task Main(string[] args)
    {
        var lambdaClient = new AmazonLambdaClient();
```

```
        Console.WriteLine("Hello AWS Lambda");
        Console.WriteLine("Let's get started with AWS Lambda by listing your
 existing Lambda functions:");

        var response = await lambdaClient.ListFunctionsAsync();
        response.Functions.ForEach(function =>
        {

 Console.WriteLine($"{function.FunctionName}\t{function.Description}");
        });
    }
}
```

- 有关 API 的详细信息，请参阅 AWS SDK for .NET API 参考中的 ListFunctions。

C++

SDK for C++

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

CMakeLists.txt CMake 文件的代码。

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS lambda)

# Set this project's name.
project("hello_lambda")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)
```

```
# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.

     # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
 may need to uncomment this
                                     # and set the proper subdirectory to the
 executables' location.

     AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_lambda.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

hello_lambda.cpp 源文件的代码。

```
#include <aws/core/Aws.h>
#include <aws/lambda/LambdaClient.h>
#include <aws/lambda/model/ListFunctionsRequest.h>
#include <iostream>

/*
 *  A "Hello Lambda" starter application which initializes an AWS Lambda (Lambda)
 client and lists the Lambda functions.
```

```
 *
 *  main function
 *
 *  Usage: 'hello_lambda'
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::Lambda::LambdaClient lambdaClient(clientConfig);
        std::vector<Aws::String> functions;
        Aws::String marker; // Used for pagination.

        do {
            Aws::Lambda::Model::ListFunctionsRequest request;
            if (!marker.empty()) {
                request.SetMarker(marker);
            }

            Aws::Lambda::Model::ListFunctionsOutcome outcome =
 lambdaClient.ListFunctions(
                    request);

            if (outcome.IsSuccess()) {
                const Aws::Lambda::Model::ListFunctionsResult
 &listFunctionsResult = outcome.GetResult();
                std::cout << listFunctionsResult.GetFunctions().size()
                          << " lambda functions were retrieved." << std::endl;

                for (const Aws::Lambda::Model::FunctionConfiguration
 &functionConfiguration: listFunctionsResult.GetFunctions()) {
                    functions.push_back(functionConfiguration.GetFunctionName());
                    std::cout << functions.size() << "  "
                              << functionConfiguration.GetDescription() <<
 std::endl;
```

```
                    std::cout << "   "
                              <<
 Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                      functionConfiguration.GetRuntime()) << ": "
                              << functionConfiguration.GetHandler()
                              << std::endl;
                }
                marker = listFunctionsResult.GetNextMarker();
            } else {
                std::cerr << "Error with Lambda::ListFunctions. "
                          << outcome.GetError().GetMessage()
                          << std::endl;
                result = 1;
                break;
            }
        } while (!marker.empty());
    }


    Aws::ShutdownAPI(options); // Should only be called once.
    return result;
}
```

- 有关 API 的详细信息，请参阅 AWS SDK for C++ API 参考中的 ListFunctions。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
package main

import (
 "context"
 "fmt"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
)

// main uses the AWS SDK for Go (v2) to create an AWS Lambda client and list up
 to 10
// functions in your account.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  fmt.Println("Couldn't load default configuration. Have you set up your AWS
 account?")
  fmt.Println(err)
  return
 }
 lambdaClient := lambda.NewFromConfig(sdkConfig)

 maxItems := 10
 fmt.Printf("Let's list up to %v functions for your account.\n", maxItems)
 result, err := lambdaClient.ListFunctions(ctx, &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 if err != nil {
  fmt.Printf("Couldn't list functions for your account. Here's why: %v\n", err)
  return
 }
 if len(result.Functions) == 0 {
  fmt.Println("You don't have any functions!")
 } else {
  for _, function := range result.Functions {
   fmt.Printf("\t%v\n", *function.FunctionName)
  }
 }
}
```

- 有关 API 的详细信息，请参阅 AWS SDK for Go API 参考中的 ListFunctions。

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
Lambda service
     */
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- 有关 API 的详细信息，请参阅 AWS SDK for Java 2.x API 参考中的 ListFunctions。

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
  const paginator = paginateListFunctions({ client }, {});
  const functions = [];

  for await (const page of paginator) {
    const funcNames = page.Functions.map((f) => f.FunctionName);
    functions.push(...funcNames);
  }

  console.log("Functions:");
  console.log(functions.join("\n"));
  return functions;
};
```

- 有关 API 的详细信息，请参阅 AWS SDK for JavaScript API 参考中的 ListFunctions。

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
import boto3


def main():
    """
    List the Lambda functions in your AWS account.
    """
    # Create the Lambda client
    lambda_client = boto3.client("lambda")

    # Use the paginator to list the functions
    paginator = lambda_client.get_paginator("list_functions")
    response_iterator = paginator.paginate()

    print("Here are the Lambda functions in your account:")
    for page in response_iterator:
        for function in page["Functions"]:
            print(f"  {function['FunctionName']}")


if __name__ == "__main__":
    main()
```

- 有关 API 详细信息，请参阅《AWS SDK for Python (Boto3) API 参考》中的 ListFunctions。

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
require 'aws-sdk-lambda'
```

```
# Creates an AWS Lambda client using the default credentials and configuration
def lambda_client
  Aws::Lambda::Client.new
end

# Lists the Lambda functions in your AWS account, paginating the results if
 necessary
def list_lambda_functions
  lambda = lambda_client

  # Use a pagination iterator to list all functions
  functions = []
  lambda.list_functions.each_page do |page|
    functions.concat(page.functions)
  end

  # Print the name and ARN of each function
  functions.each do |function|
    puts "Function name: #{function.function_name}"
    puts "Function ARN: #{function.function_arn}"
    puts
  end

  puts "Total functions: #{functions.count}"
end

list_lambda_functions if __FILE__ == $PROGRAM_NAME
```

- 有关 API 的详细信息，请参阅 AWS SDK for Ruby API 参考中的 ListFunctions。

代码示例

- 使用 AWS SDK 的 Lambda 基本示例
  - 开始使用 Lambda
  - 了解将 Lambda 与 AWS SDK 结合使用的基础知识
  - 使用 AWS SDK 对 Lambda 执行的操作
    - 将 CreateAlias 与 CLI 配合使用
    - 将 CreateFunction 与 AWS SDK 或 CLI 配合使用
    - 将 DeleteAlias 与 CLI 配合使用

- 将 DeleteFunction 与 AWS SDK 或 CLI 配合使用
- 将 DeleteFunctionConcurrency 与 CLI 配合使用
- 将 DeleteProvisionedConcurrencyConfig 与 CLI 配合使用
- 将 GetAccountSettings 与 CLI 配合使用
- 将 GetAlias 与 CLI 配合使用
- 将 GetFunction 与 AWS SDK 或 CLI 配合使用
- 将 GetFunctionConcurrency 与 CLI 配合使用
- 将 GetFunctionConfiguration 与 CLI 配合使用
- 将 GetPolicy 与 CLI 配合使用
- 将 GetProvisionedConcurrencyConfig 与 CLI 配合使用
- 将 Invoke 与 AWS SDK 或 CLI 配合使用
- 将 ListFunctions 与 AWS SDK 或 CLI 配合使用
- 将 ListProvisionedConcurrencyConfigs 与 CLI 配合使用
- 将 ListTags 与 CLI 配合使用
- 将 ListVersionsByFunction 与 CLI 配合使用
- 将 PublishVersion 与 CLI 配合使用
- 将 PutFunctionConcurrency 与 CLI 配合使用
- 将 PutProvisionedConcurrencyConfig 与 CLI 配合使用
- 将 RemovePermission 与 CLI 配合使用
- 将 TagResource 与 CLI 配合使用
- 将 UntagResource 与 CLI 配合使用
- 将 UpdateAlias 与 CLI 配合使用
- 将 UpdateFunctionCode 与 AWS SDK 或 CLI 配合使用
- 将 UpdateFunctionConfiguration 与 AWS SDK 或 CLI 配合使用

- 适用于使用 AWS SDK 的 Lambda 的场景
  - 通过 AWS SDK 使用 Lambda 函数自动确认已知的 Amazon Cognito 用户
  - 通过 AWS SDK 使用 Lambda 函数自动迁移已知的 Amazon Cognito 用户
  - 创建 API Gateway REST API 以跟踪 COVID-19 数据
  - 创建借阅图书馆 REST API
  - 使用 Step Functions 创建 Messenger 应用程序

- 创建照片资产管理应用程序，让用户能够使用标签管理照片
- 使用 API Gateway 创建 Websocket 聊天应用程序
- 创建用于分析客户反馈和合成音频的应用程序
- 从浏览器调用 Lambda 函数
- 使用 S3 对象 Lambda 转换应用程序的数据
- 使用 API Gateway 调用 Lambda 函数
- 使用 Step Functions 调用 Lambda 函数
- 使用计划的事件调用 Lambda 函数
- 在完成 Amazon Cognito 用户身份验证后，通过 AWS SDK 使用 Lambda 函数写入自定义活动数据

- 使用 AWS SDK 的 Lambda 无服务器示例
  - 使用 Lambda 函数连接到 Amazon RDS 数据库
  - 通过 Kinesis 触发器调用 Lambda 函数
  - 通过 DynamoDB 触发器调用 Lambda 函数
  - 通过 Amazon DocumentDB 触发器调用 Lambda 函数
  - 通过 Amazon MSK 触发器调用 Lambda 函数
  - 通过 Amazon S3 触发器调用 Lambda 函数
  - 通过 Amazon SNS 触发器调用 Lambda 函数
  - 通过 Amazon SQS 触发器调用 Lambda 函数
  - 通过 Kinesis 触发器报告 Lambda 函数批处理项目失败
  - 通过 DynamoDB 触发器报告 Lambda 函数批处理项目失败
  - 报告使用 Amazon SQS 触发器进行 Lambda 函数批处理项目失败

使用 AWS SDK 的 Lambda 基本示例

以下代码示例展示了如何将 AWS Lambda 的基础知识与 AWS SDK 结合使用。

示例

- 开始使用 Lambda
- 了解将 Lambda 与 AWS SDK 结合使用的基础知识
- 使用 AWS SDK 对 Lambda 执行的操作

- 将 CreateAlias 与 CLI 配合使用
- 将 CreateFunction 与 AWS SDK 或 CLI 配合使用
- 将 DeleteAlias 与 CLI 配合使用
- 将 DeleteFunction 与 AWS SDK 或 CLI 配合使用
- 将 DeleteFunctionConcurrency 与 CLI 配合使用
- 将 DeleteProvisionedConcurrencyConfig 与 CLI 配合使用
- 将 GetAccountSettings 与 CLI 配合使用
- 将 GetAlias 与 CLI 配合使用
- 将 GetFunction 与 AWS SDK 或 CLI 配合使用
- 将 GetFunctionConcurrency 与 CLI 配合使用
- 将 GetFunctionConfiguration 与 CLI 配合使用
- 将 GetPolicy 与 CLI 配合使用
- 将 GetProvisionedConcurrencyConfig 与 CLI 配合使用
- 将 Invoke 与 AWS SDK 或 CLI 配合使用
- 将 ListFunctions 与 AWS SDK 或 CLI 配合使用
- 将 ListProvisionedConcurrencyConfigs 与 CLI 配合使用
- 将 ListTags 与 CLI 配合使用
- 将 ListVersionsByFunction 与 CLI 配合使用
- 将 PublishVersion 与 CLI 配合使用
- 将 PutFunctionConcurrency 与 CLI 配合使用
- 将 PutProvisionedConcurrencyConfig 与 CLI 配合使用
- 将 RemovePermission 与 CLI 配合使用
- 将 TagResource 与 CLI 配合使用
- 将 UntagResource 与 CLI 配合使用
- 将 UpdateAlias 与 CLI 配合使用
- 将 UpdateFunctionCode 与 AWS SDK 或 CLI 配合使用
- 将 UpdateFunctionConfiguration 与 AWS SDK 或 CLI 配合使用

开始使用 Lambda

Hello Lambda 1654

以下代码示例展示了如何开始使用 Lambda。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
namespace LambdaActions;

using Amazon.Lambda;

public class HelloLambda
{
    static async Task Main(string[] args)
    {
        var lambdaClient = new AmazonLambdaClient();

        Console.WriteLine("Hello AWS Lambda");
        Console.WriteLine("Let's get started with AWS Lambda by listing your
 existing Lambda functions:");

        var response = await lambdaClient.ListFunctionsAsync();
        response.Functions.ForEach(function =>
        {

 Console.WriteLine($"{function.FunctionName}\t{function.Description}");
        });
    }
}
```

- 有关 API 的详细信息，请参阅 AWS SDK for .NET API 参考中的 ListFunctions。

C++

SDK for C++

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

CMakeLists.txt CMake 文件的代码。

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS lambda)

# Set this project's name.
project("hello_lambda")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.
```

```
    # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
 may need to uncomment this
                                    # and set the proper subdirectory to the
 executables' location.

    AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_lambda.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

hello_lambda.cpp 源文件的代码。

```
#include <aws/core/Aws.h>
#include <aws/lambda/LambdaClient.h>
#include <aws/lambda/model/ListFunctionsRequest.h>
#include <iostream>

/*
 *  A "Hello Lambda" starter application which initializes an AWS Lambda (Lambda)
 client and lists the Lambda functions.
 *
 *  main function
 *
 *  Usage: 'hello_lambda'
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";
```

```
        Aws::Lambda::LambdaClient lambdaClient(clientConfig);
        std::vector<Aws::String> functions;
        Aws::String marker; // Used for pagination.

        do {
            Aws::Lambda::Model::ListFunctionsRequest request;
            if (!marker.empty()) {
                request.SetMarker(marker);
            }

            Aws::Lambda::Model::ListFunctionsOutcome outcome =
lambdaClient.ListFunctions(
                    request);

            if (outcome.IsSuccess()) {
                const Aws::Lambda::Model::ListFunctionsResult
&listFunctionsResult = outcome.GetResult();
                std::cout << listFunctionsResult.GetFunctions().size()
                          << " lambda functions were retrieved." << std::endl;

                for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: listFunctionsResult.GetFunctions()) {
                    functions.push_back(functionConfiguration.GetFunctionName());
                    std::cout << functions.size() << "  "
                              << functionConfiguration.GetDescription() <<
std::endl;
                    std::cout << "   "
                              <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                      functionConfiguration.GetRuntime()) << ": "
                              << functionConfiguration.GetHandler()
                              << std::endl;
                }
                marker = listFunctionsResult.GetNextMarker();
            } else {
                std::cerr << "Error with Lambda::ListFunctions. "
                          << outcome.GetError().GetMessage()
                          << std::endl;
                result = 1;
                break;
            }
        } while (!marker.empty());
    }
```

```
    Aws::ShutdownAPI(options); // Should only be called once.
    return result;
}
```

- 有关 API 的详细信息，请参阅 AWS SDK for C++ API 参考中的 ListFunctions。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
)

// main uses the AWS SDK for Go (v2) to create an AWS Lambda client and list up
 to 10
// functions in your account.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  fmt.Println("Couldn't load default configuration. Have you set up your AWS
 account?")
```

```
  fmt.Println(err)
  return
 }
 lambdaClient := lambda.NewFromConfig(sdkConfig)

 maxItems := 10
 fmt.Printf("Let's list up to %v functions for your account.\n", maxItems)
 result, err := lambdaClient.ListFunctions(ctx, &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 if err != nil {
  fmt.Printf("Couldn't list functions for your account. Here's why: %v\n", err)
  return
 }
 if len(result.Functions) == 0 {
  fmt.Println("You don't have any functions!")
 } else {
  for _, function := range result.Functions {
   fmt.Printf("\t%v\n", *function.FunctionName)
  }
 }
}
```

- 有关 API 的详细信息，请参阅 AWS SDK for Go API 参考中的 ListFunctions。

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
```

```
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
 Lambda service
     */
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
 config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- 有关 API 的详细信息，请参阅 AWS SDK for Java 2.x API 参考中的 ListFunctions。

JavaScript

适用于 JavaScript 的 SDK (v3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
  const paginator = paginateListFunctions({ client }, {});
  const functions = [];

  for await (const page of paginator) {
    const funcNames = page.Functions.map((f) => f.FunctionName);
```

```
    functions.push(...funcNames);
  }

  console.log("Functions:");
  console.log(functions.join("\n"));
  return functions;
};
```

- 有关 API 的详细信息，请参阅 AWS SDK for JavaScript API 参考中的 ListFunctions。

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
import boto3


def main():
    """
    List the Lambda functions in your AWS account.
    """
    # Create the Lambda client
    lambda_client = boto3.client("lambda")

    # Use the paginator to list the functions
    paginator = lambda_client.get_paginator("list_functions")
    response_iterator = paginator.paginate()

    print("Here are the Lambda functions in your account:")
    for page in response_iterator:
        for function in page["Functions"]:
            print(f"  {function['FunctionName']}")
```

```
if __name__ == "__main__":
    main()
```

- 有关 API 详细信息，请参阅《AWS SDK for Python (Boto3) API 参考》中的 ListFunctions。

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
require 'aws-sdk-lambda'

# Creates an AWS Lambda client using the default credentials and configuration
def lambda_client
  Aws::Lambda::Client.new
end

# Lists the Lambda functions in your AWS account, paginating the results if
 necessary
def list_lambda_functions
  lambda = lambda_client

  # Use a pagination iterator to list all functions
  functions = []
  lambda.list_functions.each_page do |page|
    functions.concat(page.functions)
  end

  # Print the name and ARN of each function
  functions.each do |function|
    puts "Function name: #{function.function_name}"
    puts "Function ARN: #{function.function_arn}"
    puts
  end
```

```
  puts "Total functions: #{functions.count}"
end

list_lambda_functions if __FILE__ == $PROGRAM_NAME
```

- 有关 API 的详细信息，请参阅 AWS SDK for Ruby API 参考中的 ListFunctions。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

了解将 Lambda 与 AWS SDK 结合使用的基础知识

以下代码示例演示了如何：

- 创建 IAM 角色和 Lambda 函数，然后上传处理程序代码。
- 使用单个参数来调用函数并获取结果。
- 更新函数代码并使用环境变量进行配置。
- 使用新参数来调用函数并获取结果。显示返回的执行日志。
- 列出您账户的函数，然后清理函数。

有关更多信息，请参阅使用控制台创建 Lambda 函数。

.NET

AWS SDK for .NET

> **Note**
>
> 在 GitHub 上查看更多内容。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

创建用于执行 Lambda 操作的方法。

```
namespace LambdaActions;
```

```
using Amazon.Lambda;
using Amazon.Lambda.Model;

/// <summary>
/// A class that implements AWS Lambda methods.
/// </summary>
public class LambdaWrapper
{
    private readonly IAmazonLambda _lambdaService;

    /// <summary>
    /// Constructor for the LambdaWrapper class.
    /// </summary>
    /// <param name="lambdaService">An initialized Lambda service client.</param>
    public LambdaWrapper(IAmazonLambda lambdaService)
    {
        _lambdaService = lambdaService;
    }

    /// <summary>
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
    /// bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the
    /// appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created
    /// Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
        {
```

```
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
 _lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }


    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }


    /// <summary>
```

```
   /// Gets information about a Lambda function.
   /// </summary>
   /// <param name="functionName">The name of the Lambda function for
   /// which to retrieve information.</param>
   /// <returns>Async Task.</returns>
   public async Task<FunctionConfiguration> GetFunctionAsync(string
functionName)
   {
       var functionRequest = new GetFunctionRequest
       {
           FunctionName = functionName,
       };

       var response = await _lambdaService.GetFunctionAsync(functionRequest);
       return response.Configuration;
   }


   /// <summary>
   /// Invoke a Lambda function.
   /// </summary>
   /// <param name="functionName">The name of the Lambda function to
   /// invoke.</param
   /// <param name="parameters">The parameter values that will be passed to the
function.</param>
   /// <returns>A System Threading Task.</returns>
   public async Task<string> InvokeFunctionAsync(
       string functionName,
       string parameters)
   {
       var payload = parameters;
       var request = new InvokeRequest
       {
           FunctionName = functionName,
           Payload = payload,
       };

       var response = await _lambdaService.InvokeAsync(request);
       MemoryStream stream = response.Payload;
       string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
       return returnValue;
   }
```

```
    /// <summary>
    /// Get a list of Lambda functions.
    /// </summary>
    /// <returns>A list of FunctionConfiguration objects.</returns>
    public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
    {
        var functionList = new List<FunctionConfiguration>();

        var functionPaginator =
            _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
        await foreach (var function in functionPaginator.Functions)
        {
            functionList.Add(function);
        }

        return functionList;
    }


    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
```

```
        Console.WriteLine($"The Function was last modified at
{response.LastModified}.");
    }


    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
environmentVariables },
        };

        var response = await
_lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }


}
```

创建运行场景的函数。

```
global using System.Threading.Tasks;
```

```
global using Amazon.IdentityManagement;
global using Amazon.Lambda;
global using LambdaActions;
global using LambdaScenarioCommon;
global using Microsoft.Extensions.DependencyInjection;
global using Microsoft.Extensions.Hosting;
global using Microsoft.Extensions.Logging;
global using Microsoft.Extensions.Logging.Console;
global using Microsoft.Extensions.Logging.Debug;


using Amazon.Lambda.Model;
using Microsoft.Extensions.Configuration;

namespace LambdaBasics;

public class LambdaBasics
{
    private static ILogger logger = null!;

    static async Task Main(string[] args)
    {
        // Set up dependency injection for the Amazon service.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddFilter("System", LogLevel.Debug)
                    .AddFilter<DebugLoggerProvider>("Microsoft",
 LogLevel.Information)
                    .AddFilter<ConsoleLoggerProvider>("Microsoft",
 LogLevel.Trace))
            .ConfigureServices((_, services) =>
            services.AddAWSService<IAmazonLambda>()
            .AddAWSService<IAmazonIdentityManagementService>()
            .AddTransient<LambdaWrapper>()
            .AddTransient<LambdaRoleWrapper>()
            .AddTransient<UIWrapper>()
        )
        .Build();

        var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("settings.json") // Load test settings from .json file.
            .AddJsonFile("settings.local.json",
            true) // Optionally load local settings.
```

```
        .Build();


        logger = LoggerFactory.Create(builder => { builder.AddConsole(); })
            .CreateLogger<LambdaBasics>();

        var lambdaWrapper = host.Services.GetRequiredService<LambdaWrapper>();
        var lambdaRoleWrapper =
host.Services.GetRequiredService<LambdaRoleWrapper>();
        var uiWrapper = host.Services.GetRequiredService<UIWrapper>();

        string functionName = configuration["FunctionName"]!;
        string roleName = configuration["RoleName"]!;
        string policyDocument = "{" +
            " \"Version\": \"2012-10-17\"," +
            " \"Statement\": [ " +
            "    {" +
            "        \"Effect\": \"Allow\"," +
            "        \"Principal\": {" +
            "            \"Service\": \"lambda.amazonaws.com\" " +
            "    }," +
            "        \"Action\": \"sts:AssumeRole\" " +
            "    }" +
            "]" +
        "}";

        var incrementHandler = configuration["IncrementHandler"];
        var calculatorHandler = configuration["CalculatorHandler"];
        var bucketName = configuration["BucketName"];
        var incrementKey = configuration["IncrementKey"];
        var calculatorKey = configuration["CalculatorKey"];
        var policyArn = configuration["PolicyArn"];

        uiWrapper.DisplayLambdaBasicsOverview();

        // Create the policy to use with the AWS Lambda functions and then attach
the
        // policy to a new role.
        var roleArn = await lambdaRoleWrapper.CreateLambdaRoleAsync(roleName,
policyDocument);

        Console.WriteLine("Waiting for role to become active.");
        uiWrapper.WaitABit(15, "Wait until the role is active before trying to
use it.");
```

```
        // Attach the appropriate AWS Identity and Access Management (IAM) role
policy to the new role.
        var success = await
lambdaRoleWrapper.AttachLambdaRolePolicyAsync(policyArn, roleName);
        uiWrapper.WaitABit(10, "Allow time for the IAM policy to be attached to
the role.");

        // Create the Lambda function using a zip file stored in an Amazon Simple
Storage Service
        // (Amazon S3) bucket.
        uiWrapper.DisplayTitle("Create Lambda Function");
        Console.WriteLine($"Creating the AWS Lambda function: {functionName}.");
        var lambdaArn = await lambdaWrapper.CreateLambdaFunctionAsync(
            functionName,
            bucketName,
            incrementKey,
            roleArn,
            incrementHandler);

        Console.WriteLine("Waiting for the new function to be available.");
        Console.WriteLine($"The AWS Lambda ARN is {lambdaArn}");

        // Get the Lambda function.
        Console.WriteLine($"Getting the {functionName} AWS Lambda function.");
        FunctionConfiguration config;
        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.State != State.Active);

        Console.WriteLine($"\nThe function, {functionName} has been created.");
        Console.WriteLine($"The runtime of this Lambda function is
{config.Runtime}.");

        uiWrapper.PressEnter();

        // List the Lambda functions.
        uiWrapper.DisplayTitle("Listing all Lambda functions.");
        var functions = await lambdaWrapper.ListFunctionsAsync();
        DisplayFunctionList(functions);
```

```
        uiWrapper.DisplayTitle("Invoke increment function");
        Console.WriteLine("Now that it has been created, invoke the Lambda
increment function.");
        string? value;
        do
        {
            Console.Write("Enter a value to increment: ");
            value = Console.ReadLine();
        }
        while (string.IsNullOrEmpty(value));

        string functionParameters = "{" +
            "\"action\": \"increment\", " +
            "\"x\": \"" + value + "\"" +
        "}";
        var answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
        Console.WriteLine($"{value} + 1 = {answer}.");

        uiWrapper.DisplayTitle("Update function");
        Console.WriteLine("Now update the Lambda function code.");
        await lambdaWrapper.UpdateFunctionCodeAsync(functionName, bucketName,
calculatorKey);

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        await lambdaWrapper.UpdateFunctionConfigurationAsync(
            functionName,
            calculatorHandler,
            new Dictionary<string, string> { { "LOG_LEVEL", "DEBUG" } });

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        uiWrapper.DisplayTitle("Call updated function");
```

```
        Console.WriteLine("Now call the updated function...");

        bool done = false;

        do
        {
            string? opSelected;

            Console.WriteLine("Select the operation to perform:");
            Console.WriteLine("\t1. add");
            Console.WriteLine("\t2. subtract");
            Console.WriteLine("\t3. multiply");
            Console.WriteLine("\t4. divide");
            Console.WriteLine("\tOr enter \"q\" to quit.");
            Console.WriteLine("Enter the number (1, 2, 3, 4, or q) of the
operation you want to perform: ");
            do
            {
                Console.Write("Your choice? ");
                opSelected = Console.ReadLine();
            }
            while (opSelected == string.Empty);

            var operation = (opSelected) switch
            {
                "1" => "add",
                "2" => "subtract",
                "3" => "multiply",
                "4" => "divide",
                "q" => "quit",
                _ => "add",
            };

            if (operation == "quit")
            {
                done = true;
            }
            else
            {
                // Get two numbers and an action from the user.
                value = string.Empty;
                do
                {
                    Console.Write("Enter the first value: ");
```

```
                value = Console.ReadLine();
            }
            while (value == string.Empty);

            string? value2;
            do
            {
                Console.Write("Enter a second value: ");
                value2 = Console.ReadLine();
            }
            while (value2 == string.Empty);

            functionParameters = "{" +
                "\"action\": \"" + operation + "\", " +
                "\"x\": \"" + value + "\"," +
                "\"y\": \"" + value2 + "\"" +
            "}";

            answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
            Console.WriteLine($"The answer when we {operation} the two
numbers is: {answer}.");
        }

        uiWrapper.PressEnter();
    } while (!done);

    // Delete the function created earlier.

    uiWrapper.DisplayTitle("Clean up resources");
    // Detach the IAM policy from the IAM role.
    Console.WriteLine("First detach the IAM policy from the role.");
    success = await lambdaRoleWrapper.DetachLambdaRolePolicyAsync(policyArn,
roleName);
    uiWrapper.WaitABit(15, "Let's wait for the policy to be fully detached
from the role.");

    Console.WriteLine("Delete the AWS Lambda function.");
    success = await lambdaWrapper.DeleteFunctionAsync(functionName);
    if (success)
    {
        Console.WriteLine($"The {functionName} function was deleted.");
    }
    else
```

```
        {
            Console.WriteLine($"Could not remove the function {functionName}");
        }

        // Now delete the IAM role created for use with the functions
        // created by the application.
        Console.WriteLine("Now we can delete the role that we created.");
        success = await lambdaRoleWrapper.DeleteLambdaRoleAsync(roleName);
        if (success)
        {
            Console.WriteLine("The role has been successfully removed.");
        }
        else
        {
            Console.WriteLine("Couldn't delete the role.");
        }

        Console.WriteLine("The Lambda Scenario is now complete.");
        uiWrapper.PressEnter();

        // Displays a formatted list of existing functions returned by the
        // LambdaMethods.ListFunctions.
        void DisplayFunctionList(List<FunctionConfiguration> functions)
        {
            functions.ForEach(functionConfig =>
            {

 Console.WriteLine($"{functionConfig.FunctionName}\t{functionConfig.Description}");
            });
        }
    }
}


namespace LambdaActions;

using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

public class LambdaRoleWrapper
{
    private readonly IAmazonIdentityManagementService _lambdaRoleService;

    public LambdaRoleWrapper(IAmazonIdentityManagementService lambdaRoleService)
```

```
    {
        _lambdaRoleService = lambdaRoleService;
    }

    /// <summary>
    /// Attach an AWS Identity and Access Management (IAM) role policy to the
    /// IAM role to be assumed by the AWS Lambda functions created for the
 scenario.
    /// </summary>
    /// <param name="policyArn">The Amazon Resource Name (ARN) of the IAM
 policy.</param>
    /// <param name="roleName">The name of the IAM role to attach the IAM policy
 to.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> AttachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
        var response = await _lambdaRoleService.AttachRolePolicyAsync(new
 AttachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    /// <summary>
    /// Create a new IAM role.
    /// </summary>
    /// <param name="roleName">The name of the IAM role to create.</param>
    /// <param name="policyDocument">The policy document for the new IAM role.</
param>
    /// <returns>A string representing the ARN for newly created role.</returns>
    public async Task<string> CreateLambdaRoleAsync(string roleName, string
 policyDocument)
    {
        var request = new CreateRoleRequest
        {
            AssumeRolePolicyDocument = policyDocument,
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.CreateRoleAsync(request);
        return response.Role.Arn;
    }

    /// <summary>
    /// Deletes an IAM role.
```

```
    /// </summary>
    /// <param name="roleName">The name of the role to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteLambdaRoleAsync(string roleName)
    {
        var request = new DeleteRoleRequest
        {
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.DeleteRoleAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    public async Task<bool> DetachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
        var response = await _lambdaRoleService.DetachRolePolicyAsync(new
 DetachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
}


namespace LambdaScenarioCommon;
public class UIWrapper
{
    public readonly string SepBar = new('-', Console.WindowWidth);

    /// <summary>
    /// Show information about the AWS Lambda Basics scenario.
    /// </summary>
    public void DisplayLambdaBasicsOverview()
    {
        Console.Clear();

        DisplayTitle("Welcome to AWS Lambda Basics");
        Console.WriteLine("This example application does the following:");
        Console.WriteLine("\t1. Creates an AWS Identity and Access Management
 (IAM) role that will be assumed by the functions we create.");
        Console.WriteLine("\t2. Attaches an IAM role policy that has Lambda
 permissions.");
```

```
        Console.WriteLine("\t3. Creates a Lambda function that increments the
value passed to it.");
        Console.WriteLine("\t4. Calls the increment function and passes a
value.");
        Console.WriteLine("\t5. Updates the code so that the function is a simple
calculator.");
        Console.WriteLine("\t6. Calls the calculator function with the values
entered.");
        Console.WriteLine("\t7. Deletes the Lambda function.");
        Console.WriteLine("\t7. Detaches the IAM role policy.");
        Console.WriteLine("\t8. Deletes the IAM role.");
        PressEnter();
    }

    /// <summary>
    /// Display a message and wait until the user presses enter.
    /// </summary>
    public void PressEnter()
    {
        Console.Write("\nPress <Enter> to continue. ");
        _ = Console.ReadLine();
        Console.WriteLine();
    }

    /// <summary>
    /// Pad a string with spaces to center it on the console display.
    /// </summary>
    /// <param name="strToCenter">The string to be centered.</param>
    /// <returns>The padded string.</returns>
    public string CenterString(string strToCenter)
    {
        var padAmount = (Console.WindowWidth - strToCenter.Length) / 2;
        var leftPad = new string(' ', padAmount);
        return $"{leftPad}{strToCenter}";
    }

    /// <summary>
    /// Display a line of hyphens, the centered text of the title and another
    /// line of hyphens.
    /// </summary>
    /// <param name="strTitle">The string to be displayed.</param>
    public void DisplayTitle(string strTitle)
    {
        Console.WriteLine(SepBar);
```

```
        Console.WriteLine(CenterString(strTitle));
        Console.WriteLine(SepBar);
    }

    /// <summary>
    /// Display a countdown and wait for a number of seconds.
    /// </summary>
    /// <param name="numSeconds">The number of seconds to wait.</param>
    public void WaitABit(int numSeconds, string msg)
    {
        Console.WriteLine(msg);

        // Wait for the requested number of seconds.
        for (int i = numSeconds; i > 0; i--)
        {
            System.Threading.Thread.Sleep(1000);
            Console.Write($"{i}...");
        }

        PressEnter();
    }
}
```

定义一个递增数字的 Lambda 处理程序。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaIncrement;

public class Function
{

    /// <summary>
    /// A simple function increments the integer parameter.
    /// </summary>
    /// <param name="input">A JSON string containing an action, which must be
```

```
    /// "increment" and a string representing the value to increment.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the incremented value of the parameter.</
returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        if (input["action"] == "increment")
        {
            int inputValue = Convert.ToInt32(input["x"]);
            return inputValue + 1;
        }
        else
        {
            return 0;
        }
    }
}
```

定义执行算术运算的第二个 Lambda 处理程序。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaCalculator;

public class Function
{

    /// <summary>
    /// A simple function that takes two number in string format and performs
    /// the requested arithmetic function.
    /// </summary>
    /// <param name="input">JSON data containing an action, and x and y values.
    /// Valid actions include: add, subtract, multiply, and divide.</param>
```

```
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the results of the calculation.</returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        var action = input["action"];
        int x = Convert.ToInt32(input["x"]);
        int y = Convert.ToInt32(input["y"]);
        int result;
        switch (action)
        {
            case "add":
                result = x + y;
                break;
            case "subtract":
                result = x - y;
                break;
            case "multiply":
                result = x * y;
                break;
            case "divide":
                if (y == 0)
                {
                    Console.Error.WriteLine("Divide by zero error.");
                    result = 0;
                }
                else
                    result = x / y;
                break;
            default:
                Console.Error.WriteLine($"{action} is not a valid operation.");
                result = 0;
                break;
        }
        return result;
    }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for .NET API 参考》中的以下主题。

- CreateFunction
- DeleteFunction
- GetFunction
- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

C++

SDK for C++

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
//! Get started with functions scenario.
/*!
 \param clientConfig: AWS client configuration.
 \return bool: Successful completion.
 */
bool AwsDoc::Lambda::getStartedWithFunctionsScenario(
        const Aws::Client::ClientConfiguration &clientConfig) {

    Aws::Lambda::LambdaClient client(clientConfig);

    // 1. Create an AWS Identity and Access Management (IAM) role for Lambda
 function.
    Aws::String roleArn;
    if (!getIamRoleArn(roleArn, clientConfig)) {
        return false;
    }

    // 2. Create a Lambda function.
    int seconds = 0;
    do {
        Aws::Lambda::Model::CreateFunctionRequest request;
```

```
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_9);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
```

```
        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
seconds
                      << " seconds elapsed." << std::endl;
            break;
        }
        else if (outcome.GetError().GetErrorType() ==
                 Aws::Lambda::LambdaErrors::INVALID_PARAMETER_VALUE &&
                 outcome.GetError().GetMessage().find("role") >= 0) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout
                        << "Waiting for the IAM role to become available as a
CreateFunction parameter. "
                        << seconds
                        << " seconds elapsed." << std::endl;

                std::cout << outcome.GetError().GetMessage() << std::endl;
            }
        }
        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (60 > seconds);

    std::cout << "The current Lambda function increments 1 by an input." <<
std::endl;

    // 3.  Invoke the Lambda function.
    {
```

```
        int increment = askQuestionForInt("Enter an increment integer: ");

        Aws::Lambda::Model::InvokeResult invokeResult;
        Aws::Utils::Json::JsonValue jsonPayload;
        jsonPayload.WithString("action", "increment");
        jsonPayload.WithInteger("number", increment);
        if (invokeLambdaFunction(jsonPayload, Aws::Lambda::Model::LogType::Tail,
                                 invokeResult, client)) {
            Aws::Utils::Json::JsonValue jsonValue(invokeResult.GetPayload());
            Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                    jsonValue.View().GetAllObjects();
            auto iter = values.find("result");
            if (iter != values.end() && iter->second.IsIntegerType()) {
                {
                    std::cout << INCREMENT_RESUlT_PREFIX
                              << iter->second.AsInteger() << std::endl;
                }
            }
            else {
                std::cout << "There was an error in execution. Here is the log."
                          << std::endl;
                Aws::Utils::ByteBuffer buffer =
Aws::Utils::HashingUtils::Base64Decode(
                        invokeResult.GetLogResult());
                std::cout << "With log " << buffer.GetUnderlyingData() <<
std::endl;
            }
        }
    }

    std::cout
            << "The Lambda function will now be updated with new code. Press
return to continue, ";
    Aws::String answer;
    std::getline(std::cin, answer);

    // 4.  Update the Lambda function code.
    {
        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
```

```
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
 client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
 std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    }

    std::cout
            << "This function uses an environment variable to control the logging
 level."
            << std::endl;
    std::cout
            << "UpdateFunctionConfiguration will be used to set the LOG_LEVEL to
 DEBUG."
            << std::endl;
```

```
    seconds = 0;

    // 5.  Update the Lambda function configuration.
    do {
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);

        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

            // RESOURCE_IN_USE: function code update not completed.
        else if (outcome.GetError().GetErrorType() !=
                 Aws::Lambda::LambdaErrors::RESOURCE_IN_USE) {
            if ((seconds % 10) == 0) { // Log status every 10 seconds.
                std::cout << "Lambda function update in progress . After " <<
seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }

    } while (0 < seconds);

    if (0 > seconds) {
        std::cerr << "Function failed to become active." << std::endl;
    }
    else {
        std::cout << "Updated function active after " << seconds << " seconds."
```

```
                  << std::endl;
    }

    std::cout
            << "\nThe new code applies an arithmetic operator to two variables, x
 an y."
            << std::endl;
    std::vector<Aws::String> operators = {"plus", "minus", "times", "divided-
by"};
    for (size_t i = 0; i < operators.size(); ++i) {
        std::cout << "   " << i + 1 << " " << operators[i] << std::endl;
    }

    // 6.  Invoke the updated Lambda function.
    do {
        int operatorIndex = askQuestionForIntRange("Select an operator index 1 -
 4 ", 1,
                                                   4);
        int x = askQuestionForInt("Enter an integer for the x value ");
        int y = askQuestionForInt("Enter an integer for the y value ");

        Aws::Utils::Json::JsonValue calculateJsonPayload;
        calculateJsonPayload.WithString("action", operators[operatorIndex - 1]);
        calculateJsonPayload.WithInteger("x", x);
        calculateJsonPayload.WithInteger("y", y);
        Aws::Lambda::Model::InvokeResult calculatedResult;
        if (invokeLambdaFunction(calculateJsonPayload,
                                 Aws::Lambda::Model::LogType::Tail,
                                 calculatedResult, client)) {
            Aws::Utils::Json::JsonValue jsonValue(calculatedResult.GetPayload());
            Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                    jsonValue.View().GetAllObjects();
            auto iter = values.find("result");
            if (iter != values.end() && iter->second.IsIntegerType()) {
                std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                          << operators[operatorIndex - 1] << " "
                          << y << " is " << iter->second.AsInteger() <<
 std::endl;
            }
            else if (iter != values.end() && iter->second.IsFloatingPointType())
 {
                std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                          << operators[operatorIndex - 1] << " "
                          << y << " is " << iter->second.AsDouble() << std::endl;
```

```
            }
            else {
                std::cout << "There was an error in execution. Here is the log."
                          << std::endl;
                Aws::Utils::ByteBuffer buffer =
Aws::Utils::HashingUtils::Base64Decode(
                        calculatedResult.GetLogResult());
                std::cout << "With log " << buffer.GetUnderlyingData() <<
std::endl;
            }
        }

        answer = askQuestion("Would you like to try another operation? (y/n) ");
    } while (answer == "y");

    std::cout
            << "A list of the lambda functions will be retrieved. Press return to
continue, ";
    std::getline(std::cin, answer);

    // 7.  List the Lambda functions.

    std::vector<Aws::String> functions;
    Aws::String marker;

    do {
        Aws::Lambda::Model::ListFunctionsRequest request;
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
                request);

        if (outcome.IsSuccess()) {
            const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
            std::cout << result.GetFunctions().size()
                      << " lambda functions were retrieved." << std::endl;

            for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
                functions.push_back(functionConfiguration.GetFunctionName());
                std::cout << functions.size() << "  "
```

```
                        << functionConfiguration.GetDescription() << std::endl;
                std::cout << "   "
                        <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                functionConfiguration.GetRuntime()) << ": "
                        << functionConfiguration.GetHandler()
                        << std::endl;
            }
            marker = result.GetNextMarker();
        }
        else {
            std::cerr << "Error with Lambda::ListFunctions. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    } while (!marker.empty());

    // 8.  Get a Lambda function.
    if (!functions.empty()) {
        std::stringstream question;
        question << "Choose a function to retrieve between 1 and " <<
functions.size()
                 << " ";
        int functionIndex = askQuestionForIntRange(question.str(), 1,

static_cast<int>(functions.size()));

        Aws::String functionName = functions[functionIndex - 1];

        Aws::Lambda::Model::GetFunctionRequest request;
        request.SetFunctionName(functionName);

        Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

        if (outcome.IsSuccess()) {
            std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                      << std::endl;
        }
        else {
            std::cerr << "Error with Lambda::GetFunction. "
                      << outcome.GetError().GetMessage()
```

```
                        << std::endl;
        }
    }

    std::cout << "The resources will be deleted. Press return to continue, ";
    std::getline(std::cin, answer);

    // 9.  Delete the Lambda function.
    bool result = deleteLambdaFunction(client);

    // 10. Delete the IAM role.
    return result && deleteIamRole(clientConfig);
}

//! Routine which invokes a Lambda function and returns the result.
/*!
 \param jsonPayload: Payload for invoke function.
 \param logType: Log type setting for invoke function.
 \param invokeResult: InvokeResult object to receive the result.
 \param client: Lambda client.
 \return bool: Successful completion.
 */
bool
AwsDoc::Lambda::invokeLambdaFunction(const Aws::Utils::Json::JsonValue
 &jsonPayload,
                                     Aws::Lambda::Model::LogType logType,
                                     Aws::Lambda::Model::InvokeResult
 &invokeResult,
                                     const Aws::Lambda::LambdaClient &client) {
    int seconds = 0;
    bool result = false;
    /*
     * In this example, the Invoke function can be called before recently created
 resources are
     * available.  The Invoke function is called repeatedly until the resources
 are
     * available.
     */
    do {
        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
 Aws::MakeShared<Aws::StringStream>(
```

```
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }

            // ACCESS_DENIED: because the role is not available yet.
            // RESOURCE_CONFLICT: because the Lambda function is being created or
 updated.
        else if ((outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::ACCESS_DENIED) ||
                 (outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::RESOURCE_CONFLICT)) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout << "Waiting for the invoke api to be available, status
 " <<
                          ((outcome.GetError().GetErrorType() ==
                            Aws::Lambda::LambdaErrors::ACCESS_DENIED ?
                            "ACCESS_DENIED" : "RESOURCE_CONFLICT")) << ". " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (seconds < 60);

    return result;
}
```

- 有关 API 的详细信息，请参阅 AWS SDK for C++ API 参考中的以下主题。

- CreateFunction
- DeleteFunction
- GetFunction
- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

创建一个展示如何开始使用 Lambda 函数的交互式场景。

```
// GetStartedFunctionsScenario shows you how to use AWS Lambda to perform the
 following
// actions:
//
//  1. Create an AWS Identity and Access Management (IAM) role and Lambda
 function, then upload handler code.
//  2. Invoke the function with a single parameter and get results.
//  3. Update the function code and configure with an environment variable.
//  4. Invoke the function with new parameters and get results. Display the
 returned execution log.
//  5. List the functions for your account, then clean up resources.
type GetStartedFunctionsScenario struct {
 sdkConfig       aws.Config
 functionWrapper actions.FunctionWrapper
 questioner      demotools.IQuestioner
 helper          IScenarioHelper
 isTestRun       bool
}
```

```
// NewGetStartedFunctionsScenario constructs a GetStartedFunctionsScenario
 instance from a configuration.
// It uses the specified config to get a Lambda client and create wrappers for
 the actions
// used in the scenario.
func NewGetStartedFunctionsScenario(sdkConfig aws.Config, questioner
 demotools.IQuestioner,
 helper IScenarioHelper) GetStartedFunctionsScenario {
 lambdaClient := lambda.NewFromConfig(sdkConfig)
 return GetStartedFunctionsScenario{
  sdkConfig:       sdkConfig,
  functionWrapper: actions.FunctionWrapper{LambdaClient: lambdaClient},
  questioner:      questioner,
  helper:          helper,
 }
}

// Run runs the interactive scenario.
func (scenario GetStartedFunctionsScenario) Run(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong with the demo.\n")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the AWS Lambda get started with functions demo.")
 log.Println(strings.Repeat("-", 88))

 role := scenario.GetOrCreateRole(ctx)
 funcName := scenario.CreateFunction(ctx, role)
 scenario.InvokeIncrement(ctx, funcName)
 scenario.UpdateFunction(ctx, funcName)
 scenario.InvokeCalculator(ctx, funcName)
 scenario.ListFunctions(ctx)
 scenario.Cleanup(ctx, role, funcName)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

```
// GetOrCreateRole checks whether the specified role exists and returns it if it
 does.
// Otherwise, a role is created that specifies Lambda as a trusted principal.
// The AWSLambdaBasicExecutionRole managed policy is attached to the role and the
 role
// is returned.
func (scenario GetStartedFunctionsScenario) GetOrCreateRole(ctx context.Context)
 *iamtypes.Role {
 var role *iamtypes.Role
 iamClient := iam.NewFromConfig(scenario.sdkConfig)
 log.Println("First, we need an IAM role that Lambda can assume.")
 roleName := scenario.questioner.Ask("Enter a name for the role:",
 demotools.NotEmpty{})
 getOutput, err := iamClient.GetRole(ctx, &iam.GetRoleInput{
  RoleName: aws.String(roleName)})
 if err != nil {
  var noSuch *iamtypes.NoSuchEntityException
  if errors.As(err, &noSuch) {
   log.Printf("Role %v doesn't exist. Creating it....\n", roleName)
  } else {
   log.Panicf("Couldn't check whether role %v exists. Here's why: %v\n",
    roleName, err)
  }
 } else {
  role = getOutput.Role
  log.Printf("Found role %v.\n", *role.RoleName)
 }
 if role == nil {
  trustPolicy := PolicyDocument{
   Version: "2012-10-17",
   Statement: []PolicyStatement{{
    Effect:    "Allow",
    Principal: map[string]string{"Service": "lambda.amazonaws.com"},
    Action:    []string{"sts:AssumeRole"},
   }},
  }
  policyArn := "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
  createOutput, err := iamClient.CreateRole(ctx, &iam.CreateRoleInput{
   AssumeRolePolicyDocument: aws.String(trustPolicy.String()),
   RoleName:                 aws.String(roleName),
  })
  if err != nil {
   log.Panicf("Couldn't create role %v. Here's why: %v\n", roleName, err)
  }
```

```
  role = createOutput.Role
  _, err = iamClient.AttachRolePolicy(ctx, &iam.AttachRolePolicyInput{
   PolicyArn: aws.String(policyArn),
   RoleName:  aws.String(roleName),
  })
  if err != nil {
   log.Panicf("Couldn't attach a policy to role %v. Here's why: %v\n", roleName,
 err)
  }
  log.Printf("Created role %v.\n", *role.RoleName)
  log.Println("Let's give AWS a few seconds to propagate resources...")
  scenario.helper.Pause(10)
 }
 log.Println(strings.Repeat("-", 88))
 return role
}

// CreateFunction creates a Lambda function and uploads a handler written in
 Python.
// The code for the Python handler is packaged as a []byte in .zip format.
func (scenario GetStartedFunctionsScenario) CreateFunction(ctx context.Context,
 role *iamtypes.Role) string {
 log.Println("Let's create a function that increments a number.\n" +
  "The function uses the 'lambda_handler_basic.py' script found in the \n" +
  "'handlers' directory of this project.")
 funcName := scenario.questioner.Ask("Enter a name for the Lambda function:",
 demotools.NotEmpty{})
 zipPackage := scenario.helper.CreateDeploymentPackage("lambda_handler_basic.py",
 fmt.Sprintf("%v.py", funcName))
 log.Printf("Creating function %v and waiting for it to be ready.", funcName)
 funcState := scenario.functionWrapper.CreateFunction(ctx, funcName,
 fmt.Sprintf("%v.lambda_handler", funcName),
  role.Arn, zipPackage)
 log.Printf("Your function is %v.", funcState)
 log.Println(strings.Repeat("-", 88))
 return funcName
}

// InvokeIncrement invokes a Lambda function that increments a number. The
 function
// parameters are contained in a Go struct that is used to serialize the
 parameters to
// a JSON payload that is passed to the function.
```

```
// The result payload is deserialized into a Go struct that contains an int
 value.
func (scenario GetStartedFunctionsScenario) InvokeIncrement(ctx context.Context,
 funcName string) {
 parameters := actions.IncrementParameters{Action: "increment"}
 log.Println("Let's invoke our function. This function increments a number.")
 parameters.Number = scenario.questioner.AskInt("Enter a number to increment:",
 demotools.NotEmpty{})
 log.Printf("Invoking %v with %v...\n", funcName, parameters.Number)
 invokeOutput := scenario.functionWrapper.Invoke(ctx, funcName, parameters,
 false)
 var payload actions.LambdaResultInt
 err := json.Unmarshal(invokeOutput.Payload, &payload)
 if err != nil {
  log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
   funcName, err)
 }
 log.Printf("Invoking %v with %v returned %v.\n", funcName, parameters.Number,
 payload)
 log.Println(strings.Repeat("-", 88))
}

// UpdateFunction updates the code for a Lambda function by uploading a simple
 arithmetic
// calculator written in Python. The code for the Python handler is packaged as a
// []byte in .zip format.
// After the code is updated, the configuration is also updated with a new log
// level that instructs the handler to log additional information.
func (scenario GetStartedFunctionsScenario) UpdateFunction(ctx context.Context,
 funcName string) {
 log.Println("Let's update the function to an arithmetic calculator.\n" +
  "The function uses the 'lambda_handler_calculator.py' script found in the \n" +
  "'handlers' directory of this project.")
 scenario.questioner.Ask("Press Enter when you're ready.")
 log.Println("Creating deployment package...")
 zipPackage :=
 scenario.helper.CreateDeploymentPackage("lambda_handler_calculator.py",
  fmt.Sprintf("%v.py", funcName))
 log.Println("...and updating the Lambda function and waiting for it to be
 ready.")
 funcState := scenario.functionWrapper.UpdateFunctionCode(ctx, funcName,
 zipPackage)
 log.Printf("Updated function %v. Its current state is %v.", funcName, funcState)
```

```
 log.Println("This function uses an environment variable to control logging
 level.")
 log.Println("Let's set it to DEBUG to get the most logging.")
 scenario.functionWrapper.UpdateFunctionConfiguration(ctx, funcName,
  map[string]string{"LOG_LEVEL": "DEBUG"})
 log.Println(strings.Repeat("-", 88))
}

// InvokeCalculator invokes the Lambda calculator function. The parameters are
 stored in a
// Go struct that is used to serialize the parameters to a JSON payload. That
 payload is then passed
// to the function.
// The result payload is deserialized to a Go struct that stores the result as
 either an
// int or float32, depending on the kind of operation that was specified.
func (scenario GetStartedFunctionsScenario) InvokeCalculator(ctx context.Context,
 funcName string) {
 wantInvoke := true
 choices := []string{"plus", "minus", "times", "divided-by"}
 for wantInvoke {
  choice := scenario.questioner.AskChoice("Select an arithmetic operation:\n",
 choices)
  x := scenario.questioner.AskInt("Enter a value for x:", demotools.NotEmpty{})
  y := scenario.questioner.AskInt("Enter a value for y:", demotools.NotEmpty{})
  log.Printf("Invoking %v %v %v...", x, choices[choice], y)
  calcParameters := actions.CalculatorParameters{
   Action: choices[choice],
   X:      x,
   Y:      y,
  }
  invokeOutput := scenario.functionWrapper.Invoke(ctx, funcName, calcParameters,
 true)
  var payload any
  if choice == 3 { // divide-by results in a float.
   payload = actions.LambdaResultFloat{}
  } else {
   payload = actions.LambdaResultInt{}
  }
  err := json.Unmarshal(invokeOutput.Payload, &payload)
  if err != nil {
   log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
    funcName, err)
  }
```

```
  log.Printf("Invoking %v with %v %v %v returned %v.\n", funcName,
   calcParameters.X, calcParameters.Action, calcParameters.Y, payload)
  scenario.questioner.Ask("Press Enter to see the logs from the call.")
  logRes, err := base64.StdEncoding.DecodeString(*invokeOutput.LogResult)
  if err != nil {
   log.Panicf("Couldn't decode log result. Here's why: %v\n", err)
  }
  log.Println(string(logRes))
  wantInvoke = scenario.questioner.AskBool("Do you want to calculate again? (y/
n)", "y")
 }
 log.Println(strings.Repeat("-", 88))
}

// ListFunctions lists up to the specified number of functions for your account.
func (scenario GetStartedFunctionsScenario) ListFunctions(ctx context.Context) {
 count := scenario.questioner.AskInt(
  "Let's list functions for your account. How many do you want to see?",
 demotools.NotEmpty{})
 functions := scenario.functionWrapper.ListFunctions(ctx, count)
 log.Printf("Found %v functions:", len(functions))
 for _, function := range functions {
  log.Printf("\t%v", *function.FunctionName)
 }
 log.Println(strings.Repeat("-", 88))
}

// Cleanup removes the IAM and Lambda resources created by the example.
func (scenario GetStartedFunctionsScenario) Cleanup(ctx context.Context, role
 *iamtypes.Role, funcName string) {
 if scenario.questioner.AskBool("Do you want to clean up resources created for
 this example? (y/n)",
  "y") {
  iamClient := iam.NewFromConfig(scenario.sdkConfig)
  policiesOutput, err := iamClient.ListAttachedRolePolicies(ctx,
   &iam.ListAttachedRolePoliciesInput{RoleName: role.RoleName})
  if err != nil {
   log.Panicf("Couldn't get policies attached to role %v. Here's why: %v\n",
    *role.RoleName, err)
  }
  for _, policy := range policiesOutput.AttachedPolicies {
   _, err = iamClient.DetachRolePolicy(ctx, &iam.DetachRolePolicyInput{
    PolicyArn: policy.PolicyArn, RoleName: role.RoleName,
   })
```

```
   if err != nil {
    log.Panicf("Couldn't detach policy %v from role %v. Here's why: %v\n",
     *policy.PolicyArn, *role.RoleName, err)
   }
  }
  _, err = iamClient.DeleteRole(ctx, &iam.DeleteRoleInput{RoleName:
 role.RoleName})
  if err != nil {
   log.Panicf("Couldn't delete role %v. Here's why: %v\n", *role.RoleName, err)
  }
  log.Printf("Deleted role %v.\n", *role.RoleName)

  scenario.functionWrapper.DeleteFunction(ctx, funcName)
  log.Printf("Deleted function %v.\n", funcName)
 } else {
  log.Println("Okay. Don't forget to delete the resources when you're done with
 them.")
 }
}
```

创建一个封装单个 Lambda 操作的结构。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}


// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(ctx context.Context, functionName
 string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(ctx,
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
```

```
  state = funcOutput.Configuration.State
 }
 return state
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(ctx context.Context, functionName
 string, handlerName string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(ctx, &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython39,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
```

```
    state = funcOutput.Configuration.State
   }
  }
  return state
}



// UpdateFunctionCode updates the code for the Lambda function specified by
  functionName.
// The existing code for the Lambda function is entirely replaced by the code in
  the
// zipPackage buffer. After the update action is called, a
  lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(ctx context.Context,
  functionName string, zipPackage *bytes.Buffer) types.State {
  var state types.State
  _, err := wrapper.LambdaClient.UpdateFunctionCode(ctx,
  &lambda.UpdateFunctionCodeInput{
   FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
  })
  if err != nil {
   log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
  functionName, err)
  } else {
   waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
   funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
    FunctionName: aws.String(functionName)}, 1*time.Minute)
   if err != nil {
    log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
  functionName, err)
   } else {
    state = funcOutput.Configuration.State
   }
  }
  return state
}



// UpdateFunctionConfiguration updates a map of environment variables configured
  for
// the Lambda function specified by functionName.
```

```
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(ctx context.Context,
 functionName string, envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(ctx,
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(ctx context.Context, maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(ctx)
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(ctx context.Context, functionName
 string) {
 _, err := wrapper.LambdaClient.DeleteFunction(ctx, &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
```

```
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(ctx context.Context, functionName string,
 parameters any, getLog bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(ctx, &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}



// IncrementParameters is used to serialize parameters to the increment Lambda
 handler.
type IncrementParameters struct {
 Action string `json:"action"`
 Number int    `json:"number"`
}

// CalculatorParameters is used to serialize parameters to the calculator Lambda
 handler.
type CalculatorParameters struct {
```

```
 Action string `json:"action"`
 X      int    `json:"x"`
 Y      int    `json:"y"`
}

// LambdaResultInt is used to deserialize an int result from a Lambda handler.
type LambdaResultInt struct {
 Result int `json:"result"`
}

// LambdaResultFloat is used to deserialize a float32 result from a Lambda
 handler.
type LambdaResultFloat struct {
 Result float32 `json:"result"`
}
```

创建一个实施函数的结构以帮助运行场景。

```
// IScenarioHelper abstracts I/O and wait functions from a scenario so that they
// can be mocked for unit testing.
type IScenarioHelper interface {
 Pause(secs int)
 CreateDeploymentPackage(sourceFile string, destinationFile string) *bytes.Buffer
}

// ScenarioHelper lets the caller specify the path to Lambda handler functions.
type ScenarioHelper struct {
 HandlerPath string
}

// Pause waits for the specified number of seconds.
func (helper *ScenarioHelper) Pause(secs int) {
 time.Sleep(time.Duration(secs) * time.Second)
}

// CreateDeploymentPackage creates an AWS Lambda deployment package from a source
 file. The
// deployment package is stored in .zip format in a bytes.Buffer. The buffer can
 be
// used to pass a []byte to Lambda when creating the function.
```

```
// The specified destinationFile is the name to give the file when it's deployed
 to Lambda.
func (helper *ScenarioHelper) CreateDeploymentPackage(sourceFile string,
 destinationFile string) *bytes.Buffer {
 var err error
 buffer := &bytes.Buffer{}
 writer := zip.NewWriter(buffer)
 zFile, err := writer.Create(destinationFile)
 if err != nil {
  log.Panicf("Couldn't create destination archive %v. Here's why: %v\n",
 destinationFile, err)
 }
 sourceBody, err := os.ReadFile(fmt.Sprintf("%v/%v", helper.HandlerPath,
 sourceFile))
 if err != nil {
  log.Panicf("Couldn't read handler source file %v. Here's why: %v\n",
   sourceFile, err)
 } else {
  _, err = zFile.Write(sourceBody)
  if err != nil {
   log.Panicf("Couldn't write handler %v to zip archive. Here's why: %v\n",
    sourceFile, err)
  }
 }
 err = writer.Close()
 if err != nil {
  log.Panicf("Couldn't close zip writer. Here's why: %v\n", err)
 }
 return buffer
}
```

定义一个递增数字的 Lambda 处理程序。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
```

```
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

定义执行算术运算的第二个 Lambda 处理程序。

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
```

```
    """
    Accepts an action and two numbers, performs the specified action on the
numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

- 有关 API 详细信息，请参阅《AWS SDK for Go API 参考》中的以下主题。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions

- UpdateFunctionCode
- UpdateFunctionConfiguration

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
/*
 *  Lambda function names appear as:
 *
 *  arn:aws:lambda:us-west-2:335556666777:function:HelloFunction
 *
 *  To find this value, look at the function in the AWS Management Console.
 *
 *  Before running this Java code example, set up your development environment,
 including your credentials.
 *
 *  For more information, see this documentation topic:
 *
 *  https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 *  This example performs the following tasks:
 *
 * 1. Creates an AWS Lambda function.
 * 2. Gets a specific AWS Lambda function.
 * 3. Lists all Lambda functions.
 * 4. Invokes a Lambda function.
 * 5. Updates the Lambda function code and invokes it again.
 * 6. Updates a Lambda function's configuration value.
 * 7. Deletes a Lambda function.
 */

public class LambdaScenario {
```

```
   public static final String DASHES = new String(new char[80]).replace("\0",
"-");

   public static void main(String[] args) throws InterruptedException {
       final String usage = """

           Usage:
               <functionName> <role> <handler> <bucketName> <key>\s

           Where:
               functionName - The name of the Lambda function.\s
               role - The AWS Identity and Access Management (IAM) service role
that has Lambda permissions.\s
               handler - The fully qualified method name (for example,
example.Handler::handleRequest).\s
               bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
name that contains the .zip or .jar used to update the Lambda function's code.\s
               key - The Amazon S3 key name that represents the .zip or .jar
(for example, LambdaHello-1.0-SNAPSHOT.jar).
               """;

       if (args.length != 5) {
             System.out.println(usage);
             return;
       }

       String functionName = args[0];
       String role = args[1];
       String handler = args[2];
       String bucketName = args[3];
       String key = args[4];
       LambdaClient awsLambda = LambdaClient.builder()
           .build();

       System.out.println(DASHES);
       System.out.println("Welcome to the AWS Lambda Basics scenario.");
       System.out.println(DASHES);

       System.out.println(DASHES);
       System.out.println("1. Create an AWS Lambda function.");
       String funArn = createLambdaFunction(awsLambda, functionName, key,
bucketName, role, handler);
       System.out.println("The AWS Lambda ARN is " + funArn);
       System.out.println(DASHES);
```

```
        System.out.println(DASHES);
        System.out.println("2. Get the " + functionName + " AWS Lambda
function.");
        getFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("3. List all AWS Lambda functions.");
        listFunctions(awsLambda);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Invoke the Lambda function.");
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("5. Update the Lambda function code and invoke it
again.");
        updateFunctionCode(awsLambda, functionName, bucketName, key);
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Update a Lambda function's configuration value.");
        updateFunctionConfiguration(awsLambda, functionName, handler);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Delete the AWS Lambda function.");
        LambdaScenario.deleteLambdaFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("The AWS Lambda scenario completed successfully");
        System.out.println(DASHES);
        awsLambda.close();
    }
```

```
   /**
    * Creates a new Lambda function in AWS using the AWS Lambda Java API.
    *
    * @param awsLambda    the AWS Lambda client used to interact with the AWS
Lambda service
    * @param functionName the name of the Lambda function to create
    * @param key          the S3 key of the function code
    * @param bucketName   the name of the S3 bucket containing the function code
    * @param role         the IAM role to assign to the Lambda function
    * @param handler      the fully qualified class name of the function handler
    * @return the Amazon Resource Name (ARN) of the created Lambda function
    */
   public static String createLambdaFunction(LambdaClient awsLambda,
                                              String functionName,
                                              String key,
                                              String bucketName,
                                              String role,
                                              String handler) {

       try {
           LambdaWaiter waiter = awsLambda.waiter();
           FunctionCode code = FunctionCode.builder()
               .s3Key(key)
               .s3Bucket(bucketName)
               .build();

           CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
               .functionName(functionName)
               .description("Created by the Lambda Java API")
               .code(code)
               .handler(handler)
               .runtime(Runtime.JAVA17)
               .role(role)
               .build();

           // Create a Lambda function using a waiter
           CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
           GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
               .functionName(functionName)
               .build();
           WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
```

```
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }

    /**
     * Retrieves information about an AWS Lambda function.
     *
     * @param awsLambda    an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to retrieve
information about
     */
    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();

            GetFunctionResponse response =
awsLambda.getFunction(functionRequest);
            System.out.println("The runtime of this Lambda function is " +
response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
Lambda service
     */
```

```
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Invokes a specific AWS Lambda function.
     *
     * @param awsLambda    an instance of {@link LambdaClient} to interact with
the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to be invoked
     */
    public static void invokeFunction(LambdaClient awsLambda, String
functionName) {
        InvokeResponse res;
        try {
            // Need a SdkBytes instance for the payload.
            JSONObject jsonObj = new JSONObject();
            jsonObj.put("inputValue", "2000");
            String json = jsonObj.toString();
            SdkBytes payload = SdkBytes.fromUtf8String(json);

            InvokeRequest request = InvokeRequest.builder()
                .functionName(functionName)
                .payload(payload)
                .build();

            res = awsLambda.invoke(request);
            String value = res.payload().asUtf8String();
            System.out.println(value);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
```

```
        }
    }

    /**
     * Updates the code for an AWS Lambda function.
     *
     * @param awsLambda  the AWS Lambda client
     * @param functionName the name of the Lambda function to update
     * @param bucketName the name of the S3 bucket where the function code is
located
     * @param key the key (file name) of the function code in the S3 bucket
     * @throws LambdaException if there is an error updating the function code
     */
    public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
        try {
            LambdaWaiter waiter = awsLambda.waiter();
            UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
                .functionName(functionName)
                .publish(true)
                .s3Bucket(bucketName)
                .s3Key(key)
                .build();

            UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
            GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .build();

            WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
waiter
                .waitUntilFunctionUpdated(getFunctionConfigRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The last modified value is " +
response.lastModified());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

```
   /**
    * Updates the configuration of an AWS Lambda function.
    *
    * @param awsLambda     the {@link LambdaClient} instance to use for the AWS
Lambda operation
    * @param functionName  the name of the AWS Lambda function to update
    * @param handler       the new handler for the AWS Lambda function
    *
    * @throws LambdaException if there is an error while updating the function
configuration
    */
   public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
       try {
           UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .build();

           awsLambda.updateFunctionConfiguration(configurationRequest);

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }

   /**
    * Deletes an AWS Lambda function.
    *
    * @param awsLambda     an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
    * @param functionName  the name of the Lambda function to be deleted
    *
    * @throws LambdaException if an error occurs while deleting the Lambda
function
    */
   public static void deleteLambdaFunction(LambdaClient awsLambda, String
functionName) {
       try {
           DeleteFunctionRequest request = DeleteFunctionRequest.builder()
```

```
                .functionName(functionName)
                .build();

            awsLambda.deleteFunction(request);
            System.out.println("The " + functionName + " function was deleted");

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- 有关 API 的详细信息，请参阅 AWS SDK for Java 2.x API 参考中的以下主题。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

创建 AWS Identity and Access Management (IAM) 角色，该角色授予 Lambda 写入日志的权限。

```
    logger.log(`Creating role (${NAME_ROLE_LAMBDA})...`);
    const response = await createRole(NAME_ROLE_LAMBDA);
```

```
import { AttachRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 * @param {string} roleName
 */
export const attachRolePolicy = (policyArn, roleName) => {
  const command = new AttachRolePolicyCommand({
    PolicyArn: policyArn,
    RoleName: roleName,
  });

  return client.send(command);
};
```

创建 Lambda 函数并上传处理程序代码。

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

调用单参数函数并得出结果。

```
const invoke = async (funcName, payload) => {
```

```
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

更新函数代码并使用环境变量配置其 Lambda 环境。

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};

const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  return client.send(command);
};
```

列出您账户的函数。

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

删除 IAM 角色和 Lambda 函数。

```
import { DeleteRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 */
export const deleteRole = (roleName) => {
  const command = new DeleteRoleCommand({ RoleName: roleName });
  return client.send(command);
};

/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- 有关 API 详细信息，请参阅《AWS SDK for JavaScript API 参考》中的以下主题。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode

- UpdateFunctionConfiguration

Kotlin

适用于 Kotlin 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
suspend fun main(args: Array<String>) {
    val usage = """
        Usage:
            <functionName> <role> <handler> <bucketName> <updatedBucketName>
 <key>

        Where:
            functionName - The name of the AWS Lambda function.
            role - The AWS Identity and Access Management (IAM) service role that
 has AWS Lambda permissions.
            handler - The fully qualified method name (for example,
 example.Handler::handleRequest).
            bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
 name that contains the ZIP or JAR used for the Lambda function's code.
            updatedBucketName - The Amazon S3 bucket name that contains the .zip
 or .jar used to update the Lambda function's code.
            key - The Amazon S3 key name that represents the .zip or .jar file
 (for example, LambdaHello-1.0-SNAPSHOT.jar).
            """

    if (args.size != 6) {
        println(usage)
        exitProcess(1)
    }

    val functionName = args[0]
    val role = args[1]
    val handler = args[2]
    val bucketName = args[3]
```

```
    val updatedBucketName = args[4]
    val key = args[5]

    println("Creating a Lambda function named $functionName.")
    val funArn = createScFunction(functionName, bucketName, key, handler, role)
    println("The AWS Lambda ARN is $funArn")

    // Get a specific Lambda function.
    println("Getting the $functionName AWS Lambda function.")
    getFunction(functionName)

    // List the Lambda functions.
    println("Listing all AWS Lambda functions.")
    listFunctionsSc()

    // Invoke the Lambda function.
    println("*** Invoke the Lambda function.")
    invokeFunctionSc(functionName)

    // Update the AWS Lambda function code.
    println("*** Update the Lambda function code.")
    updateFunctionCode(functionName, updatedBucketName, key)

    // println("*** Invoke the function again after updating the code.")
    invokeFunctionSc(functionName)

    // Update the AWS Lambda function configuration.
    println("Update the run time of the function.")
    updateFunctionConfiguration(functionName, handler)

    // Delete the AWS Lambda function.
    println("Delete the AWS Lambda function.")
    delFunction(functionName)
}

suspend fun createScFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String,
): String {
    val functionCode =
        FunctionCode {
```

```
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java8
        }

    // Create a Lambda function using a waiter
    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn.toString()
    }
}

suspend fun getFunction(functionNameVal: String) {
    val functionRequest =
        GetFunctionRequest {
            functionName = functionNameVal
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val response = awsLambda.getFunction(functionRequest)
        println("The runtime of this Lambda function is
 ${response.configuration?.runtime}")
    }
}

suspend fun listFunctionsSc() {
    val request =
        ListFunctionsRequest {
            maxItems = 10
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
```

```
        val response = awsLambda.listFunctions(request)
        response.functions?.forEach { function ->
            println("The function name is ${function.functionName}")
        }
    }
}

suspend fun invokeFunctionSc(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            payload = byteArray
            logType = LogType.Tail
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("The function payload is
 ${res.payload?.toString(Charsets.UTF_8)}")
    }
}

suspend fun updateFunctionCode(
    functionNameVal: String?,
    bucketName: String?,
    key: String?,
) {
    val functionCodeRequest =
        UpdateFunctionCodeRequest {
            functionName = functionNameVal
            publish = true
            s3Bucket = bucketName
            s3Key = key
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val response = awsLambda.updateFunctionCode(functionCodeRequest)
        awsLambda.waitUntilFunctionUpdated {
            functionName = functionNameVal
        }
        println("The last modified value is " + response.lastModified)
    }
```

```
}

suspend fun updateFunctionConfiguration(
    functionNameVal: String?,
    handlerVal: String?,
) {
    val configurationRequest =
        UpdateFunctionConfigurationRequest {
            functionName = functionNameVal
            handler = handlerVal
            runtime = Runtime.Java11
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.updateFunctionConfiguration(configurationRequest)
    }
}

suspend fun delFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Kotlin API 参考》中的以下主题。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
namespace Lambda;

use Aws\S3\S3Client;
use GuzzleHttp\Psr7\Stream;
use Iam\IAMService;

class GettingStartedWithLambda
{
    public function run()
    {
        echo("\n");
        echo("---------------------------------------\n");
        print("Welcome to the AWS Lambda getting started demo using PHP!\n");
        echo("---------------------------------------\n");

        $clientArgs = [
            'region' => 'us-west-2',
            'version' => 'latest',
            'profile' => 'default',
        ];
        $uniqid = uniqid();

        $iamService = new IAMService();
        $s3client = new S3Client($clientArgs);
        $lambdaService = new LambdaService();

        echo "First, let's create a role to run our Lambda code.\n";
        $roleName = "test-lambda-role-$uniqid";
        $rolePolicyDocument = "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
                {
```

```
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"Service\": \"lambda.amazonaws.com\"
                    },
                    \"Action\": \"sts:AssumeRole\"
                }
            ]
        }";
        $role = $iamService->createRole($roleName, $rolePolicyDocument);
        echo "Created role {$role['RoleName']}.\n";

        $iamService->attachRolePolicy(
            $role['RoleName'],
            "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        );
        echo "Attached the AWSLambdaBasicExecutionRole to {$role['RoleName']}.
\n";

        echo "\nNow let's create an S3 bucket and upload our Lambda code there.
\n";
        $bucketName = "test-example-bucket-$uniqid";
        $s3client->createBucket([
            'Bucket' => $bucketName,
        ]);
        echo "Created bucket $bucketName.\n";

        $functionName = "doc_example_lambda_$uniqid";
        $codeBasic = __DIR__ . "/lambda_handler_basic.zip";
        $handler = "lambda_handler_basic";
        $file = file_get_contents($codeBasic);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "Uploaded the Lambda code.\n";

        $createLambdaFunction = $lambdaService->createFunction($functionName,
 $role, $bucketName, $handler);
        // Wait until the function has finished being created.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['State'] == "Pending");
```

```
        echo "Created Lambda function {$getLambdaFunction['Configuration']
['FunctionName']}.\n";

        sleep(1);

        echo "\nOk, let's invoke that Lambda code.\n";
        $basicParams = [
            'action' => 'increment',
            'number' => 3,
        ];
        /** @var Stream $invokeFunction */
        $invokeFunction = $lambdaService->invoke($functionName, $basicParams)
['Payload'];
        $result = json_decode($invokeFunction->getContents())->result;
        echo "After invoking the Lambda code with the input of
 {$basicParams['number']} we received $result.\n";

        echo "\nSince that's working, let's update the Lambda code.\n";
        $codeCalculator = "lambda_handler_calculator.zip";
        $handlerCalculator = "lambda_handler_calculator";
        echo "First, put the new code into the S3 bucket.\n";
        $file = file_get_contents($codeCalculator);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "New code uploaded.\n";

        $lambdaService->updateFunctionCode($functionName, $bucketName,
 $functionName);
        // Wait for the Lambda code to finish updating.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "New Lambda code uploaded.\n";

        $environment = [
            'Variable' => ['Variables' => ['LOG_LEVEL' => 'DEBUG']],
        ];
        $lambdaService->updateFunctionConfiguration($functionName,
 $handlerCalculator, $environment);
```

```
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "Lambda code updated with new handler and a LOG_LEVEL of DEBUG for
 more information.\n";

        echo "Invoke the new code with some new data.\n";
        $calculatorParams = [
            'action' => 'plus',
            'x' => 5,
            'y' => 4,
        ];
        $invokeFunction = $lambdaService->invoke($functionName,
 $calculatorParams, "Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "Indeed, {$calculatorParams['x']} + {$calculatorParams['y']} does
 equal $result.\n";
        echo "Here's the extra debug info: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nBut what happens if you try to divide by zero?\n";
        $divZeroParams = [
            'action' => 'divide',
            'x' => 5,
            'y' => 0,
        ];
        $invokeFunction = $lambdaService->invoke($functionName, $divZeroParams,
 "Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "You get a |$result| result.\n";
        echo "And an error message: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nHere's all the Lambda functions you have in this Region:\n";
        $listLambdaFunctions = $lambdaService->listFunctions(5);
        $allLambdaFunctions = $listLambdaFunctions['Functions'];
        $next = $listLambdaFunctions->get('NextMarker');
        while ($next != false) {
            $listLambdaFunctions = $lambdaService->listFunctions(5, $next);
            $next = $listLambdaFunctions->get('NextMarker');
            $allLambdaFunctions = array_merge($allLambdaFunctions,
 $listLambdaFunctions['Functions']);
```

```
        }
        foreach ($allLambdaFunctions as $function) {
            echo "{$function['FunctionName']}\n";
        }

        echo "\n\nAnd don't forget to clean up your data!\n";

        $lambdaService->deleteFunction($functionName);
        echo "Deleted Lambda function.\n";
        $iamService->deleteRole($role['RoleName']);
        echo "Deleted Role.\n";
        $deleteObjects = $s3client->listObjectsV2([
            'Bucket' => $bucketName,
        ]);
        $deleteObjects = $s3client->deleteObjects([
            'Bucket' => $bucketName,
            'Delete' => [
                'Objects' => $deleteObjects['Contents'],
            ]
        ]);
        echo "Deleted all objects from the S3 bucket.\n";
        $s3client->deleteBucket(['Bucket' => $bucketName]);
        echo "Deleted the bucket.\n";
    }
}
```

- 有关 API 的详细信息，请参阅 AWS SDK for PHP API 参考中的以下主题。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

定义一个递增数字的 Lambda 处理程序。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

定义执行算术运算的第二个 Lambda 处理程序。

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
```

```
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

创建包装 Lambda 操作的函数。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    @staticmethod
    def create_deployment_package(source_file, destination_file):
        """
        Creates a Lambda deployment package in .zip format in an in-memory
 buffer. This
        buffer can be passed directly to Lambda when creating the function.

        :param source_file: The name of the file that contains the Lambda handler
                            function.
        :param destination_file: The name to give the file when it's deployed to
 Lambda.
        :return: The deployment package.
        """
        buffer = io.BytesIO()
        with zipfile.ZipFile(buffer, "w") as zipped:
            zipped.write(source_file, destination_file)
        buffer.seek(0)
        return buffer.read()

    def get_iam_role(self, iam_role_name):
        """
        Get an AWS Identity and Access Management (IAM) role.
```

```
        :param iam_role_name: The name of the role to retrieve.
        :return: The IAM role.
        """
        role = None
        try:
            temp_role = self.iam_resource.Role(iam_role_name)
            temp_role.load()
            role = temp_role
            logger.info("Got IAM role %s", role.name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "NoSuchEntity":
                logger.info("IAM role %s does not exist.", iam_role_name)
            else:
                logger.error(
                    "Couldn't get IAM role %s. Here's why: %s: %s",
                    iam_role_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return role

    def create_iam_role_for_lambda(self, iam_role_name):
        """
        Creates an IAM role that grants the Lambda function basic permissions. If
a
        role with the specified name already exists, it is used for the demo.

        :param iam_role_name: The name of the role to create.
        :return: The role and a value that indicates whether the role is newly
created.
        """
        role = self.get_iam_role(iam_role_name)
        if role is not None:
            return role, False

        lambda_assume_role_policy = {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {"Service": "lambda.amazonaws.com"},
                    "Action": "sts:AssumeRole",
```

```
                }
            ],
        }
        policy_arn = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

        try:
            role = self.iam_resource.create_role(
                RoleName=iam_role_name,
                AssumeRolePolicyDocument=json.dumps(lambda_assume_role_policy),
            )
            logger.info("Created role %s.", role.name)
            role.attach_policy(PolicyArn=policy_arn)
            logger.info("Attached basic execution policy to role %s.", role.name)
        except ClientError as error:
            if error.response["Error"]["Code"] == "EntityAlreadyExists":
                role = self.iam_resource.Role(iam_role_name)
                logger.warning("The role %s already exists. Using it.",
 iam_role_name)
            else:
                logger.exception(
                    "Couldn't create role %s or attach policy %s.",
                    iam_role_name,
                    policy_arn,
                )
                raise

        return role, True

    def get_function(self, function_name):
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
 self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
```

```
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.9",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
```

```
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise


    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
dict
                                is serialized to JSON before it is sent to
Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
included in
                        the response.
        :return: The response from the function invocation.
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
```

```
        return response


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
```

```
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

创建运行场景的函数。

```
class UpdateFunctionWaiter(CustomWaiter):
    """A custom waiter that waits until a function is successfully updated."""

    def __init__(self, client):
        super().__init__(
```

```
            "UpdateSuccess",
            "GetFunction",
            "Configuration.LastUpdateStatus",
            {"Successful": WaitState.SUCCESS, "Failed": WaitState.FAILURE},
            client,
        )

    def wait(self, function_name):
        self._wait(FunctionName=function_name)


def run_scenario(lambda_client, iam_resource, basic_file, calculator_file,
 lambda_name):
    """
    Runs the scenario.

    :param lambda_client: A Boto3 Lambda client.
    :param iam_resource: A Boto3 IAM resource.
    :param basic_file: The name of the file that contains the basic Lambda
 handler.
    :param calculator_file: The name of the file that contains the calculator
 Lambda handler.
    :param lambda_name: The name to give resources created for the scenario, such
 as the
                        IAM role and the Lambda function.
    """
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the AWS Lambda getting started with functions demo.")
    print("-" * 88)

    wrapper = LambdaWrapper(lambda_client, iam_resource)

    print("Checking for IAM role for Lambda...")
    iam_role, should_wait = wrapper.create_iam_role_for_lambda(lambda_name)
    if should_wait:
        logger.info("Giving AWS time to create resources...")
        wait(10)

    print(f"Looking for function {lambda_name}...")
    function = wrapper.get_function(lambda_name)
    if function is None:
        print("Zipping the Python script into a deployment package...")
```

```
        deployment_package = wrapper.create_deployment_package(
            basic_file, f"{lambda_name}.py"
        )
        print(f"...and creating the {lambda_name} Lambda function.")
        wrapper.create_function(
            lambda_name, f"{lambda_name}.lambda_handler", iam_role,
deployment_package
        )
    else:
        print(f"Function {lambda_name} already exists.")
    print("-" * 88)

    print(f"Let's invoke {lambda_name}. This function increments a number.")
    action_params = {
        "action": "increment",
        "number": q.ask("Give me a number to increment: ", q.is_int),
    }
    print(f"Invoking {lambda_name}...")
    response = wrapper.invoke_function(lambda_name, action_params)
    print(
        f"Incrementing {action_params['number']} resulted in "
        f"{json.load(response['Payload'])}"
    )
    print("-" * 88)

    print(f"Let's update the function to an arithmetic calculator.")
    q.ask("Press Enter when you're ready.")
    print("Creating a new deployment package...")
    deployment_package = wrapper.create_deployment_package(
        calculator_file, f"{lambda_name}.py"
    )
    print(f"...and updating the {lambda_name} Lambda function.")
    update_waiter = UpdateFunctionWaiter(lambda_client)
    wrapper.update_function_code(lambda_name, deployment_package)
    update_waiter.wait(lambda_name)
    print(f"This function uses an environment variable to control logging
level.")
    print(f"Let's set it to DEBUG to get the most logging.")
    wrapper.update_function_configuration(
        lambda_name, {"LOG_LEVEL": logging.getLevelName(logging.DEBUG)}
    )

    actions = ["plus", "minus", "times", "divided-by"]
    want_invoke = True
```

```
    while want_invoke:
        print(f"Let's invoke {lambda_name}. You can invoke these actions:")
        for index, action in enumerate(actions):
            print(f"{index + 1}: {action}")
        action_params = {}
        action_index = q.ask(
            "Enter the number of the action you want to take: ",
            q.is_int,
            q.in_range(1, len(actions)),
        )
        action_params["action"] = actions[action_index - 1]
        print(f"You've chosen to invoke 'x {action_params['action']} y'.")
        action_params["x"] = q.ask("Enter a value for x: ", q.is_int)
        action_params["y"] = q.ask("Enter a value for y: ", q.is_int)
        print(f"Invoking {lambda_name}...")
        response = wrapper.invoke_function(lambda_name, action_params, True)
        print(
            f"Calculating {action_params['x']} {action_params['action']}
{action_params['y']} "
            f"resulted in {json.load(response['Payload'])}"
        )
        q.ask("Press Enter to see the logs from the call.")
        print(base64.b64decode(response["LogResult"]).decode())
        want_invoke = q.ask("That was fun. Shall we do it again? (y/n) ",
q.is_yesno)
    print("-" * 88)

    if q.ask(
        "Do you want to list all of the functions in your account? (y/n) ",
q.is_yesno
    ):
        wrapper.list_functions()
    print("-" * 88)

    if q.ask("Ready to delete the function and role? (y/n) ", q.is_yesno):
        for policy in iam_role.attached_policies.all():
            policy.detach_role(RoleName=iam_role.name)
        iam_role.delete()
        print(f"Deleted role {lambda_name}.")
        wrapper.delete_function(lambda_name)
        print(f"Deleted function {lambda_name}.")

    print("\nThanks for watching!")
    print("-" * 88)
```

```
if __name__ == "__main__":
    try:
        run_scenario(
            boto3.client("lambda"),
            boto3.resource("iam"),
            "lambda_handler_basic.py",
            "lambda_handler_calculator.py",
            "doc_example_lambda_calculator",
        )
    except Exception:
        logging.exception("Something went wrong with the demo!")
```

- 有关 API 详细信息，请参阅《AWS SDK for Python (Boto3) API 参考》中的以下主题。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

为能够写入日志的 Lambda 函数设置必备的 IAM 权限。

```
# Get an AWS Identity and Access Management (IAM) role.
#
# @param iam_role_name: The name of the role to retrieve.
```

```
  # @param action: Whether to create or destroy the IAM apparatus.
  # @return: The IAM role.
  def manage_iam(iam_role_name, action)
    case action
    when 'create'
      create_iam_role(iam_role_name)
    when 'destroy'
      destroy_iam_role(iam_role_name)
    else
      raise "Incorrect action provided. Must provide 'create' or 'destroy'"
    end
  end

  private

  def create_iam_role(iam_role_name)
    role_policy = {
      'Version': '2012-10-17',
      'Statement': [
        {
          'Effect': 'Allow',
          'Principal': { 'Service': 'lambda.amazonaws.com' },
          'Action': 'sts:AssumeRole'
        }
      ]
    }
    role = @iam_client.create_role(
      role_name: iam_role_name,
      assume_role_policy_document: role_policy.to_json
    )
    @iam_client.attach_role_policy(
      {
        policy_arn: 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole',
        role_name: iam_role_name
      }
    )
    wait_for_role_to_exist(iam_role_name)
    @logger.debug("Successfully created IAM role: #{role['role']['arn']}")
    sleep(10)
    [role, role_policy.to_json]
  end

  def destroy_iam_role(iam_role_name)
```

```
    @iam_client.detach_role_policy(
      {
        policy_arn: 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole',
        role_name: iam_role_name
      }
    )
    @iam_client.delete_role(role_name: iam_role_name)
    @logger.debug("Detached policy & deleted IAM role: #{iam_role_name}")
  end

  def wait_for_role_to_exist(iam_role_name)
    @iam_client.wait_until(:role_exists, { role_name: iam_role_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  end
```

定义一个 Lambda 处理程序，用以递增作为调用参数提供的数字。

```
require 'logger'

# A function that increments a whole number by one (1) and logs the result.
# Requires a manually-provided runtime parameter, 'number', which must be Int
#
# @param event [Hash] Parameters sent when the function is invoked
# @param context [Hash] Methods and properties that provide information
# about the invocation, function, and execution environment.
# @return incremented_number [String] The incremented number.
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  log_level = ENV['LOG_LEVEL']
  logger.level = case log_level
                 when 'debug'
                   Logger::DEBUG
                 when 'info'
                   Logger::INFO
                 else
                   Logger::ERROR
                 end
  logger.debug('This is a debug log message.')
  logger.info('This is an info log message. Code executed successfully!')
```

```
  number = event['number'].to_i
  incremented_number = number + 1
  logger.info("You provided #{number.round} and it was incremented to
 #{incremented_number.round}")
  incremented_number.round.to_s
end
```

将 Lambda 函数压缩到部署程序包中。

```
  # Creates a Lambda deployment package in .zip format.
  #
  # @param source_file: The name of the object, without suffix, for the Lambda
 file and zip.
  # @return: The deployment package.
  def create_deployment_package(source_file)
    Dir.chdir(File.dirname(__FILE__))
    if File.exist?('lambda_function.zip')
      File.delete('lambda_function.zip')
      @logger.debug('Deleting old zip: lambda_function.zip')
    end
    Zip::File.open('lambda_function.zip', create: true) do |zipfile|
      zipfile.add('lambda_function.rb', "#{source_file}.rb")
    end
    @logger.debug("Zipping #{source_file}.rb into: lambda_function.zip.")
    File.read('lambda_function.zip').to_s
  rescue StandardError => e
    @logger.error("There was an error creating deployment package:\n
 #{e.message}")
  end
```

新建 Lambda 函数。

```
  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
 code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
```

```
    response = @lambda_client.create_function({
                                                role: role_arn.to_s,
                                                function_name: function_name,
                                                handler: handler_name,
                                                runtime: 'ruby2.7',
                                                code: {
                                                  zip_file: deployment_package
                                                },
                                                environment: {
                                                  variables: {
                                                    'LOG_LEVEL' => 'info'
                                                  }
                                                }
                                              })
    @lambda_client.wait_until(:function_active_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
    response
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating #{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

使用可选的运行时参数调用 Lambda 函数。

```
  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name }
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
#{e.message}")
  end
```

更新 Lambda 函数的配置，以注入新的环境变量。

```
 # Updates the environment variables for a Lambda function.
 # @param function_name: The name of the function to update.
 # @param log_level: The log level of the function.
 # @return: Data about the update, including the status.
 def update_function_configuration(function_name, log_level)
   @lambda_client.update_function_configuration({
                                                  function_name: function_name,
                                                  environment: {
                                                    variables: {
                                                      'LOG_LEVEL' => log_level
                                                    }
                                                  }
                                                })
   @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
 end
```

使用包含不同代码的不同部署程序包更新 Lambda 函数的代码。

```
 # Updates the code for a Lambda function by submitting a .zip archive that
contains
 # the code for the function.
 #
 # @param function_name: The name of the function to update.
 # @param deployment_package: The function code to update, packaged as bytes in
 #                            .zip format.
 # @return: Data about the update, including the status.
 def update_function_code(function_name, deployment_package)
   @lambda_client.update_function_code(
     function_name: function_name,
     zip_file: deployment_package
```

```
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

使用内置分页工具列出所有现有的 Lambda 函数。

```
# Lists the Lambda functions for the current account.
def list_functions
  functions = []
  @lambda_client.list_functions.each do |response|
    response['functions'].each do |function|
      functions.append(function['function_name'])
    end
  end
  functions
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error listing functions:\n #{e.message}")
end
```

删除特定的 Lambda 函数。

```
# Deletes a Lambda function.
# @param function_name: The name of the function to delete.
def delete_function(function_name)
  print "Deleting function: #{function_name}..."
  @lambda_client.delete_function(
    function_name: function_name
  )
  print 'Done!'.green
rescue Aws::Lambda::Errors::ServiceException => e
```

```
    @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
  end
```

- 有关 API 详细信息，请参阅《AWS SDK for Ruby API 参考》中的以下主题。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

在这种情况下使用的带依赖项的 Cargo.toml。

```
[package]
name = "lambda-code-examples"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
reference/manifest.html

[dependencies]
aws-config = { version = "1.0.1", features = ["behavior-version-latest"] }
aws-sdk-ec2 = { version = "1.3.0" }
aws-sdk-iam = { version = "1.3.0" }
aws-sdk-lambda = { version = "1.3.0" }
aws-sdk-s3 = { version = "1.4.0" }
```

```
aws-smithy-types = { version = "1.0.1" }
aws-types = { version = "1.0.1" }
clap = { version = "~4.4", features = ["derive"] }
tokio = { version = "1.20.1", features = ["full"] }
tracing-subscriber = { version = "0.3.15", features = ["env-filter"] }
tracing = "0.1.37"
serde_json = "1.0.94"
anyhow = "1.0.71"
uuid = { version = "1.3.3", features = ["v4"] }
lambda_runtime = "0.8.0"
serde = "1.0.164"
```

可就这种情况简化 Lambda 调用的一组实用程序。此文件在 crate 中是 src/ations.rs。

```
use anyhow::anyhow;
use aws_sdk_iam::operation::{create_role::CreateRoleError,
 delete_role::DeleteRoleOutput};
use aws_sdk_lambda::{
    operation::{
        delete_function::DeleteFunctionOutput, get_function::GetFunctionOutput,
        invoke::InvokeOutput, list_functions::ListFunctionsOutput,
        update_function_code::UpdateFunctionCodeOutput,
        update_function_configuration::UpdateFunctionConfigurationOutput,
    },
    primitives::ByteStream,
    types::{Environment, FunctionCode, LastUpdateStatus, State},
};
use aws_sdk_s3::{
    error::ErrorMetadata,
    operation::{delete_bucket::DeleteBucketOutput,
 delete_object::DeleteObjectOutput},
    types::CreateBucketConfiguration,
};
use aws_smithy_types::Blob;
use serde::{ser::SerializeMap, Serialize};
use std::{fmt::Display, path::PathBuf, str::FromStr, time::Duration};
use tracing::{debug, info, warn};

/* Operation describes  */
#[derive(Clone, Copy, Debug, Serialize)]
pub enum Operation {
```

```
    #[serde(rename = "plus")]
    Plus,
    #[serde(rename = "minus")]
    Minus,
    #[serde(rename = "times")]
    Times,
    #[serde(rename = "divided-by")]
    DividedBy,
}

impl FromStr for Operation {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        match s {
            "plus" => Ok(Operation::Plus),
            "minus" => Ok(Operation::Minus),
            "times" => Ok(Operation::Times),
            "divided-by" => Ok(Operation::DividedBy),
            _ => Err(anyhow!("Unknown operation {s}")),
        }
    }
}

impl Display for Operation {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Operation::Plus => write!(f, "plus"),
            Operation::Minus => write!(f, "minus"),
            Operation::Times => write!(f, "times"),
            Operation::DividedBy => write!(f, "divided-by"),
        }
    }
}

/**
 * InvokeArgs will be serialized as JSON and sent to the AWS Lambda handler.
 */
#[derive(Debug)]
pub enum InvokeArgs {
    Increment(i32),
    Arithmetic(Operation, i32, i32),
}
```

```
impl Serialize for InvokeArgs {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        match self {
            InvokeArgs::Increment(i) => serializer.serialize_i32(*i),
            InvokeArgs::Arithmetic(o, i, j) => {
                let mut map: S::SerializeMap =
 serializer.serialize_map(Some(3))?;
                map.serialize_key(&"op".to_string())?;
                map.serialize_value(&o.to_string())?;
                map.serialize_key(&"i".to_string())?;
                map.serialize_value(&i)?;
                map.serialize_key(&"j".to_string())?;
                map.serialize_value(&j)?;
                map.end()
            }
        }
    }
}

/** A policy document allowing Lambda to execute this function on the account's
 behalf. */
const ROLE_POLICY_DOCUMENT: &str = r#"{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": { "Service": "lambda.amazonaws.com" },
            "Action": "sts:AssumeRole"
        }
    ]
}"#;

/**
 * A LambdaManager gathers all the resources necessary to run the Lambda example
 scenario.
 * This includes instantiated aws_sdk clients and details of resource names.
 */
pub struct LambdaManager {
    iam_client: aws_sdk_iam::Client,
    lambda_client: aws_sdk_lambda::Client,
    s3_client: aws_sdk_s3::Client,
```

```
    lambda_name: String,
    role_name: String,
    bucket: String,
    own_bucket: bool,
}

// These unit type structs provide nominal typing on top of String parameters for
 LambdaManager::new
pub struct LambdaName(pub String);
pub struct RoleName(pub String);
pub struct Bucket(pub String);
pub struct OwnBucket(pub bool);

impl LambdaManager {
    pub fn new(
        iam_client: aws_sdk_iam::Client,
        lambda_client: aws_sdk_lambda::Client,
        s3_client: aws_sdk_s3::Client,
        lambda_name: LambdaName,
        role_name: RoleName,
        bucket: Bucket,
        own_bucket: OwnBucket,
    ) -> Self {
        Self {
            iam_client,
            lambda_client,
            s3_client,
            lambda_name: lambda_name.0,
            role_name: role_name.0,
            bucket: bucket.0,
            own_bucket: own_bucket.0,
        }
    }

    /**
     * Load the AWS configuration from the environment.
     * Look up lambda_name and bucket if none are given, or generate a random
 name if not present in the environment.
     * If the bucket name is provided, the caller needs to have created the
 bucket.
     * If the bucket name is generated, it will be created.
     */
    pub async fn load_from_env(lambda_name: Option<String>, bucket:
 Option<String>) -> Self {
```

```
        let sdk_config = aws_config::load_from_env().await;
        let lambda_name = LambdaName(lambda_name.unwrap_or_else(|| {
            std::env::var("LAMBDA_NAME").unwrap_or_else(|_|
"rust_lambda_example".to_string())
        }));
        let role_name = RoleName(format!("{}_role", lambda_name.0));
        let (bucket, own_bucket) =
            match bucket {
                Some(bucket) => (Bucket(bucket), false),
                None => (
                    Bucket(std::env::var("LAMBDA_BUCKET").unwrap_or_else(|_| {
                        format!("rust-lambda-example-{}", uuid::Uuid::new_v4())
                    })),
                    true,
                ),
            };

        let s3_client = aws_sdk_s3::Client::new(&sdk_config);

        if own_bucket {
            info!("Creating bucket for demo: {}", bucket.0);
            s3_client
                .create_bucket()
                .bucket(bucket.0.clone())
                .create_bucket_configuration(
                    CreateBucketConfiguration::builder()

.location_constraint(aws_sdk_s3::types::BucketLocationConstraint::from(
                            sdk_config.region().unwrap().as_ref(),
                        ))
                        .build(),
                )
                .send()
                .await
                .unwrap();
        }

        Self::new(
            aws_sdk_iam::Client::new(&sdk_config),
            aws_sdk_lambda::Client::new(&sdk_config),
            s3_client,
            lambda_name,
            role_name,
            bucket,
```

```
            OwnBucket(own_bucket),
        )
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }

    /**
     * Create a function, uploading from a zip file.
     */
    pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
 anyhow::Error> {
        let code = self.prepare_function(zip_file, None).await?;

        let key = code.s3_key().unwrap().to_string();
```

```
        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Create an IAM execution role for the managed Lambda function.
     * If the role already exists, use that instead.
     */
    async fn create_role(&self) -> Result<aws_sdk_iam::types::Role,
CreateRoleError> {
        info!("Creating execution role for function");
        let get_role = self
            .iam_client
            .get_role()
            .role_name(self.role_name.clone())
            .send()
            .await;
        if let Ok(get_role) = get_role {
            if let Some(role) = get_role.role {
```

```
                return Ok(role);
            }
        }

        let create_role = self
            .iam_client
            .create_role()
            .role_name(self.role_name.clone())
            .assume_role_policy_document(ROLE_POLICY_DOCUMENT)
            .send()
            .await;

        match create_role {
            Ok(create_role) => match create_role.role {
                Some(role) => Ok(role),
                None => Err(CreateRoleError::generic(
                    ErrorMetadata::builder()
                        .message("CreateRole returned empty success")
                        .build(),
                )),
            },
            Err(err) => Err(err.into_service_error()),
        }
    }

    /**
     * Poll `is_function_ready` with a 1-second delay. It returns when the
function is ready or when there's an error checking the function's state.
     */
    pub async fn wait_for_function_ready(&self) -> Result<(), anyhow::Error> {
        info!("Waiting for function");
        while !self.is_function_ready(None).await? {
            info!("Function is not ready, sleeping 1s");
            tokio::time::sleep(Duration::from_secs(1)).await;
        }
        Ok(())
    }

    /**
     * Check if a Lambda function is ready to be invoked.
     * A Lambda function is ready for this scenario when its state is active and
its LastUpdateStatus is Successful.
     * Additionally, if a sha256 is provided, the function must have that as its
current code hash.
```

```
     * Any missing properties or failed requests will be reported as an Err.
     */
    async fn is_function_ready(
        &self,
        expected_code_sha256: Option<&str>,
    ) -> Result<bool, anyhow::Error> {
        match self.get_function().await {
            Ok(func) => {
                if let Some(config) = func.configuration() {
                    if let Some(state) = config.state() {
                        info!(?state, "Checking if function is active");
                        if !matches!(state, State::Active) {
                            return Ok(false);
                        }
                    }
                    match config.last_update_status() {
                        Some(last_update_status) => {
                            info!(?last_update_status, "Checking if function is
ready");
                            match last_update_status {
                                LastUpdateStatus::Successful => {
                                    // continue
                                }
                                LastUpdateStatus::Failed |
LastUpdateStatus::InProgress => {
                                    return Ok(false);
                                }
                                unknown => {
                                    warn!(
                                        status_variant = unknown.as_str(),
                                        "LastUpdateStatus unknown"
                                    );
                                    return Err(anyhow!(
                                        "Unknown LastUpdateStatus, fn config is
{config:?}"
                                    ));
                                }
                            }
                        }
                        None => {
                            warn!("Missing last update status");
                            return Ok(false);
                        }
                    };
```

```
                    if expected_code_sha256.is_none() {
                        return Ok(true);
                    }
                    if let Some(code_sha256) = config.code_sha256() {
                        return Ok(code_sha256 ==
expected_code_sha256.unwrap_or_default());
                    }
                }
            }
            Err(e) => {
                warn!(?e, "Could not get function while waiting");
            }
        }
        Ok(false)
    }

    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
```

```
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }

    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
```

```
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(updated)
    }

    /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
    pub async fn delete_function(
        &self,
        location: Option<String>,
    ) -> (
        Result<DeleteFunctionOutput, anyhow::Error>,
        Result<DeleteRoleOutput, anyhow::Error>,
        Option<Result<DeleteObjectOutput, anyhow::Error>>,
    ) {
        info!("Deleting lambda function {}", self.lambda_name);
        let delete_function = self
            .lambda_client
            .delete_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        info!("Deleting iam role {}", self.role_name);
        let delete_role = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
            if let Some(location) = location {
```

```
                info!("Deleting object {location}");
                Some(
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }

    pub async fn cleanup(
        &self,
        location: Option<String>,
    ) -> (
        (
            Result<DeleteFunctionOutput, anyhow::Error>,
            Result<DeleteRoleOutput, anyhow::Error>,
            Option<Result<DeleteObjectOutput, anyhow::Error>>,
        ),
        Option<Result<DeleteBucketOutput, anyhow::Error>>,
    ) {
        let delete_function = self.delete_function(location).await;

        let delete_bucket = if self.own_bucket {
            info!("Deleting bucket {}", self.bucket);
            if delete_function.2.is_none() ||
delete_function.2.as_ref().unwrap().is_ok() {
                Some(
                    self.s3_client
                        .delete_bucket()
                        .bucket(self.bucket.clone())
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
```

```
                None
            }
        } else {
            info!("No bucket to clean up");
            None
        };

        (delete_function, delete_bucket)
    }
}

/**
 * Testing occurs primarily as an integration test running the `scenario` bin
 successfully.
 * Each action relies deeply on the internal workings and state of Amazon Simple
 Storage Service (Amazon S3), Lambda, and IAM working together.
 * It is therefore infeasible to mock the clients to test the individual actions.
 */
#[cfg(test)]
mod test {
    use super::{InvokeArgs, Operation};
    use serde_json::json;

    /** Make sure that the JSON output of serializing InvokeArgs is what's
 expected by the calculator. */
    #[test]
    fn test_serialize() {
        assert_eq!(json!(InvokeArgs::Increment(5)), 5);
        assert_eq!(
            json!(InvokeArgs::Arithmetic(Operation::Plus, 5, 7)).to_string(),
            r#"{"op":"plus","i":5,"j":7}"#.to_string(),
        );
    }
}
```

一个从头到尾运行场景的二进制文件，使用命令行标志来控制某些行为。此文件在 crate 中是 src/bin/scenario.rs。

```
/*
## Service actions
```

```
Service actions wrap the SDK call, taking a client and any specific parameters
 necessary for the call.

* CreateFunction
* GetFunction
* ListFunctions
* Invoke
* UpdateFunctionCode
* UpdateFunctionConfiguration
* DeleteFunction

## Scenario
A scenario runs at a command prompt and prints output to the user on the result
 of each service action. A scenario can run in one of two ways: straight through,
 printing out progress as it goes, or as an interactive question/answer script.

## Getting started with functions

Use an SDK to manage AWS Lambda functions: create a function, invoke it, update
 its code, invoke it again, view its output and logs, and delete it.

This scenario uses two Lambda handlers:
_Note: Handlers don't use AWS SDK API calls._

The increment handler is straightforward:

1. It accepts a number, increments it, and returns the new value.
2. It performs simple logging of the result.

The arithmetic handler is more complex:
1. It accepts a set of actions ['plus', 'minus', 'times', 'divided-by'] and two
 numbers, and returns the result of the calculation.
2. It uses an environment variable to control log level (such as DEBUG, INFO,
 WARNING, ERROR).
It logs a few things at different levels, such as:
    * DEBUG: Full event data.
    * INFO: The calculation result.
    * WARN~ING~: When a divide by zero error occurs.
    * This will be the typical `RUST_LOG` variable.


The steps of the scenario are:
```

```
1. Create an AWS Identity and Access Management (IAM) role that meets the
 following requirements:
    * Has an assume_role policy that grants 'lambda.amazonaws.com' the
 'sts:AssumeRole' action.
    * Attaches the 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' managed role.
    * _You must wait for ~10 seconds after the role is created before you can use
 it!_
2. Create a function (CreateFunction) for the increment handler by packaging it
 as a zip and doing one of the following:
    * Adding it with CreateFunction Code.ZipFile.
    * --or--
    * Uploading it to Amazon Simple Storage Service (Amazon S3) and adding it
 with CreateFunction Code.S3Bucket/S3Key.
    * _Note: Zipping the file does not have to be done in code._
    * If you have a waiter, use it to wait until the function is active.
 Otherwise, call GetFunction until State is Active.
3. Invoke the function with a number and print the result.
4. Update the function (UpdateFunctionCode) to the arithmetic handler by
 packaging it as a zip and doing one of the following:
    * Adding it with UpdateFunctionCode ZipFile.
    * --or--
    * Uploading it to Amazon S3 and adding it with UpdateFunctionCode S3Bucket/
S3Key.
5. Call GetFunction until Configuration.LastUpdateStatus is 'Successful' (or
 'Failed').
6. Update the environment variable by calling UpdateFunctionConfiguration and
 pass it a log level, such as:
    * Environment={'Variables': {'RUST_LOG': 'TRACE'}}
7. Invoke the function with an action from the list and a couple of values.
 Include LogType='Tail' to get logs in the result. Print the result of the
 calculation and the log.
8. [Optional] Invoke the function to provoke a divide-by-zero error and show the
 log result.
9. List all functions for the account, using pagination (ListFunctions).
10. Delete the function (DeleteFunction).
11. Delete the role.

Each step should use the function created in Service Actions to abstract calling
 the SDK.
 */

use aws_sdk_lambda::{operation::invoke::InvokeOutput, types::Environment};
use clap::Parser;
```

```
use std::{collections::HashMap, path::PathBuf};
use tracing::{debug, info, warn};
use tracing_subscriber::EnvFilter;

use lambda_code_examples::actions::{
    InvokeArgs::{Arithmetic, Increment},
    LambdaManager, Operation,
};

#[derive(Debug, Parser)]
pub struct Opt {
    /// The AWS Region.
    #[structopt(short, long)]
    pub region: Option<String>,

    // The bucket to use for the FunctionCode.
    #[structopt(short, long)]
    pub bucket: Option<String>,

    // The name of the Lambda function.
    #[structopt(short, long)]
    pub lambda_name: Option<String>,

    // The number to increment.
    #[structopt(short, long, default_value = "12")]
    pub inc: i32,

    // The left operand.
    #[structopt(long, default_value = "19")]
    pub num_a: i32,

    // The right operand.
    #[structopt(long, default_value = "23")]
    pub num_b: i32,

    // The arithmetic operation.
    #[structopt(short, long, default_value = "plus")]
    pub operation: Operation,

    #[structopt(long)]
    pub cleanup: Option<bool>,

    #[structopt(long)]
    pub no_cleanup: Option<bool>,
```

```
}

fn code_path(lambda: &str) -> PathBuf {
    PathBuf::from(format!("../target/lambda/{lambda}/bootstrap.zip"))
}

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}

async fn main_block(
    opt: &Opt,
    manager: &LambdaManager,
    code_location: String,
) -> Result<(), anyhow::Error> {
    let invoke = manager.invoke(Increment(opt.inc)).await?;
    log_invoke_output(&invoke, "Invoked function configured as increment");

    let update_code = manager
        .update_function_code(code_path("arithmetic"), code_location.clone())
        .await?;

    let code_sha256 = update_code.code_sha256().unwrap_or("Unknown SHA");
    info!(?code_sha256, "Updated function code with arithmetic.zip");

    let arithmetic_args = Arithmetic(opt.operation, opt.num_a, opt.num_b);
    let invoke = manager.invoke(arithmetic_args).await?;
    log_invoke_output(&invoke, "Invoked function configured as arithmetic");

    let update = manager
        .update_function_configuration(
            Environment::builder()
                .set_variables(Some(HashMap::from([(
                    "RUST_LOG".to_string(),
```

```
                    "trace".to_string(),
                )]))
                .build(),
        )
        .await?;
    let updated_environment = update.environment();
    info!(?updated_environment, "Updated function configuration");

    let invoke = manager
        .invoke(Arithmetic(opt.operation, opt.num_a, opt.num_b))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with increased logging",
    );

    let invoke = manager
        .invoke(Arithmetic(Operation::DividedBy, opt.num_a, 0))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with divide by zero",
    );

    Ok::<(), anyhow::Error>(())
}

#[tokio::main]
async fn main() {
    tracing_subscriber::fmt()
        .without_time()
        .with_file(true)
        .with_line_number(true)
        .with_env_filter(EnvFilter::from_default_env())
        .init();

    let opt = Opt::parse();
    let manager = LambdaManager::load_from_env(opt.lambda_name.clone(),
 opt.bucket.clone()).await;

    let key = match manager.create_function(code_path("increment")).await {
        Ok(init) => {
            info!(?init, "Created function, initially with increment.zip");
            let run_block = main_block(&opt, &manager, init.clone()).await;
```

```
            info!(?run_block, "Finished running example, cleaning up");
            Some(init)
        }
        Err(err) => {
            warn!(?err, "Error happened when initializing function");
            None
        }
    };

    if Some(false) == opt.cleanup || Some(true) == opt.no_cleanup {
        info!("Skipping cleanup")
    } else {
        let delete = manager.cleanup(key).await;
        info!(?delete, "Deleted function & cleaned up resources");
    }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Rust API 参考》中的以下主题。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    TRY.
        "Create an AWS Identity and Access Management (IAM) role that grants AWS
 Lambda permission to write to logs."
        DATA(lv_policy_document) = `{` &&
            `"Version":"2012-10-17",` &&
                  `"Statement": [` &&
                    `{` &&
                      `"Effect": "Allow",` &&
                      `"Action": [` &&
                        `"sts:AssumeRole"` &&
                      `],` &&
                      `"Principal": {` &&
                        `"Service": [` &&
                          `"lambda.amazonaws.com"` &&
                        `]` &&
                      `}` &&
                    `}` &&
                  `]` &&
                `}`.
        TRY.
            DATA(lo_create_role_output) =  lo_iam->createrole(
                    iv_rolename = iv_role_name
                    iv_assumerolepolicydocument = lv_policy_document
                    iv_description = 'Grant lambda permission to write to logs'
                ).
            MESSAGE 'IAM role created.' TYPE 'I'.
            WAIT UP TO 10 SECONDS.            " Make sure that the IAM role is
 ready for use. "
          CATCH /aws1/cx_iamentityalrdyexex.
            MESSAGE 'IAM role already exists.' TYPE 'E'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iammalformedplydocex.
            MESSAGE 'Policy document in the request is malformed.' TYPE 'E'.
        ENDTRY.

        TRY.
            lo_iam->attachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole'
            ).
            MESSAGE 'Attached policy to the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
```

```
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Create a Lambda function and upload handler code. "
        " Lambda function performs 'increment' action on a number. "
        TRY.
            lo_lmd->createfunction(
                 iv_functionname = iv_function_name
                 iv_runtime = `python3.9`
                 iv_role = lo_create_role_output->get_role( )->get_arn( )
                 iv_handler = iv_handler
                 io_code = io_initial_zip_file
                 iv_description = 'AWS Lambda code example'
             ).
            MESSAGE 'Lambda function created.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        " Verify the function is in Active state "
        WHILE lo_lmd->getfunction( iv_functionname = iv_function_name )-
>get_configuration( )->ask_state( ) <> 'Active'.
          IF sy-index = 10.
            EXIT.               " Maximum 10 seconds. "
          ENDIF.
          WAIT UP TO 1 SECONDS.
        ENDWHILE.

        "Invoke the function with a single parameter and get results."
        TRY.
            DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
```

```
                `"action": "increment",`  &&
                `"number": 10` &&
              `}`
            ).
            DATA(lo_initial_invoke_output) =  lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json
                  ).
            ov_initial_invoke_payload = lo_initial_invoke_output->get_payload( ).
           " ov_initial_invoke_payload is returned for testing purposes. "
            DATA(lo_writer_json) = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_initial_invoke_payload RESULT
 XML lo_writer_json.
            DATA(lv_result) = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " Update the function code and configure its Lambda environment with an
 environment variable. "
        " Lambda function is updated to perform 'decrement' action also. "
        TRY.
            lo_lmd->updatefunctioncode(
                  iv_functionname = iv_function_name
                  iv_zipfile = io_updated_zip_file
              ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function code updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
```

```
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        TRY.
            DATA lt_variables TYPE /aws1/
cl_lmdenvironmentvaria00=>tt_environmentvariables.
            DATA ls_variable LIKE LINE OF lt_variables.
            ls_variable-key = 'LOG_LEVEL'.
            ls_variable-value = NEW /aws1/cl_lmdenvironmentvaria00( iv_value =
 'info' ).
            INSERT ls_variable INTO TABLE lt_variables.

            lo_lmd->updatefunctionconfiguration(
                  iv_functionname = iv_function_name
                  io_environment = NEW /aws1/cl_lmdenvironment( it_variables =
 lt_variables )
              ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourceconflictex.
            MESSAGE 'Resource already exists or another operation is in
 progress.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        "Invoke the function with new parameters and get results. Display the
 execution log that's returned from the invocation."
        TRY.
            lv_json = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "decrement",`  &&
                `"number": 10` &&
              `}`
            ).
            DATA(lo_updated_invoke_output) =  lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json
                  ).
            ov_updated_invoke_payload = lo_updated_invoke_output->get_payload( ).
           " ov_updated_invoke_payload is returned for testing purposes. "
```

```
            lo_writer_json = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_updated_invoke_payload RESULT
 XML lo_writer_json.
            lv_result = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " List the functions for your account. "
        TRY.
            DATA(lo_list_output) = lo_lmd->listfunctions( ).
            DATA(lt_functions) = lo_list_output->get_functions( ).
            MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
        ENDTRY.

        " Delete the Lambda function. "
        TRY.
            lo_lmd->deletefunction( iv_functionname = iv_function_name ).
            MESSAGE 'Lambda function deleted.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        " Detach role policy. "
        TRY.
            lo_iam->detachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole'
            ).
```

```
            MESSAGE 'Detached policy from the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Delete the IAM role. "
        TRY.
            lo_iam->deleterole( iv_rolename = iv_role_name ).
            MESSAGE 'IAM role deleted.' TYPE 'I'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

      CATCH /aws1/cx_rt_service_generic INTO lo_exception.
        DATA(lv_error) = lo_exception->get_longtext( ).
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- 有关 API 详细信息，请参阅适用于 SAP ABAP 的 AWS SDK 的 API 参考中的以下主题。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

使用 AWS SDK 对 Lambda 执行的操作

以下代码示例演示了如何使用 AWS SDK 来执行各个 Lambda 操作。每个示例都包含一个指向 GitHub 的链接，您可以在其中找到有关设置和运行代码的说明。

这些代码节选调用了 Lambda API，是必须在上下文中运行的大型程序的代码节选。您可以在适用于使用 AWS SDK 的 Lambda 的场景 中结合上下文查看操作。

以下示例仅包括最常用的操作。有关完整列表，请参阅 AWS Lambda API 参考。

示例

- 将 CreateAlias 与 CLI 配合使用
- 将 CreateFunction 与 AWS SDK 或 CLI 配合使用
- 将 DeleteAlias 与 CLI 配合使用
- 将 DeleteFunction 与 AWS SDK 或 CLI 配合使用
- 将 DeleteFunctionConcurrency 与 CLI 配合使用
- 将 DeleteProvisionedConcurrencyConfig 与 CLI 配合使用
- 将 GetAccountSettings 与 CLI 配合使用
- 将 GetAlias 与 CLI 配合使用
- 将 GetFunction 与 AWS SDK 或 CLI 配合使用
- 将 GetFunctionConcurrency 与 CLI 配合使用
- 将 GetFunctionConfiguration 与 CLI 配合使用
- 将 GetPolicy 与 CLI 配合使用
- 将 GetProvisionedConcurrencyConfig 与 CLI 配合使用
- 将 Invoke 与 AWS SDK 或 CLI 配合使用
- 将 ListFunctions 与 AWS SDK 或 CLI 配合使用
- 将 ListProvisionedConcurrencyConfigs 与 CLI 配合使用
- 将 ListTags 与 CLI 配合使用
- 将 ListVersionsByFunction 与 CLI 配合使用
- 将 PublishVersion 与 CLI 配合使用
- 将 PutFunctionConcurrency 与 CLI 配合使用

- 将 PutProvisionedConcurrencyConfig 与 CLI 配合使用
- 将 RemovePermission 与 CLI 配合使用
- 将 TagResource 与 CLI 配合使用
- 将 UntagResource 与 CLI 配合使用
- 将 UpdateAlias 与 CLI 配合使用
- 将 UpdateFunctionCode 与 AWS SDK 或 CLI 配合使用
- 将 UpdateFunctionConfiguration 与 AWS SDK 或 CLI 配合使用

将 **CreateAlias** 与 CLI 配合使用

以下代码示例演示如何使用 CreateAlias。

CLI

AWS CLI

为 Lambda 函数创建别名

以下 create-alias 示例会创建名为 LIVE 的别名，该别名指向 my-function Lambda 函数的版本 1。

```
aws lambda create-alias \
    --function-name my-function \
    --description "alias for live version of function" \
    --function-version 1 \
    --name LIVE
```

输出：

```
{
    "FunctionVersion": "1",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "Description": "alias for live version of function"
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数别名。

- 有关 API 详细信息，请参阅《AWS CLI 命令参考》中的 CreateAlias。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例为指定版本和路由配置创建了新的 Lambda 别名，可指定相应版本收到的调用请求的百分比。

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- 有关 API 的详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 CreateAlias。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **CreateFunction** 与 AWS SDK 或 CLI 配合使用

以下代码示例演示如何使用 `CreateFunction`。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- 了解基础知识

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /// <summary>
```

```
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
    /// bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the
    /// appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created
    /// Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
        {
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
_lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for .NET API 参考》中的 CreateFunction。

C++

SDK for C++

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_9);
#endif
```

```
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
 seconds
                      << " seconds elapsed." << std::endl;
            break;
        }

        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
```

```
            return false;
        }
```

- 有关 API 详细信息，请参阅《AWS SDK for C++ API 参考》中的 CreateFunction。

CLI

AWS CLI

创建 Lambda 函数

以下 `create-function` 示例创建一个名为 `my-function` 的 Lambda 函数。

```
aws lambda create-function \
    --function-name my-function \
    --runtime nodejs18.x \
    --zip-file fileb://my-function.zip \
    --handler my-function.handler \
    --role arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
tges6bf4
```

`my-function.zip` 的内容：

```
This file is a deployment package that contains your function code and any
 dependencies.
```

输出：

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "PFn4S+er27qk+UuZSTKEQfNKG/XNn7QJs90mJgq6oH8=",
    "FunctionName": "my-function",
    "CodeSize": 308,
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
```

```
    "Timeout": 3,
    "LastModified": "2023-10-14T22:26:11.234+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs18.x",
    "Description": ""
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数选项。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 CreateFunction。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(ctx context.Context, functionName
 string, handlerName string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
```

```
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(ctx, &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython39,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Go API 参考》中的 CreateFunction。

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /**
     * Creates a new Lambda function in AWS using the AWS Lambda Java API.
     *
     * @param awsLambda    the AWS Lambda client used to interact with the AWS
Lambda service
     * @param functionName the name of the Lambda function to create
     * @param key          the S3 key of the function code
     * @param bucketName   the name of the S3 bucket containing the function code
     * @param role         the IAM role to assign to the Lambda function
     * @param handler      the fully qualified class name of the function handler
     * @return the Amazon Resource Name (ARN) of the created Lambda function
     */
    public static String createLambdaFunction(LambdaClient awsLambda,
                                               String functionName,
                                               String key,
                                               String bucketName,
                                               String role,
                                               String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
            FunctionCode code = FunctionCode.builder()
                .s3Key(key)
                .s3Bucket(bucketName)
                .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                .functionName(functionName)
                .description("Created by the Lambda Java API")
                .code(code)
                .handler(handler)
```

```
                .runtime(Runtime.JAVA17)
                .role(role)
                .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
 awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
 waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for Java 2.x API 参考》中的 CreateFunction。

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
```

```
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- 有关 API 详细信息，请参阅《AWS SDK for JavaScript API 参考》中的 CreateFunction。

Kotlin

适用于 Kotlin 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
suspend fun createNewFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String,
): String? {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
```

```
            role = myRole
            runtime = Runtime.Java8
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn
    }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Kotlin API 参考》中的 CreateFunction。

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    public function createFunction($functionName, $role, $bucketName, $handler)
    {
        //This assumes the Lambda function is in an S3 bucket.
        return $this->customWaiter(function () use ($functionName, $role,
$bucketName, $handler) {
            return $this->lambdaClient->createFunction([
                'Code' => [
                    'S3Bucket' => $bucketName,
                    'S3Key' => $functionName,
                ],
                'FunctionName' => $functionName,
                'Role' => $role['Arn'],
                'Runtime' => 'python3.9',
                'Handler' => "$handler.lambda_handler",
            ]);
        });
```

```
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for PHP API 参考》中的 CreateFunction。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例在 AWS Lambda 中创建了一个名为 MyFunction 的新 C#（dotnetcore1.0 运行时系统）函数，从本地文件系统上的 zip 文件为该函数提供编译后的二进制文件（可以使用相对路径或绝对路径）。C# Lambda 函数使用名称 AssemblyName::Namespace.ClassName::MethodName 为函数指定处理程序。您应适当地替换处理程序规范中的程序集名称（不带 .dll 后缀）、命名空间、类名和方法名等部分。新函数将根据提供的值设置环境变量“envvar1”和“envvar2”。

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

输出：

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
LastModified     : 2016-12-29T23:50:14.207+0000
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
Runtime          : dotnetcore1.0
Timeout          : 3
Version          : $LATEST
```

```
VpcConfig        :
```

示例 2：本示例与前例类似，但要先将函数二进制文件上传到 Amazon S3 存储桶（该存储桶必须与预期的 Lambda 函数位于同一区域），然后在创建函数时引用生成的 S3 对象。

```
Write-S3Object -BucketName amzn-s3-demo-bucket -Key MyFunctionBinaries.zip -
File .\MyFunctionBinaries.zip
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -BucketName amzn-s3-demo-bucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 CreateFunction。

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.
```

```
        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.9",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn
```

- 有关 API 详细信息，请参阅《AWS SDK for Python (Boto3) API 参考》中的 CreateFunction。

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
 code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
                                                role: role_arn.to_s,
                                                function_name: function_name,
                                                handler: handler_name,
                                                runtime: 'ruby2.7',
                                                code: {
                                                  zip_file: deployment_package
                                                },
                                                environment: {
                                                  variables: {
                                                    'LOG_LEVEL' => 'info'
                                                  }
```

```
                                                  }
                                                })
   @lambda_client.wait_until(:function_active_v2, { function_name:
function_name }) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
   response
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error creating #{function_name}:\n #{e.message}")
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
 end
```

- 有关 API 详细信息，请参阅《AWS SDK for Ruby API 参考》中的 CreateFunction。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
   /**
    * Create a function, uploading from a zip file.
    */
   pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
       let code = self.prepare_function(zip_file, None).await?;

       let key = code.s3_key().unwrap().to_string();

       let role = self.create_role().await.map_err(|e| anyhow!(e))?;

       info!("Created iam role, waiting 15s for it to become active");
       tokio::time::sleep(Duration::from_secs(15)).await;
```

```
        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
```

```
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for Rust API 参考》中的 CreateFunction。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    TRY.
        lo_lmd->createfunction(
            iv_functionname = iv_function_name
            iv_runtime = `python3.9`
            iv_role = iv_role_arn
            iv_handler = iv_handler
            io_code = io_zip_file
            iv_description = 'AWS Lambda code example'
        ).
        MESSAGE 'Lambda function created.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodestorageexcdex.
        MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
```

```
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 有关 API 详细信息，请参阅《AWS SDK for SAP ABAP API 参考》中的 CreateFunction。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **DeleteAlias** 与 CLI 配合使用

以下代码示例演示如何使用 DeleteAlias。

CLI

AWS CLI

删除 Lambda 函数别名

以下 delete-alias 示例从 my-function Lambda 函数中删除了名为 LIVE 的别名。

```
aws lambda delete-alias \
    --function-name my-function \
    --name LIVE
```

此命令不生成任何输出。

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数别名。

- 有关 API 详细信息，请参阅《AWS CLI 命令参考》中的 DeleteAlias。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例删除了命令中提到的 Lambda 函数别名。

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- 有关 API 的详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 DeleteAlias。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **DeleteFunction** 与 AWS SDK 或 CLI 配合使用

以下代码示例演示如何使用 DeleteFunction。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- 了解基础知识

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
```

```
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for .NET API 参考》中的 DeleteFunction。

C++

SDK for C++

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    Aws::Lambda::Model::DeleteFunctionRequest request;
    request.SetFunctionName(LAMBDA_NAME);

    Aws::Lambda::Model::DeleteFunctionOutcome outcome = client.DeleteFunction(
            request);
```

```
    if (outcome.IsSuccess()) {
        std::cout << "The lambda function was successfully deleted." <<
 std::endl;
    }
    else {
        std::cerr << "Error with Lambda::DeleteFunction. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for C++ API 参考》中的 DeleteFunction。

CLI

AWS CLI

示例 1：按函数名称删除 Lambda 函数

以下 delete-function 示例删除通过指定函数名称命名为 my-function 的 Lambda 函数。

```
aws lambda delete-function \
    --function-name my-function
```

此命令不生成任何输出。

示例 2：按函数 ARN 删除 Lambda 函数

以下 delete-function 示例删除通过指定函数 ARN 命名为 my-function 的 Lambda 函数。

```
aws lambda delete-function \
    --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function
```

此命令不生成任何输出。

示例 3：按部分函数 ARN 删除 Lambda 函数

以下 delete-function 示例删除通过指定函数的部分 ARN 命名为 my-function 的 Lambda 函数。

```
aws lambda delete-function \
```

```
    --function-name 123456789012:function:my-function
```

此命令不生成任何输出。

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数选项。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 DeleteFunction。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(ctx context.Context, functionName
 string) {
 _, err := wrapper.LambdaClient.DeleteFunction(ctx, &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Go API 参考》中的 DeleteFunction。

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
   /**
    * Deletes an AWS Lambda function.
    *
    * @param awsLambda     an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
    * @param functionName  the name of the Lambda function to be deleted
    *
    * @throws LambdaException if an error occurs while deleting the Lambda
function
    */
   public static void deleteLambdaFunction(LambdaClient awsLambda, String
functionName) {
       try {
           DeleteFunctionRequest request = DeleteFunctionRequest.builder()
               .functionName(functionName)
               .build();

           awsLambda.deleteFunction(request);
           System.out.println("The " + functionName + " function was deleted");

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- 有关 API 详细信息，请参阅《AWS SDK for Java 2.x API 参考》中的 DeleteFunction。

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- 有关 API 详细信息，请参阅《AWS SDK for JavaScript API 参考》中的 DeleteFunction。

Kotlin

适用于 Kotlin 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
suspend fun delLambdaFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
```

```
        println("$myFunctionName was deleted")
    }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Kotlin API 参考》中的 DeleteFunction。

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    public function deleteFunction($functionName)
    {
        return $this->lambdaClient->deleteFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for PHP API 参考》中的 DeleteFunction。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例删除了特定版本的 Lambda 函数

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 DeleteFunction。

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise
```

- 有关 API 详细信息，请参阅《AWS SDK for Python (Boto3) API 参考》中的 DeleteFunction。

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deletes a Lambda function.
  # @param function_name: The name of the function to delete.
  def delete_function(function_name)
    print "Deleting function: #{function_name}..."
    @lambda_client.delete_function(
      function_name: function_name
    )
    print 'Done!'.green
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
  end
```

- 有关 API 详细信息，请参阅《AWS SDK for Ruby API 参考》中的 DeleteFunction。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
   /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
   pub async fn delete_function(
       &self,
       location: Option<String>,
   ) -> (
       Result<DeleteFunctionOutput, anyhow::Error>,
       Result<DeleteRoleOutput, anyhow::Error>,
       Option<Result<DeleteObjectOutput, anyhow::Error>>,
   ) {
       info!("Deleting lambda function {}", self.lambda_name);
       let delete_function = self
           .lambda_client
           .delete_function()
           .function_name(self.lambda_name.clone())
           .send()
           .await
           .map_err(anyhow::Error::from);

       info!("Deleting iam role {}", self.role_name);
       let delete_role = self
           .iam_client
           .delete_role()
           .role_name(self.role_name.clone())
           .send()
           .await
           .map_err(anyhow::Error::from);

       let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
           if let Some(location) = location {
               info!("Deleting object {location}");
               Some(
```

```
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for Rust API 参考》中的 DeleteFunction。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    TRY.
        lo_lmd->deletefunction( iv_functionname = iv_function_name ).
        MESSAGE 'Lambda function deleted.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 有关 API 详细信息，请参阅《AWS SDK for SAP ABAP API 参考》中的 DeleteFunction。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **DeleteFunctionConcurrency** 与 CLI 配合使用

以下代码示例演示如何使用 DeleteFunctionConcurrency。

CLI

AWS CLI

从函数中删除预留的并发执行限制

以下 delete-function-concurrency 示例从 my-function 函数中删除了预留的并发执行限制。

```
aws lambda delete-function-concurrency \
    --function-name  my-function
```

此命令不生成任何输出。

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的为 Lambda 函数预留并发。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 DeleteFunctionConcurrency。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例删除了 Lambda 函数的函数并发。

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 DeleteFunctionConcurrency。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 `DeleteProvisionedConcurrencyConfig` 与 CLI 配合使用

以下代码示例演示如何使用 `DeleteProvisionedConcurrencyConfig`。

CLI

AWS CLI

删除预置并发配置

以下 `delete-provisioned-concurrency-config` 示例删除了指定函数 `GREEN` 别名的预置并发配置。

```
aws lambda delete-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier GREEN
```

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》 中的 DeleteProvisionedConcurrencyConfig。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例删除了特定别名的预置并发配置。

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 DeleteProvisionedConcurrencyConfig。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **GetAccountSettings** 与 CLI 配合使用

以下代码示例演示如何使用 GetAccountSettings。

CLI

AWS CLI

在 AWS 区域中检索关于账户的详细信息

以下 get-account-settings 示例展示了账户的 Lambda 限制和使用信息。

```
aws lambda get-account-settings
```

输出：

```
{
    "AccountLimit": {
        "CodeSizeUnzipped": 262144000,
        "UnreservedConcurrentExecutions": 1000,
        "ConcurrentExecutions": 1000,
        "CodeSizeZipped": 52428800,
        "TotalCodeSize": 80530636800
    },
    "AccountUsage": {
        "FunctionCount": 4,
        "TotalCodeSize": 9426
    }
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的 AWS Lambda 限制。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 GetAccountSettings。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例是为了比较账户限制与账户使用情况

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

输出：

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 GetAccountSettings。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **GetAlias** 与 CLI 配合使用

以下代码示例演示如何使用 GetAlias。

CLI

AWS CLI

检索关于函数别名的详细信息

以下 get-alias 示例展示了 my-function Lambda 函数中名为 LIVE 的别名的详细信息。

```
aws lambda get-alias \
    --function-name my-function \
    --name LIVE
```

输出：

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
```

```
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数别名。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 GetAlias。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例检索了特定 Lambda 函数别名的路由配置权重。

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

输出：

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 GetAlias。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **GetFunction** 与 AWS SDK 或 CLI 配合使用

以下代码示例演示如何使用 GetFunction。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- 了解基础知识

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for .NET API 参考》中的 GetFunction。

C++

SDK for C++

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::GetFunctionRequest request;
        request.SetFunctionName(functionName);

        Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

        if (outcome.IsSuccess()) {
            std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                      << std::endl;
        }
        else {
            std::cerr << "Error with Lambda::GetFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- 有关 API 详细信息，请参阅《AWS SDK for C++ API 参考》中的 GetFunction。

CLI

AWS CLI

检索有关函数的信息

以下 `get-function` 示例显示有关 `my-function` 函数的信息：

```
aws lambda get-function \
    --function-name  my-function
```

输出：

```
{
```

```
    "Concurrency": {
        "ReservedConcurrentExecutions": 100
    },
    "Code": {
        "RepositoryType": "S3",
        "Location": "https://awslambda-us-west-2-tasks.s3.us-
west-2.amazonaws.com/snapshots/123456789012/my-function..."
    },
    "Configuration": {
        "TracingConfig": {
            "Mode": "PassThrough"
        },
        "Version": "$LATEST",
        "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
        "FunctionName": "my-function",
        "VpcConfig": {
            "SubnetIds": [],
            "VpcId": "",
            "SecurityGroupIds": []
        },
        "MemorySize": 128,
        "RevisionId": "28f0fb31-5c5c-43d3-8955-03e76c5c1075",
        "CodeSize": 304,
        "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
        "Handler": "index.handler",
        "Role": "arn:aws:iam::123456789012:role/service-role/helloWorldPython-
role-uy3l9qyq",
        "Timeout": 3,
        "LastModified": "2019-09-24T18:20:35.054+0000",
        "Runtime": "nodejs10.x",
        "Description": ""
    }
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数选项。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 GetFunction。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(ctx context.Context, functionName
 string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(ctx,
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Go API 参考》中的 GetFunction。

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
   /**
    * Retrieves information about an AWS Lambda function.
    *
    * @param awsLambda    an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
    * @param functionName the name of the AWS Lambda function to retrieve
information about
    */
   public static void getFunction(LambdaClient awsLambda, String functionName) {
       try {
           GetFunctionRequest functionRequest = GetFunctionRequest.builder()
               .functionName(functionName)
               .build();

           GetFunctionResponse response =
awsLambda.getFunction(functionRequest);
           System.out.println("The runtime of this Lambda function is " +
response.configuration().runtime());

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- 有关 API 详细信息，请参阅《AWS SDK for Java 2.x API 参考》中的 GetFunction。

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- 有关 API 详细信息，请参阅《AWS SDK for JavaScript API 参考》中的 GetFunction。

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    public function getFunction($functionName)
    {
        return $this->lambdaClient->getFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for PHP API 参考》中的 GetFunction。

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def get_function(self, function_name):
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
 self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response
```

- 有关 API 详细信息，请参阅《AWS SDK for Python (Boto3) API 参考》中的 GetFunction。

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Gets data about a Lambda function.
  #
  # @param function_name: The name of the function.
  # @return response: The function data, or nil if no such function exists.
  def get_function(function_name)
    @lambda_client.get_function(
      {
        function_name: function_name
      }
    )
  rescue Aws::Lambda::Errors::ResourceNotFoundException => e
    @logger.debug("Could not find function: #{function_name}:\n #{e.message}")
    nil
  end
```

- 有关 API 详细信息，请参阅《AWS SDK for Ruby API 参考》中的 GetFunction。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for Rust API 参考》中的 GetFunction。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    TRY.
        oo_result =  lo_lmd->getfunction( iv_functionname = iv_function_name ).
     " oo_result is returned for testing purposes. "
        MESSAGE 'Lambda function information retrieved.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
```

```
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 有关 API 详细信息，请参阅《AWS SDK for SAP ABAP API 参考》中的 GetFunction。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 GetFunctionConcurrency 与 CLI 配合使用

以下代码示例演示如何使用 GetFunctionConcurrency。

CLI

AWS CLI

查看函数的预留并发设置

以下 get-function-concurrency 示例检索了指定函数的预留并发设置。

```
aws lambda get-function-concurrency \
    --function-name my-function
```

输出：

```
{
    "ReservedConcurrentExecutions": 250
}
```

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 GetFunctionConcurrency。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例获取了 Lambda 函数的预留并发

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

输出：

```
ReservedConcurrentExecutions
----------------------------
100
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 GetFunctionConcurrency。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **GetFunctionConfiguration** 与 CLI 配合使用

以下代码示例演示如何使用 GetFunctionConfiguration。

CLI

AWS CLI

检索 Lambda 函数的版本特定设置

以下 get-function-configuration 示例展示了 my-function 函数版本 2 的设置。

```
aws lambda get-function-configuration \
    --function-name  my-function:2
```

输出：

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
```

```
    "MemorySize": 256,
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
    "Handler": "index.handler"
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数选项。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 GetFunctionConfiguration。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例返回了 Lambda 函数的版本特定配置。

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

输出：

```
CodeSha256                  : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                    : 1426
DeadLetterConfig            : Amazon.Lambda.Model.DeadLetterConfig
Description                 : Verson 3 to test Aliases
Environment                 : Amazon.Lambda.Model.EnvironmentResponse
```

```
FunctionArn                : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
                             :PowershellAlias
FunctionName               : MylambdaFunction123
Handler                    : lambda_function.launch_instance
KMSKeyArn                  :
LastModified               : 2019-12-25T09:52:59.872+0000
LastUpdateStatus           : Successful
LastUpdateStatusReason     :
LastUpdateStatusReasonCode :
Layers                     : {}
MasterArn                  :
MemorySize                 : 128
RevisionId                 : 5d7de38b-87f2-4260-8f8a-e87280e10c33
Role                       : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                    : python3.8
State                      : Active
StateReason                :
StateReasonCode            :
Timeout                    : 600
TracingConfig              : Amazon.Lambda.Model.TracingConfigResponse
Version                    : 4
VpcConfig                  : Amazon.Lambda.Model.VpcConfigDetail
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 GetFunctionConfiguration。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **GetPolicy** 与 CLI 配合使用

以下代码示例演示如何使用 GetPolicy。

CLI

AWS CLI

检索函数、版本或别名的基于资源的 IAM 策略

以下 get-policy 示例展示了有关 my-function Lambda 函数的策略信息。

```
aws lambda get-policy \
```

```
    --function-name my-function
```

输出：

```
{
    "Policy": {
        "Version":"2012-10-17",
        "Id":"default",
        "Statement":
        [
            {
                "Sid":"iot-events",
                "Effect":"Allow",
                "Principal": {"Service":"iotevents.amazonaws.com"},
                "Action":"lambda:InvokeFunction",
                "Resource":"arn:aws:lambda:us-west-2:123456789012:function:my-
function"
            }
        ]
    },
    "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668"
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的对 AWS Lambda 使用基于资源的策略。

- 有关 API 详细信息，请参阅《AWS CLI 命令参考》中的 GetPolicy。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例展示了 Lambda 函数的函数策略

```
Get-LMPolicy -FunctionName test -Select Policy
```

输出：

```
{"Version":"2012-10-17","Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 GetPolicy。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 GetProvisionedConcurrencyConfig 与 CLI 配合使用

以下代码示例演示如何使用 GetProvisionedConcurrencyConfig。

CLI

AWS CLI

查看预置并发配置

以下 get-provisioned-concurrency-config 示例展示了指定函数 BLUE 别名的预置并发配置的详细信息。

```
aws lambda get-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE
```

输出：

```
{
    "RequestedProvisionedConcurrentExecutions": 100,
    "AvailableProvisionedConcurrentExecutions": 100,
    "AllocatedProvisionedConcurrentExecutions": 100,
    "Status": "READY",
    "LastModified": "2019-12-31T20:28:49+0000"
}
```

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 GetProvisionedConcurrencyConfig。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例获取了 Lambda 函数指定别名的预置并发配置。

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

输出：

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- 有关 API 的详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 GetProvisionedConcurrencyConfig。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **Invoke** 与 AWS SDK 或 CLI 配合使用

以下代码示例演示如何使用 `Invoke`。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- 了解基础知识

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /// <summary>
```

```
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for .NET API 参考》中的 Invoke。

C++

SDK for C++

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
```

```
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }

        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
```

- 有关 API 详细信息，请参阅《AWS SDK for C++ API 参考》中的 Invoke。

CLI

AWS CLI

示例 1：同步调用 Lambda 函数

以下 `invoke` 示例同步调用该 `my-function` 函数。如果使用 `cli-binary-format` CLI 版本 2，则 AWS 选项是必需的。有关更多信息，请参阅《AWS 命令行界面版本 2 的用户指南》中 AWS CLI 支持的全局命令行选项。

```
aws lambda invoke \
```

```
    --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

输出：

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的同步调用。

示例 2：异步调用 Lambda 函数

以下 invoke 示例异步调用该 my-function 函数。如果使用 cli-binary-format CLI 版本 2，则 AWS 选项是必需的。有关更多信息，请参阅《AWS 命令行界面版本 2 的用户指南》中 AWS CLI 支持的全局命令行选项。

```
aws lambda invoke \
    --function-name my-function \
    --invocation-type Event \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

输出：

```
{
    "StatusCode": 202
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的异步调用。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 Invoke。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(ctx context.Context, functionName string,
 parameters any, getLog bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(ctx, &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
```

```
 return invokeOutput
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Go API 参考》中的 Invoke。

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
   /**
    * Invokes a specific AWS Lambda function.
    *
    * @param awsLambda    an instance of {@link LambdaClient} to interact with
the AWS Lambda service
    * @param functionName the name of the AWS Lambda function to be invoked
    */
   public static void invokeFunction(LambdaClient awsLambda, String
functionName) {
       InvokeResponse res;
       try {
           // Need a SdkBytes instance for the payload.
           JSONObject jsonObj = new JSONObject();
           jsonObj.put("inputValue", "2000");
           String json = jsonObj.toString();
           SdkBytes payload = SdkBytes.fromUtf8String(json);

           InvokeRequest request = InvokeRequest.builder()
               .functionName(functionName)
               .payload(payload)
               .build();

           res = awsLambda.invoke(request);
           String value = res.payload().asUtf8String();
           System.out.println(value);
```

```
        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for Java 2.x API 参考》中的 Invoke。

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

- 有关 API 详细信息，请参阅《AWS SDK for JavaScript API 参考》中的 Invoke。

Kotlin

适用于 Kotlin 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
suspend fun invokeFunction(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            logType = LogType.Tail
            payload = byteArray
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("${res.payload?.toString(Charsets.UTF_8)}")
        println("The log result is ${res.logResult}")
    }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Kotlin API 参考》中的 Invoke。

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    public function invoke($functionName, $params, $logType = 'None')
    {
        return $this->lambdaClient->invoke([
            'FunctionName' => $functionName,
            'Payload' => json_encode($params),
            'LogType' => $logType,
        ]);
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for PHP API 参考》中的 Invoke。

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
 dict
                                is serialized to JSON before it is sent to
 Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
 included in
                        the response.
        :return: The response from the function invocation.
```

```
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response
```

- 有关 API 详细信息，请参阅《AWS SDK for Python (Boto3) API 参考》中的 Invoke。

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
```

```
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name }
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
 #{e.message}")
  end
```

- 有关 API 详细信息，请参阅《AWS SDK for Ruby API 参考》中的 Invoke。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
 anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
```

```
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Rust API 参考》中的调用。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    TRY.
        DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
          `{`  &&
            `"action": "increment",`  &&
            `"number": 10` &&
          `}`
        ).
        oo_result =  lo_lmd->invoke(                  " oo_result is returned for
testing purposes. "
                iv_functionname = iv_function_name
                iv_payload = lv_json
             ).
        MESSAGE 'Lambda function invoked.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvrequestcontex.
        MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidzipfileex.
        MESSAGE 'The deployment package could not be unzipped.' TYPE 'E'.
      CATCH /aws1/cx_lmdrequesttoolargeex.
```

```
        MESSAGE 'Invoke request body JSON input limit was exceeded by the request
payload.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
      CATCH /aws1/cx_lmdunsuppedmediatyp00.
        MESSAGE 'Invoke request body does not have JSON as its content type.'
TYPE 'E'.
    ENDTRY.
```

- 有关 API 详细信息，请参阅《AWS SDK for SAP ABAP API 参考》中的 Invoke。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **ListFunctions** 与 AWS SDK 或 CLI 配合使用

以下代码示例演示如何使用 `ListFunctions`。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- 了解基础知识

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /// <summary>
    /// Get a list of Lambda functions.
    /// </summary>
    /// <returns>A list of FunctionConfiguration objects.</returns>
    public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
    {
        var functionList = new List<FunctionConfiguration>();

        var functionPaginator =
            _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
        await foreach (var function in functionPaginator.Functions)
        {
            functionList.Add(function);
        }

        return functionList;
    }
```

- 有关 API 的详细信息，请参阅 AWS SDK for .NET API 参考中的 ListFunctions。

C++

SDK for C++

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    std::vector<Aws::String> functions;
    Aws::String marker;
```

```
   do {
       Aws::Lambda::Model::ListFunctionsRequest request;
       if (!marker.empty()) {
           request.SetMarker(marker);
       }

       Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
               request);

       if (outcome.IsSuccess()) {
           const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
           std::cout << result.GetFunctions().size()
                     << " lambda functions were retrieved." << std::endl;

           for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
               functions.push_back(functionConfiguration.GetFunctionName());
               std::cout << functions.size() << "  "
                         << functionConfiguration.GetDescription() << std::endl;
               std::cout << "   "
                         <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                 functionConfiguration.GetRuntime()) << ": "
                         << functionConfiguration.GetHandler()
                         << std::endl;
           }
           marker = result.GetNextMarker();
       }
       else {
           std::cerr << "Error with Lambda::ListFunctions. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
       }
   } while (!marker.empty());
```

- 有关 API 的详细信息，请参阅 AWS SDK for C++ API 参考中的 ListFunctions。

CLI

AWS CLI

检索 Lambda 函数列表

以下 `list-functions` 示例显示当前用户所有函数的列表。

```
aws lambda list-functions
```

输出：

```
{
    "Functions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
            "FunctionName": "helloworld",
            "MemorySize": 128,
            "RevisionId": "1718e831-badf-4253-9518-d0644210af7b",
            "CodeSize": 294,
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:helloworld",
            "Handler": "helloworld.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-
role-zgur6bf4",
            "Timeout": 3,
            "LastModified": "2023-09-23T18:32:33.857+0000",
            "Runtime": "nodejs18.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
```

```
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2023-10-01T16:47:28.490+0000",
            "Runtime": "nodejs18.x",
            "Description": ""
        },
        {
            "Layers": [
                {
                    "CodeSize": 41784542,
                    "Arn": "arn:aws:lambda:us-
west-2:420165488524:layer:AWSLambda-Python37-SciPy1x:2"
                },
                {
                    "CodeSize": 4121,
                    "Arn": "arn:aws:lambda:us-
west-2:123456789012:layer:pythonLayer:1"
                }
            ],
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "ZQukCqxtkqFgyF2cU41Avj99TKQ/hNihPtDtRcc08mI=",
            "FunctionName": "my-python-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 128,
            "RevisionId": "80b4eabc-acf7-4ea8-919a-e874c213707d",
            "CodeSize": 299,
```

```
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
python-function",
            "Handler": "lambda_function.lambda_handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/my-python-
function-role-z5g7dr6n",
            "Timeout": 3,
            "LastModified": "2023-10-01T19:40:41.643+0000",
            "Runtime": "python3.11",
            "Description": ""
        }
    ]
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数选项。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 ListFunctions。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(ctx context.Context, maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
```

```
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(ctx)
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}
```

- 有关 API 的详细信息，请参阅 AWS SDK for Go API 参考中的 ListFunctions。

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

- 有关 API 的详细信息，请参阅 AWS SDK for JavaScript API 参考中的 ListFunctions。

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    public function listFunctions($maxItems = 50, $marker = null)
    {
        if (is_null($marker)) {
            return $this->lambdaClient->listFunctions([
                'MaxItems' => $maxItems,
            ]);
        }

        return $this->lambdaClient->listFunctions([
            'Marker' => $marker,
            'MaxItems' => $maxItems,
        ]);
    }
```

- 有关 API 的详细信息，请参阅 AWS SDK for PHP API 参考中的 ListFunctions。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例展示了代码大小已排序的所有 Lambda 函数

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

输出：

```
FunctionName                                                  Runtime   Timeout
 CodeSize
```

```
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 ListFunctions。

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
```

```
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- 有关 API 详细信息，请参阅《AWS SDK for Python (Boto3) API 参考》中的 ListFunctions。

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Lists the Lambda functions for the current account.
  def list_functions
    functions = []
    @lambda_client.list_functions.each do |response|
      response['functions'].each do |function|
        functions.append(function['function_name'])
      end
    end
    functions
```

```
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error listing functions:\n #{e.message}")
  end
```

- 有关 API 的详细信息，请参阅 AWS SDK for Ruby API 参考中的 ListFunctions。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
   /** List all Lambda functions in the current Region. */
   pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
       info!("Listing lambda functions");
       self.lambda_client
           .list_functions()
           .send()
           .await
           .map_err(anyhow::Error::from)
   }
```

- 有关 API 详细信息，请参阅《AWS SDK for Rust API 参考》中的 ListFunctions。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    TRY.
        oo_result =  lo_lmd->listfunctions( ).       " oo_result is returned for
testing purposes. "
        DATA(lt_functions) = oo_result->get_functions( ).
        MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 有关 API 详细信息，请参阅《AWS SDK for SAP ABAP API 参考》中的 ListFunctions。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **ListProvisionedConcurrencyConfigs** 与 CLI 配合使用

以下代码示例演示如何使用 ListProvisionedConcurrencyConfigs。

CLI

AWS CLI

获取预置并发配置列表

以下 list-provisioned-concurrency-configs 示例列出了指定函数的预置并发配置。

```
aws lambda list-provisioned-concurrency-configs \
    --function-name my-function
```

输出：

```
{
    "ProvisionedConcurrencyConfigs": [
        {
```

```
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:GREEN",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:29:00+0000"
        },
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:BLUE",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:28:49+0000"
        }
    ]
}
```

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》 中的 ListProvisionedConcurrencyConfigs。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例检索了 Lambda 函数的预置并发配置列表。

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 ListProvisionedConcurrencyConfigs。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **ListTags** 与 CLI 配合使用

以下代码示例演示如何使用 ListTags。

CLI

AWS CLI

检索 Lambda 函数的标签列表

以下 `list-tags` 示例展示了附加到 `my-function` Lambda 函数的标签。

```
aws lambda list-tags \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function
```

输出：

```
{
    "Tags": {
        "Category": "Web Tools",
        "Department": "Sales"
    }
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的标记 Lambda 函数。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 ListTags。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例检索了当前在指定函数上设置的标签及其值。

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

输出：

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 ListTags。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **ListVersionsByFunction** 与 CLI 配合使用

以下代码示例演示如何使用 `ListVersionsByFunction`。

CLI

AWS CLI

检索函数的版本列表

以下 `list-versions-by-function` 示例展示了 `my-function` Lambda 函数的版本列表。

```
aws lambda list-versions-by-function \
    --function-name my-function
```

输出：

```
{
    "Versions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:$LATEST",
```

```
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "1",
            "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "949c8914-012e-4795-998c-e467121951b1",
            "CodeSize": 304,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:1",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-09-26T20:28:40.438+0000",
            "Runtime": "nodejs10.x",
            "Description": "new version"
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "2",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
```

```
            },
            "MemorySize": 256,
            "RevisionId": "cd669f21-0f3d-4e1c-9566-948837f2e2ea",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": "newer version"
        }
    ]
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数别名。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 ListVersionsByFunction。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例返回了 Lambda 函数各个版本的版本特定配置列表。

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

输出：

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
```

```
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 ListVersionsByFunction。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **PublishVersion** 与 CLI 配合使用

以下代码示例演示如何使用 PublishVersion。

CLI

AWS CLI

发布函数的新版本

以下 publish-version 示例发布了 my-function Lambda 函数的新版本。

```
aws lambda publish-version \
    --function-name my-function
```

输出：

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
    "FunctionName": "my-function",
    "CodeSize": 294,
    "RevisionId": "f31d3d39-cc63-4520-97d4-43cd44c94c20",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:3",
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2019-09-23T18:32:33.857+0000",
```

```
    "Handler": "my-function.handler",
    "Runtime": "nodejs10.x",
    "Description": ""
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数别名。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 PublishVersion。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例创建了 Lambda 函数代码的现有快照版本

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 PublishVersion。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **PutFunctionConcurrency** 与 CLI 配合使用

以下代码示例演示如何使用 PutFunctionConcurrency。

CLI

AWS CLI

配置函数的预留并发限制

以下 put-function-concurrency 示例为 my-function 函数配置了 100 个预留并发执行。

```
aws lambda put-function-concurrency \
    --function-name  my-function  \
    --reserved-concurrent-executions 100
```

输出：

```
{
    "ReservedConcurrentExecutions": 100
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的为 Lambda 函数预留并发。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 PutFunctionConcurrency。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例将函数的并发设置作为一个整体应用。

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 PutFunctionConcurrency。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **PutProvisionedConcurrencyConfig** 与 CLI 配合使用

以下代码示例演示如何使用 PutProvisionedConcurrencyConfig。

CLI

AWS CLI

分配预置并发

以下 put-provisioned-concurrency-config 示例为指定函数的 BLUE 别名分配了 100 个预置并发。

```
aws lambda put-provisioned-concurrency-config \
    --function-name my-function \
```

```
    --qualifier BLUE \
    --provisioned-concurrent-executions 100
```

输出：

```
{
    "Requested ProvisionedConcurrentExecutions": 100,
    "Allocated ProvisionedConcurrentExecutions": 0,
    "Status": "IN_PROGRESS",
    "LastModified": "2019-11-21T19:32:12+0000"
}
```

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 PutProvisionedConcurrencyConfig。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例将预置并发配置添加到了函数的别名

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 PutProvisionedConcurrencyConfig。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **RemovePermission** 与 CLI 配合使用

以下代码示例演示如何使用 RemovePermission。

CLI

AWS CLI

从现有 Lambda 函数中删除权限

以下 remove-permission 示例删除了调用名为 my-function 的函数的权限。

```
aws lambda remove-permission \
    --function-name my-function \
    --statement-id sns
```

此命令不生成任何输出。

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的对 AWS Lambda 使用基于资源的策略。

- 有关 API 的详细信息，请参阅《AWS CLI Command Reference》中的 RemovePermission。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例删除了 Lambda 函数指定 StatementId 的函数策略。

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 RemovePermission。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 `TagResource` 与 CLI 配合使用

以下代码示例演示如何使用 `TagResource`。

CLI

AWS CLI

将标签添加到现有 Lambda 函数

以下 `tag-resource` 示例向指定的 Lambda 函数添加了键名称为 `DEPARTMENT` 和值为 `Department A` 的标签。

```
aws lambda tag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tags "DEPARTMENT=Department A"
```

此命令不生成任何输出。

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的标记 Lambda 函数。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 TagResource。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例将三个标签（Washington、Oregon 和 California）及其关联值添加到了由函数 ARN 标识的指定函数。

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 TagResource。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **UntagResource** 与 CLI 配合使用

以下代码示例演示如何使用 UntagResource。

CLI

AWS CLI

从现有 Lambda 函数中删除标签

以下 untag-resource 示例从 my-function Lambda 函数中删除了键名称为 DEPARTMENT 的标签。

```
aws lambda untag-resource \
```

```
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tag-keys DEPARTMENT
```

此命令不生成任何输出。

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的标记 Lambda 函数。

- 有关 API 详细信息，请参阅《AWS CLI 命令参考》中的 UntagResource。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例从函数中删除了提供的标签。除非指定了 -Force 开关，否则在继续操作之前，cmdlet 会提示您进行确认。调用服务一次，即可删除标签。

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

示例 2：本示例从函数中删除了提供的标签。除非指定了 -Force 开关，否则在继续操作之前，cmdlet 会提示您进行确认。根据提供的标签调用了一次服务。

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 UntagResource。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **UpdateAlias** 与 CLI 配合使用

以下代码示例演示如何使用 `UpdateAlias`。

CLI

AWS CLI

更新函数别名

以下 update-alias 示例会更新名为 LIVE 的别名，该别名指向 my-function Lambda 函数的版本 3。

```
aws lambda update-alias \
    --function-name my-function \
    --function-version 3 \
    --name LIVE
```

输出：

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数别名。

- 有关 API 详细信息，请参阅《AWS CLI 命令参考》中的 UpdateAlias。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例更新了现有 Lambda 函数别名的配置。RoutingConfiguration 值得到更新，将 60%（0.6）的流量转移到版本 1

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 UpdateAlias。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **UpdateFunctionCode** 与 AWS SDK 或 CLI 配合使用

以下代码示例演示如何使用 UpdateFunctionCode。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- 了解基础知识

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };
```

```
        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
 {response.LastModified}.");
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for .NET API 参考》中的 UpdateFunctionCode。

C++

SDK for C++

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
```

```
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- 有关 API 详细信息，请参阅《AWS SDK for C++ API 参考》中的 UpdateFunctionCode。

CLI

AWS CLI

更新 Lambda 函数代码

以下 `update-function-code` 示例将 `my-function` 函数的未发布（$LATEST）版本的代码替换为指定 zip 文件的内容。

```
aws lambda update-function-code \
    --function-name  my-function \
    --zip-file fileb://my-function.zip
```

输出：

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数选项。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 UpdateFunctionCode。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
// FunctionWrapper encapsulates function actions used in the examples.
```

```
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(ctx context.Context,
 functionName string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.UpdateFunctionCode(ctx,
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Go API 参考》中的 UpdateFunctionCode。

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /**
     * Retrieves information about an AWS Lambda function.
     *
     * @param awsLambda    an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to retrieve
information about
     */
    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();

            GetFunctionResponse response =
awsLambda.getFunction(functionRequest);
            System.out.println("The runtime of this Lambda function is " +
response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for Java 2.x API 参考》中的 UpdateFunctionCode。

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- 有关 API 详细信息，请参阅《AWS SDK for JavaScript API 参考》中的 UpdateFunctionCode。

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
```

```
    {
        return $this->lambdaClient->updateFunctionCode([
            'FunctionName' => $functionName,
            'S3Bucket' => $s3Bucket,
            'S3Key' => $s3Key,
        ]);
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for PHP API 参考》中的 UpdateFunctionCode。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例使用指定 zip 文件中包含的新内容来更新名为“MyFunction”的函数。对于 C# .NET 核心 Lambda 函数，zip 文件应包含编译后的程序集。

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

示例 2：本示例与前例类似，但使用包含了已更新代码的 Amazon S3 对象来更新函数。

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 UpdateFunctionCode。

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper:
```

```
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- 有关 API 详细信息，请参阅《AWS SDK for Python (Boto3) API 参考》中的 UpdateFunctionCode。

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the code for a Lambda function by submitting a .zip archive that
 contains
  # the code for the function.
  #
  # @param function_name: The name of the function to update.
  # @param deployment_package: The function code to update, packaged as bytes in
  #                            .zip format.
  # @return: Data about the update, including the status.
  def update_function_code(function_name, deployment_package)
    @lambda_client.update_function_code(
      function_name: function_name,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
 function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
 #{function_name}:\n #{e.message}")
```

```
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

- 有关 API 详细信息，请参阅《AWS SDK for Ruby API 参考》中的 UpdateFunctionCode。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
```

```
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for Rust API 参考》中的 UpdateFunctionCode。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
   TRY.
       oo_result = lo_lmd->updatefunctioncode(     " oo_result is returned for
testing purposes. "
            iv_functionname = iv_function_name
            iv_zipfile = io_zip_file
         ).

       MESSAGE 'Lambda function code updated.' TYPE 'I'.
     CATCH /aws1/cx_lmdcodesigningcfgno00.
       MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodestorageexcdex.
       MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodeverification00.
       MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvalidcodesigex.
       MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
   ENDTRY.
```

- 有关 API 详细信息，请参阅《AWS SDK for SAP ABAP API 参考》中的 UpdateFunctionCode。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **UpdateFunctionConfiguration** 与 AWS SDK 或 CLI 配合使用

以下代码示例演示如何使用 UpdateFunctionConfiguration。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- 了解基础知识

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
```

```
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
environmentVariables },
        };

        var response = await
_lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for .NET API 参考》中的 UpdateFunctionConfiguration。

C++

SDK for C++

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
```

```
        request.SetEnvironment(environment);

        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
 client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- 有关 API 详细信息，请参阅《AWS SDK for C++ API 参考》中的 UpdateFunctionConfiguration。

CLI

AWS CLI

修改函数的配置

以下 update-function-configuration 示例将 my-function 函数的未发布（$LATEST）版本的内存大小修改为 256 MB。

```
aws lambda update-function-configuration \
    --function-name  my-function \
    --memory-size 256
```

输出：

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
```

```
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

有关更多信息，请参阅《AWS Lambda 开发人员指南》中的配置 AWS Lambda 函数选项。

- 有关 API 详细信息，请参阅《AWS CLI Command Reference》中的 UpdateFunctionConfiguration。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}
```

```
// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(ctx context.Context,
 functionName string, envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(ctx,
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Go API 参考》中的 UpdateFunctionConfiguration。

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /**
     * Updates the configuration of an AWS Lambda function.
     *
     * @param awsLambda     the {@link LambdaClient} instance to use for the AWS
 Lambda operation
     * @param functionName  the name of the AWS Lambda function to update
     * @param handler       the new handler for the AWS Lambda function
     *
```

```
     * @throws LambdaException if there is an error while updating the function
configuration
     */
    public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
        try {
            UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .build();

            awsLambda.updateFunctionConfiguration(configurationRequest);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for Java 2.x API 参考》中的 UpdateFunctionConfiguration。

JavaScript

适用于 JavaScript 的 SDK (v3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
```

```
  });
  return client.send(command);
};
```

- 有关 API 详细信息，请参阅《AWS SDK for JavaScript API 参考》中的 UpdateFunctionConfiguration。

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    public function updateFunctionConfiguration($functionName, $handler,
$environment = '')
    {
        return $this->lambdaClient->updateFunctionConfiguration([
            'FunctionName' => $functionName,
            'Handler' => "$handler.lambda_handler",
            'Environment' => $environment,
        ]);
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for PHP API 参考》中的 UpdateFunctionConfiguration。

PowerShell

适用于 PowerShell 的工具

示例 1：本示例更新了现有 Lambda 函数配置

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
```

```
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- 有关 API 详细信息，请参阅《AWS Tools for PowerShell Cmdlet Reference》中的 UpdateFunctionConfiguration。

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
```

```
            )
            raise
        else:
            return response
```

- 有关 API 详细信息，请参阅《AWS SDK for Python (Boto3) API 参考》中的 UpdateFunctionConfiguration。

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       'LOG_LEVEL' => log_level
```

```
                                                    }
                                                  }
                                                })
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

- 有关 API 详细信息，请参阅《AWS SDK for Ruby API 参考》中的 UpdateFunctionConfiguration。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
```

```
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(updated)
    }
```

- 有关 API 详细信息，请参阅《AWS SDK for Rust API 参考》中的 UpdateFunctionConfiguration。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

```
    TRY.
        oo_result = lo_lmd->updatefunctionconfiguration(     " oo_result is
returned for testing purposes. "
              iv_functionname = iv_function_name
              iv_runtime = iv_runtime
              iv_description  = 'Updated Lambda function'
              iv_memorysize  = iv_memory_size
          ).

        MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 有关 API 详细信息，请参阅《AWS SDK for SAP ABAP API 参考》中的 UpdateFunctionConfiguration。

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

适用于使用 AWS SDK 的 Lambda 的场景

以下代码示例显示如何通过 AWS SDK 实施 Lambda 中的常见场景。这些场景向您展示了如何通过调用 Lambda 中的多个函数或与其他 AWS 服务 结合来完成特定任务。每个场景都包含完整源代码的链接，您可以在其中找到有关如何设置和运行代码的说明。

场景以中等水平的经验为目标，可帮助您结合具体环境了解服务操作。

示例

- 通过 AWS SDK 使用 Lambda 函数自动确认已知的 Amazon Cognito 用户
- 通过 AWS SDK 使用 Lambda 函数自动迁移已知的 Amazon Cognito 用户
- 创建 API Gateway REST API 以跟踪 COVID-19 数据
- 创建借阅图书馆 REST API
- 使用 Step Functions 创建 Messenger 应用程序
- 创建照片资产管理应用程序，让用户能够使用标签管理照片

- 使用 API Gateway 创建 Websocket 聊天应用程序
- 创建用于分析客户反馈和合成音频的应用程序
- 从浏览器调用 Lambda 函数
- 使用 S3 对象 Lambda 转换应用程序的数据
- 使用 API Gateway 调用 Lambda 函数
- 使用 Step Functions 调用 Lambda 函数
- 使用计划的事件调用 Lambda 函数
- 在完成 Amazon Cognito 用户身份验证后，通过 AWS SDK 使用 Lambda 函数写入自定义活动数据

通过 AWS SDK 使用 Lambda 函数自动确认已知的 Amazon Cognito 用户

以下代码示例显示了如何使用 Lambda 函数自动确认已知的 Amazon Cognito 用户。

- 配置用户池以调用 `PreSignUp` 触发器的 Lambda 函数。
- 将用户注册到 Amazon Cognito
- Lambda 函数会扫描 DynamoDB 表并自动确认已知用户。
- 以新用户身份登录，然后清理资源。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

在命令提示符中运行交互式场景。

```
// AutoConfirm separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type AutoConfirm struct {
 helper       IScenarioHelper
```

```
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewAutoConfirm constructs a new auto confirm runner.
func NewAutoConfirm(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) AutoConfirm {
 scenario := AutoConfirm{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddPreSignUpTrigger adds a Lambda handler as an invocation target for the
 PreSignUp trigger.
func (runner *AutoConfirm) AddPreSignUpTrigger(ctx context.Context, userPoolId
 string, functionArn string) {
 log.Printf("Let's add a Lambda function to handle the PreSignUp trigger from
 Cognito.\n" +
  "This trigger happens when a user signs up, and lets your function take action
 before the main Cognito\n" +
  "sign up processing occurs.\n")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.PreSignUp, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the PreSignUp
 trigger.\n",
  functionArn, userPoolId)
}

// SignUpUser signs up a user from the known user table with a password you
 specify.
func (runner *AutoConfirm) SignUpUser(ctx context.Context, clientId string,
 usersTable string) (string, string) {
```

```
 log.Println("Let's sign up a user to your Cognito user pool. When the user's
 email matches an email in the\n" +
  "DynamoDB known users table, it is automatically verified and the user is
 confirmed.")

 knownUsers, err := runner.helper.GetKnownUsers(ctx, usersTable)
 if err != nil {
  panic(err)
 }
 userChoice := runner.questioner.AskChoice("Which user do you want to use?\n",
 knownUsers.UserNameList())
 user := knownUsers.Users[userChoice]

 var signedUp bool
 var userConfirmed bool
 password := runner.questioner.AskPassword("Enter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !signedUp {
  log.Printf("Signing up user '%v' with email '%v' to Cognito.\n", user.UserName,
 user.UserEmail)
  userConfirmed, err = runner.cognitoActor.SignUp(ctx, clientId, user.UserName,
 password, user.UserEmail)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("Enter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   signedUp = true
  }
 }
 log.Printf("User %v signed up, confirmed = %v.\n", user.UserName, userConfirmed)

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// SignInUser signs in a user.
func (runner *AutoConfirm) SignInUser(ctx context.Context, clientId string,
 userName string, password string) string {
```

```
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 log.Printf("Let's sign in as %v...\n", userName)
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 log.Println(strings.Repeat("-", 88))
 return *authResult.AccessToken
}

// Run runs the scenario.
func (runner *AutoConfirm) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(ctx, stackOutputs["TableName"])

 runner.AddPreSignUpTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["AutoConfirmFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers, actions.PreSignUp)
 userName, password := runner.SignUpUser(ctx, stackOutputs["UserPoolClientId"],
 stackOutputs["TableName"])
 runner.helper.ListRecentLogEvents(ctx, stackOutputs["AutoConfirmFunction"])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
  runner.SignInUser(ctx, stackOutputs["UserPoolClientId"], userName, password))

 runner.resources.Cleanup(ctx)
```

```
 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

使用 Lambda 函数处理 PreSignUp 触发器。

```
const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PreSignUp event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be confirmed and verified.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsPreSignup) (events.CognitoEventUserPoolsPreSignup,
 error) {
 log.Printf("Received presignup from %v for user '%v'", event.TriggerSource,
 event.UserName)
 if event.TriggerSource != "PreSignUp_SignUp" {
  // Other trigger sources, such as PreSignUp_AdminInitiateAuth, ignore the
 response from this handler.
```

```
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserEmail: event.Request.UserAttributes["email"],
 }
 log.Printf("Looking up email %v in table %v.\n", user.UserEmail, tableName)
 output, err := h.dynamoClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key:       user.GetKey(),
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Error looking up email %v.\n", user.UserEmail)
  return event, err
 }
 if output.Item == nil {
  log.Printf("Email %v not found. Email verification is required.\n",
 user.UserEmail)
  return event, err
 }

 err = attributevalue.UnmarshalMap(output.Item, &user)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB item. Here's why: %v\n", err)
  return event, err
 }

 if user.UserName != event.UserName {
  log.Printf("UserEmail %v found, but stored UserName '%v' does not match
 supplied UserName '%v'. Verification is required.\n",
   user.UserEmail, user.UserName, event.UserName)
 } else {
  log.Printf("UserEmail %v found with matching UserName %v. User is confirmed.
\n", user.UserEmail, user.UserName)
  event.Response.AutoConfirmUser = true
  event.Response.AutoVerifyEmail = true
 }

 return event, err
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
```

```
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

创建一个执行常见任务的结构。

```
// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
```

```
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}
```

```
// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

创建一个封装 Amazon Cognito 操作的结构。

```
type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
```

```
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}
```

```
// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}
```

```
// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
```

```
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
```

```
 }
 return err
}
```

创建一个封装 DynamoDB 操作的结构。

```
// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
```

```
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
```

```
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

创建一个封装 CloudWatch Logs 操作的结构。

```
type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
```

```
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

创建一个封装 AWS CloudFormation 操作的结构。

```
// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}
```

```
// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

清理资源。

```
// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}
```

```
// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Go API 参考》中的以下主题。

- DeleteUser
- InitiateAuth
- SignUp
- UpdateUserPool

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 AWS SDK 使用 Lambda 函数自动迁移已知的 Amazon Cognito 用户

以下代码示例显示了如何使用 Lambda 函数自动迁移已知的 Amazon Cognito 用户。

- 配置用户池以调用 `MigrateUser` 触发器的 Lambda 函数。
- 使用不在用户池中的用户名和电子邮件地址登录 Amazon Cognito。
- Lambda 函数会扫描 DynamoDB 表并自动将已知用户迁移到该用户池。
- 执行“忘记密码”流程可重置已迁移用户的密码。
- 以新用户身份登录，然后清理资源。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

在命令提示符中运行交互式场景。

```
import (
 "context"
 "errors"
 "fmt"
 "log"
 "strings"
```

```
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// MigrateUser separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type MigrateUser struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewMigrateUser constructs a new migrate user runner.
func NewMigrateUser(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) MigrateUser {
 scenario := MigrateUser{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddMigrateUserTrigger adds a Lambda handler as an invocation target for the
 MigrateUser trigger.
func (runner *MigrateUser) AddMigrateUserTrigger(ctx context.Context, userPoolId
 string, functionArn string) {
 log.Printf("Let's add a Lambda function to handle the MigrateUser trigger from
 Cognito.\n" +
  "This trigger happens when an unknown user signs in, and lets your function
 take action before Cognito\n" +
  "rejects the user.\n\n")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
```

```
  actions.TriggerInfo{Trigger: actions.UserMigration, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the MigrateUser
 trigger.\n",
  functionArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser adds a new user to the known users table and signs that user in to
 Amazon Cognito.
func (runner *MigrateUser) SignInUser(ctx context.Context, usersTable string,
 clientId string) (bool, actions.User) {
 log.Println("Let's sign in a user to your Cognito user pool. When the username
 and email matches an entry in the\n" +
  "DynamoDB known users table, the email is automatically verified and the user
 is migrated to the Cognito user pool.")

 user := actions.User{}
 user.UserName = runner.questioner.Ask("\nEnter a username:")
 user.UserEmail = runner.questioner.Ask("\nEnter an email that you own. This
 email will be used to confirm user migration\n" +
  "during this example:")

 runner.helper.AddKnownUser(ctx, usersTable, user)

 var err error
 var resetRequired *types.PasswordResetRequiredException
 var authResult *types.AuthenticationResultType
 signedIn := false
 for !signedIn && resetRequired == nil {
  log.Printf("Signing in to Cognito as user '%v'. The expected result is a
 PasswordResetRequiredException.\n\n", user.UserName)
  authResult, err = runner.cognitoActor.SignIn(ctx, clientId, user.UserName, "_")
  if err != nil {
   if errors.As(err, &resetRequired) {
    log.Printf("\nUser '%v' is not in the Cognito user pool but was found in the
 DynamoDB known users table.\n"+
     "User migration is started and a password reset is required.",
 user.UserName)
   } else {
```

```
    panic(err)
   }
  } else {
   log.Printf("User '%v' successfully signed in. This is unexpected and probably
 means you have not\n"+
    "cleaned up a previous run of this scenario, so the user exist in the Cognito
 user pool.\n"+
    "You can continue this example and select to clean up resources, or manually
 remove\n"+
    "the user from your user pool and try again.", user.UserName)
   runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
   signedIn = true
  }
 }

 log.Println(strings.Repeat("-", 88))
 return resetRequired != nil, user
}

// ResetPassword starts a password recovery flow.
func (runner *MigrateUser) ResetPassword(ctx context.Context, clientId string,
 user actions.User) {
 wantCode := runner.questioner.AskBool(fmt.Sprintf("In order to migrate the user
 to Cognito, you must be able to receive a confirmation\n"+
  "code by email at %v. Do you want to send a code (y/n)?", user.UserEmail), "y")
 if !wantCode {
  log.Println("To complete this example and successfully migrate a user to
 Cognito, you must enter an email\n" +
   "you own that can receive a confirmation code.")
  return
 }
 codeDelivery, err := runner.cognitoActor.ForgotPassword(ctx, clientId,
 user.UserName)
 if err != nil {
  panic(err)
 }
 log.Printf("\nA confirmation code has been sent to %v.",
 *codeDelivery.Destination)
 code := runner.questioner.Ask("Check your email and enter it here:")

 confirmed := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
```

```
  "(the password will not display as you type):", 8)
 for !confirmed {
  log.Printf("\nConfirming password reset for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.ConfirmForgotPassword(ctx, clientId, code,
 user.UserName, password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   confirmed = true
  }
 }
 log.Printf("User '%v' successfully confirmed and migrated.\n", user.UserName)
 log.Println("Signing in with your username and password...")
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, user.UserName,
 password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)

 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *MigrateUser) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))
```

```
 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]

 runner.AddMigrateUserTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["MigrateUserFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers,
 actions.UserMigration)
 resetNeeded, user := runner.SignInUser(ctx, stackOutputs["TableName"],
 stackOutputs["UserPoolClientId"])
 if resetNeeded {
  runner.helper.ListRecentLogEvents(ctx, stackOutputs["MigrateUserFunction"])
  runner.ResetPassword(ctx, stackOutputs["UserPoolClientId"], user)
 }

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

使用 Lambda 函数处理 `MigrateUser` 触发器。

```
const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

type handler struct {
 dynamoClient *dynamodb.Client
}
```

```
// HandleRequest handles the MigrateUser event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be migrated to the user pool.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsMigrateUser)
 (events.CognitoEventUserPoolsMigrateUser, error) {
 log.Printf("Received migrate trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 if event.TriggerSource != "UserMigration_Authentication" {
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName: event.UserName,
 }
 log.Printf("Looking up user '%v' in table %v.\n", user.UserName, tableName)
 filterEx := expression.Name("UserName").Equal(expression.Value(user.UserName))
 expr, err := expression.NewBuilder().WithFilter(filterEx).Build()
 if err != nil {
  log.Printf("Error building expression to query for user '%v'.\n",
 user.UserName)
  return event, err
 }
 output, err := h.dynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName:                 aws.String(tableName),
  FilterExpression:          expr.Filter(),
  ExpressionAttributeNames:  expr.Names(),
  ExpressionAttributeValues: expr.Values(),
 })
 if err != nil {
  log.Printf("Error looking up user '%v'.\n", user.UserName)
  return event, err
 }
 if len(output.Items) == 0 {
  log.Printf("User '%v' not found, not migrating user.\n", user.UserName)
  return event, err
 }

 var users []UserInfo
 err = attributevalue.UnmarshalListOfMaps(output.Items, &users)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB items. Here's why: %v\n", err)
  return event, err
 }
```

```
 user = users[0]
 log.Printf("UserName '%v' found with email %v. User is migrated and must reset
 password.\n", user.UserName, user.UserEmail)
 event.CognitoEventUserPoolsMigrateUserResponse.UserAttributes =
 map[string]string{
  "email":          user.UserEmail,
  "email_verified": "true", // email_verified is required for the forgot password
 flow.
 }
 event.CognitoEventUserPoolsMigrateUserResponse.FinalUserStatus =
 "RESET_REQUIRED"
 event.CognitoEventUserPoolsMigrateUserResponse.MessageAction = "SUPPRESS"

 return event, err
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

创建一个执行常见任务的结构。

```
// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
```

```
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
```

```
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
```

```
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

创建一个封装 Amazon Cognito 操作的结构。

```
type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}
```

```
// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
```

```
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
```

```
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}
```

```
// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}
```

```
// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

创建一个封装 DynamoDB 操作的结构。

```
// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}
```

```
// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
```

```
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

创建一个封装 CloudWatch Logs 操作的结构。

```
type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
```

```
 } else {
  events = output.Events
 }
 return events, err
}
```

创建一个封装 AWS CloudFormation 操作的结构。

```
// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

清理资源。

```
// Resources keeps track of AWS resources created during an example and handles
```

```
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
```

```
    triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
  HandlerArn: nil}
   }
   err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
  triggerList...)
   if err != nil {
    log.Println("Couldn't update Cognito triggers during cleanup.")
    panic(err)
   }
   log.Println("Removed Cognito triggers from user pool.")
  } else {
   log.Println("Be sure to remove resources when you're done with them to avoid
  unexpected charges!")
  }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Go API 参考》中的以下主题。
  - ConfirmForgotPassword
  - DeleteUser
  - ForgotPassword
  - InitiateAuth
  - SignUp
  - UpdateUserPool

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

创建 API Gateway REST API 以跟踪 COVID-19 数据

以下代码示例显示如何创建 REST API，该 API 模拟一个使用虚构数据跟踪美国每日 COVID-19 病例的系统。

Python

SDK for Python (Boto3)

说明如何将 AWS Chalice 与 AWS SDK for Python (Boto3) 结合使用，以创建一个使用 Amazon API Gateway、AWS Lambda 和 Amazon DynamoDB 的无服务器 REST API。REST API 模拟一个使用虚构数据跟踪美国每日 COVID-19 病例的系统。了解如何：

- 使用 AWS Chalice 在 Lambda 函数中定义路由，可以调用这些函数来处理通过 API Gateway 发出的 REST 请求。
- 使用 Lambda 函数在 DynamoDB 表中检索数据并存储数据以处理 REST 请求。
- 在 AWS CloudFormation 模板中定义表结构和安全角色资源。
- 使用 AWS Chalice 和 CloudFormation 打包和部署所有必要的资源。
- 使用 CloudFormation 清理所有创建的资源。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- API Gateway
- AWS CloudFormation
- DynamoDB
- Lambda

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

创建借阅图书馆 REST API

以下代码示例显示如何创建借阅图书馆，其中顾客可以使用由 Amazon Aurora 数据库支持的 REST API 借阅和归还图书。

Python

SDK for Python (Boto3)

展示如何将 AWS SDK for Python (Boto3) 和 Amazon Relational Database Service (Amazon RDS) API 与 AWS Chalice 结合使用，以创建由 Amazon Aurora 数据库支持的 REST API。此

Web 服务是完全无服务器的，代表简单的借阅图书馆，其中顾客可以借阅和归还图书。了解如何：

- 创建和管理无服务器 Aurora 数据库集群。
- 使用 AWS Secrets Manager 管理数据库凭证。
- 实施一个数据存储层，该层使用 Amazon RDS 将数据移入和移出数据库。
- 使用 AWS Chalice 将无服务器 REST API 部署到 Amazon API Gateway 和 AWS Lambda。
- 使用请求软件包向 Web 服务发送请求。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- API Gateway
- Aurora
- Lambda
- Secrets Manager

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

使用 Step Functions 创建 Messenger 应用程序

以下代码示例显示如何创建从数据库表中检索消息记录的 AWS Step Functions Messenger 应用程序。

Python

SDK for Python (Boto3)

显示如何结合使用 AWS SDK for Python (Boto3) 和 AWS Step Functions 以创建从 Amazon DynamoDB 表中检索消息记录并使用 Amazon Simple Queue Service (Amazon SQS) 发送消息记录的 Messenger 应用程序。状态机与 AWS Lambda 函数集成以扫描数据库中是否有未发送的消息。

- 创建检索并更新 Amazon DynamoDB 表中的消息记录的状态机。
- 更新状态机定义以便也将消息发送到 Amazon Simple Queue Service (Amazon SQS)。
- 启动和停止状态机运行。

- 使用服务集成从状态机连接到 Lambda、DynamoDB 和 Amazon SQS。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- DynamoDB
- Lambda
- Amazon SQS
- Step Functions

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

创建照片资产管理应用程序，让用户能够使用标签管理照片

以下代码示例演示如何创建无服务器应用程序，让用户能够使用标签管理照片。

.NET

AWS SDK for .NET

演示如何开发照片资产管理应用程序，该应用程序使用 Amazon Rekognition 检测图像中的标签并将其存储以供日后检索。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

要深入了解这个例子的起源，请参阅 AWS 社区上的博文。

本示例中使用的服务

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK for C++

演示如何开发照片资产管理应用程序，该应用程序使用 Amazon Rekognition 检测图像中的标签并将其存储以供日后检索。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

要深入了解这个例子的起源，请参阅 AWS 社区上的博文。

本示例中使用的服务

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

SDK for Java 2.x

演示如何开发照片资产管理应用程序，该应用程序使用 Amazon Rekognition 检测图像中的标签并将其存储以供日后检索。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

要深入了解这个例子的起源，请参阅 AWS 社区上的博文。

本示例中使用的服务

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3

- Amazon SNS

JavaScript

SDK for JavaScript (v3)

演示如何开发照片资产管理应用程序，该应用程序使用 Amazon Rekognition 检测图像中的标签并将其存储以供日后检索。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

要深入了解这个例子的起源，请参阅 AWS 社区上的博文。

本示例中使用的服务

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

适用于 Kotlin 的 SDK

演示如何开发照片资产管理应用程序，该应用程序使用 Amazon Rekognition 检测图像中的标签并将其存储以供日后检索。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

要深入了解这个例子的起源，请参阅 AWS 社区上的博文。

本示例中使用的服务

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition

- Amazon S3
- Amazon SNS

PHP

适用于 PHP 的 SDK

演示如何开发照片资产管理应用程序，该应用程序使用 Amazon Rekognition 检测图像中的标签并将其存储以供日后检索。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

要深入了解这个例子的起源，请参阅 AWS 社区上的博文。

本示例中使用的服务

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

适用于 Rust 的 SDK

演示如何开发照片资产管理应用程序，该应用程序使用 Amazon Rekognition 检测图像中的标签并将其存储以供日后检索。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

要深入了解这个例子的起源，请参阅 AWS 社区上的博文。

本示例中使用的服务

- API Gateway
- DynamoDB
- Lambda

- Amazon Rekognition
- Amazon S3
- Amazon SNS

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

使用 API Gateway 创建 Websocket 聊天应用程序

以下代码示例显示如何创建由基于 Amazon API Gateway 构建的 Websocket API 提供服务的聊天应用程序。

Python

SDK for Python (Boto3)

显示如何将 AWS SDK for Python (Boto3) 与 Amazon API Gateway V2 结合使用，以创建与 AWS Lambda 和 Amazon DynamoDB 集成的 Websocket API。

- 创建由 API Gateway 提供服务的 Websocket API。
- 定义在 DynamoDB 中存储连接并向其他聊天参与者发布消息的 Lambda 处理程序。
- 连接到 Websocket 聊天应用程序并使用 WebSocket 软件包发送消息。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- API Gateway
- DynamoDB
- Lambda

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

创建用于分析客户反馈和合成音频的应用程序

以下代码示例显示如何创建应用程序来分析客户意见卡、翻译其母语、确定其情绪并根据译后的文本生成音频文件。

.NET

AWS SDK for .NET

此示例应用程序可分析并存储客户反馈卡。具体来说，它满足了纽约市一家虚构酒店的需求。酒店以实体意见卡的形式收集来自不同语种的客人的反馈。该反馈通过 Web 客户端上传到应用程序中。意见卡图片上传后，将执行以下步骤：

- 使用 Amazon Textract 从图片中提取文本。
- Amazon Comprehend 确定所提取文本的情绪及其语言。
- 使用 Amazon Translate 将所提取文本翻译为英语。
- Amazon Polly 根据所提取文本合成音频文件。

完整的应用程序可使用 AWS CDK 进行部署。有关源代码和部署说明，请参阅 GitHub 中的项目。

本示例中使用的服务

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Java

SDK for Java 2.x

此示例应用程序可分析并存储客户反馈卡。具体来说，它满足了纽约市一家虚构酒店的需求。酒店以实体意见卡的形式收集来自不同语种的客人的反馈。该反馈通过 Web 客户端上传到应用程序中。意见卡图片上传后，将执行以下步骤：

- 使用 Amazon Textract 从图片中提取文本。
- Amazon Comprehend 确定所提取文本的情绪及其语言。
- 使用 Amazon Translate 将所提取文本翻译为英语。
- Amazon Polly 根据所提取文本合成音频文件。

完整的应用程序可使用 AWS CDK 进行部署。有关源代码和部署说明，请参阅 GitHub 中的项目。

本示例中使用的服务

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

JavaScript

SDK for JavaScript (v3)

此示例应用程序可分析并存储客户反馈卡。具体来说，它满足了纽约市一家虚构酒店的需求。酒店以实体意见卡的形式收集来自不同语种的客人的反馈。该反馈通过 Web 客户端上传到应用程序中。意见卡图片上传后，将执行以下步骤：

- 使用 Amazon Textract 从图片中提取文本。
- Amazon Comprehend 确定所提取文本的情绪及其语言。
- 使用 Amazon Translate 将所提取文本翻译为英语。
- Amazon Polly 根据所提取文本合成音频文件。

完整的应用程序可使用 AWS CDK 进行部署。有关源代码和部署说明，请参阅 GitHub 中的项目。以下摘录显示了 AWS SDK for JavaScript 在 Lambda 函数中是如何使用的。

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string}} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
```

```
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
  const { Languages } = await comprehendClient.send(
    detectDominantLanguageCommand,
  );

  const languageCode = Languages[0].LanguageCode;

  const detectSentimentCommand = new DetectSentimentCommand({
    Text: extractTextOutput.source_text,
    LanguageCode: languageCode,
  });

  const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

  return {
    sentiment: Sentiment,
    language_code: languageCode,
  };
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
 eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();

  const detectDocumentTextCommand = new DetectDocumentTextCommand({
    Document: {
      S3Object: {
        Bucket: eventBridgeS3Event.bucket,
        Name: eventBridgeS3Event.object,
      },
```

```
    },
  });

  // Textract returns a list of blocks. A block can be a line, a page, word, etc.
  // Each block also contains geometry of the detected text.
  // For more information on the Block type, see https://docs.aws.amazon.com/
textract/latest/dg/API_Block.html.
  const { Blocks } = await textractClient.send(detectDocumentTextCommand);

  // For the purpose of this example, we are only interested in words.
  const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
    (b) => b.Text,
  );

  return extractedWords.join(" ");
};
```

```
import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string}}
 sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });

  const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

  const audioKey = `${sourceDestinationConfig.object}.mp3`;

  // Store the audio file in S3.
  const s3Client = new S3Client();
```

```
  const upload = new Upload({
    client: s3Client,
    params: {
      Bucket: sourceDestinationConfig.bucket,
      Key: audioKey,
      Body: AudioStream,
      ContentType: "audio/mp3",
    },
  });

  await upload.done();
  return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
 * @param {{ extracted_text: string, source_language_code: string}}
 textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);

  return { translated_text: TranslatedText };
};
```

本示例中使用的服务

- Amazon Comprehend
- Lambda

- Amazon Polly
- Amazon Textract
- Amazon Translate

Ruby

适用于 Ruby 的 SDK

此示例应用程序可分析并存储客户反馈卡。具体来说，它满足了纽约市一家虚构酒店的需求。酒店以实体意见卡的形式收集来自不同语种的客人的反馈。该反馈通过 Web 客户端上传到应用程序中。意见卡图片上传后，将执行以下步骤：

- 使用 Amazon Textract 从图片中提取文本。
- Amazon Comprehend 确定所提取文本的情绪及其语言。
- 使用 Amazon Translate 将所提取文本翻译为英语。
- Amazon Polly 根据所提取文本合成音频文件。

完整的应用程序可使用 AWS CDK 进行部署。有关源代码和部署说明，请参阅 GitHub 中的项目。

本示例中使用的服务

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

有关 AWS 软件开发工具包开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

从浏览器调用 Lambda 函数

以下代码示例显示如何从浏览器调用 AWS Lambda 函数。

JavaScript

适用于 JavaScript 的 SDK（v2）

您可以创建一个基于浏览器的应用程序，此应用程序使用 AWS Lambda 函数通过用户选择来更新 Amazon DynamoDB 表。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- DynamoDB
- Lambda

SDK for JavaScript (v3)

您可以创建一个基于浏览器的应用程序，此应用程序使用 AWS Lambda 函数通过用户选择来更新 Amazon DynamoDB 表。此应用程序使用 AWS SDK for JavaScript v3。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- DynamoDB
- Lambda

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

使用 S3 对象 Lambda 转换应用程序的数据

下面的代码示例显示如何使用 S3 对象 Lambda 转换应用程序的数据。

.NET

AWS SDK for .NET

显示如何将自定义代码添加到标准 S3 GET 请求中，来修改从 S3 检索的请求对象，从而使该对象适合发出请求的客户端或应用程序的需要。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- Lambda
- Amazon S3

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

使用 API Gateway 调用 Lambda 函数

以下代码示例展示如何创建通过 Amazon API Gateway 调用的 AWS Lambda 函数。

Java

SDK for Java 2.x

展示如何使用 Lambda Java 运行时 API 创建 AWS Lambda 函数。此示例调用不同的 AWS 服务以执行特定的使用案例。此示例展示了如何创建通过 Amazon API Gateway 调用的 Lambda 函数，该函数扫描 Amazon DynamoDB 表获取工作周年纪念日，并使用 Amazon Simple Notification Service (Amazon SNS)向员工发送文本消息，祝贺他们的周年纪念日。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

JavaScript

适用于 JavaScript 的 SDK（v3）

展示如何使用 Lambda JavaScript 运行时 API 创建 AWS Lambda 函数。此示例调用不同的 AWS 服务以执行特定的应用场景。此示例展示了如何创建通过 Amazon API Gateway 调用的 Lambda 函数，该函数扫描 Amazon DynamoDB 表获取工作周年纪念日，并使用 Amazon Simple Notification Service (Amazon SNS)向员工发送文本消息，祝贺他们的周年纪念日。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

该示例也可在 AWS SDK for JavaScript v3 开发人员指南中找到。

本示例中使用的服务

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

此示例显示如何创建和使用以 AWS Lambda 函数为目标的 Amazon API Gateway REST API。Lambda 处理程序演示了如何基于 HTTP 方法进行路由；如何从查询字符串、标头和正文中获取数据；以及如何返回 JSON 响应。

- 部署 Lambda 函数。
- 使用 API Gateway 创建 REST API
- 创建以 Lambda 函数为目标的 REST 资源。
- 授予允许 API Gateway 调用 Lambda 函数的权限。
- 使用请求软件包向 REST API 发送请求。
- 清理演示期间创建的所有资源。

此示例最好在 GitHub 上查看。有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- API Gateway
- Lambda

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

使用 Step Functions 调用 Lambda 函数

以下代码示例展示如何创建 AWS Step Functions 状态机来按顺序调用 AWS Lambda 函数。

Java

SDK for Java 2.x

展示如何使用 AWS Step Functions 和 AWS SDK for Java 2.x 创建 AWS 无服务器工作流。每个工作流步骤都通过使用 AWS Lambda 函数实现。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的软件开发工具包版本的详细信息。

使用计划的事件调用 Lambda 函数

以下代码示例展示如何创建通过 Amazon EventBridge 计划事件调用的 AWS Lambda 函数。

Java

SDK for Java 2.x

展示如何创建调用 AWS Lambda 函数的 Amazon EventBridge 计划事件。将 EventBridge 配置为使用 cron 表达式来计划调用 Lambda 函数的时间。在本示例中，您使用 Lambda Java 运行时 API 创建 Lambda 函数。此示例调用不同的 AWS 服务以执行特定应用场景。此示例展示了如何创建一个应用程序，在其一周年纪念日时向员工发送移动短信表示祝贺。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

适用于 JavaScript 的 SDK（v3）

展示如何创建调用 AWS Lambda 函数的 Amazon EventBridge 计划事件。将 EventBridge 配置为使用 cron 表达式来计划调用 Lambda 函数的时间。在本示例中，您使用 Lambda JavaScript 运行时 API 创建 Lambda 函数。此示例调用不同的 AWS 服务以执行特定应用场景。此示例展示了如何创建一个应用程序，在其一周年纪念日时向员工发送移动短信表示祝贺。

有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

该示例也可在 AWS SDK for JavaScript v3 开发人员指南中找到。

本示例中使用的服务

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

此示例说明了如何注册 AWS Lambda 函数作为计划的 Amazon EventBridge 事件的目标。Lambda 处理程序将友好消息和完整的事件数据写入 Amazon CloudWatch Logs 以供以后检索。

- 部署 Lambda 函数。
- 创建 EventBridge 计划的事件，并将 Lambda 函数作为目标。
- 授予允许 EventBridge 调用 Lambda 函数的权限。
- 打印 CloudWatch Logs 中的最新数据以显示计划调用的结果。
- 清理演示期间创建的所有资源。

此示例最好在 GitHub 上查看。有关完整的源代码以及如何设置和运行的说明，请参阅 GitHub 上的完整示例。

本示例中使用的服务

- CloudWatch Logs
- EventBridge

- Lambda

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

在完成 Amazon Cognito 用户身份验证后，通过 AWS SDK 使用 Lambda 函数写入自定义活动数据

以下代码示例显示了在完成 Amazon Cognito 用户身份验证后如何使用 Lambda 函数写入自定义活动数据。

- 使用管理员功能将用户添加到用户池。
- 配置用户池以调用 `PostAuthentication` 触发器的 Lambda 函数。
- 将新用户登录到 Amazon Cognito 控制台。
- Lambda 函数会将自定义信息写入 CloudWatch Logs 和 DynamoDB 表。
- 从 DynamoDB 表获取并显示自定义数据，然后清理资源。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。查找完整示例，学习如何在 AWS 代码示例存储库中进行设置和运行。

在命令提示符中运行交互式场景。

```
// ActivityLog separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type ActivityLog struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
```

```
}

// NewActivityLog constructs a new activity log runner.
func NewActivityLog(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) ActivityLog {
 scenario := ActivityLog{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddUserToPool selects a user from the known users table and uses administrator
 credentials to add the user to the user pool.
func (runner *ActivityLog) AddUserToPool(ctx context.Context, userPoolId string,
 tableName string) (string, string) {
 log.Println("To facilitate this example, let's add a user to the user pool using
 administrator privileges.")
 users, err := runner.helper.GetKnownUsers(ctx, tableName)
 if err != nil {
  panic(err)
 }
 user := users.Users[0]
 log.Printf("Adding known user %v to the user pool.\n", user.UserName)
 err = runner.cognitoActor.AdminCreateUser(ctx, userPoolId, user.UserName,
 user.UserEmail)
 if err != nil {
  panic(err)
 }
 pwSet := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !pwSet {
  log.Printf("\nSetting password for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.AdminSetUserPassword(ctx, userPoolId, user.UserName,
 password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
```

```
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   pwSet = true
  }
 }

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// AddActivityLogTrigger adds a Lambda handler as an invocation target for the
 PostAuthentication trigger.
func (runner *ActivityLog) AddActivityLogTrigger(ctx context.Context, userPoolId
 string, activityLogArn string) {
 log.Println("Let's add a Lambda function to handle the PostAuthentication
 trigger from Cognito.\n" +
  "This trigger happens after a user is authenticated, and lets your function
 take action, such as logging\n" +
  "the outcome.")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.PostAuthentication, HandlerArn:
 aws.String(activityLogArn)})
 if err != nil {
  panic(err)
 }
 runner.resources.triggers = append(runner.resources.triggers,
 actions.PostAuthentication)
 log.Printf("Lambda function %v added to user pool %v to handle
 PostAuthentication Cognito trigger.\n",
  activityLogArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser signs in as the specified user.
func (runner *ActivityLog) SignInUser(ctx context.Context, clientId string,
 userName string, password string) {
 log.Printf("Now we'll sign in user %v and check the results in the logs and the
 DynamoDB table.", userName)
```

```
 runner.questioner.Ask("Press Enter when you're ready.")
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Println("Sign in successful.",
  "The PostAuthentication Lambda handler writes custom information to CloudWatch
 Logs.")

 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
}

// GetKnownUserLastLogin gets the login info for a user from the Amazon DynamoDB
 table and displays it.
func (runner *ActivityLog) GetKnownUserLastLogin(ctx context.Context, tableName
 string, userName string) {
 log.Println("The PostAuthentication handler also writes login data to the
 DynamoDB table.")
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 users, err := runner.helper.GetKnownUsers(ctx, tableName)
 if err != nil {
  panic(err)
 }
 for _, user := range users.Users {
  if user.UserName == userName {
   log.Println("The last login info for the user in the known users table is:")
   log.Printf("\t%+v", *user.LastLogin)
  }
 }
 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *ActivityLog) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")
```

```
 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(ctx, stackOutputs["TableName"])
 userName, password := runner.AddUserToPool(ctx, stackOutputs["UserPoolId"],
 stackOutputs["TableName"])

 runner.AddActivityLogTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["ActivityLogFunctionArn"])
 runner.SignInUser(ctx, stackOutputs["UserPoolClientId"], userName, password)
 runner.helper.ListRecentLogEvents(ctx, stackOutputs["ActivityLogFunction"])
 runner.GetKnownUserLastLogin(ctx, stackOutputs["TableName"], userName)

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

使用 Lambda 函数处理 PostAuthentication 触发器。

```
const TABLE_NAME = "TABLE_NAME"

// LoginInfo defines structured login data that can be marshalled to a DynamoDB
 format.
type LoginInfo struct {
 UserPoolId string `dynamodbav:"UserPoolId"`
 ClientId   string `dynamodbav:"ClientId"`
 Time       string `dynamodbav:"Time"`
}

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
```

```
 UserName  string    `dynamodbav:"UserName"`
 UserEmail string    `dynamodbav:"UserEmail"`
 LastLogin LoginInfo `dynamodbav:"LastLogin"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PostAuthentication event by writing custom data to
 the logs and
// to an Amazon DynamoDB table.
func (h *handler) HandleRequest(ctx context.Context,
 event events.CognitoEventUserPoolsPostAuthentication)
 (events.CognitoEventUserPoolsPostAuthentication, error) {
 log.Printf("Received post authentication trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName:  event.UserName,
  UserEmail: event.Request.UserAttributes["email"],
  LastLogin: LoginInfo{
   UserPoolId: event.UserPoolID,
   ClientId:   event.CallerContext.ClientID,
   Time:       time.Now().Format(time.UnixDate),
  },
 }
 // Write to CloudWatch Logs.
 fmt.Printf("%#v", user)

 // Also write to an external system. This examples uses DynamoDB to demonstrate.
 userMap, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshal to DynamoDB map. Here's why: %v\n", err)
 } else if len(userMap) == 0 {
```

```
  log.Printf("User info marshaled to an empty map.")
 } else {
  _, err := h.dynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
   Item:      userMap,
   TableName: aws.String(tableName),
  })
  if err != nil {
   log.Printf("Couldn't write to DynamoDB. Here's why: %v\n", err)
  } else {
   log.Printf("Wrote user info to DynamoDB table %v.\n", tableName)
  }
 }

 return event, nil
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

创建一个执行常见任务的结构。

```
// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
```

```
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
```

```
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
```

```
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

创建一个封装 Amazon Cognito 操作的结构。

```
type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
```

```
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
```

```
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}
```

```
// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}
```

```
// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
```

```
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

创建一个封装 DynamoDB 操作的结构。

```
// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
```

```
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
```

```
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

创建一个封装 CloudWatch Logs 操作的结构。

```
type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
```

```
  events = output.Events
 }
 return events, err
}
```

创建一个封装 AWS CloudFormation 操作的结构。

```
// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

清理资源。

```
// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
```

```
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
```

```
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- 有关 API 详细信息，请参阅《AWS SDK for Go API 参考》中的以下主题。
  - AdminCreateUser
  - AdminSetUserPassword
  - DeleteUser
  - InitiateAuth
  - UpdateUserPool

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

使用 AWS SDK 的 Lambda 无服务器示例

以下代码示例显示如何将 Lambda 与 AWS SDK 一起使用。

示例

- 使用 Lambda 函数连接到 Amazon RDS 数据库
- 通过 Kinesis 触发器调用 Lambda 函数
- 通过 DynamoDB 触发器调用 Lambda 函数
- 通过 Amazon DocumentDB 触发器调用 Lambda 函数
- 通过 Amazon MSK 触发器调用 Lambda 函数
- 通过 Amazon S3 触发器调用 Lambda 函数

- 通过 Amazon SNS 触发器调用 Lambda 函数
- 通过 Amazon SQS 触发器调用 Lambda 函数
- 通过 Kinesis 触发器报告 Lambda 函数批处理项目失败
- 通过 DynamoDB 触发器报告 Lambda 函数批处理项目失败
- 报告使用 Amazon SQS 触发器进行 Lambda 函数批处理项目失败

使用 Lambda 函数连接到 Amazon RDS 数据库

以下代码示例显示如何实现连接到 RDS 数据库的 Lambda 函数。该函数发出一个简单的数据库请求并返回结果。

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 Go 连接到 Amazon RDS 数据库。

```
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"
 "os"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
```

```
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = os.Getenv("DatabaseName")
 var dbUser string = os.Getenv("DatabaseUser")
 var dbHost string = os.Getenv("DBHost") // Add hostname without https
 var dbPort int = os.Getenv("Port")      // Add port number
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
 var region string = os.Getenv("AWS_REGION")

 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }

 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)
```

```
 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 Java 连接到 Amazon RDS 数据库。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rdsdata.RdsDataClient;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementRequest;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementResponse;
import software.amazon.awssdk.services.rdsdata.model.Field;
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class RdsLambdaHandler implements
 RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
 event, Context context) {
        APIGatewayProxyResponseEvent response = new
 APIGatewayProxyResponseEvent();

        try {
            // Obtain auth token
            String token = createAuthToken();

            // Define connection configuration
            String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true",
                    System.getenv("ProxyHostName"),
                    System.getenv("Port"),
                    System.getenv("DBName"));

            // Establish a connection to the database
            try (Connection connection =
 DriverManager.getConnection(connectionString, System.getenv("DBUserName"),
 token);
                 PreparedStatement statement =
 connection.prepareStatement("SELECT ? + ? AS sum")) {

                statement.setInt(1, 3);
                statement.setInt(2, 2);

                try (ResultSet resultSet = statement.executeQuery()) {
                    if (resultSet.next()) {
                        int sum = resultSet.getInt("sum");
                        response.setStatusCode(200);
                        response.setBody("The selected sum is: " + sum);
                    }
                }
            }
```

```
        } catch (Exception e) {
            response.setStatusCode(500);
            response.setBody("Error: " + e.getMessage());
        }

        return response;
    }

    private String createAuthToken() {
        // Create RDS Data Service client
        RdsDataClient rdsDataClient = RdsDataClient.builder()
                .region(Region.of(System.getenv("AWS_REGION")))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        // Define authentication request
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
                .resourceArn(System.getenv("ProxyHostName"))
                .secretArn(System.getenv("DBUserName"))
                .database(System.getenv("DBName"))
                .sql("SELECT 'RDS IAM Authentication'")
                .build();

        // Execute request and obtain authentication token
        ExecuteStatementResponse response =
 rdsDataClient.executeStatement(request);
        Field tokenField = response.records().get(0).get(0);

        return tokenField.stringValue();
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 JavaScript 连接到 Amazon RDS 数据库。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
```

```
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

在 Lambda 函数中使用 TypeScript 连接到 Amazon RDS 数据库。

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that
 the DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!


async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });
```

```
    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
            host: proxy_host_name,
            user: db_user_name,
            password: token,
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
        const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
        console.log('result:', rows);
        return rows;
    }
    catch (err) {
        console.log(err);
    }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number;
 body: string }> => {
    // Execute database flow
    const result = await dbOps();

    // Return error is result is undefined
    if (result == undefined)
        return {
            statusCode: 500,
            body: JSON.stringify(`Error with connection to DB host`)
        }

    // Return result
    return {
        statusCode: 200,
        body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
    };
};
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 PHP 连接到 Amazon RDS 数据库。

```
<?php
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;
use Aws\Rds\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }


    private function getAuthToken(): string {
        // Define connection authentication parameters
        $dbConnection = [
            'hostname' => getenv('DB_HOSTNAME'),
            'port' => getenv('DB_PORT'),
            'username' => getenv('DB_USERNAME'),
            'region' => getenv('AWS_REGION'),
        ];
```

```
        // Create RDS AuthTokenGenerator object
        $generator = new
AuthTokenGenerator(CredentialProvider::defaultProvider());

        // Request authorization token from RDS, specifying the username
        return $generator->createToken(
            $dbConnection['hostname'] . ':' . $dbConnection['port'],
            $dbConnection['region'],
            $dbConnection['username']
        );
    }

    private function getQueryResults() {
        // Obtain auth token
        $token = $this->getAuthToken();

        // Define connection configuration
        $connectionConfig = [
            'host' => getenv('DB_HOSTNAME'),
            'user' => getenv('DB_USERNAME'),
            'password' => $token,
            'database' => getenv('DB_NAME'),
        ];

        // Create the connection to the DB
        $conn = new PDO(

"mysql:host={$connectionConfig['host']};dbname={$connectionConfig['database']}",
            $connectionConfig['user'],
            $connectionConfig['password'],
            [
                PDO::MYSQL_ATTR_SSL_CA => '/path/to/rds-ca-2019-root.pem',
                PDO::MYSQL_ATTR_SSL_VERIFY_SERVER_CERT => true,
            ]
        );

        // Obtain the result of the query
        $stmt = $conn->prepare('SELECT ?+? AS sum');
        $stmt->execute([3, 2]);

        return $stmt->fetch(PDO::FETCH_ASSOC);
    }
```

```
    /**
     * @param mixed $event
     * @param Context $context
     * @return array
     */
    public function handle(mixed $event, Context $context): array
    {
        $this->logger->info("Processing query");

        // Execute database flow
        $result = $this->getQueryResults();

        return [
            'sum' => $result['sum']
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 Python 连接到 Amazon RDS 数据库。

```
import json
import os
import boto3
import pymysql

# RDS settings
proxy_host_name = os.environ['PROXY_HOST_NAME']
port = int(os.environ['PORT'])
db_name = os.environ['DB_NAME']
```

```
db_user_name = os.environ['DB_USER_NAME']
aws_region = os.environ['AWS_REGION']


# Fetch RDS Auth Token
def get_auth_token():
    client = boto3.client('rds')
    token = client.generate_db_auth_token(
        DBHostname=proxy_host_name,
        Port=port
        DBUsername=db_user_name
        Region=aws_region
    )
    return token

def lambda_handler(event, context):
    token = get_auth_token()
    try:
        connection = pymysql.connect(
            host=proxy_host_name,
            user=db_user_name,
            password=token,
            db=db_name,
            port=port,
            ssl={'ca': 'Amazon RDS'}  # Ensure you have the CA bundle for SSL
 connection
        )

        with connection.cursor() as cursor:
            cursor.execute('SELECT %s + %s AS sum', (3, 2))
            result = cursor.fetchone()

        return result

    except Exception as e:
        return (f"Error: {str(e)}")  # Return an error message if an exception
 occurs
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 Ruby 连接到 Amazon RDS 数据库。

```
# Ruby code here.

require 'aws-sdk-rds'
require 'json'
require 'mysql2'

def lambda_handler(event:, context:)
  endpoint = ENV['DBEndpoint'] # Add the endpoint without https"
  port = ENV['Port']           # 3306
  user = ENV['DBUser']
  region = ENV['DBRegion']     # 'us-east-1'
  db_name = ENV['DBName']

  credentials = Aws::Credentials.new(
    ENV['AWS_ACCESS_KEY_ID'],
    ENV['AWS_SECRET_ACCESS_KEY'],
    ENV['AWS_SESSION_TOKEN']
  )
  rds_client = Aws::RDS::AuthTokenGenerator.new(
    region: region,
    credentials: credentials
  )

  token = rds_client.auth_token(
    endpoint: endpoint+ ':' + port,
    user_name: user,
    region: region
  )

  begin
    conn = Mysql2::Client.new(
```

```
      host: endpoint,
      username: user,
      password: token,
      port: port,
      database: db_name,
      sslca: '/var/task/global-bundle.pem',
      sslverify: true,
      enable_cleartext_plugin: true
    )
    a = 3
    b = 2
    result = conn.query("SELECT #{a} + #{b} AS sum").first['sum']
    puts result
    conn.close
    {
      statusCode: 200,
      body: result.to_json
    }
  rescue => e
    puts "Database connection failed due to #{e}"
  end
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

在 Lambda 函数中使用 Rust 连接到 Amazon RDS 数据库。

```
use aws_config::BehaviorVersion;
use aws_credential_types::provider::ProvideCredentials;
use aws_sigv4::{
    http_request::{sign, SignableBody, SignableRequest, SigningSettings},
    sign::v4,
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
```

```
use serde_json::{json, Value};
use sqlx::postgres::PgConnectOptions;
use std::env;
use std::time::{Duration, SystemTime};

const RDS_CERTS: &[u8] = include_bytes!("global-bundle.pem");

async fn generate_rds_iam_token(
    db_hostname: &str,
    port: u16,
    db_username: &str,
) -> Result<String, Error> {
    let config = aws_config::load_defaults(BehaviorVersion::v2024_03_28()).await;

    let credentials = config
        .credentials_provider()
        .expect("no credentials provider found")
        .provide_credentials()
        .await
        .expect("unable to load credentials");
    let identity = credentials.into();
    let region = config.region().unwrap().to_string();

    let mut signing_settings = SigningSettings::default();
    signing_settings.expires_in = Some(Duration::from_secs(900));
    signing_settings.signature_location =
 aws_sigv4::http_request::SignatureLocation::QueryParams;

    let signing_params = v4::SigningParams::builder()
        .identity(&identity)
        .region(&region)
        .name("rds-db")
        .time(SystemTime::now())
        .settings(signing_settings)
        .build()?;

    let url = format!(
        "https://{db_hostname}:{port}/?Action=connect&DBUser={db_user}",
        db_hostname = db_hostname,
        port = port,
        db_user = db_username
    );

    let signable_request =
```

```
        SignableRequest::new("GET", &url, std::iter::empty(),
 SignableBody::Bytes(&[]))
            .expect("signable request");

    let (signing_instructions, _signature) =
        sign(signable_request, &signing_params.into())?.into_parts();

    let mut url = url::Url::parse(&url).unwrap();
    for (name, value) in signing_instructions.params() {
        url.query_pairs_mut().append_pair(name, &value);
    }

    let response = url.to_string().split_off("https://".len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(handler)).await
}

async fn handler(_event: LambdaEvent<Value>) -> Result<Value, Error> {
    let db_host = env::var("DB_HOSTNAME").expect("DB_HOSTNAME must be set");
    let db_port = env::var("DB_PORT")
        .expect("DB_PORT must be set")
        .parse::<u16>()
        .expect("PORT must be a valid number");
    let db_name = env::var("DB_NAME").expect("DB_NAME must be set");
    let db_user_name = env::var("DB_USERNAME").expect("DB_USERNAME must be set");

    let token = generate_rds_iam_token(&db_host, db_port, &db_user_name).await?;

    let opts = PgConnectOptions::new()
        .host(&db_host)
        .port(db_port)
        .username(&db_user_name)
        .password(&token)
        .database(&db_name)
        .ssl_root_cert_from_pem(RDS_CERTS.to_vec())
        .ssl_mode(sqlx::postgres::PgSslMode::Require);

    let pool = sqlx::postgres::PgPoolOptions::new()
        .connect_with(opts)
```

```
        .await?;

    let result: i32 = sqlx::query_scalar("SELECT $1 + $2")
        .bind(3)
        .bind(2)
        .fetch_one(&pool)
        .await?;

    println!("Result: {:?}", result);

    Ok(json!({
        "statusCode": 200,
        "content-type": "text/plain",
        "body": format!("The selected sum is: {result}")
    }))
}
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 Kinesis 触发器调用 Lambda 函数

以下代码示例演示了如何实现一个 Lambda 函数，该函数接收通过接收来自 Kinesis 流的记录而触发的事件。该函数检索 Kinesis 有效负载，将 Base64 解码，并记录下记录内容。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 .NET 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
```

```
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }
```

```
    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
```

```
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Java 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
```

```
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

适用于 JavaScript 的 SDK (v3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 JavaScript 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
```

```
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

通过 TypeScript 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
```

```
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 PHP 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';
```

```
class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 Kinesis 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 Kinesis 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
```

```
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Rust 将 Kinesis 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }
```

```
    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 DynamoDB 触发器调用 Lambda 函数

以下代码示例演示了如何实现一个 Lambda 函数，该函数接收通过接收来自 DynamoDB 流的记录而触发的事件。该函数检索 DynamoDB 有效负载，并记录下记录内容。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 .NET 将 DynamoDB 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
```

```
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 DynamoDB 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }
```

```
 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Java 将 DynamoDB 事件与 Lambda 结合使用。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
```

```
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

适用于 JavaScript 的 SDK (v3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 JavaScript 将 DynamoDB 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

使用 TypeScript 将 DynamoDB 事件与 Lambda 结合使用。

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 PHP 将 DynamoDB 事件与 Lambda 结合使用。

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;
```

```
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Python 将 DynamoDB 事件与 Lambda 结合使用。

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 DynamoDB 事件与 Lambda 结合使用。

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Rust 将 DynamoDB 事件与 Lambda 结合使用。

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
```

```
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())
```

```
}
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 Amazon DocumentDB 触发器调用 Lambda 函数

以下代码示例演示了如何实现一个 Lambda 函数，该函数接收通过接收来自 DynamoDB 更改流的记录而触发的事件。该函数检索 DocumentDB 有效负载，并记录下记录内容。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 .NET 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
using Amazon.Lambda.Core;
using System.Text.Json;
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;
//Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaDocDb;

public class Function
{

     /// <summary>
    /// Lambda function entry point to process Amazon DocumentDB events.
    /// </summary>
```

```
    /// <param name="event">The Amazon DocumentDB event.</param>
    /// <param name="context">The Lambda context object.</param>
    /// <returns>A string to indicate successful processing.</returns>
    public string FunctionHandler(Event evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Events)
        {
            ProcessDocumentDBEvent(record, context);
        }

        return "OK";
    }

     private void ProcessDocumentDBEvent(DocumentDBEventRecord record,
ILambdaContext context)
    {

        var eventData = record.Event;
        var operationType = eventData.OperationType;
        var databaseName = eventData.Ns.Db;
        var collectionName = eventData.Ns.Coll;
        var fullDocument = JsonSerializer.Serialize(eventData.FullDocument, new
JsonSerializerOptions { WriteIndented = true });

        context.Logger.LogLine($"Operation type: {operationType}");
        context.Logger.LogLine($"Database: {databaseName}");
        context.Logger.LogLine($"Collection: {collectionName}");
        context.Logger.LogLine($"Full document:\n{fullDocument}");
    }



    public class Event
    {
        [JsonPropertyName("eventSourceArn")]
        public string EventSourceArn { get; set; }

        [JsonPropertyName("events")]
        public List<DocumentDBEventRecord> Events { get; set; }

        [JsonPropertyName("eventSource")]
        public string EventSource { get; set; }
    }
```

```
public class DocumentDBEventRecord
{
    [JsonPropertyName("event")]
    public EventData Event { get; set; }
}

public class EventData
{
    [JsonPropertyName("_id")]
    public IdData Id { get; set; }

    [JsonPropertyName("clusterTime")]
    public ClusterTime ClusterTime { get; set; }

    [JsonPropertyName("documentKey")]
    public DocumentKey DocumentKey { get; set; }

    [JsonPropertyName("fullDocument")]
    public Dictionary<string, object> FullDocument { get; set; }

    [JsonPropertyName("ns")]
    public Namespace Ns { get; set; }

    [JsonPropertyName("operationType")]
    public string OperationType { get; set; }
}

public class IdData
{
    [JsonPropertyName("_data")]
    public string Data { get; set; }
}

public class ClusterTime
{
    [JsonPropertyName("$timestamp")]
    public Timestamp Timestamp { get; set; }
}

public class Timestamp
{
    [JsonPropertyName("t")]
    public long T { get; set; }
```

```
        [JsonPropertyName("i")]
        public int I { get; set; }
    }

    public class DocumentKey
    {
        [JsonPropertyName("_id")]
        public Id Id { get; set; }
    }

    public class Id
    {
        [JsonPropertyName("$oid")]
        public string Oid { get; set; }
    }

    public class Namespace
    {
        [JsonPropertyName("db")]
        public string Db { get; set; }

        [JsonPropertyName("coll")]
        public string Coll { get; set; }
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
package main
```

```
import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 JavaScript 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

使用 TypeScript 将 Amazon DocumentDB 事件与 Lambda 结合使用

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from 'aws-
lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
```

```
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 PHP 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
<?php

require __DIR__.'/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Handler;

class DocumentDBEventHandler implements Handler
{
    public function handle($event, Context $context): string
    {

        $events = $event['events'] ?? [];
```

```
        foreach ($events as $record) {
            $this->logDocumentDBEvent($record['event']);
        }
        return 'OK';
    }

    private function logDocumentDBEvent($event): void
    {
        // Extract information from the event record

        $operationType = $event['operationType'] ?? 'Unknown';
        $db = $event['ns']['db'] ?? 'Unknown';
        $collection = $event['ns']['coll'] ?? 'Unknown';
        $fullDocument = $event['fullDocument'] ?? [];

        // Log the event details

        echo "Operation type: $operationType\n";
        echo "Database: $db\n";
        echo "Collection: $collection\n";
        echo "Full document: " . json_encode($fullDocument, JSON_PRETTY_PRINT) .
 "\n";
    }
}
return new DocumentDBEventHandler();
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
```

```
            log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Ruby 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}
```

```
  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Rust 将 Amazon DocumentDB 事件与 Lambda 结合使用。

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::documentdb::{DocumentDbEvent, DocumentDbInnerEvent},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<DocumentDbEvent>) ->Result<(),
 Error> {

    tracing::info!("Event Source ARN: {:?}", event.payload.event_source_arn);
    tracing::info!("Event Source: {:?}", event.payload.event_source);

    let records = &event.payload.events;
```

```
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_document_db_event(record);
    }

    tracing::info!("Document db records processed");

    // Prepare the response
    Ok(())

}

fn log_document_db_event(record: &DocumentDbInnerEvent)-> Result<(), Error>{
    tracing::info!("Change Event: {:?}", record.event);

    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 Amazon MSK 触发器调用 Lambda 函数

以下代码示例展示了如何实现 Lambda 函数，该函数接收通过接收来自 Amazon MSK 集群的记录而触发的事件。该函数检索 MSK 有效负载，并记录下记录内容。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 .NET 将 Amazon MSK 事件与 Lambda 结合使用。

```
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KafkaEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace MSKLambda;

public class Function
{


    /// <param name="input">The event for the Lambda function handler to
 process.</param>
    /// <param name="context">The ILambdaContext that provides methods for
 logging and describing the Lambda environment.</param>
    /// <returns></returns>
    public void FunctionHandler(KafkaEvent evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Records)
```

```
        {
            Console.WriteLine("Key:" + record.Key);
            foreach (var eventRecord in record.Value)
            {
                var valueBytes = eventRecord.Value.ToArray();
                var valueText = Encoding.UTF8.GetString(valueBytes);

                Console.WriteLine("Message:" + valueText);
            }
        }
    }


}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Go 将 Amazon MSK 事件与 Lambda 结合使用。

```
package main

import (
 "encoding/base64"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.KafkaEvent) {
 for key, records := range event.Records {
  fmt.Println("Key:", key)
```

```
  for _, record := range records {
   fmt.Println("Record:", record)

   decodedValue, _ := base64.StdEncoding.DecodeString(record.Value)
   message := string(decodedValue)
   fmt.Println("Message:", message)
  }
 }
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Java 将 Amazon MSK 事件与 Lambda 结合使用。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent.KafkaEventRecord;

import java.util.Base64;
import java.util.Map;

public class Example implements RequestHandler<KafkaEvent, Void> {

    @Override
    public Void handleRequest(KafkaEvent event, Context context) {
        for (Map.Entry<String, java.util.List<KafkaEventRecord>> entry :
 event.getRecords().entrySet()) {
```

```
            String key = entry.getKey();
            System.out.println("Key: " + key);

            for (KafkaEventRecord record : entry.getValue()) {
                System.out.println("Record: " + record);

                byte[] value = Base64.getDecoder().decode(record.getValue());
                String message = new String(value);
                System.out.println("Message: " + message);
            }
        }

        return null;
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 JavaScript 将 Amazon MSK 事件与 Lambda 结合使用。

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
```

```
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 PHP 将 Amazon MSK 事件与 Lambda 结合使用。

```
<?php
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

// using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kafka\KafkaEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): void
    {
        $kafkaEvent = new KafkaEvent($event);
```

```
        $this->logger->info("Processing records");
        $records = $kafkaEvent->getRecords();

        foreach ($records as $record) {
            try {
                $key = $record->getKey();
                $this->logger->info("Key: $key");

                $values = $record->getValue();
                $this->logger->info(json_encode($values));

                foreach ($values as $value) {
                    $this->logger->info("Value: $value");
                }

            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Python 将 Amazon MSK 事件与 Lambda 结合使用。

```
import base64
```

```
def lambda_handler(event, context):
    # Iterate through keys
    for key in event['records']:
        print('Key:', key)
        # Iterate through records
        for record in event['records'][key]:
            print('Record:', record)
            # Decode base64
            msg = base64.b64decode(record['value']).decode('utf-8')
            print('Message:', msg)
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 Amazon MSK 事件与 Lambda 结合使用。

```
require 'base64'

def lambda_handler(event:, context:)
  # Iterate through keys
  event['records'].each do |key, records|
    puts "Key: #{key}"

    # Iterate through records
    records.each do |record|
      puts "Record: #{record}"

      # Decode base64
      msg = Base64.decode64(record['value'])
      puts "Message: #{msg}"
    end
  end
```

```
end
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 Amazon S3 触发器调用 Lambda 函数

以下代码示例演示了如何实现一个 Lambda 函数，该函数接收通过将对象上传到 S3 存储桶而触发的事件。该函数从事件参数中检索 S3 存储桶名称和对象密钥，并调用 Amazon S3 API 来检索和记录对象的内容类型。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 .NET 将 S3 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace S3Integration
{
    public class Function
    {
```

```
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key = HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket, key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 S3 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
```

```
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Java 将 S3 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotifi

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey + " of
 type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String bucket,
 String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 JavaScript 将 S3 事件与 Lambda 结合使用。

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g,
 ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

使用 TypeScript 将 S3 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
```

```
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, '
 '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure
 they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 PHP 将 S3 事件与 Lambda 结合使用。

```
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';
```

```
class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as this
 function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 S3 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
 encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and
 your bucket is in the same region as this function.'.format(key, bucket))
        raise e
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 S3 事件与 Lambda 结合使用。

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']['key'],
 Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they exist
 and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Rust 将 S3 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
```

```
    tracing::info!(records = ?evt.payload.records.len(), "Received request from
 SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket = evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket
 name to exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object key to
 exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 Amazon SNS 触发器调用 Lambda 函数

以下代码示例演示了如何实现一个 Lambda 函数，该函数接收通过接收来自 SNS 主题的消息而触发的事件。该函数从事件参数检索消息并记录每条消息的内容。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 .NET 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");
```

```
            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
```

```
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Java 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;
```

```
    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 JavaScript 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

使用 TypeScript 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
```

```
    throw err;
  }
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 PHP 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
```

```
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 SNS 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here
```

```
    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 SNS 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Rust 将 SNS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 Amazon SQS 触发器调用 Lambda 函数

以下代码示例演示了如何实现一个 Lambda 函数，该函数接收通过接收来自 SQS 队列的消息而触发的事件。该函数从事件参数检索消息并记录每条消息的内容。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 .NET 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Java 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }
```

```
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 JavaScript 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

通过 TypeScript 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";
```

```
export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 PHP 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
```

```
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 SQS 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 Ruby 将 SQS 事件与 Lambda 结合使用。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
```

```
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

通过 Rust 将 SQS 事件与 Lambda 结合使用。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
```

```
        .init();

    run(service_fn(function_handler)).await
}
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 Kinesis 触发器报告 Lambda 函数批处理项目失败

以下代码示例显示如何为接收来自 Kinesis 流的事件的 Lambda 函数实现部分批处理响应。该函数在响应中报告批处理项目失败，并指示 Lambda 稍后重试这些消息。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 .NET 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
```

```
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
```

```
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告通过 Go 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
```

```
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Java 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Javascript 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
```

```
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

报告使用 TypeScript 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
```

```
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告通过 PHP 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
```

```
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Python 进行 Lambda Kinesis 批处理项目失败。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告通过 Ruby 进行 Lambda Kinesis 批处理项目失败。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end
```

```
def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告通过 Rust 进行 Lambda Kinesis 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
```

```
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
```

```
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 DynamoDB 触发器报告 Lambda 函数批处理项目失败

以下代码示例显示如何为接收来自 DynamoDB 流的事件的 Lambda 函数实现部分批处理响应。该函数在响应中报告批处理项目失败，并指示 Lambda 稍后重试这些消息。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 .NET 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Go 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
```

```
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Java 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 Serializable> {

    @Override
```

```
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

使用 JavaScript 报告 Lambda 的 DynamoDB 批处理项目失败。

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

使用 TypeScript 报告 Lambda 的 DynamoDB 批处理项目失败。

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }
```

```
  return { batchItemFailures: batchItemFailures };
};
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 PHP 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
```

```
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Python 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Ruby 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Rust 通过 Lambda 进行 DynamoDB 批处理项目失败。

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;
```

```
    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }
```

```
    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

报告使用 Amazon SQS 触发器进行 Lambda 函数批处理项目失败

以下代码示例显示如何为接收来自 SQS 队列的事件的 Lambda 函数实现部分批处理响应。该函数在响应中报告批处理项目失败，并指示 Lambda 稍后重试这些消息。

.NET

AWS SDK for .NET

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 .NET 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
```

```
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

适用于 Go V2 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Go 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "encoding/json"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {

  if /* Your message processing condition here */ {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
```

```
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Java 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context) {

        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();
        String messageId = "";
        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
```

```
                //process your message
                messageId = message.getMessageId();
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(messageId));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

适用于 JavaScript 的 SDK（v3）

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 JavaScript 进行 Lambda SQS 批处理项目失败。

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
    const batchItemFailures = [];
    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
    console.log(`Processed message: ${record.body}`);
```

```
}
```

报告使用 TypeScript 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure, SQSRecord }
 from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

适用于 PHP 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 PHP 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS records");
```

```
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Python 进行 Lambda SQS 批处理项目失败。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event["Records"]:
            try:
                # process message
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

适用于 Ruby 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Ruby 进行 Lambda SQS 批处理项目失败。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
  end
end
```

Rust

适用于 Rust 的 SDK

> **Note**
>
> 查看 GitHub，了解更多信息。在无服务器示例存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Rust 进行 Lambda SQS 批处理项目失败。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
```

```
    run(service_fn(function_handler)).await
}
```

有关 AWS SDK 开发人员指南和代码示例的完整列表，请参阅 将 Lambda 与 AWS SDK 配合使用。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

Lambda 配额

> **Important**
>
> 新的 AWS 账户 减少了并发和内存配额。AWS 将根据您的使用情况自动提高这些配额。

计算和存储

Lambda 为可用来运行和存储函数的计算和存储资源量设置配额。并发执行和存储限额按 AWS 区域适用。弹性网络接口（ENI）限额按虚拟私有云（VPC）适用，不按区域适用。以下限额可以在默认值的基础上增加。有关更多信息，请参阅 Service Quotas 用户指南中的请求增加配额。

资源	默认配额	最多可增加到
并发执行	1000	数十万
用于上传的功能（.zip 文件存档）和层的存储。每个函数版本和层版本都会消耗存储。 有关管理代码存储的最佳实践，请参阅 Serverless Land 中的 Monitoring Lambda code storage。	75 GB	千吉字节
用于定义为容器映像的函数的存储 这些映像存储在 Amazon ECR 中。	请参阅 Amazon ECR 服务限额。	
每个虚拟私有云（VPC）的弹性网络接口 **Note** 此配额与其他服务（如 Amazon Elastic File System (Amazon EFS)）共享。请参阅 Amazon VPC 配额。	500	千

有关并发以及 Lambda 如何根据流量扩展函数并发的详细信息，请参阅 了解 Lambda 函数扩展。

函数配置、部署和执行

以下配额适用于函数配置、部署和执行。除非另有说明，否则无法对其进行更改。

> **Note**
>
> Lambda 文档、日志消息和控制台中使用缩写 MB（而不是 MiB）来表示 1024 KB。

资源	配额
函数内存分配	128 MB 到 10240 MB，以 1 MB 为增量。 注意：Lambda 根据配置的内存量按比例分配 CPU 处理能力。请使用 Memory (MB)（内存 (MB)）设置增加分配给函数的内存和 CPU 处理能力。大小为 1769MB 时，函数的处理能力与一个 vCPU 等效。
函数超时	900 秒（15 分钟）
函数环境变量	对于与函数关联的所有环境变量总共为 4 KB
函数基于资源的策略	20 KB
函数层	五层
函数并发扩展限制	对于每个函数，每 10 秒 1,000 个执行环境
调用负载（请求和响应）	请求和响应各为 6 MB（同步） 每个流式处理响应 20 MB（同步。流式处理响应的有效负载大小可以从默认值开始增加。请联系 AWS Support 以进一步查询。）

资源	配额
	256 KB（异步） 请求行和标头值的总组合大小为 1MB
流式处理响应的带宽	函数响应的前 6MB 无上限 对于大于 6MB 的响应，其余响应为 2Mbps
部署程序包（.zip 文件存档）大小	50 MB（通过 Lambda API 或软件开发工具包上传时已压缩）。上载较大文件请使用 Amazon S3。 50 MB（通过 Lambda 控制台上传时） 250 MB 部署包内容的最大大小，包括层和自定义运行时。（已解压缩）
容器映像设置大小	16 KB
容器映像代码包大小	10GB（最大未压缩图像大小，包括所有图层）
测试事件（控制台编辑器）	10
/tmp 目录存储	介于 512 MB 与 10,240 MB 之间，以 1 MB 为增量
文件描述符	1024
执行进程/线程	1024

Lambda API 请求

以下配额与 Lambda API 请求相关联。

资源	限额
每个区域的每个函数的调用请求数（同步）	您的执行环境的每个实例每秒最多处理 10 个请求。换言之，总调用限制是并发限制的 10 倍。请参阅 了解 Lambda 函数扩展。
每个区域的每个函数的调用请求数（异步）	您的执行环境的每个实例都可以处理无限数量的请求。换言之，总调用限制仅基于函数可用的并发。请参阅 了解 Lambda 函数扩展。
每个函数版本或别名的调用请求数（每秒请求数）	10 x 分配的预配置并发 Note 此配额仅适用于使用预置并发的函数。
GetFunction API 请求	每秒 100 个请求。无法增加。
GetPolicy API 请求	每秒 15 个请求。无法增加。
控制面板 API 请求的剩余部分（不包括调用、GetFunction 和 GetPolicy 请求）	所有 API 每秒 15 个请求（而不是每个 API 每秒 15 个请求）。无法增加。

其他服务

其他服务（如 AWS Identity and Access Management (IAM)、Amazon CloudFront (Lambda@Edge) 和 Amazon Virtual Private Cloud (Amazon VPC)）的配额可能会影响您的 Lambda 函数。有关更多信息，请参阅《Amazon Web Services 一般参考》中的 AWS 服务 限额，以及 使用来自其 AWS 他服务的事件调用 Lambda。

文档历史记录

下表介绍了 2018 年 5 月之后对 AWS Lambda 开发人员指南的重要更改。如需有关文档更新的通知，您可以订阅 RSS 源。

变更	说明	日期
在新区域支持 SnapStart	Lambda SnapStart 现已在以下区域推出：欧洲（西班牙）、欧洲（苏黎世）、亚太地区（墨尔本）、亚太地区（海得拉巴）和中东（阿联酋）。	2024 年 1 月 12 日
AWS 托管式策略更新	服务配额更新了现有的 AWS 托管策略（`AWSLambda VPCAccessExecution Role`）。	2024 年 1 月 5 日
Python 3.12 运行时系统	Lambda 现在支持 Python 3.12 作为托管运行时系统和容器基础映像。有关更多信息，请参阅 AWS Compute Blog 上的 Python 3.12 runtime now available in AWS Lambda。	2023 年 12 月 14 日
Java 21 运行时系统	Lambda 现在支持 Java 21 作为托管运行时系统和容器基础映像（`java21`）。	2023 年 11 月 16 日
Node.js 20.x 运行时系统	Lambda 现在支持 Node.js 20 作为托管运行时系统和容器基础映像（`nodejs20.x`）。有关更多信息，请参阅 AWS 计算博客上的 AWS Lambda 现已推出 Node.js 20.x 运行时系统。	2023 年 11 月 14 日

provided.al2023 运行时系统	Lambda 现在支持 Amazon Linux 2023 作为托管运行时系统和容器基础映像。有关更多信息，请参阅 AWS 计算博客上的AWS Lambda 的 Amazon Linux 2023 运行时系统简介。	2023 年 11 月 9 日
对双堆栈子网的 IPv6 支持	Lambda 现在支持到双堆栈子网的出站 IPv6 流量。有关更多信息，请参阅 IPv6 支持。	2023 年 10 月 12 日
测试无服务器函数和应用程序	了解在云端调试和自动执行测试无服务器函数的技术。现在，Python 和 Typescript 语言部分中包含测试章节和资源。有关详细信息，请参阅测试无服务器函数和应用程序。	2023 年 6 月 16 日
Ruby 3.2 运行时系统	Lambda 现在支持适用于 Ruby 3.2 的新运行时系统。有关更多信息，请参阅使用 Ruby 构建 Lambda 函数。	2023 年 6 月 7 日
响应流式处理	Lambda 现支持来自函数的流式处理响应。有关更多信息，请参阅配置 Lambda 函数以流式处理响应。	2023 年 4 月 6 日
异步调用指标	Lambda 发布异步调用指标。有关更多信息，请参阅异步调用指标。	2023 年 2 月 9 日

运行时版本控制	Lambda 发布了新的运行时版本，其中包括安全更新、错误修复和新功能。现在，您可以控制您的函数何时更新到新的运行时版本。有关更多信息，请参阅 Lambda runtime updates（Lambda 运行时更新）。	2023 年 1 月 23 日
Lambda SnapStart	使用 Lambda SnapStart 可以缩短 Java 函数的启动时间，而无需预置额外资源或实施复杂的性能优化。有关更多信息，请参阅 Improving startup performance with Lambda SnapStart（使用 Lambda SnapStart 提高启动性能）。	2022 年 11 月 28 日
Node.js 18 运行时	Lambda 现支持 Node.js 18 新运行时。Node.js 18 使用 Amazon Linux 2。有关详细信息，请参阅使用 Node.js 构建 Lambda 函数。	2022 年 11 月 18 日
lambda:SourceFunctionArn 条件键	对于 AWS 资源，`lambda:SourceFunctionArn` 条件键通过 Lambda 函数的 ARN 筛选对资源的访问。有关详细信息，请参阅使用 Lambda 执行环境凭证。	2022 年 7 月 1 日
Node.js 16 运行时	Lambda 现支持 Node.js 16 新运行时。Node.js 16 使用 Amazon Linux 2。有关详细信息，请参阅使用 Node.js 构建 Lambda 函数。	2022 年 5 月 11 日

Lambda 函数 URL	Lambda 现在支持函数 URL，即 Lambda 函数的专用 HTTP(S) 端点。有关详细信息，请参阅 Lambda 函数 URL。	2022 年 4 月 6 日
AWS Lambda 控制台中的共享测试事件	Lambda 现在支持与同一 AWS 账户 中的其他用户共享测试事件。有关详细信息，请参阅在控制台中测试 Lambda 函数。	2022 年 3 月 16 日
基于资源的策略中的 PrincipalOrgId	Lambda 现在支持向 AWS Organizations 中的企业授予权限。有关详细信息，请参阅 使用 AWS Lambda 的基于资源的策略。	2022 年 3 月 11 日
.NET 6 运行时	Lambda 现在支持 .NET 6 的新运行时。有关详细信息，请参阅 Lambda 运行时。	2022 年 2 月 23 日
Kinesis、DynamoDB 和 Amazon SQS 事件源的事件筛选	Lambda 现在支持对 Kinesis、DynamoDB 和 Amazon SQS 事件源进行事件筛选。有关详细信息，请参阅 Lambda 事件筛选。	2021 年 11 月 24 日
用于 Amazon MSK 和自行管理的 Apache Kafka 事件源的 mTLS 身份验证	Lambda 现在支持对 Amazon MSK 和自行管理的 Apache Kafka 事件源进行 mTLS 身份验证。有关详细信息，请参阅将 Lambda 与 Amazon MSK 结合使用。	2021 年 11 月 19 日
Graviton2 上的 Lambda	Lambda 现在为使用 arm64 架构的函数支持 Graviton2。有关详细信息，请参阅 Lambda 指令集架构。	2021 年 9 月 29 日

Python 3.9 运行时	Lambda 现在支持 Python 3.9 的新运行时。有关详细信息，请参阅 Lambda 运行时。	2021 年 8 月 16 日
Node.js、Python 和 Java 的新运行时版本	新运行时版本可用于 Node.js、Python 和 Java。有关详细信息，请参阅 Lambda 运行时。	2021 年 7 月 21 日
Lambda 现在支持 RabbitMQ 作为事件源	Lambda 现支持 Amazon MQ for RabbitMQ 作为事件源。Amazon MQ 是 Apache ActiveMQ 和 RabbitMQ 的托管式消息代理服务，您可轻松地在云中设置和操作消息代理。有关详细信息，请参阅将 Lambda 与 Amazon MQ 结合使用。	2021 年 7 月 7 日
Lambda 上的自行管理的 Kafka SASL/PLAY 身份验证	目前，SASL/PLAIN 是 Lambda 上自行管理的 Kafka 事件源支持的身份验证机制。已在其自行管理的 Kafka 集群上使用 SAL/PLAY 的客户现在可以轻松地使用 Lambda 构建消费者应用程序，而无需修改身份验证方式。有关详细信息，请参阅将 Lambda 与自行管理的 Apache Kafka 结合使用。	2021 年 6 月 29 日
Lambda 扩展 API	Lambda 扩展通用版。使用扩展来增强 Lambda 函数。可以使用 Lambda 合作伙伴提供的扩展，也可以创建自己的 Lambda 扩展。有关详细信息，请参阅 Lambda 扩展 API。	2021 年 5 月 24 日

新增 Lambda 控制台体验	为了提升性能和一致性，已对 Lambda 控制台进行了重新设计。	2021 年 3 月 2 日
Node.js 14 运行时	Lambda 现支持 Node.js 14 新运行时。Node.js 14 使用 Amazon Linux 2。有关详细信息，请参阅使用 Node.js 构建 Lambda 函数。	2021 年 1 月 27 日
Lambda 容器镜像	Lambda 现支持定义为容器镜像的函数。您可以将容器工具的灵活性与 Lambda 的敏捷性和操作简易性相结合，以构建应用程序。有关详细信息，请参阅将容器镜像与 Lambda 结合使用。	2020 年 12 月 1 日
Lambda 函数的代码签名	Lambda 现支持代码签名。管理员可以将 Lambda 函数配置为在部署时仅接受签名的代码。Lambda 会检查签名以确保代码未经更改或篡改。此外，Lambda 可确保在接受部署之前代码已由可信开发人员签名。有关详细信息，请参阅为 Lambda 配置代码签名。	2020 年 11 月 23 日
预览：Lambda Rambda Runtime Logs API	Lambda 现支持 Runtime Logs API。Lambda 扩展可使用 Logs API 订阅执行环境中的日志流。有关详细信息，请参阅 Lambda Runtime Logs API。	2020 年 11 月 12 日

Amazon MQ 的新事件源	Lambda 现支持 Amazon MQ 作为事件源。使用 Lambda 函数处理来自 Amazon MQ 消息代理的记录。有关详细信息，请参阅将 Lambda 与 Amazon MQ 结合使用。	2020 年 11 月 5 日
预览：Lambda 扩展 API	使用 Lambda 扩展来增强 Lambda 函数。可以使用 Lambda 合作伙伴提供的扩展，也可以创建自己的 Lambda 扩展。有关详细信息，请参阅 Lambda 扩展 API。	2020 年 10 月 8 日
在 AL2 上支持 Java 8 和 自定义运行时	Lambda 目前在 Amazon Linux 2 上支持 Java 8 和 自定义运行时。有关详细信息，请参阅 Lambda 运行时。	2020 年 8 月 12 日
Amazon Managed Streaming for Apache Kafka 的新事件源	Lambda 现支持 Amazon MSK 作为事件源。将 Lambda 函数与 Amazon MSK 结合使用来处理 Kafka 主题中的记录。有关详细信息，请参阅将 Lambda 与 Amazon MSK 结合使用。	2020 年 8 月 11 日
Amazon VPC 设置的 IAM 条件键	您现在可以将特定于 Lambda 的条件键用于 VPC 设置。例如，您可以要求组织中的所有函数都连接到 VPC。您还可以指定函数的用户可以使用和不能使用的子网和安全组。有关详细信息，请参阅为 IAM 函数配置 VPC。	2020 年 8 月 10 日

Kinesis HTTP/2 流使用者的并发设置	现在，您可以对具有增强扇出功能（HTTP/2 流）的 Kinesis 使用者使用以下并发设置：ParallelizationFactor、MaximumRetryAttempts、MaximumRecordAgeInSeconds、DestinationConfig 和 BisectBatchOnFunctionError。有关详细信息，请参阅将 AWS Lambda 与 Amazon Kinesis 结合使用。	2020 年 7 月 7 日
Kinesis HTTP/2 流使用者的批处理时段	您现在可以为 HTTP/2 流配置批处理时段 (MaximumBatchingWindowInSeconds)。Lambda 会一直从流中读取记录，直到收集完整批处理，或者直到批处理时段到期。有关详细信息，请参阅将 AWS Lambda 与 Amazon Kinesis 结合使用。	2020 年 6 月 18 日
支持 Amazon EFS 文件系统	现在，您可以将 Amazon EFS 文件系统连接到 Lambda 函数以进行共享网络文件访问。有关详细信息，请参阅配置 Lambda 函数的文件系统访问。	2020 年 6 月 16 日
Lambda 控制台中的 AWS CDK 示例应用程序	Lambda 控制台现在包括对 TypeScript 使用 AWS Cloud Development Kit (AWS CDK) 的示例应用程序。AWS CDK 是一个框架，允许您使用 TypeScript、Python、Java 或 .NET 定义应用程序资源。	2020 年 6 月 1 日

在 AWS Lambda 中支持 .NET Core 3.1.0 运行时	AWS Lambda 现在支持 .NET Core 3.1.0 运行时。有关详细信息，请参阅 .NET Core CLI。	2020 年 3 月 31 日
支持 API Gateway HTTP API	更新和扩展了将 Lambda 与 API Gateway 结合使用的文档，包括对 HTTP API 的支持。添加了使用 AWS CloudFormation 创建 API 和函数的示例应用程序。有关详细信息，请参阅将 Lambda 与 Amazon API Gateway 结合使用。	2020 年 3 月 23 日
Ruby 2.7	已提供适用于 Ruby 2.7 的新运行时 ruby2.7，它是第一个使用 Amazon Linux 2 的 Ruby 运行时。有关详细信息，请参阅使用 Ruby 构建 Lambda 函数。	2020 年 2 月 19 日
并发指标	Lambda 现在报告所有函数、别名和版本的 `ConcurrentExecutions` 指标。您可以在函数的监控页面上查看此指标的图表。以前，仅在账户级别和为使用预留并发的函数报告 `ConcurrentExecutions` 。有关详细信息，请参阅 AWS Lambda 函数指标。	2020 年 2 月 18 日

更新功能状态	默认情况下，现在对所有函数强制执行函数状态。当您将函数连接到 VPC 时，Lambda 会创建共享的弹性网络接口。这样，您的函数就可以在不创建额外的网络接口的情况下向上扩展。在此期间，您无法对函数执行其他操作，包括更新其配置和发布版本。在某些情况下，调用也会受到影响。Lambda API 提供了有关函数当前状态的详细信息。 此更新正在分阶段发布。有关详细信息，请参阅AWS计算博客上的 VPC 网络的已更新 Lambda 状态生命周期。有关状态的更多信息，请参阅 AWS Lambda 函数状态。	2020 年 1 月 24 日
对函数配置 API 输出的更新	对于连接到 VPC 的函数，将原因代码添加到 StateReasonCode（InvalidSubnet、InvalidSecurityGroup）和 LastUpdateStatusReasonCode（SubnetOutOfIPAddresses、InvalidSubnet、InvalidSecurityGroup）。有关状态的更多信息，请参阅 AWS Lambda 函数状态。	2020 年 1 月 20 日

预配置并发	现在，您可为函数版本或别名分配预配置并发。预配置并发使函数能够在延迟不发生波动的情况下进行扩展。有关详细信息，请参阅管理 Lambda 函数的并发。	2019 年 12 月 3 日
创建数据库代理	现在，您可以使用 Lambda 控制台为 Lambda 函数创建数据库代理。数据库代理使函数能够在不耗尽数据库连接的情况下达到高并发级别。有关详细信息，请参阅为 Lambda 函数配置数据库访问。	2019 年 12 月 3 日
对持续时间指标的百分位支持	现在，您可以基于百分位筛选持续时间指标。有关详细信息，请参阅 AWS Lambda 指标。	2019 年 11 月 26 日
增加流事件源的并发	DynamoDB 流和 Kinesis 流事件源映射的新选项使您能够一次处理每个分区中的多个批处理。当您增加每个分区的并发批处理时，函数的并发最多可以是流中分区数的 10 倍。有关详细信息，请参阅 Lambda 事件源映射。	2019 年 11 月 25 日

函数状态	当您创建或更新函数时，它会进入挂起状态，同时 Lambda 会预置资源来为其提供支持。如果您将函数连接到 VPC，Lambda 可以立即创建共享的弹性网络接口，而不是在调用函数时创建网络接口。这将为连接 VPC 的函数带来更好的性能，但可能需要更新您的自动化。有关详细信息，请参阅 AWS Lambda 函数状态。	2019 年 11 月 25 日
异步调用的错误处理选项	异步调用可以使用新的配置选项。您可以配置 Lambda 以限制重试并设置最长事件期限。有关详细信息，请参阅配置异步调用的错误处理。	2019 年 11 月 25 日
流事件源的错误处理	从流读取的事件源映射可以使用新的配置选项。您可以配置 DynamoDB stream 流和 Kinesis 流事件源映射，以便限制重试并设置最长记录期限。出现错误时，您可以将事件源映射配置为在重试之前拆分批处理，并将失败批处理的调用记录发送到队列或主题。有关详细信息，请参阅 Lambda 事件源映射。	2019 年 11 月 25 日

异步调用目标	现在，您可以配置 Lambda 以将异步调用记录发送给另一个服务。调用记录包含有关事件、上下文和函数响应的详细信息。您可以将调用记录发送到 SQS 队列、SNS 主题、Lambda 函数或 EventBridge 事件总线。有关详细信息，请参阅配置异步调用目标。	2019 年 11 月 25 日
Node.js、Python 和 Java 的新运行时	新运行时可用于 Node.js 12、Python 3.8 和 Java 11。有关详细信息，请参阅 Lambda 运行时。	2019 年 11 月 18 日
Amazon SQS 事件源的 FIFO 队列支持	现在，您可以创建从先进先出 (FIFO) 队列读取的事件源映射。以前只支持标准队列。有关详细信息，请参阅将 Lambda 与 Amazon SQS 结合使用。	2019 年 11 月 18 日
在 Lambda 控制台中创建应用程序	现在可在 Lambda 控制台中创建应用程序。有关说明，请参阅在 Lambda 控制台中管理应用程序。	2019 年 10 月 31 日

在 Lambda 控制台（测试版）中创建应用程序	现在，您可以在 Lambda 控制台中创建具有集成式持续交付管道的 Lambda 应用程序。该控制台提供了示例应用程序，可将这些应用程序用作您自己的项目的起点。选择 AWS CodeCommit 和 GitHub 来实施源代码控制。每次将更改推送到存储库时，包含的管道都会自动构建并部署它们。有关说明，请参阅在 Lambda 控制台中管理应用程序。	2019 年 10 月 3 日
针对 VPC 连接函数的性能改进	Lambda 现在使用一种新的弹性网络接口，该接口由 Virtual Private Cloud (VPC) 子网中的所有函数共享。当您将一个函数连接到 VPC 时，Lambda 为每个所选安全组和子网组合创建一个网络接口。当共享网络接口可用时，此函数在扩展时不再需要创建其他网络接口。这大大缩短了启动时间。有关详细信息，请参阅配置 Lambda 函数以访问 VPC 中的资源。	2019 年 9 月 3 日
流批量设置	现在，您可以为 Amazon DynamoDB 和 Amazon Kinesis 事件源映射配置一个批处理时间窗口。配置一个最高 5 分钟批处理时间窗口以便在完整批处理可用之前对传入的记录进行缓存。这可以在流处于不活动状态时减少您函数的调用次数。	2019 年 8 月 29 日

CloudWatch Logs Insights 集成	Lambda 控制台中的监控页面现在包括来自 Amazon CloudWatch Logs Insights 的报告。	2019 年 6 月 18 日
Amazon Linux 2018.03	Lambda 执行环境正在更新为使用 Amazon Linux 2018.03。有关详细信息，请参阅执行环境。	2019 年 5 月 21 日
Node.js 10	针对 Node.js 10 和 nodejs10.x 提供了新的运行时。此运行时使用 Node.js 10.15，并将定期使用 Node.js 10 的最新小版本进行更新。Node.js 10 也是可使用 Amazon Linux 2 的首个运行时。有关详细信息，请参阅使用 Node.js 构建 Lambda 函数。	2019 年 5 月 13 日
GetLayerVersionByArn API	使用 GetLayerVersionByArn API 下载层版本信息，使用版本 ARN 作为输入。与 GetLayerVersion 相比，GetLayerVersionByArn 让您可以直接使用 ARN 而不是将其解析以获取层名称和版本号。	2019 年 4 月 25 日
Ruby	AWS Lambda 现在通过一个新的运行时支持 Ruby 2.5。有关详细信息，请参阅使用 Ruby 构建 Lambda 函数。	2018 年 11 月 29 日

图层	利用 Lambda 层，您可以从您的函数代码单独打包并部署库、自定义运行时及其他依赖项。与其他账户或在全球范围内共享您的层。有关详细信息，请参阅 Lambda 层。	2018 年 11 月 29 日
自定义运行时	构建自定义运行时以采用您的常用编程语言运行 Lambda 函数。有关详细信息，请参阅自定义 Lambda 运行时。	2018 年 11 月 29 日
Application Load Balancer 触发器	Elastic Load Balancing 现在支持 Lambda 函数作为 Application Load Balancers 的目标。有关详细信息，请参阅将 Lambda 与 Application Load Balancer 结合使用。	2018 年 11 月 29 日
使用 Kinesis HTTP/2 流使用者作为触发器	您可以使用 Kinesis HTTP/2 数据流使用者将事件发送到 AWS Lambda。流使用者具有来自数据流中每个分区的专用读取吞吐量，并使用 HTTP/2 来最大程度地降低延迟。有关详细信息，请参阅将 Lambda 与 Kinesis 结合使用。	2018 年 11 月 19 日
Python 3.7	AWS Lambda 现在通过一个新运行时支持 Python 3.7。有关更多信息，请参阅使用 Python 构建 Lambda 函数。	2018 年 11 月 19 日

异步函数调用的有效负载限制提高	异步调用的最大负载大小从 128 KB 增加到 256 KB，这与 Amazon SNS 触发器的最大消息大小相匹配。有关详细信息，请参阅 Lambda 配额。	2018 年 11 月 16 日
AWS GovCloud (US-East) 区域	现已在 AWS GovCloud（美国东部）区域推出 AWS Lambda。	2018 年 11 月 12 日
已将 AWS SAM 主题移至单独的开发人员指南	许多主题都重点说明了如何使用 AWS Serverless Application Model (AWS SAM) 构建无服务器应用程序。这些主题已移至《AWS Serverless Application Model 开发人员指南》。	2018 年 10 月 25 日
在控制台中查看 Lambda 应用程序	您可以在 Lambda 控制台中的 Applications（应用程序）页面上查看 Lambda 应用程序的状态。此页面显示了 AWS CloudFormation 堆栈的状态。它包括页面的链接，您可以在这些页面中查看堆栈资源的更多信息。您还可以查看应用程序的聚合指标并创建自定义监控控制面板。	2018 年 11 月 10 日
函数执行超时限制	要允许长时间运行的函数，最大可配置执行超时从 5 分钟增加到 15 分钟。有关详细信息，请参阅 Lambda 限制。	2018 年 10 月 10 日

AWS Lambda 支持 PowerShell Core 语言	AWS Lambda 现在支持 PowerShell Core 语言。有关更多信息，请参阅在 PowerShell 中编写 Lambda 函数的编程模型。	2018 年 9 月 11 日
在 AWS Lambda 中支持 .NET Core 2.1.0 运行时	AWS Lambda 现在支持 .NET Core 2.1.0 运行时。有关更多信息，请参阅 .NET Core CLI。	2018 年 7 月 9 日
现在可通过 RSS 更新	您现在可以订阅 RSS 源，以了解本指南的版本。	2018 年 7 月 5 日
支持 Amazon SQS 作为事件源	AWS Lambda 现在支持 Amazon Simple Queue Service (Amazon SQS) 作为事件源。有关更多信息，请参阅调用 Lambda 函数。	2018 年 6 月 28 日
中国（宁夏）区域	现已在中国（宁夏）区域推出 AWS Lambda。有关 Lambda 区域和端点的更多信息，请参阅《AWS 一般参考》中的区域和端点。	2018 年 6 月 28 日

早期更新

下表描述 2018 年 6 月之前发布的每个 AWS Lambda 开发人员指南中的重要变化。

更改	描述	日期
Node.js 运行时 8.10 的运行时支持	AWS Lambda 现在支持 Node.js 运行时 v8.10。有关更多信息，请参阅使用 Node.js 构建 Lambda 函数：	2018 年 4 月 2 日

更改	描述	日期
函数和别名修订 ID	AWS Lambda 现在支持您的函数版本和别名上的修订 ID。当您更新您的函数版本或别名资源时，您可以使用这些 ID 跟踪和应用条件更新。	2018 年 1 月 25 日
对 Go 和 .NET 2.0 的运行时支持	AWS Lambda 增加了对 Go 和 .NET 2.0 的运行时支持。有关更多信息，请参阅 使用 Go 构建 Lambda 函数 和 使用 C# 构建 Lambda 函数：	2018 年 1 月 15 日
控制台再设计	AWS Lambda 推出了全新 Lambda 控制台以简化您的体验，并添加了 Cloud9 代码编辑器以使您能够更好地调试和修改函数代码。	2017 年 11 月 30 日
设置单个函数的并发限制	AWS Lambda 现在支持设置单个函数的并发限制。有关更多信息，请参阅为函数配置预留并发：	2017 年 11 月 30 日
使用别名转移流量	AWS Lambda 现在支持使用别名转移流量。有关更多信息，请参阅创建 Lambda 函数的滚动部署：	2017 年 11 月 28 日
逐步代码部署	AWS Lambda 现在支持通过使用 Code Deploy 安全部署新版本的 Lambda 函数。有关更多信息，请参阅逐步代码部署。	2017 年 11 月 28 日
中国（北京）区域	现已在中国（北京）区域推出 AWS Lambda。有关 Lambda 区域和端点的更多信息，请参阅《AWS 一般参考》中的区域和端点。	2017 年 11 月 9 日
推出 SAM Local	AWS Lambda 推出 SAM Local（现称为 SAM CLI），一种 AWS CLI 工具，可在将无服务器应用程序上载到 Lambda 运行时前，为您提供在本地开发、测试和分析环境。有关更多信息，请参阅测试和调试无服务器应用程序。	2017 年 8 月 11 日
加拿大 (中部) 区域	现已在加拿大（中部）区域推出 AWS Lambda。有关 Lambda 区域和端点的更多信息，请参阅《AWS 一般参考》中的区域和端点。	2017 年 6 月 22 日

更改	描述	日期
South America (São Paulo) Region	现已在南美洲（圣保罗）区域推出 AWS Lambda。有关 Lambda 区域和端点的更多信息，请参阅《AWS 一般参考》中的区域和端点。	2017 年 6 月 6 日
AWS Lambda 支持 AWS X-Ray。	Lambda 引入了对 X-Ray 的支持，这样您就可以使用 Lambda 应用程序检测、分析和优化性能问题。有关更多信息，请参阅使用 AWS X-Ray 可视化 Lambda 函数调用：	2017 年 4 月 19 日
亚太地区 (孟买) 区域	现已在亚太地区 (孟买) 区域推出 AWS Lambda。有关 Lambda 区域和端点的更多信息，请参阅《AWS 一般参考》中的区域和端点。	2017 年 3 月 28 日
AWS Lambda 现在支持 Node.js 运行时 v6.10	AWS Lambda 增加了对 Node.js 运行时 v6.10 的支持。有关更多信息，请参阅使用 Node.js 构建 Lambda 函数：	2017 年 3 月 22 日
欧洲（伦敦）区域	现已在欧洲（伦敦）区域推出 AWS Lambda。有关 Lambda 区域和端点的更多信息，请参阅《AWS 一般参考》中的区域和端点。	2017 年 2 月 1 日
AWS Lambda 支持 .NET 运行时、Lambda@Edge (预览版)、死信队列和无服务器应用程序自动部署。	AWS Lambda 增加对 C# 的支持。有关更多信息，请参阅使用 C# 构建 Lambda 函数： 您可使用 Lambda@Edge 在AWS边缘站点上运行 Lambda 函数以响应 CloudFront 事件。有关更多信息，请参阅使用 Lambda @Edge 在边缘进行自定义。	2016 年 12 月 3 日
AWS Lambda 可将 Amazon Lex 添加为受支持的事件源。	使用 Lambda 和 Amazon Lex，您可以为 Slack 和 Facebook 等各种服务快速构建聊天机器人。	2016 年 11 月 30 日
US West (N. California) Region	现已在美国西部（加利福尼亚北部）区域推出 AWS Lambda。有关 Lambda 区域和端点的更多信息，请参阅《AWS 一般参考》中的区域和端点。	2016 年 11 月 21 日

更改	描述	日期
推出 AWS SAM 创建和部署基于 Lambda 的应用程序，以及将环境变量用于 Lambda 函数配置设置。	AWS SAM：您可以使用 AWS SAM 定义用于在无服务器应用程序内表示资源的语法。要部署您的应用程序，只需在 AWS CloudFormation 模板文件（在 JSON 或 YAML 中写入）中作为应用程序的一部分来指定资源及其相关权限策略，打包您的部署项目，然后部署该模板。 环境变量：您可以使用环境变量为 Lambda 函数指定函数代码以外的配置设置。有关更多信息，请参阅使用 Lambda 环境变量配置代码中的值：	2016 年 11 月 18 日
亚太区域（首尔）	现已在亚太地区 (首尔) 区域推出 AWS Lambda。有关 Lambda 区域和端点的更多信息，请参阅《AWS 一般参考》中的区域和端点。	2016 年 8 月 29 日
Asia Pacific (Sydney) Region	现已在亚太地区（悉尼）区域中推出 Lambda。有关 Lambda 区域和端点的更多信息，请参阅《AWS 一般参考》中的区域和端点。	2016 年 6 月 23 日
Lambda 控制台更新	已更新 Lambda 控制台以简化角色创建过程。	2016 年 6 月 23 日
AWS Lambda 现在支持 Node.js 运行时 v4.3	AWS Lambda 增加了对 Node.js 运行时 v4.3 的支持。有关更多信息，请参阅使用 Node.js 构建 Lambda 函数：	2016 年 4 月 7 日
欧洲（法兰克福）区域	现已在欧洲（法兰克福）区域中推出 Lambda。有关 Lambda 区域和端点的更多信息，请参阅《AWS 一般参考》中的区域和端点。	2016 年 3 月 14 日
VPC 支持	您现在可以配置 Lambda 函数来访问 VPC 中的资源。有关更多信息，请参阅授予 Lambda 函数访问 Amazon VPC 中资源的权限：	2016 年 2 月 11 日
已更新 Lambda 运行时。	更新了执行环境。	2015 年 11 月 4 日

更改	描述	日期
版本控制支持、用于开发 Lambda 函数代码的 Python、计划的事件和执行时间增加	您现在可以使用 Python 开发您的 Lambda 函数代码。有关更多信息，请参阅使用 Python 构建 Lambda 函数： 版本控制：您可以保留 Lambda 函数的一个或多个版本。利用版本控制，您可以控制在不同的环境（例如，开发、测试或生产环境）中执行的 Lambda 函数版本。有关更多信息，请参阅管理 Lambda 函数版本： 计划的事件：您还可以使用 Lambda 控制台将 Lambda 设置为定期调用您的代码。您可以指定一个固定速率（小时数、天数或周数）或指定一个 cron 表达式。有关更多信息，请参阅 按计划调用 Lambda 函数。 执行时间增加：您现在可以设置您的 Lambda 函数运行最多五分钟以允许更长时间运行的函数，例如大量数据注入和处理作业。	2015 年 10 月 8 日
对于 DynamoDB Streams 的支持	DynamoDB Streams 现在普遍可用，您可以在 DynamoDB 可用的所有区域使用它。您可以为自己的表启用 DynamoDB Streams，并使用 Lambda 函数作为该表的触发器。触发器是为响应对 DynamoDB 表做出的更新而采取的自定义操作。有关示例演练的信息，请参阅教程：将 AWS Lambda 与 Amazon DynamoDB Streams 结合使用：	2015 年 7 月 14 日
Lambda 现在支持通过兼容 REST 的客户端调用 Lambda 函数。	以前，要从 Web、移动设备或 IoT 应用程序调用 Lambda 函数，您需要AWS开发工具包（例如：AWS SDK for Java、AWS SDK for Android 或 AWS SDK for iOS）。现在，Lambda 支持在兼容 REST 的客户端上通过可借助 Amazon API Gateway 创建的自定义 API 调用 Lambda 函数。您可以向 Lambda 函数端点 URL 发送请求。您可以在该端点上配置安全性以允许开放性访问，利用 AWS Identity and Access Management (IAM) 授权访问，或使用 API 密钥限制其他人对您的 Lambda 函数的访问。 有关示例入门练习，请参阅使用 Amazon API Gateway 端点调用 Lambda 函数：	2015 年 7 月 09 日

更改	描述	日期
现在，Lambda 控制台可提供蓝图，轻松创建 Lambda 函数并对其进行测试。	Lambda 控制台提供了一组蓝图。每个蓝图为您的 Lambda 函数提供了示例事件源配置和示例代码，您可以使用它们轻松地创建基于 Lambda 的应用程序。所有 Lambda 入门练习现在都使用这些蓝图。	2015 年 7 月 09 日
Lambda 现在支持使用 Java 编写 Lambda 函数。	您现在可以使用 Java 编写 Lambda 代码。有关更多信息，请参阅使用 Java 构建 Lambda 函数：	2015 年 6 月 15 日
在创建或更新 Lambda 函数时，Lambda 现在支持以函数 .zip 的形式指定 Amazon S3 对象。	可以将 Lambda 函数部署程序包（.zip 文件）上载到要创建 Lambda 函数的同一区域中的 Amazon S3 存储桶中。然后，您可以在创建或更新 Lambda 函数时指定存储桶名称和对象键名称。	2015 年 5 月 28 日
Lambda 现已普遍增加对移动后端的支持	Lambda 现在可普遍用于生产环境。此外，该版本还推出了一些新的功能，用户使用 Lambda 可轻松构建手机、平板电脑和物联网 (IoT) 后端（可自动扩展而无需预置或管理基础设施）。Lambda 现在支持实时（同步）和异步事件。其他功能包括更简单的事件源配置和管理。引入了针对 Lambda 函数的资源策略，简化了权限模型和编程模型。	2015 年 4 月 9 日
预览版	AWS Lambda 开发人员指南 预览版。	2014 年 11 月 13 日