适用于 Java 的 AWS 开发工具包

开发人员指南

Table of Contents

适用于 Java 的 AWS 开发工具包 2.0 开发人员指南

适用于 Java 的 AWS 开发工具包为 Amazon Web Services 提供 Java API。利用此开发工具包，您可以轻松构建使用 Amazon S3、Amazon EC2、DynamoDB 等的 Java 应用程序。我们将定期向 AWS SDK for Java 添加对新服务的支持。有关特定版本中的更改和功能的列表，请参阅更改日志。

版本 2.0 中的新增功能

AWS SDK for Java 2.0 是对版本 1.x 代码库的重大改写。它基于 Java 8 构建并增加了几个请求次数较多的功能。其中包括对非阻塞 I/O 的支持以及在运行时插入不同 HTTP 实现的功能。有关更多信息，请参阅 AWS 博客。

> Important
>
> 目前，版本 2.0 中的 Amazon Kinesis、Amazon DynamoDB 和 Amazon CloudWatch 模块已处于生产就绪状态，可用于生产环境中。所有其他模块都是预览版本，不建议用于生产中。

对 1.0 的支持

我们目前不会删除对适用于 Java 的 AWS 开发工具包 1.x 版本的支持。随着我们更接近于最终生产版本，我们将分享一份有关持续 1.x 支持的详细计划，这与我们推出其他 AWS 开发工具包的主要版本的方式相似。

其他资源

除了本指南外，还有以下适用于 AWS SDK for Java 开发人员的有价值的在线资源：

- 适用于 Java 的 AWS 开发工具包 2.0 参考
- Java 开发人员博客
- Java 开发人员论坛
- GitHub:
  - 文档源
  - 开发工具包源
- @awsforjava (Twitter)

为开发人员预览做出贡献

开发人员还可以通过以下渠道提供反馈：

- 在 GitHub 上提交问题：
  - 提交文档问题
  - 提交开发工具包问题

- 在AWS SDK for Java 2.0 Gitter 通道上加入有关开发工具包的非正式聊天
- 向 aws-java-sdk-v2-feedback@amazon.com 匿名提交反馈。此电子邮件由适用于 Java 的 AWS 开发工具包团队监控。
- 在文档或开发工具包源 GitHub 存储库中提交拉取请求以便为开发工具包开发做出贡献。

Eclipse IDE 支持

AWS Toolkit for Eclipse 目前不支持AWS SDK for Java 2.0。要将 AWS Toolkit for Eclipse 与AWS SDK for Java 2.0 结合使用，您应使用 Eclipse 中的 Maven 工具来在 2.0 SDK 上添加依赖项。

开发适用于 Android 的 AWS 应用程序

对于 Android 开发人员，Amazon Web Services 发布了专用于 Android 开发的开发工具包：适用于 Android 的 AWS Mobile 开发工具包。请参阅适用于 Android 的 AWS 移动开发工具包开发人员指南了解完整的文档。

AWS SDK for Java 2.0 开发人员预览版入门

此部分提供有关如何安装、设置和使用 AWS SDK for Java 的信息。

主题

注册 AWS 并创建 IAM 用户

要使用AWS SDK for Java访问 Amazon Web Services (AWS)，您需要 AWS 账户和 AWS 凭证。为了提高 AWS 账户的安全性，我们建议您使用 IAM 用户（而不使用 AWS 账户凭证）来提供访问凭证。

> **Note**
>
> 有关 IAM 用户以及它们对于账户的安全性为何十分重要的概述，请参阅 Amazon Web Services General Reference 中的 AWS 安全凭证。

注册 AWS

1. 打开 https://aws.amazon.com/ 并单击 Sign Up。
2. 按照屏幕上的说明进行操作。在注册过程中，您会接到一个电话，需要您使用电话按键输入 PIN 码。

接着，创建一个 IAM 用户并下载 (或复制) 它的秘密访问密钥。

创建 IAM 用户

1. 转到 IAM 控制台 (您可能需要首先登录 AWS)。
2. 单击侧边栏中的 Users 以查看您的 IAM 用户。
3. 如果您未设置任何 IAM 用户，则单击 Create New Users 创建一个用户。
4. 在列表中选择您将用来访问 AWS 的 IAM 用户。
5. 打开 Security Credentials 选项卡，然后单击 Create Access Key。

   > **Note**
   >
   > 对于任何给定的 IAM 用户最多可以有两个活动访问密钥。如果您的 IAM 用户已经有两个访问密钥，您将需要先删除其中的一个访问密钥，然后再创建新密钥。
6. 在所得到的对话框中，单击 Download Credentials 按钮以将凭证文件下载到您的计算机上，或者单击 Show User Security Credentials 以查看 IAM 用户的访问密钥 ID 和秘密访问密钥 (您可以复制和粘贴)。

   > **Important**
   >
   > 在关闭该对话框之后，就无法获取密码访问密钥了。但是，您可以删除与它相关联的访问密钥 ID 并创建新密钥。

接着，在 AWS 共享凭证文件或环境中设置凭证 (p. 5)。

设置AWS SDK for Java 2.0 开发人员预览

本主题介绍如何在您的项目中设置和使用AWS SDK for Java。

先决条件

要使用 AWS SDK for Java，必须拥有：

- 合适的 Java 开发环境 (p. 5)。
- AWS 账户和访问密钥。有关说明，请参阅注册 AWS 并创建 IAM 用户 (p. 3)。
- 在您的环境中设置 AWS 凭证（访问密钥），或使用由 AWS CLI 和其他开发工具包使用的共享凭证文件。有关更多信息，请参阅设置用于开发的 AWS 凭证和区域 (p. 5)。

在您的项目中包含开发工具包

根据您的生成系统或 IDE，使用下列方法之一：

- Apache Maven- 如果使用 Apache Maven，您只能将所需的开发工具包组件或整个开发工具包（不推荐）指定为项目中的依赖项。请参阅将开发工具包与 Apache Maven 一起使用 (p. 6)。
- Gradle - 如果使用 Gradle，您可将 Maven 材料清单 (BOM) 导入到 Gradle 项目，以便自动管理开发工具包依赖项。请参阅将开发工具包与 Gradle 一起使用 (p. 9)。

> **Note**
>
> 可使用支持 MavenCentral 作为项目源的任何生成系统。但是，我们不会为开发人员预览提供可下载的 zip。

编译开发工具包

您可以使用 Maven 构建AWS SDK for Java。Maven 将一步完成下载所有必需的依赖项、构建开发工具包和安装开发工具包。有关安装说明和更多信息，请参阅 http://maven.apache.org/。

编译开发工具包

1. 打开适用于 Java 的 AWS 开发工具包 2.0 (GitHub)。

   > **Note**
   >
   > 开发工具包的版本 1.0 在适用于 Java 的 AWS 开发工具包 1.x (GitHub) 的 GitHub 中也有提供。

2. 单击 Clone or download (克隆或下载) 按钮以选择您的下载选项。
3. 在终端窗口中，导航到下载了开发工具包源文件的目录。
4. 使用以下命令构建并安装开发工具包（需要 Maven）。

   ```
   mvn clean install
   ```

   生成的 `.jar` 文件会构建到 `target` 目录中。
5. （可选）使用以下命令构建 API 参考文档。

```
mvn javadoc:javadoc
```

该文档内置于每个服务的 target/site/apidocs/ 目录中。

安装 Java 开发环境

AWS SDK for Java要求使用 Java SE Development Kit 8.0 或更高版本。可以从 http://www.oracle.com/technetwork/java/javase/downloads/ 下载最新的 Java 软件。

选择 JVM

为了让使用AWS SDK for Java的基于服务器的应用程序获得最佳性能，我们建议您使用 Java 虚拟机 (JVM) 的 64 位版本。此 JVM 仅在服务器模式下运行，即使在运行时指定了 -Client 选项也是如此。

设置用于开发的 AWS 凭证和区域

要使用 AWS SDK for Java 连接到任何支持的服务，您必须提供 AWS 凭证。AWS 开发工具包和 CLI 使用提供程序链 在多个不同的位置（包括系统/用户环境变量和本地 AWS 配置文件）查找 AWS 凭证。

本主题提供有关使用 AWS SDK for Java 为本地应用程序开发设置 AWS 凭证的基本信息。如果您需要设置在 Amazon EC2 实例中使用的凭证，或者您使用 Eclipse IDE 进行开发，请改为参阅以下主题：

- 在使用 EC2 实例时，创建一个 IAM 角色，然后向 EC2 实例授予对该角色的访问权限，如为 Amazon EC2 配置 IAM 角色（高级）(p. 22)中所述。
- 使用 AWS Toolkit for Eclipse 设置 Eclipse 中的 AWS 凭证。请参阅 AWS Toolkit for Eclipse 用户指南中的设置 AWS 凭证。

设置 AWS 凭证

您可以通过多种方式设置凭证以供AWS SDK for Java使用。但是，推荐使用以下方法：

- 在本地系统上的 AWS 凭证配置文件中设置凭证，该配置文件位于：
  - Linux, OS X, or Unix 中的 ~/.aws/credentials
  - Windows 中的 C:\Users*USERNAME*\.aws\credentials

  此文件应包含以下格式的行：

```
[default]
aws_access_key_id = your_access_key_id
aws_secret_access_key = your_secret_access_key
```

  用您自己的 AWS 凭证值替换值 your_access_key_id 和 your_secret_access_key。
- 设置 AWS_ACCESS_KEY_ID 和 AWS_SECRET_ACCESS_KEY 环境变量。

  要在 Linux, OS X, or Unix 上设置这些变量，请使用 **export** ：

```
export AWS_ACCESS_KEY_ID=your_access_key_id
export AWS_SECRET_ACCESS_KEY=your_secret_access_key
```

  要在 Windows 上设置这些变量，请使用 **set** ：

```
set AWS_ACCESS_KEY_ID=your_access_key_id
set AWS_SECRET_ACCESS_KEY=your_secret_access_key
```

- 对于 EC2 实例，请指定一个 IAM 角色，然后向该角色授予对 EC2 实例的访问权。有关其工作方式的详细探讨，请参阅 Amazon EC2 User Guide for Linux Instances 中的 Amazon EC2 的 IAM 角色。

在您使用这些方法之一设置 AWS 凭证后，AWS SDK for Java将使用默认凭证提供程序链自动加载这些凭证。有关在 Java 应用程序中使用 AWS 凭证的更多信息，请参阅使用 AWS 凭证 (p. 12)。

设置 AWS 区域

您应使用AWS SDK for Java设置要用于访问 AWS 服务的默认 AWS 区域。要获得最佳网络性能，请选择在地理位置上靠近您 (或您的客户) 的区域。

> Note
>
> 如果您未 选择区域，则需要区域的服务调用将失败。

您可以使用类似于这样的方法设置凭证以设置默认 AWS 区域：

- 在本地系统上的 AWS Config 文件中设置 AWS 区域，该文件位于：
  - Linux, OS X, or Unix 中的 `~/.aws/config`
  - Windows 中的 `C:\Users\USERNAME\.aws\config`

  此文件应包含以下格式的行：

```
[default]
region = your_aws_region
```

  用所需的 AWS 区域 (例如“us-west-2”) 替换 your_aws_region。
- 设置 `AWS_REGION` 环境变量。

  在 Linux, OS X, or Unix 上，请使用 **`export`** ：

```
export AWS_REGION=your_aws_region
```

  在 Windows 上，请使用 **`set`** ：

```
set AWS_REGION=your_aws_region
```

  其中，your_aws_region 是所需的 AWS 区域名称。

有关选择区域的信息，请参阅AWS 区域选择 (p. 15)。

将开发工具包与 Apache Maven 一起使用

您可以使用 Apache Maven 配置和构建AWS SDK for Java项目或构建开发工具包本身。

> Note
>
> 您必须安装 Maven 才能使用本主题中的指导信息。如果尚未安装 Maven，请访问 http://maven.apache.org/ 下载并进行安装。

创建新的 Maven 程序包

要创建基本 Maven 程序包，请打开终端（命令行）窗口并运行以下命令。

```
mvn -B archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DgroupId=org.example.basicapp \
  -DartifactId=myapp
```

将 org.example.basicapp 替换为您的应用程序的完整程序包命名空间。将 myapp 替换为您的项目名称（这将成为您的项目的目录名称）。

默认情况下，Maven 使用 quickstart 原型为您创建项目模板。这将创建一个 Java 1.5 项目。您必须将应用程序更新到 Java 1.8，以便与AWS SDK for Java 2.0 兼容。要更新到 Java 1.8，请将以下内容添加到 `pom.xml` 文件。

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
         <source>1.8</source>
         <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

可以通过向 `archetype:generate` 命令中添加 `-DarchetypeArtifactId` 参数来选择要使用的特定原型。要跳过步骤以更新 `pom.xml` 文件，您可以使用以下原型从头开始创建 Java 1.8 项目。

```
mvn archetype:generate -B \
   -DarchetypeGroupId=pl.org.miki \
   -DarchetypeArtifactId=java8-quickstart-archetype \
   -DarchetypeVersion=1.0.0 \
   -DgroupId=com.example \
   -DartifactId=sdk-sandbox \
   -Dversion=1.0 \
   -Dpackage=com.example
```

还提供了更多原型。请参阅 Maven 原型以获得随 Maven 打包的原型的列表。

> **Note**
>
> 有关创建和配置 Maven 项目的更多信息，请参阅 Maven 入门指南。

将开发工具包配置为 Maven 依赖项

要在项目中使用AWS SDK for Java，您需要在项目的 `pom.xml` 文件中将该工具包声明为依赖项。您可以导入单个组件 (p. 7)或整个开发工具包 (p. 8)。强烈建议您仅拉入所需的组件而不是整个开发工具包。

指定单独的开发工具包模块（推荐）

要选择单独的开发工具包模块，请使用 Maven 的AWS SDK for Java材料清单 (BOM)。这将确保您指定的模块使用相同版本的开发工具包，并且它们相互兼容。

要使用 BOM，请向应用程序的 pom.xml 文件添加 <dependencyManagement> 部分。将 bom 作为依赖项添加并指定要使用的开发工具包的版本。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>software.amazon.awssdk</groupId>
      <artifactId>bom</artifactId>
      <version>2.0.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

如果您要测试开发人员预览版，请将 preview-* 添加到版本中。

> **Note**
>
> 您不能在一个项目中混合使用 AWS SDK for Java 2.0 开发人员预览版模块和非预览版模块。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>software.amazon.awssdk</groupId>
      <artifactId>bom</artifactId>
      <version>2.0.0-preview-10</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

要查看 Maven Central 中提供的最新版本的AWS SDK for Java BOM，请参阅 https://mvnrepository.com/artifact/software.amazon.awssdk/bom。此页面还显示了由可包含在项目的 pom.xml 文件的 <dependencies> 部分中的 BOM 管理的模块（依赖项）。

现在，可以从开发工具包中选择用于应用程序的单个模块。由于您已经在 BOM 中声明了开发工具包版本，因此无需为每个组件都指定版本号。

```
<dependencies>
  <dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>kinesis</artifactId>
  </dependency>
  <dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>dynamodb</artifactId>
  </dependency>
</dependencies>
```

导入所有开发工具包模块（不推荐）

要将整个 开发工具包作为依赖项拉入，请不要使用 BOM 方法。只需在 pom.xml 中声明它即可，如下所示。

```
<dependencies>
  <dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>aws-java-sdk</artifactId>
```

```
    <version>2.0.0</version>
  </dependency>
</dependencies>
```

构建项目

在设置项目后，可以使用 Maven 的 `package` 命令构建它。

```
mvn package
```

这会在 `target` 目录中创建 `.jar` 文件。

将开发工具包与 Gradle 一起使用

要在 Gradle 项目中使用AWS SDK for Java，请将 Spring 的依赖项管理插件用于 Gradle。您可以使用此插件导入开发工具包的 Maven 材料清单 (BOM) 来管理您的项目的开发工具包依赖项。

配置适用于 Gradle 的开发工具包

1. 向 `build.gradle` 文件中添加依赖项管理插件.

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "io.spring.gradle:dependency-management-plugin:1.0.3.RELEASE"
    }
}

apply plugin: "io.spring.dependency-management"
```

2. 将 BOM 添加到该文件的 dependencyManagement 部分.

```
dependencyManagement {
    imports {
        mavenBom 'software.amazon.awssdk:bom:2.0.0'
    }
}
```

如果您要测试开发人员预览版，请将 `preview-*` 添加到版本中。

> **Note**
>
> 您不能在一个项目中混合使用 AWS SDK for Java 2.0 开发人员预览版模块和非预览版模块。

```
dependencyManagement {
    imports {
        mavenBom 'software.amazon.awssdk:bom:2.0.0-preview-10'
    }
}
```

3. 在依赖项 部分中指定要使用的开发工具包模块。

```
dependencies {
    compile 'software.amazon.awssdk:kinesis'
```

```
    testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

Gradle 会自动使用 BOM 中的信息来解析开发工具包依赖项的正确版本。

以下是完整的 `build.gradle` 文件：

```
group 'aws.test'
version '1.0'

apply plugin: 'java'

sourceCompatibility = 1.8

repositories {
  mavenCentral()
}

buildscript {
  repositories {
      mavenCentral()
  }
  dependencies {
      classpath "io.spring.gradle:dependency-management-plugin:1.0.3.RELEASE"
  }
}

apply plugin: "io.spring.dependency-management"

dependencyManagement {
  imports {
      mavenBom 'software.amazon.awssdk:bom:2.0.0'
  }
}

dependencies {
  compile 'software.amazon.awssdk:kinesis'
  testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

> **Note**
>
> 有关使用 BOM 指定开发工具包依赖项的更多详细信息，请参阅将开发工具包与 Apache Maven 一起使用 (p. 6)。

使用AWS SDK for Java 2.0 开发人员预览

使用 AWS SDK for Java 进行编程的重要常规信息，适用于可与开发工具包结合使用的所有服务。

主题

创建服务客户端

要提交请求至 Amazon Web Services，您首先要创建一个服务客户端对象。在 2.0 版的开发工具包中，您只能使用服务客户端生成器创建客户端。

每个 AWS 服务都有一个服务接口，提供与服务 API 中各项操作对应的方法。例如，Amazon DynamoDB 的服务接口名为 DynamoDbClient。每个服务接口都有静态工厂生成器方法，可用于构建服务接口的实施。

获取客户端生成器

要获取客户端的实例，请使用静态工厂方法 `builder`。然后，使用生成器中的 setter 对它进行自定义，如以下示例所示。

在AWS SDK for Java 2.0 中，为 setter 提供的名称中不含 `with` 前缀。

```
DynamoDBClient client = DynamoDBClient.builder()
                        .region(Region.US_WEST_2)
                        .credentialsProvider(ProfileCredentialsProvider.builder()
                                    .profileName("myProfile")
                                    .build())
                        .build();
```

Note

常用的 setter 方法会返回 `builder` 对象，由此可以将方法调用组合起来，这样不仅方便而且代码更加便于阅读。在配置需要的属性后，可以调用 `build` 方法创建客户端。客户端一经创建便不可变。创建带不同设置的客户端的唯一方法是构建新的客户端。

使用 DefaultClient

客户端生成器包含名为 `create` 的另一个工厂方法。此方法将使用默认配置创建服务客户端。该客户端使用默认提供程序链加载凭证和 AWS 区域。如果不能根据运行应用程序的环境确定凭证或区域，则对

create 的调用失败。有关如何确定凭证和区域的更多信息，请参阅使用 AWS 凭证 (p. 12)和 AWS 区域选择 (p. 15)。

创建默认客户端

```
DynamoDBClient client = DynamoDBClient.create();
```

客户端生命周期

开发工具包中的服务客户端是线程安全的。为了获得最佳性能，应将其作为永久对象。每个客户端自己有连接池资源，当客户端收集到垃圾时相应资源会释放。AWS SDK for Java 2.0 中的客户端现在扩展了 AutoClosable 接口。为了实现最佳实践，请通过调用 close 方法显式关闭客户端。

关闭客户端

```
DynamoDBClient client = DynamoDBClient.create();
client.close();
```

使用 AWS 凭证

要向 Amazon Web Services 提交请求，您必须为 AWS SDK for Java 提供 AWS 凭证。您可以通过下列方式来执行此操作：

- 使用默认凭证提供程序链 (推荐).
- 使用特定的凭证提供程序或提供程序链 (或创建您自己的)。
- 自行提供凭证。这些凭证可以是 AWS 账户凭证、IAM 凭证或从 AWS STS 获取的临时凭证。

> **Important**
>
> 出于安全考虑，强烈建议 您使用 IAM 用户凭证 而非 AWS 账户凭证来进行 AWS 访问。有关更多信息，请参阅 Amazon Web Services General Reference 中的 AWS 安全凭证。

使用默认凭证提供程序链

在初始化新服务客户端而不提供任何参数时，AWS SDK for Java将尝试使用由 DefaultCredentialsProvider 类实现的默认凭证提供程序链 来查找 AWS 凭证。默认凭证提供程序链将按此顺序查找凭证：

1. Java 系统属性 – aws.accessKeyId 和 aws.secretAccessKey。AWS SDK for Java使用 SystemPropertyCredentialsProvider 加载这些凭证。
2. 环境变量 – AWS_ACCESS_KEY_ID 和 AWS_SECRET_ACCESS_KEY。AWS SDK for Java 使用 EnvironmentVariableCredentialsProvider 类加载这些凭证。
3. 默认凭证配置文件 – 通常位于 ~/.aws/credentials (此位置可能因平台而异)，此凭证文件由多个 AWS 开发工具包和 AWS CLI 共享。AWS SDK for Java 使用 ProfileCredentialsProvider 加载这些凭证。

   您可以使用由 AWS CLI 提供的 aws configure 命令创建凭证文件，也可以通过用文本编辑器编辑文件来创建它。有关凭证文件格式的信息，请参阅 AWS 凭证文件格式 (p. 13)。
4. Amazon ECS 容器凭证 – 如果设置了环境变量 AWS_CONTAINER_CREDENTIALS_RELATIVE_URI，则从 Amazon ECS 加载凭证。AWS SDK for Java 使用 ContainerCredentialsProvider 加载这些凭证。
5. 实例配置文件凭证 - 在 Amazon EC2 实例上使用，并通过 Amazon EC2 元数据服务传送。AWS SDK for Java 使用 InstanceProfileCredentialsProvider 加载这些凭证。

设置凭证

要使用 AWS 凭证，必须在上述位置中的至少一个位置设置该凭证。有关设置凭证的信息，请参阅以下主题：

- 有关在环境或默认凭证配置文件中指定凭证，请参阅设置用于开发的 AWS 凭证和区域 (p. 5)。
- 要设置 Java 系统属性，请参阅官方 Java 教程网站中的系统属性教程。
- 要设置和使用与 EC2 实例一起使用的实例配置文件凭证，请参阅为 Amazon EC2 配置 IAM 角色（高级） (p. 22)。

设置备用凭证配置文件

默认情况下，AWS SDK for Java 使用默认配置文件，但可通过几种方式自定义源自凭证文件的配置文件。

您可以使用 `AWS_PROFILE` 环境变量来更改开发工具包所加载的配置文件。

例如，在 Linux, OS X, or Unix 上，您可运行以下命令来将配置文件更改为 myProfile。

```
export AWS_PROFILE="myProfile"
```

在 Windows 上，您将使用以下命令。

```
set AWS_PROFILE="myProfile"
```

设置 `AWS_PROFILE` 环境变量将影响所有正式支持的 AWS 开发工具包和工具（包括 AWS CLI 和 AWS CLI for PowerShell）的凭证加载。如果只需要更改 Java 应用程序的配置文件，则可改用系统属性 aws.profile。

设置备用凭证文件位置

AWS SDK for Java 会自动从默认凭证文件位置加载 AWS 凭证。但是，您也可以通过在 `AWS_CREDENTIAL_PROFILES_FILE` 环境变量中设置凭证文件的完整路径来指定位置。

您可以使用此功能临时更改 AWS SDK for Java 查找凭证文件的位置 (例如，通过使用命令行设置此变量)。或者，您也可以在您的用户环境或系统环境中设置该环境变量，在用户范围或系统范围内对其进行更改。

覆盖默认凭证文件位置

- 将 `AWS_CREDENTIAL_PROFILES_FILE` 环境变量设置为 AWS 凭证文件的位置。
  - 在 Linux, OS X, or Unix 上，请使用 **`export`** ：

    ```
    export AWS_CREDENTIAL_PROFILES_FILE=path/to/credentials_file
    ```

  - 在 Windows 上，请使用 **`set`** ：

    ```
    set AWS_CREDENTIAL_PROFILES_FILE=path/to/credentials_file
    ```

AWS 凭证文件格式

在使用 `aws configure` 命令创建 AWS 凭证文件时，该命令将采用以下格式创建一个文件。

```
[default]
aws_access_key_id={YOUR_ACCESS_KEY_ID}
aws_secret_access_key={YOUR_SECRET_ACCESS_KEY}
```

```
[profile2]
aws_access_key_id={YOUR_ACCESS_KEY_ID}
aws_secret_access_key={YOUR_SECRET_ACCESS_KEY}
```

在方括号中指定配置文件名 (例如：`[default]`)，后跟该配置文件中的可配置字段作为键值对。您的凭证文件可包含多个配置文件，可使用 `aws configure --profile` *`PROFILE_NAME`* 选择要配置的配置文件来添加或编辑这些配置文件。除了访问密钥和秘密访问密钥之外，您还可以使用 `aws_session_token` 字段指定会话令牌。

加载凭证

在设置凭证后，可使用默认凭证提供程序链来加载这些凭证。

为此，应该实例化 AWS 服务客户端而不向生成器显式提供凭证，如下所示。

```
S3Client s3 = S3Client.builder()
                        .region(Region.US_WEST_2)
                        .build();
```

指定凭证提供程序或提供程序链

您可以通过客户端生成器来指定一个不同于默认凭证提供程序链的凭证提供程序。

应该向将 AwsCredentialsProvider 接口作为输入的客户端生成器提供凭证提供程序或提供程序链的实例。下例展示使用环境凭证的具体情况。

```
S3Client s3 = S3Client.builder()

 .credentialsProvider(EnvironmentVariableCredentialsProvider.create())
                        .build();
```

有关AWS SDK for Java提供的凭证提供程序和提供程序链的完整列表，请参阅 AwsCredentialsProvider 中的所有已知实现类。

> **Note**
>
> 您可以使用此方法提供您创建的凭证提供程序或提供程序链，方式为使用您自己的可实现 `AwsCredentialsProvider` 接口的凭证提供程序。

明确指定凭证

如果默认凭证链和特定的或自定义的提供程序或提供程序链都不适用于您的代码，您可以通过自行提供来明确设置这些凭证。如果您已使用 AWS STS 获得临时凭证，请使用此方法指定用于 AWS 访问的凭证。

向 AWS 客户端明确提供凭证

1. 实例化一个提供 AwsCredentials 接口的类，例如 AwsSessionCredentials。为该类提供您将用于连接的 AWS 访问密钥和私有密钥。
2. 使用 `AwsCredentials` 对象创建 StaticCredentialsProvider。
3. 使用 `StaticCredentialsProvider` 配置客户端生成器并构建客户端。

以下是示例。

```
AwsSessionCredentials awsCreds = AwsSessionCredentials.create(
      "access_key_id",
```

```
      "secret_key_id",
      "session_token");
S3Client s32 = S3Client.builder()
                       .credentialsProvider(StaticCredentialsProvider.create(awsCreds))
                       .build();
```

更多信息

- 注册 AWS 并创建 IAM 用户 (p. 3)
- 设置用于开发的 AWS 凭证和区域 (p. 5)
- 为 Amazon EC2 配置 IAM 角色（高级） (p. 22)

AWS 区域选择

使用区域可以访问实际位于特定地理区域的 AWS 服务。它可以用于保证冗余，并保证您的数据和应用程序接近您和用户访问它们的位置。

在 AWS SDK for Java 2.0 中，已将版本 1.x 中的所有其他区域相关的类折叠到一个 Region 类中。您可以将此类用于所有区域相关的操作，例如，检索有关区域的元数据或检查服务在区域中是否可用。

选择区域

您可以指定区域名称，开发工具包将自动为您选择适当的终端节点。

要显式设置区域，建议您使用 Region 类中定义的常量。这是所有公开可用区域的枚举。要使用该类中的区域创建客户端，请使用以下代码。

```
EC2Client ec2 = EC2Client.builder()
                    .region(Region.US_WEST_2)
                    .build();
```

如果您尝试使用的区域不是 `Region` 类中的常量之一，则可使用 `of` 方法创建一个新区域。可使用此功能访问新区域而无需升级开发工具包。

```
Region newRegion = Region.of("us-east-42");
EC2Client ec2 = EC2Client.builder()
                    .region(newRegion)
                    .build();
```

> **Note**
>
> 使用生成器所构建的客户端不可改变，而且不能更改区域。如果要为同一项服务使用多个 AWS 区域，请创建多个客户端，即每个区域一个客户端。

选择特定终端节点

只需调用 `endpointOverride` 方法，即可将各个 AWS 客户端配置为使用一个区域内的特定终端节点。

例如，要将 Amazon EC2 客户端配置为使用 欧洲（爱尔兰）区域，请使用以下代码。

```
EC2Client ec2 = EC2Client.builder()
                             .region(Region.EU_WEST_1)
                             .endpointOverride(URI.create("https://ec2.eu-
west-1.amazonaws.com"))
```

```
                .build();
```

有关所有 AWS 服务的区域及其相应终端节点的最新列表，请参阅区域和终端节点。

根据环境自动确定 AWS 区域

在 Amazon EC2 或 AWS Lambda 上运行时，您可能需要将客户端配置为使用代码运行于的同一区域。由此可以将代码从其运行的环境中脱离，更轻松地将应用程序部署到多个区域以减少延迟并保证冗余。

要使用默认的凭证/区域提供程序链来根据环境确定区域，请使用客户端生成器的 `create` 方法。

```
EC2Client ec2 = EC2Client.create();
```

如果您没有使用 `region` 方法显式设置区域，开发工具包将参考默认区域提供程序链来尝试并确定要使用的区域。

默认区域提供程序链

区域查找过程如下：

1. 通过生成器本身使用 `region` 明确设置的所有区域优先于其他所有区域。
2. 系统会检查 `AWS_REGION` 环境变量。如果已设置该变量，将使用对应区域配置客户端。

   Note

   该环境变量通过 Lambda 容器设置。
3. 开发工具包将检查 AWS 共享配置文件 (通常位于 `~/.aws/config`)。如果 region 属性存在，则开发工具包会使用它。
   - `AWS_CONFIG_FILE` 环境变量可用于自定义共享配置文件的位置。
   - 可以使用 `AWS_PROFILE` 环境变量或 aws.profile 系统属性来自定义开发工具包加载的配置文件。
4. 开发工具包将尝试使用 Amazon EC2 实例元数据服务，为当前运行的 Amazon EC2 实例确定区域。
5. 如果开发工具包此时仍不能确定区域，则客户端创建将失败并返回异常。

开发 AWS 应用程序的一个常用方法是使用共享配置文件 (如使用默认凭证提供程序链 (p. 12)中所述) 在本地开发时设置区域，而在 AWS 基础设施上运行时依赖默认区域提供程序链确定区域。这可以明显简化客户端创建，并保证应用程序的便携性。

查看 AWS 区域的服务可用性

要确认特定的 AWS 服务在某个区域内是否可用，请对要检查的服务使用 `serviceMetadata` 和 `region` 方法。

```
DynamoDBClient.serviceMetadata().regions().forEach(System.out::println);
```

请参阅 Region 类文档以了解可指定的区域，并使用服务的终端节点前缀进行查询。

异步编程

AWS SDK for Java 2.0 具有真正的非阻塞异步客户端，可以跨多个线程实现高并发度。AWS SDK for Java 1.11.x 具有异步客户端，该客户端是围绕线程池和阻塞同步客户端（未提供非阻塞 I/O 的所有优势）的包装程序。

同步方法会阻止执行您的线程，直到客户端接收到服务的响应。异步方法会立即返回，并控制调用的线程，而不必等待响应。

由于异步方法在收到响应之前返回，所以需要通过某种方法接收返回的响应。AWS SDK for Java 2.0 异步客户端方法将返回 CompletableFuture 对象，该对象可让您在响应准备就绪时访问响应。

非流式操作

对于非流式操作，异步方法调用与同步方法类似，只不过AWS SDK for Java中的异步方法会返回 CompletableFuture 对象，该对象包含之后的 异步操作的结果。

在结果可用时使用要完成的操作调用 `CompletableFuture` `whenComplete()` 方法。`CompletableFuture` 会实现 `Future` 接口，以便您也可以通过调用 `get()` 方法来获取响应对象。

以下示例演示一个调用 DynamoDB 函数以获取表列表的异步操作，该操作收到可包含 ListTablesResponse 对象的 `CompletableFuture`。在调用 `whenComplete()` 时定义的操作仅在异步调用完成时完成。

```
import java.util.concurrent.CompletableFuture;

public class DynamoDBAsync {

    public static void main(String[] args) throws InterruptedException {
        // Creates a default async client with credentials and regions loaded from the
 environment
        DynamoDbAsyncClient client = DynamoDbAsyncClient.create();
        CompletableFuture<ListTablesResponse> response =
 client.listTables(ListTablesRequest.builder()

 .build());

        // Map the response to another CompletableFuture containing just the table names
        CompletableFuture<List<String>> tableNames =
 response.thenApply(ListTablesResponse::tableNames);
        // When future is complete (either successfully or in error) handle the response
        tableNames.whenComplete((tables, err) -> {
            try {
             if (tables != null) {
                    tables.forEach(System.out::println);
                } else {
                    // Handle error
                    err.printStackTrace();
                }
            } finally {
                // Lets the application shut down. Only close the client when you are
 completely done with it.
                client.close();
            }
        });

        tableNames.join();
    }
}
```

流式操作

对于流式操作，您必须提供 AsyncRequestBody 来以递增方式提供内容，或者提供 AsyncResponseTransformer 来接收和处理响应。

下面是使用 `PutObject` 操作将文件异步上传到 Amazon S3 的示例。

```
public class S3AsyncOps {
```

```
   private static final String BUCKET = "sample-bucket";
 private static final String KEY = "testfile.in";

 public static void main(String[] args) {
     S3AsyncClient client = S3AsyncClient.create();
        CompletableFuture<PutObjectResponse> future = client.putObject(
                PutObjectRequest.builder()
                                .bucket(BUCKET)
                                .key(KEY)
                                .build(),
                AsyncRequestBody.fromFile(Paths.get("myfile.in"))
        );
        future.whenComplete((resp, err) -> {
            try {
                if (resp != null) {
                    System.out.println("my response: " + resp);
                } else {
                    // Handle error
                    err.printStackTrace();
                }
            } finally {
                // Lets the application shut down. Only close the client when you are
 completely done with it.
                client.close();
            }
        });

        future.join();
    }
}
```

下面是使用 `GetObject` 操作从 Amazon S3 中异步获取文件的示例。

```
public class S3AsyncStreamOps {

   private static final String BUCKET = "sample-bucket";
 private static final String KEY = "testfile.out";

 public static void main(String[] args) {
     S3AsyncClient client = S3AsyncClient.create();
      final CompletableFuture<GetObjectResponse> futureGet = client.getObject(
                GetObjectRequest.builder()
                                .bucket(BUCKET)
                                .key(KEY)
                                .build(),
                AsyncResponseTransformer.toFile(Paths.get("myfile.out")));
      futureGet.whenComplete((resp, err) -> {
            try {
                if (resp != null) {
                    System.out.println(resp);
                } else {
                    // Handle error
                    err.printStackTrace();
                }
            } finally {
                // Lets the application shut down. Only close the client when you are
 completely done with it
                client.close();
            }
        });

      futureGet.join();
    }
```

```
}
```

HTTP/2 编程

HTTP/2 是 HTTP 协议的一个主要修订。这一新版本具有多个增强功能以提高性能：

- 二进制数据编码提供了更高效的数据传输。
- 标头压缩可减少客户端下载的开销字节数，同时帮助客户端更快地获取内容。这对于受带宽限制的移动客户端尤其有用。
- 双向异步通信（多路复用）可让客户端和 AWS 之间的多个请求和响应消息同时通过单一连接（而不是通过多个连接）进行传输，这样可以提高性能。

升级到最新开发工具包的开发人员将自动使用 HTTP/2（如果他们使用的服务支持）。新编程接口无缝地利用 HTTP/2 功能并提供新的方法来构建应用程序。

AWS SDK for Java 2.0 提供了新的实施 HTTP/2 协议的 API 进行事件流式传输。有关如何使用这些新 API 的示例，请参阅使用适用于 Java 的 AWS 开发工具包的 Kinesis 示例 (p. 76)。

异常处理

要使用开发工具包构建高质量的应用程序，必须了解 AWS SDK for Java 在什么情况下会引发异常以及它以什么方式引发异常。接下来几节介绍开发工具包引发异常的几种不同情况，以及如何正确地处理这些异常。

为什么使用取消选中的异常？

出于以下原因，AWS SDK for Java 使用运行时 (或取消选中的) 异常而不是选中的异常：

- 使开发人员能够精细控制要处理哪些错误，而不是必须处理无关紧要的异常情况 (这会导致代码极其冗长)
- 避免大型应用程序因使用选中的异常而固有的可扩展性问题

一般来说，小型应用程序使用选中的异常是可以的，但随着应用程序的大小和复杂程度增加，这样做就会出现问题。

SdkServiceException（和子类）

SdkServiceException 是在使用 AWS SDK for Java 时最常遇到的异常。该异常是指来自 AWS 服务的错误响应。例如，如果您尝试终止不存在的 Amazon EC2 实例，EC2 会返回错误响应，而且引发的 `SdkServiceException` 中会包含该错误响应的所有详细信息。在某些情况下，会引发 `SdkServiceException` 的一个子类，使开发人员能够通过捕获模块精细控制如何处理错误情况。

当您遇到 `SdkServiceException` 时，您就会知道，您的请求已成功发送到 AWS 服务，但无法成功处理。这可能是因为请求的参数中存在错误，或者是因为服务端的问题。

`SdkServiceException` 为您提供很多信息，例如：

- 返回的 HTTP 状态代码
- 返回的 AWS 错误代码
- 来自服务的详细错误消息
- 已失败请求的 AWS 请求 ID

`SdkServiceException` 还包括 ErrorType 字段，此字段指出请求失败原因是调用方的错误（请求具有非法值）还是 AWS 服务的错误（内部服务错误）。

SdkClientException

SdkClientException 指示在尝试将请求发送到 AWS 或者在尝试解析来自 AWS 的响应时，Java 客户端代码内出现问题。在一般情况下，`SdkClientException` 比 `SdkServiceException` 严重，前者指示出现严重问题，导致客户端无法对 AWS 服务进行服务调用。例如，如果您在尝试对一个客户端执行操作时网络连接不可用，AWS SDK for Java 会引发 `SdkClientException`。

记录 AWS SDK for Java 调用

AWS SDK for Java通过 Slf4j 进行检测，它是一个抽象层，支持在运行时使用多种日志记录系统中的一个。

支持的日志记录系统包括 Java Logging Framework、Apache Log4j 和其他系统。本主题介绍如何使用 Log4j。无需对您的应用程序代码进行任何更改，就可以使用开发工具包的日志记录功能。

要了解有关 Log4j 的更多信息，请参阅 Apache 网站。

添加 Log4J JAR

要将 Log4j 与开发工具包结合使用，您需要从 Log4j 网站下载 Log4j JAR 或通过在 pom.xml 文件中的 Log4j 上添加依赖项来使用 Maven。该开发工具包不包括 JAR。

Log4j 配置文件

Log4j 使用配置文件 log4j2.xml。配置文件示例如下所示。要了解有关配置文件中使用的值的更多信息，请参阅 Log4j 配置手册。

将配置文件置于类路径上的目录中。Log4j JAR 和 log4j2.xml 文件不需要位于同一目录中。

log4j2.xml 配置文件会指定日志记录级别、发送日志记录输出的位置（例如发送到文件或控制台）和输出格式等属性。日志级别是记录器生成输出的粒度。Log4j 支持多个日志记录层次结构的概念。可以为每级层次结构单独设置日志记录级别。AWS SDK for Java 支持以下两个日志记录层次结构：

- software.amazon.awssdk
- org.apache.http.wire

设置类路径

Log4j JAR 和 log4j2.xml 文件都必须位于类路径中。要为 Maven 中的 Sl4j 配置 log4j 绑定，您可以将以下内容添加到 pom.xml：

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
```

```
  <artifactId>log4j-slf4j-impl</artifactId>
</dependency>
```

如果您使用 Eclipse IDE，可以打开菜单并导航到 Project | Properties | Java Build Path 来设置类路径。

特定服务的错误消息和警告

我们建议您始终将“software.amazon.awssdk”记录器层次结构设置为“WARN”，以保证不会错过来自客户端库的任何重要消息。例如，如果 Amazon S3 客户端检测到应用程序没有正确关闭`InputStream`而且可能会泄漏资源，那么 S3 客户端将通过警告消息来进行报告。另外，由此可确保客户端在处理请求或响应遇到任何问题时会记录相应消息。

以下 log4j2.xml 文件将 `rootLogger` 设置为 WARN，也就是包含“software.amazon.awssdk”层次结构中所有记录器发送的警告和错误消息。您也可以将 software.amazon.awssdk 记录器显式设置为 WARN。

```
<Configuration status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{YYYY-MM-dd HH:mm:ss} [%t] %-5p %c:%L - %m%n" />
    </Console>
  </Appenders>

  <Loggers>
    <Root level="WARN">
      <AppenderRef ref="ConsoleAppender"/>
    </Root>
    <Logger name="software.amazon.awssdk" level="WARN" />
  </Loggers>
</Configuration>
```

请求/响应摘要日志记录

对 AWS 服务的所有请求都会生成一个 AWS 请求 ID，如果您遇到与 AWS 服务处理请求有关的问题，可以使用它。如果调用任何服务失败，可以通过开发工具包中的 Exception 对象以编程方式访问 AWS 请求 ID，还可以通过“software.amazon.awssdk.request”记录器中的 DEBUG 日志级别报告 AWS 请求 ID。

以下 log4j2.xml 文件将启用请求和响应的汇总。

```
<Configuration status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{YYYY-MM-dd HH:mm:ss} [%t] %-5p %c:%L - %m%n" />
    </Console>
  </Appenders>

  <Loggers>
    <Root level="WARN">
      <AppenderRef ref="ConsoleAppender"/>
    </Root>
    <Logger name="software.amazon.awssdk" level="WARN" />
    <Logger name="software.amazon.awssdk.request" level="DEBUG" />
  </Loggers>
</Configuration>
```

以下是日志输出的示例：

```
2018-01-28 19:31:56 [main] DEBUG software.amazon.awssdk.request:Logger.java:78 - Sending
 Request: software.amazon.awssdk.http.DefaultSdkHttpFullRequest@3a80515c
```

详细线路日志记录

在某些情况下，查看 AWS SDK for Java 发送和接收的确切请求和响应可能很有用。如果确实需要访问相关信息，可以通过 Apache HttpClient 记录器临时启用它。如果在 `apache.http.wire` 记录器中启用 DEBUG 级别，会记录所有请求和响应数据。

> Warning
>
> 我们建议只出于调试目的使用线路日志记录。由于线路日志记录可能记录敏感数据，因此应在您的生产环境中禁用它。它会记录完整的请求或响应而不加密，即使对于 HTTPS 调用也是如此。对于大型请求（例如，将文件上传到 Amazon S3）或响应，详细线路日志记录也可能显著影响应用程序的性能。

以下 log4j2.xml 文件会在 Apache HttpClient 中启用完整线路日志记录。

```
<Configuration status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{YYYY-MM-dd HH:mm:ss} [%t] %-5p %c:%L - %m%n" />
    </Console>
  </Appenders>

  <Loggers>
    <Root level="WARN">
     <AppenderRef ref="ConsoleAppender"/>
    </Root>
    <Logger name="software.amazon.awssdk" level="WARN" />
    <Logger name="software.amazon.awssdk.request" level="DEBUG" />
    <Logger name="org.apache.http.wire" level="DEBUG" />
  </Loggers>
</Configuration>
```

使用 Apache 进行线路日志记录需要 log4j-1.2-api 上的其他 Maven 依赖项，因为 Apache 在后台使用了 1.2。如果启用了线路日志记录，请将以下内容添加到 pom.xml 文件。

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-1.2-api</artifactId>
</dependency>
```

为 Amazon EC2 配置 IAM 角色（高级）

必须使用由 AWS 颁发的凭证对发送到 AWS 服务的所有请求进行加密签名。可以使用 IAM 角色 方便地授予对 Amazon EC2 实例上的 AWS 资源的安全访问权。

本主题介绍如何将 IAM 角色用于 Amazon EC2 上运行的AWS SDK for Java应用程序。有关 IAM 实例的更多信息，请参阅 Amazon EC2 User Guide for Linux Instances 中的 Amazon EC2 的 IAM 角色。

默认提供程序链和 Amazon EC2 实例配置文件

如果您的应用程序使用 `create` 方法创建了一个 AWS 客户端，该客户端将按照以下顺序使用默认凭证提供程序链 搜索凭证：

1. Java 系统属性：`aws.accessKeyId` 和 `aws.secretKey`。
2. 系统环境变量：`AWS_ACCESS_KEY_ID` 和 `AWS_SECRET_ACCESS_KEY`。
3. 默认凭证文件 (在不同平台上该文件位于不同位置)。

4. 在 Amazon ECS 环境变量中：`AWS_CONTAINER_CREDENTIALS_RELATIVE_URI`。
5. 实例配置文件凭证，包含在与 EC2 实例的 IAM 角色关联的实例元数据中。

仅当在 Amazon EC2 实例上运行您的应用程序时，默认提供程序链中的最终步骤才可用。但是，当与 Amazon EC2 实例一起使用时，它将提供最大的易用性和最高安全性。您还可以将 InstanceProfileCredentialsProvider 实例直接传递给客户端构造函数，这样无需执行整个默认提供程序链即可获取实例配置文件凭证。

例如：

```
S3Client s3 = S3Client.builder()
              .credentialsProvider(InstanceProfileCredentialsProvider.builder().build())
              .build();
```

当您使用此方法时，开发工具包将检索与 Amazon EC2 实例（位于其实例配置文件中）的关联 IAM 角色的关联凭证具有相同权限的临时 AWS 凭证。尽管这些凭证是临时凭证，而且最终会过期，但 `InstanceProfileCredentialsProvider` 会定期为您刷新它们，保证您收到的凭证可继续访问 AWS。

演练：将 IAM 角色用于 Amazon EC2 实例

本演练将介绍如何使用 IAM 角色从 Amazon S3 中检索对象以管理访问。

创建 IAM 角色

创建授予对 Amazon S3 的只读访问权的 IAM 角色。

创建 IAM 角色

1. 打开 IAM 控制台。
2. 在导航窗格中，选择 Roles (角色)，然后选择 Create New Role (创建新角色)。
3. 在 Select Role Type (选择角色类型) 页上的 AWS Service Roles (AWS 服务角色) 下，选择 Amazon EC2。
4. 在 Attach Policy (附加策略) 页上，选择 Amazon S3 Read Only Access (Amazon S3 只读访问权限)，然后选择 Next Step (下一步)。
5. 输入角色名称，然后选择 Next Step。请记住此名称，

   因为在启动 Amazon EC2 实例时会用到它。
6. 在 Review 页面上，选择 Create Role。

启动 EC2 实例并指定您的 IAM 角色

您可通过 Amazon EC2 控制台，使用 IAM 角色启动 Amazon EC2 实例。

要使用控制台启动 Amazon EC2 实例，请参阅 Amazon EC2 User Guide for Linux Instances 中的 Amazon EC2 Linux 实例入门。

到达 Review Instance Launch 页面时，选择 Edit instance details。在 IAM role 中，选择您之前创建的 IAM 角色。按指示完成该过程。

> Note
>
> 您需要创建或使用现有安全组和密钥对才能连接到该实例。

利用此 IAM 和 Amazon EC2 设置，您可以将应用程序部署到 EC2 实例，它将具有对 Amazon S3 服务的读取访问权限。

AWS SDK for Java 2.0 代码示例

此部分提供了有关使用适用于特定使用案例的 AWS SDK for Java 2.0 的编程示例。

> Important
>
> 目前，Amazon Kinesis、Amazon DynamoDB 和 Amazon CloudWatch 模块已处于生产就绪状态，可用于生产环境中。所有其他模块都是预览版本，我们不建议您将它们用于生产环境中。

主题

使用 AWS SDK for Java (Developer Preview) 的 Amazon S3 示例

此部分提供使用适用于 Java 的 AWS 开发工具包 2.0 对 Amazon S3 进行编程的示例。

> Important
>
> 这是一个预览版，不建议用于生产环境。

以下示例仅包含演示每种方法所需的代码。完整的示例代码在 GitHub 上提供。您可以从中下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行它们。

主题

创建、列出和删除 Amazon S3 存储桶

> Important
>
> 这是一个预览版，不建议用于生产环境。

Amazon S3 中的每个对象（文件）必须位于存储桶 中。存储桶表示对象的集合（容器）。每个存储桶必须具有一个唯一键（名称）。有关存储桶及其配置的详细信息，请参阅 Amazon S3 Developer Guide 中的使用 Amazon S3 存储桶。

> Note
>
> 最佳实践

建议您对 Amazon S3 存储桶启用 AbortIncompleteMultipartUpload 生命周期规则。
该规则指示 Amazon S3 中止在启动后没有在指定天数内完成的分段上传。当超过设置的时间限制时，Amazon S3 将中止上传，然后删除未完成的上传数据。
有关更多信息，请参阅 Amazon S3 User Guide 中的使用版本控制的存储桶的生命周期配置。

Note

这些代码段假定您了解使用适用于 Java 的 AWS 开发工具包 2.0 开发人员预览 (p. 11)中的内容，并且已使用设置用于开发的 AWS 凭证和区域 (p. 5)中的信息配置默认 AWS 凭证。

主题

创建存储桶

构建 CreateBucketRequest 并提供存储桶名称。将其传递到 S3Client 的 `createBucket` 方法。使用 S3Client 执行其他操作（例如列出或删除存储桶），如后面的示例中所示。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.CreateBucketConfiguration;
import software.amazon.awssdk.services.s3.model.CreateBucketRequest;
```

代码

```
Region region = Region.US_WEST_2;
S3Client s3 = S3Client.builder().region(region).build();
String bucket = "bucket" + System.currentTimeMillis();
CreateBucketRequest createBucketRequest = CreateBucketRequest
        .builder()
        .bucket(bucket)
        .createBucketConfiguration(CreateBucketConfiguration.builder()

 .locationConstraint(region.value())
                                                                .build())
        .build();
s3.createBucket(createBucketRequest);
```

请参阅 GitHub 上的完整示例。

列出存储桶

构建 ListBucketRequest。使用 S3Client 的 `listBuckets` 方法检索存储桶列表。如果请求成功，将返回 ListBucketsResponse。使用此响应对象可检索存储桶列表。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.CreateBucketConfiguration;
import software.amazon.awssdk.services.s3.model.CreateBucketRequest;
import software.amazon.awssdk.services.s3.model.DeleteBucketRequest;
import software.amazon.awssdk.services.s3.model.ListBucketsRequest;
```

```
import software.amazon.awssdk.services.s3.model.ListBucketsResponse;
```

代码

```
Region region = Region.US_WEST_2;
S3Client s3 = S3Client.builder().region(region).build();
// List buckets
ListBucketsRequest listBucketsRequest = ListBucketsRequest.builder().build();
ListBucketsResponse listBucketsResponse = s3.listBuckets(listBucketsRequest);
listBucketsResponse.buckets().stream().forEach(x -> System.out.println(x.name()));
```

请参阅 GitHub 上的完整示例。

删除存储桶

在删除 Amazon S3 存储桶前，必须先确保存储桶为空，否则该服务将返回错误。如果您的存储桶受版本控制，则必须同时删除位于存储桶中的所有受版本控制对象。

主题

删除存储桶中的对象

构建 ListObjectsV2Request 并使用 S3Client 的 `listObjects` 方法检索存储桶中的对象的列表。然后，在每个对象上使用 `deleteObject` 方法以删除它。

导入

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.CreateBucketConfiguration;
import software.amazon.awssdk.services.s3.model.CreateBucketRequest;
import software.amazon.awssdk.services.s3.model.DeleteBucketRequest;
import software.amazon.awssdk.services.s3.model.DeleteObjectRequest;
import software.amazon.awssdk.services.s3.model.ListObjectsV2Request;
import software.amazon.awssdk.services.s3.model.ListObjectsV2Response;
```

代码

```
ListObjectsV2Request listObjectsV2Request =
 ListObjectsV2Request.builder().bucket(bucket2).build();
ListObjectsV2Response listObjectsV2Response;
do {
    listObjectsV2Response = s3.listObjectsV2(listObjectsV2Request);
    for (S3Object s3Object : listObjectsV2Response.contents()) {

 s3.deleteObject(DeleteObjectRequest.builder().bucket(bucket2).key(s3Object.key()).build());
    }

    listObjectsV2Request = ListObjectsV2Request.builder().bucket(bucket2)

 .continuationToken(listObjectsV2Response.nextContinuationToken())
                                                 .build();

} while (listObjectsV2Response.isTruncated());
```

请参阅 GitHub 上的完整示例。

删除空存储桶

使用存储桶名称构建 DeleteBucketRequest 并将其传递到 S3Client 的 `deleteBucket` 方法。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.CreateBucketConfiguration;
import software.amazon.awssdk.services.s3.model.CreateBucketRequest;
import software.amazon.awssdk.services.s3.model.DeleteBucketRequest;
```

代码

```
Region region = Region.US_WEST_2;
S3Client s3 = S3Client.builder().region(region).build();
DeleteBucketRequest deleteBucketRequest =
 DeleteBucketRequest.builder().bucket(bucket).build();
s3.deleteBucket(deleteBucketRequest);
```

请参阅 GitHub 上的完整示例。

在 Amazon S3 对象上执行操作

> **Important**
>
> 这是一个预览版，不建议用于生产环境。

Amazon S3 对象表示一个文件或数据集合。每个对象都必须包含在一个存储桶 (p. 24)中。

> **Note**
>
> 最佳实践
> 建议您对 Amazon S3 存储桶启用 AbortIncompleteMultipartUpload 生命周期规则。
> 该规则指示 Amazon S3 中止在启动后没有在指定天数内完成的分段上传。当超过设置的时间限制时，Amazon S3 将中止上传，然后删除未完成的上传数据。
> 有关更多信息，请参阅 Amazon S3 User Guide 中的使用版本控制的存储桶的生命周期配置。

> **Note**
>
> 这些代码段假定您了解使用适用于 Java 的 AWS 开发工具包 2.0 开发人员预览 (p. 11)中的内容，并且已使用设置用于开发的 AWS 凭证和区域 (p. 5)中的信息配置默认 AWS 凭证。

主题

上传对象

构建 PutObjectRequest 并提供存储桶名称和密钥名称。然后，将 S3Client 的 `putObject` 方法与包含对象内容和 `PutObjectRequest` 对象的 RequestBody 结合使用。存储桶必须存在，否则服务将返回错误。

导入

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.ListObjectsV2Response;
import software.amazon.awssdk.services.s3.paginators.ListObjectsV2Iterable;
```

代码

```
Region region = Region.US_WEST_2;
s3 = S3Client.builder().region(region).build();

String bucket = "bucket" + System.currentTimeMillis();
String key = "key";

// Put Object
s3.putObject(PutObjectRequest.builder().bucket(bucket).key(key)
                             .build(),
             RequestBody.fromByteBuffer(getRandomByteBuffer(10_000)));
```

请参阅 GitHub 上的完整示例。

分段上传对象

使用 S3Client 的 `createMultipartUpload` 方法获取上传 ID。然后，使用 `uploadPart` 方法上传每个段。最后，使用 S3Client 的 `completeMultipartUpload` 方法告知 Amazon S3 合并所有已上传的段并完成上传操作。

导入

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.CompleteMultipartUploadRequest;
import software.amazon.awssdk.services.s3.model.CompletedMultipartUpload;
import software.amazon.awssdk.services.s3.model.CompletedPart;
import software.amazon.awssdk.services.s3.model.CreateMultipartUploadRequest;
import software.amazon.awssdk.services.s3.model.CreateMultipartUploadResponse;
import software.amazon.awssdk.services.s3.model.S3Object;
```

代码

```
CreateMultipartUploadRequest createMultipartUploadRequest =
 CreateMultipartUploadRequest.builder()

 .bucket(bucketName).key(key)

 .build();
CreateMultipartUploadResponse response =
 s3.createMultipartUpload(createMultipartUploadRequest);
String uploadId = response.uploadId();
System.out.println(uploadId);

// Upload all the different parts of the object
UploadPartRequest uploadPartRequest1 =
 UploadPartRequest.builder().bucket(bucketName).key(key)
                                                    .uploadId(uploadId)
                                                    .partNumber(1).build();
String etag1 = s3.uploadPart(uploadPartRequest1,
 RequestBody.fromByteBuffer(getRandomByteBuffer(5 * MB))).eTag();
CompletedPart part1 = CompletedPart.builder().partNumber(1).eTag(etag1).build();

UploadPartRequest uploadPartRequest2 =
 UploadPartRequest.builder().bucket(bucketName).key(key)
                                                    .uploadId(uploadId)
```

```
                                                        .partNumber(2).build();
String etag2 = s3.uploadPart(uploadPartRequest2,
 RequestBody.fromByteBuffer(getRandomByteBuffer(3 * MB))).eTag();
CompletedPart part2 = CompletedPart.builder().partNumber(2).eTag(etag2).build();


// Finally call completeMultipartUpload operation to tell S3 to merge all uploaded
// parts and finish the multipart operation.
CompletedMultipartUpload completedMultipartUpload =
 CompletedMultipartUpload.builder().parts(part1, part2).build();
CompleteMultipartUploadRequest completeMultipartUploadRequest =

 CompleteMultipartUploadRequest.builder().bucket(bucketName).key(key).uploadId(uploadId)
                                     .multipartUpload(completedMultipartUpload).build();
s3.completeMultipartUpload(completeMultipartUploadRequest);
```

请参阅 GitHub 上的完整示例。

下载对象

构建 GetObjectRequest 并提供存储桶名称和密钥名称。使用 S3Client 的 `getObject` 方法，并向其传递 `GetObjectRequest` 对象和 `ResponseTransformer` 对象。`ResponseTransformer` 将创建一个将响应内容写入到指定的文件或流的响应处理程序。

以下示例指定要将对象内容写入到的文件名。

导入

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.GetObjectRequest;
import software.amazon.awssdk.core.sync.RequestBody;
```

代码

```
// Get Object
s3.getObject(GetObjectRequest.builder().bucket(bucket).key(key).build(),
  ResponseTransformer.toFile(Paths.get("multiPartKey")));
```

请参阅 GitHub 上的完整示例。

删除数据元

构建 DeleteObjectRequest 并提供存储桶名称和密钥名称。使用 S3Client 的 `deleteObject` 方法，并向其传递要删除的存储桶和对象的名称。指定的存储桶和对象键必须存在，否则服务将返回错误。

导入

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.DeleteObjectRequest;
```

代码

```
// Delete Object
DeleteObjectRequest deleteObjectRequest =
 DeleteObjectRequest.builder().bucket(bucket).key(key).build();
s3.deleteObject(deleteObjectRequest);
```

请参阅 GitHub 上的完整示例。

使用 AWS SDK for Java (Developer Preview) 的 Amazon SQS 示例

本节提供使用适用于 Java 的 AWS 开发工具包 2.0 对 Amazon SQS 进行编程的示例。

> Important
>
> 这是一个预览版，不建议用于生产环境。

以下示例仅包含演示每种方法所需的代码。完整的示例代码在 GitHub 上提供。您可以从中下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行它们。

主题

使用 Amazon SQS 消息队列

> Important
>
> 这是一个预览版，不建议用于生产环境。

消息队列是用于在 Amazon SQS 中可靠地发送消息的逻辑容器。有两种类型的队列：标准和先进先出 (FIFO)。要了解有关队列以及这些类型之间的差异的更多信息，请参阅 Amazon SQS 开发人员指南。

本主题介绍如何使用 AWS SDK for Java 来创建、列出、删除和获取 Amazon SQS 队列的 URL。

创建队列

使用 SQSClient`createQueue` 方法，然后提供一个描述队列参数的 CreateQueueRequest 对象。

导入

```
import software.amazon.awssdk.services.sqs.SQSClient;
import software.amazon.awssdk.services.sqs.model.CreateQueueRequest;
```

代码

```
CreateQueueRequest createQueueRequest =
 CreateQueueRequest.builder().queueName(queueName).build();
sqsClient.createQueue(createQueueRequest);
```

请参阅 GitHub 上的完整示例。

列出队列

要列出您的账户的 Amazon SQS 队列，请使用 ListQueuesRequest 对象调用 SQSClient`listQueues` 方法。

使用 `listQueues` 重载（不带任何参数）将返回所有队列，最多 1,000 个队列。您可以向 `ListQueuesRequest` 对象提供一个队列名称前缀，以将结果限制为与该前缀匹配的队列。

导入

```
import software.amazon.awssdk.services.sqs.model.ListQueuesRequest;
import software.amazon.awssdk.services.sqs.model.ListQueuesResponse;
```

代码

```
String prefix = "que";
ListQueuesRequest listQueuesRequest =
 ListQueuesRequest.builder().queueNamePrefix(prefix).build();
ListQueuesResponse listQueuesResponse = sqsClient.listQueues(listQueuesRequest);
for (String url : listQueuesResponse.queueUrls()) {
 System.out.println(url);
}
```

请参阅 GitHub 上的完整示例。

获取队列的 URL

通过以下方式调用 SQSClient`getQueueUrl` 方法：使用 GetQueueUrlRequest 对象。

导入

```
import software.amazon.awssdk.services.sqs.model.GetQueueUrlRequest;
import software.amazon.awssdk.services.sqs.model.GetQueueUrlResponse;
```

代码

```
GetQueueUrlResponse getQueueUrlResponse =
        sqsClient.getQueueUrl(GetQueueUrlRequest.builder().queueName(queueName).build());
String queueUrl = getQueueUrlResponse.queueUrl();
System.out.println(queueUrl);
```

请参阅 GitHub 上的完整示例。

删除队列

向 DeleteMessageRequest 对象提供队列的 URL (p. 31)。然后调用 SQSClient`deleteQueue` 方法。

导入

```
import software.amazon.awssdk.services.sqs.model.DeleteMessageRequest;
import software.amazon.awssdk.services.sqs.model.DeleteQueueRequest;
```

代码

```
DeleteQueueRequest deleteQueueRequest =
 DeleteQueueRequest.builder().queueUrl(queueUrl).build();
sqsClient.deleteQueue(deleteQueueRequest);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon SQS 队列在 Amazon SQS Developer Guide 中的工作方式
- Amazon SQS API Reference 中的 CreateQueue

- Amazon SQS API Reference 中的 GetQueueUrl
- Amazon SQS API Reference 中的 ListQueues
- Amazon SQS API Reference 中的 DeleteQueues

发送、接收和删除 Amazon SQS 消息

> Important
>
> 这是一个预览版，不建议用于生产环境。

消息是可由分布式组件发送和接收的一段数据。始终使用 SQS 队列 (p. 30)发送消息。

发送消息

通过调用 SQSClient 客户端 `sendMessage` 方法，将单个消息添加到 Amazon SQS 队列。提供包含该队列的 URL (p. 31)、消息正文和可选延迟值 (以秒为单位) 的 SendMessageRequest 对象。

导入

```
import software.amazon.awssdk.services.sqs.SQSClient;
import software.amazon.awssdk.services.sqs.model.SendMessageRequest;
```

代码

```
sqsClient.sendMessage(SendMessageRequest.builder()
        .queueUrl(queueUrl)
        .messageBody("Hello world!")
        .delaySeconds(10)
        .build());
```

在一个请求中发送多条消息

通过使用 SQSClient`sendMessageBatch` 方法，在单个请求中发送多条消息。此方法采用了 SendMessageBatchRequest，后者包含队列 URL 和要发送的消息的列表。（每条消息都是一个 SendMessageBatchRequestEntry。）您也可以通过设置消息上的延迟值来延迟发送特定消息。

导入

```
import software.amazon.awssdk.services.sqs.SQSClient;
import software.amazon.awssdk.services.sqs.model.SendMessageBatchRequest;
import software.amazon.awssdk.services.sqs.model.SendMessageBatchRequestEntry;
```

代码

```
SendMessageBatchRequest sendMessageBatchRequest = SendMessageBatchRequest.builder()
        .queueUrl(queueUrl)
        .entries(SendMessageBatchRequestEntry.builder().id("id1").messageBody("Hello from
 msg 1").build(),
                SendMessageBatchRequestEntry.builder().id("id2").messageBody("msg
 2").delaySeconds(10).build())
        .build();
sqsClient.sendMessageBatch(sendMessageBatchRequest);
```

请参阅 GitHub 上的完整示例。

检索消息

通过调用 SQSClient`receiveMessage` 方法，检索当前位于队列中的任何消息。此方法采用了 ReceiveMessageRequest，后者包含队列 URL。您也可以指定要返回的消息的最大数量。消息将作为一系列 Message 对象返回。

导入

```
import software.amazon.awssdk.services.sqs.SQSClient;
import software.amazon.awssdk.services.sqs.model.ReceiveMessageRequest;
```

代码

```
ReceiveMessageRequest receiveMessageRequest = ReceiveMessageRequest.builder()
        .queueUrl(queueUrl)
        .maxNumberOfMessages(5)
        .build();
List<Message> messages= sqsClient.receiveMessage(receiveMessageRequest).messages();
```

收到后删除消息

在收到消息并处理其内容后，可通过将消息的接收句柄和队列 URL 发送到 SQSClient`deleteMessage` 方法来从队列中删除消息。

导入

```
import software.amazon.awssdk.services.sqs.SQSClient;
import software.amazon.awssdk.services.sqs.model.DeleteQueueRequest;
```

代码

```
for (Message message : messages) {
    DeleteMessageRequest deleteMessageRequest = DeleteMessageRequest.builder()
            .queueUrl(queueUrl)
            .receiptHandle(message.receiptHandle())
            .build();
    sqsClient.deleteMessage(deleteMessageRequest);
}
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon SQS 队列在 Amazon SQS Developer Guide 中的工作方式
- Amazon SQS API Reference 中的 SendMessage
- Amazon SQS API Reference 中的 SendMessageBatch
- Amazon SQS API Reference 中的 ReceiveMessage
- Amazon SQS API Reference 中的 DeleteMessage

使用 AWS SDK for Java 的 CloudWatch 示例

此部分提供了使用适用于 Java 的 AWS 开发工具包 2.0 对 CloudWatch 进行编程的示例。

Amazon CloudWatch 可实时监控您的 Amazon Web Services (AWS) 资源以及您在 AWS 中运行的应用程序。您可以使用 CloudWatch 收集和跟踪指标，这些指标是您可衡量的相关资源和应用程序的变量。CloudWatch 警报可根据您定义的规则发送通知或者对您所监控的资源自动进行更改。

有关 CloudWatch 的更多信息，请参阅 Amazon CloudWatch 用户指南。

以下示例仅包含演示每种方法所需的代码。完整的示例代码在 GitHub 上提供。您可以从中下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行它们。

主题

从 CloudWatch 获取指标

列出指标

要列出 CloudWatch 指标，请创建 ListMetricsRequest 并调用 CloudWatchClient 的 `listMetrics` 方法。您可以使用 `ListMetricsRequest` 通过命名空间、指标名称或维度筛选返回的指标。

> **Note**
>
> Amazon CloudWatch User Guide 中的 Amazon CloudWatch 指标和维度参考中提供了 AWS 服务发布的指标和维度列表。

导入

```
import software.amazon.awssdk.services.cloudwatch.CloudWatchClient;
import software.amazon.awssdk.services.cloudwatch.model.ListMetricsRequest;
import software.amazon.awssdk.services.cloudwatch.model.ListMetricsResponse;
import software.amazon.awssdk.services.cloudwatch.model.Metric;
```

代码

```
CloudWatchClient cw =
  CloudWatchClient.builder().build();

boolean done = false;
String next_token = null;

while(!done) {

 ListMetricsResponse response;

 if (next_token == null) {
  ListMetricsRequest request = ListMetricsRequest.builder()
    .namespace(namespace)
    .build();

  response = cw.listMetrics(request);
 }
 else {
  ListMetricsRequest request = ListMetricsRequest.builder()
```

```
                .namespace(namespace)
                .nextToken(next_token)
                .build();

  response = cw.listMetrics(request);
 }

    for(Metric metric : response.metrics()) {
        System.out.printf(
            "Retrieved metric %s", metric.metricName());
        System.out.println();
    }

    if(response.nextToken() == null) {
        done = true;
    }
    else {
     next_token = response.nextToken();
    }
}
```

调用指标的 `getMetrics` 方法可在 ListMetricsResponse 中返回指标。

结果可以分页。要检索下一批结果，请对响应对象调用 `nextToken` 并使用该令牌值构建新的请求对象。然后使用新请求再次调用 `listMetrics` 方法。

请参阅 GitHub 上的完整示例。

更多信息

- Amazon CloudWatch API Reference 中的 ListMetrics。

发布自定义指标数据

许多 AWS 服务以“AWS/”开头的命名空间发布它们自己的指标。您也可以使用自己的命名空间发布自定义指标数据 (不以“AWS/”开头即可)。

发布自定义指标数据

要发布自己的指标数据，请使用 PutMetricDataRequest 调用 CloudWatchClient 的 `putMetricData` 方法。`PutMetricDataRequest` 必须包括数据要使用的自定义命名空间，还必须在 MetricDatum 对象中包含有关该数据点本身的信息。

> **Note**
>
> 您无法指定以“AWS/”开头的命名空间。以“AWS/”开头的命名空间为 Amazon Web Services 产品预留。

导入

```
import software.amazon.awssdk.services.cloudwatch.CloudWatchClient;
import software.amazon.awssdk.services.cloudwatch.model.Dimension;
import software.amazon.awssdk.services.cloudwatch.model.MetricDatum;
import software.amazon.awssdk.services.cloudwatch.model.PutMetricDataRequest;
import software.amazon.awssdk.services.cloudwatch.model.PutMetricDataResponse;
import software.amazon.awssdk.services.cloudwatch.model.StandardUnit;
```

代码

```
CloudWatchClient cw =
  CloudWatchClient.builder().build();

Dimension dimension = Dimension.builder()
    .name("UNIQUE_PAGES")
    .value("URLS").build();

MetricDatum datum = MetricDatum.builder()
    .metricName("PAGES_VISITED")
    .unit(StandardUnit.NONE)
    .value(data_point)
    .dimensions(dimension).build();

PutMetricDataRequest request = PutMetricDataRequest.builder()
    .namespace("SITE/TRAFFIC")
    .metricData(datum).build();

PutMetricDataResponse response = cw.putMetricData(request);

System.out.printf("Successfully put data point %f", data_point);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon CloudWatch User Guide 中的使用 Amazon CloudWatch 指标。
- Amazon CloudWatch User Guide 中的 AWS 命名空间。
- Amazon CloudWatch API Reference 中的 PutMetricData。

使用 CloudWatch 警报

创建警报

根据 CloudWatch 指标创建警报，请使用已填充警报条件的 PutMetricAlarmRequest 调用 CloudWatchClient 的 `putMetricAlarm` 方法。

导入

```
import software.amazon.awssdk.services.cloudwatch.CloudWatchClient;
import software.amazon.awssdk.services.cloudwatch.model.ComparisonOperator;
import software.amazon.awssdk.services.cloudwatch.model.Dimension;
import software.amazon.awssdk.services.cloudwatch.model.PutMetricAlarmRequest;
import software.amazon.awssdk.services.cloudwatch.model.PutMetricAlarmResponse;
import software.amazon.awssdk.services.cloudwatch.model.StandardUnit;
import software.amazon.awssdk.services.cloudwatch.model.Statistic;
```

代码

```
CloudWatchClient cw =
  CloudWatchClient.builder().build();

Dimension dimension = Dimension.builder()
    .name("InstanceId")
    .value(instanceId).build();

PutMetricAlarmRequest request = PutMetricAlarmRequest.builder()
```

```
    .alarmName(alarmName)
    .comparisonOperator(
        ComparisonOperator.GREATER_THAN_THRESHOLD)
    .evaluationPeriods(1)
    .metricName("CPUUtilization")
    .namespace("AWS/EC2")
    .period(60)
    .statistic(Statistic.AVERAGE)
    .threshold(70.0)
    .actionsEnabled(false)
    .alarmDescription(
        "Alarm when server CPU utilization exceeds 70%")
    .unit(StandardUnit.SECONDS)
    .dimensions(dimension)
    .build();

PutMetricAlarmResponse response = cw.putMetricAlarm(request);
```

请参阅 GitHub 上的完整示例。

列出警报

要列出您已创建的 CloudWatch 警报，请使用您用来设置结果选项的 DescribeAlarmsRequest 调用 CloudWatchClient 的 `describeAlarms` 方法。

导入

```
import software.amazon.awssdk.services.cloudwatch.CloudWatchClient;
import software.amazon.awssdk.services.cloudwatch.model.DescribeAlarmsRequest;
import software.amazon.awssdk.services.cloudwatch.model.DescribeAlarmsResponse;
import software.amazon.awssdk.services.cloudwatch.model.MetricAlarm;
```

代码

```
CloudWatchClient cw = CloudWatchClient.builder().build();

boolean done = false;
String new_token = null;

while(!done) {
 DescribeAlarmsResponse response;
 if (new_token == null) {
  DescribeAlarmsRequest request = DescribeAlarmsRequest.builder().build();
        response = cw.describeAlarms(request);
 }
 else {
  DescribeAlarmsRequest request = DescribeAlarmsRequest.builder()
    .nextToken(new_token)
    .build();
        response = cw.describeAlarms(request);
 }


    for(MetricAlarm alarm : response.metricAlarms()) {
        System.out.printf("Retrieved alarm %s", alarm.alarmName());
    }

    if(response.nextToken() == null) {
        done = true;
    }
    else {
```

```
            new_token = response.nextToken();
        }
    }
```

警报列表可以通过在 `describeAlarms` 返回的 DescribeAlarmsResponse 上调用 `MetricAlarms` 获得。

结果可以分页。要检索下一批结果，请对响应对象调用 `nextToken` 并使用该令牌值构建新的请求对象。然后使用新请求再次调用 `describeAlarms` 方法。

> **Note**
>
> 您还可以使用 CloudWatchClient 的 `describeAlarmsForMetric` 方法检索特定指标的警报。它的使用类似于 `describeAlarms`。

请参阅 GitHub 上的完整示例。

删除警报

要删除 CloudWatch 警报，请使用 DeleteAlarmsRequest (包含您要删除的一个或更多警报名称) 调用 CloudWatchClient 的 `deleteAlarms` 方法。

导入

```
import software.amazon.awssdk.services.cloudwatch.CloudWatchClient;
import software.amazon.awssdk.services.cloudwatch.model.DeleteAlarmsRequest;
import software.amazon.awssdk.services.cloudwatch.model.DeleteAlarmsResponse;
```

代码

```
CloudWatchClient cw = CloudWatchClient.builder().build();


DeleteAlarmsRequest request = DeleteAlarmsRequest.builder()
    .alarmNames(alarm_name).build();

DeleteAlarmsResponse response = cw.deleteAlarms(request);

System.out.printf("Successfully deleted alarm %s", alarm_name);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon CloudWatch User Guide 中的创建 Amazon CloudWatch 警报
- Amazon CloudWatch API Reference 中的 PutMetricAlarm
- Amazon CloudWatch API Reference 中的 DescribeAlarms
- Amazon CloudWatch API Reference 中的 DeleteAlarms

在 CloudWatch 中使用警报操作

利用 CloudWatch 警报操作，您可创建执行自动停止、终止、重启或恢复 Amazon EC2 实例等操作的警报。

> **Note**
>
> 通过在创建警报 (p. 36)时使用 PutMetricAlarmRequest 的 `alarmActions` 方法，可以将警报操作添加到警报。

启用警报操作

要启用 CloudWatch 警报的警报操作，请使用 EnableAlarmActionsRequest (包含一个或多个您要启用的警报的名称) 调用 CloudWatchClient 的 `enableAlarmActions`。

导入

```
import software.amazon.awssdk.services.cloudwatch.CloudWatchClient;
import software.amazon.awssdk.services.cloudwatch.model.EnableAlarmActionsRequest;
import software.amazon.awssdk.services.cloudwatch.model.EnableAlarmActionsResponse;
```

代码

```
CloudWatchClient cw =
  CloudWatchClient.builder().build();

EnableAlarmActionsRequest request = EnableAlarmActionsRequest.builder()
    .alarmNames(alarm).build();

EnableAlarmActionsResponse response = cw.enableAlarmActions(request);
```

请参阅 GitHub 上的完整示例。

禁用警报操作

要禁用 CloudWatch 警报的警报操作，请使用 DisableAlarmActionsRequest (包含一个或多个您要禁用的警报的名称) 调用 CloudWatchClient 的 `disableAlarmActions`。

导入

```
import software.amazon.awssdk.services.cloudwatch.CloudWatchClient;
import software.amazon.awssdk.services.cloudwatch.model.DisableAlarmActionsRequest;
import software.amazon.awssdk.services.cloudwatch.model.DisableAlarmActionsResponse;
```

代码

```
CloudWatchClient cw = CloudWatchClient.builder().build();


DisableAlarmActionsRequest request = DisableAlarmActionsRequest.builder()
    .alarmNames(alarmName).build();

DisableAlarmActionsResponse response = cw.disableAlarmActions(request);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon CloudWatch User Guide 中的创建警报以停止、终止、重启或恢复实例
- Amazon CloudWatch API Reference 中的 PutMetricAlarm
- Amazon CloudWatch API Reference 中的 EnableAlarmActions
- Amazon CloudWatch API Reference 中的 DisableAlarmActions

将事件发送到 CloudWatch

CloudWatch Events 提供几乎实时的系统事件流，这些事件描述 AWS 资源中对 Amazon EC2 实例、Lambda 函数、Kinesis 流、Amazon ECS 任务、Step Functions 状态机、Amazon SNS 主题、Amazon SQS 队列或内置目标的更改。通过使用简单的规则，您可以匹配事件并将事件路由到一个或多个目标函数或流。

添加事件

要添加自定义 CloudWatch 事件，请使用包含一个或多个 PutEventsRequestEntry 对象 (提供每个事件的详细信息) 的 PutEventsRequest 对象调用 CloudWatchEventsClient 的 `putEvents` 方法。您可以为条目指定多个参数，例如事件的来源和类型、与事件相关联的资源等等。

> **Note**
>
> 对于每个 `putEvents` 调用，您最多可以指定 10 个事件。

导入

```
import software.amazon.awssdk.services.cloudwatchevents.CloudWatchEventsClient;
import software.amazon.awssdk.services.cloudwatchevents.model.PutEventsRequest;
import software.amazon.awssdk.services.cloudwatchevents.model.PutEventsRequestEntry;
import software.amazon.awssdk.services.cloudwatchevents.model.PutEventsResponse;
```

代码

```
CloudWatchEventsClient cwe =
  CloudWatchEventsClient.builder().build();

final String EVENT_DETAILS =
    "{ \"key1\": \"value1\", \"key2\": \"value2\" }";

PutEventsRequestEntry request_entry = PutEventsRequestEntry.builder()
    .detail(EVENT_DETAILS)
    .detailType("sampleSubmitted")
    .resources(resource_arn)
    .source("aws-sdk-java-cloudwatch-example").build();

PutEventsRequest request = PutEventsRequest.builder()
    .entries(request_entry).build();

PutEventsResponse response = cwe.putEvents(request);
```

请参阅 GitHub 上的完整示例。

添加规则

要创建或更新规则，请使用包含规则名称和可选参数的 PutRuleRequest 调用 CloudWatchEventsClient 的 `putRule` 方法，可选参数如事件模式、与规则相关联的 IAM 角色以及描述规则运行频率的计划表达式。

导入

```
import software.amazon.awssdk.services.cloudwatchevents.CloudWatchEventsClient;
import software.amazon.awssdk.services.cloudwatchevents.model.PutRuleRequest;
import software.amazon.awssdk.services.cloudwatchevents.model.PutRuleResponse;
import software.amazon.awssdk.services.cloudwatchevents.model.RuleState;
```

代码

```
CloudWatchEventsClient cwe =
  CloudWatchEventsClient.builder().build();

PutRuleRequest request = PutRuleRequest.builder()
    .name(rule_name)
    .roleArn(role_arn)
    .scheduleExpression("rate(5 minutes)")
    .state(RuleState.ENABLED)
    .build();

PutRuleResponse response = cwe.putRule(request);
```

请参阅 GitHub 上的完整示例。

添加目标

目标是触发规则时调用的资源。举例来说，目标包括 Amazon EC2 实例、Lambda 函数、Kinesis 流、Amazon ECS 任务、Step Functions 状态机和内置目标。

要向规则添加目标，请使用 PutTargetsRequest (包含要更新的规则和要添加到规则的目标列表) 来调用 CloudWatchEventsClient 的 `putTargets` 方法。

导入

```
import software.amazon.awssdk.services.cloudwatchevents.CloudWatchEventsClient;
import software.amazon.awssdk.services.cloudwatchevents.model.PutTargetsRequest;
import software.amazon.awssdk.services.cloudwatchevents.model.PutTargetsResponse;
import software.amazon.awssdk.services.cloudwatchevents.model.Target;
```

代码

```
CloudWatchEventsClient cwe =
  CloudWatchEventsClient.builder().build();

Target target = Target.builder()
    .arn(function_arn)
    .id(target_id)
    .build();

PutTargetsRequest request = PutTargetsRequest.builder()
    .targets(target)
    .rule(rule_name)
    .build();

PutTargetsResponse response = cwe.putTargets(request);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon CloudWatch Events User Guide 中的使用 PutEvents 添加事件
- Amazon CloudWatch Events User Guide 中的规则的计划表达式
- Amazon CloudWatch Events User Guide 中的 CloudWatch Events 的事件类型
- Amazon CloudWatch Events User Guide 中的事件和事件模式
- Amazon CloudWatch Events API Reference 中的 PutEvents

- Amazon CloudWatch Events API Reference 中的 PutTargets
- Amazon CloudWatch Events API Reference 中的 PutRule

使用 AWS SDK for Java 的 DynamoDB 示例

本节提供了使用适用于 Java 的 AWS 开发工具包 2.0 对 DynamoDB 进行编程的示例。

以下示例仅包含演示每种方法所需的代码。完整的示例代码在 GitHub 上提供。您可以从中下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行它们。

主题

在 DynamoDB 中使用表

表是 DynamoDB 数据库中所有项目的容器。您必须先创建表，然后才能在 DynamoDB 中添加或删除数据。

对于每个表，您必须定义：

- 表名称，它对于您的账户和所在区域都是唯一的。
- 一个主键，每个值对于它都必须是唯一的；表中的任意两个项目不能具有相同的主键值。

  主键可以是简单主键 (包含单个分区 (HASH) 键) 或复合主键 (包含一个分区和一个排序 (RANGE) 键)。

  每个键值均有一个由 ScalarAttributeType 类枚举的关联的数据类型。键值可以是二进制 (B)、数字 (N) 或字符串 (S)。有关更多信息，请参阅 Amazon DynamoDB Developer Guide 中的命名规则和数据类型。
- 预置吞吐量 是定义为表保留的读取/写入容量单位数的值。

  Note

  Amazon DynamoDB 定价基于您为表设置的预置吞吐量值，因此您应只为表保留可能需要的容量。
  表的预置吞吐量可随时修改，以便您能够在需要更改时调整容量。

创建表

使用 DynamoDBClient 的 `createTable` 方法可创建新的 DynamoDB 表。您需要构造表属性和表架构，二者用于标识表的主键。您还必须提供初始预置吞吐量值和表名。

Note

如果使用您所选名称的表已存在，则将引发 DynamoDBException。

导入

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
```

创建具有简单主键的表

此代码使用简单主键 (“Name”) 创建表。

代码

```
CreateTableRequest request = CreateTableRequest.builder()
  .attributeDefinitions(AttributeDefinition.builder()
    .attributeName("Name")
    .attributeType(ScalarAttributeType.S)
    .build())
  .keySchema(KeySchemaElement.builder()
    .attributeName("Name")
    .keyType(KeyType.HASH)
    .build())
  .provisionedThroughput(ProvisionedThroughput.builder()
    .readCapacityUnits(new Long(10))
    .writeCapacityUnits(new Long(10))
    .build())
  .tableName(table_name)
  .build();

DynamoDbClient ddb = DynamoDbClient.create();

try {
    CreateTableResponse response = ddb.createTable(request);
    System.out.println(response.tableDescription().tableName());
} catch (DynamoDbException e) {
    System.err.println(e.errorMessage());
    System.exit(1);
}
System.out.println("Done!");
```

请参阅 GitHub 上的完整示例。

创建具有复合主键的表

添加另一个 AttributeDefinition 和 KeySchemaElement 到 CreateTableRequest。

代码

```
CreateTableRequest request = CreateTableRequest.builder()
    .attributeDefinitions(
          AttributeDefinition.builder()
          .attributeName("Language")
          .attributeType(ScalarAttributeType.S)
          .build(),
          AttributeDefinition.builder()
          .attributeName("Greeting")
          .attributeType(ScalarAttributeType.S)
          .build())
    .keySchema(
      KeySchemaElement.builder()
      .attributeName("Language")
      .keyType(KeyType.HASH)
      .build(),
      KeySchemaElement.builder()
      .attributeName("Greeting")
      .keyType(KeyType.RANGE)
      .build())
    .provisionedThroughput(
      ProvisionedThroughput.builder()
      .readCapacityUnits(new Long(10))
```

```
      .writeCapacityUnits(new Long(10)).build())
    .tableName(table_name)
    .build();

DynamoDbClient ddb = DynamoDbClient.create();

try {
    CreateTableResponse result = ddb.createTable(request);
} catch (DynamoDbException e) {
    System.err.println(e.errorMessage());
    System.exit(1);
}
System.out.println("Done!");
```

请参阅 GitHub 上的完整示例。

列出表

您可以通过调用 DynamoDBClient 的 `listTables` 方法列出特定区域中的表。

> **Note**
>
> 如果您的账户和区域没有该已命名的表，则将引发 ResourceNotFoundException 异常。

导入

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;
import software.amazon.awssdk.services.dynamodb.model.ListTablesRequest;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
```

代码

```
DynamoDbClient ddb = DynamoDbClient.create();

boolean more_tables = true;
String last_name = null;

while(more_tables) {
    try {
        ListTablesResponse response = null;
        if (last_name == null) {
         ListTablesRequest request = ListTablesRequest.builder().build();
            response = ddb.listTables(request);
        }
        else {
         ListTablesRequest request = ListTablesRequest.builder()
           .exclusiveStartTableName(last_name).build();
            response = ddb.listTables(request);
        }

        List<String> table_names = response.tableNames();

        if (table_names.size() > 0) {
            for (String cur_name : table_names) {
                System.out.format("* %s\n", cur_name);
            }
        } else {
            System.out.println("No tables found!");
            System.exit(0);
```

```
        }

        last_name = response.lastEvaluatedTableName();
        if (last_name == null) {
            more_tables = false;
        }
    } catch (DynamoDbException e) {
        System.err.println(e.errorMessage());
        System.exit(1);
    }
}
System.out.println("\nDone!");
```

默认情况下，每次调用将返回最多 100 个表 - 对返回的 ListTablesResponse 对象使用 `lastEvaluatedTableName` 可获得评估的上一个表。可使用此值在上一列出的最后一个返回值后开始列出。

请参阅 GitHub 上的完整示例。

描述表 (获取相关信息)

调用 DynamoDBClient 的 `describeTable` 方法。

> **Note**
>
> 如果您的账户和区域没有该已命名的表，则将引发 ResourceNotFoundException 异常。

导入

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughputDescription;
import software.amazon.awssdk.services.dynamodb.model.TableDescription;
```

代码

```
DynamoDbClient ddb = DynamoDbClient.create();

DescribeTableRequest request = DescribeTableRequest.builder()
  .tableName(table_name)
  .build();

try {
    TableDescription table_info =
       ddb.describeTable(request).table();

    if (table_info != null) {
        System.out.format("Table name  : %s\n",
              table_info.tableName());
        System.out.format("Table ARN   : %s\n",
              table_info.tableArn());
        System.out.format("Status      : %s\n",
              table_info.tableStatus());
        System.out.format("Item count  : %d\n",
              table_info.itemCount().longValue());
        System.out.format("Size (bytes): %d\n",
              table_info.tableSizeBytes().longValue());

        ProvisionedThroughputDescription throughput_info =
           table_info.provisionedThroughput();
```

```
        System.out.println("Throughput");
        System.out.format("  Read Capacity : %d\n",
              throughput_info.readCapacityUnits().longValue());
        System.out.format("  Write Capacity: %d\n",
              throughput_info.writeCapacityUnits().longValue());

        List<AttributeDefinition> attributes =
           table_info.attributeDefinitions();
        System.out.println("Attributes");
        for (AttributeDefinition a : attributes) {
            System.out.format("  %s (%s)\n",
                  a.attributeName(), a.attributeType());
        }
    }
} catch (DynamoDbException e) {
    System.err.println(e.errorMessage());
    System.exit(1);
}
System.out.println("\nDone!");
```

请参阅 GitHub 上的完整示例。

修改 (更新) 表

您可以通过调用 DynamoDBClient 的 `updateTable` 方法随时修改表的预置吞吐量值。

> **Note**
>
> 如果您的账户和区域没有该已命名的表，则将引发 ResourceNotFoundException 异常。

导入

```
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateTableRequest;
```

代码

```
ProvisionedThroughput table_throughput = ProvisionedThroughput.builder()
  .readCapacityUnits(read_capacity)
  .writeCapacityUnits(write_capacity)
  .build();

DynamoDbClient ddb = DynamoDbClient.create();

UpdateTableRequest request = UpdateTableRequest.builder()
  .provisionedThroughput(table_throughput)
  .tableName(table_name)
  .build();

try {
    ddb.updateTable(request);
} catch (DynamoDbException e) {
    System.err.println(e.errorMessage());
    System.exit(1);
}

System.out.println("Done!");
```

请参阅 GitHub 上的完整示例。

删除表

调用 DynamoDBClient 的 `deleteTable` 方法并向其传递表名称。

> Note
>
> 如果您的账户和区域没有该已命名的表，则将引发 ResourceNotFoundException 异常。

导入

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;

import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;
```

代码

```
DynamoDbClient ddb = DynamoDbClient.create();

DeleteTableRequest request = DeleteTableRequest.builder()
  .tableName(table_name)
  .build();

try {
    ddb.deleteTable(request);
} catch (DynamoDbException e) {
    System.err.println(e.errorMessage());
    System.exit(1);
}
System.out.println("Done!");
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon DynamoDB Developer Guide 中的表使用指南
- Amazon DynamoDB Developer Guide 中的在 DynamoDB 中使用表

在 DynamoDB 中处理项目

在 DynamoDB 中，项目是属性的集合，每个项目都包括一个名称和一个值。属性值可以为标量、集或文档类型。有关更多信息，请参阅 Amazon DynamoDB Developer Guide 中的命名规则和数据类型。

检索 (获取) 表中的项目

调用 DynamoDBClient 的 `getItem` 方法，并向其传递 GetItemRequest 对象，包含您所需项目的表名称和主键值。该方法返回包含该项目的所有属性的 GetItemResponse 对象。您可以在 `GetItemRequest` 中指定一个或多个投影表达式以检索特定属性。

可以使用所返回 `GetItemResponse` 对象的 `item()` 方法，检索与项目关联的映射 (键 (字符串) 和值 (AttributeValue) 对)。

导入

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
```

```
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
```

代码

```
HashMap<String,AttributeValue> key_to_get =
    new HashMap<String,AttributeValue>();

key_to_get.put("Name", AttributeValue.builder()
  .s(name).build());

GetItemRequest request = null;
if (projection_expression != null) {
    request = GetItemRequest.builder()
        .key(key_to_get)
        .tableName(table_name)
        .projectionExpression(projection_expression)
        .build();
} else {
    request = GetItemRequest.builder()
        .key(key_to_get)
        .tableName(table_name)
        .build();
}

DynamoDbClient ddb = DynamoDbClient.create();

try {
    Map<String,AttributeValue> returned_item =
       ddb.getItem(request).item();
    if (returned_item != null) {
        Set<String> keys = returned_item.keySet();
        for (String key : keys) {
            System.out.format("%s: %s\n",
                    key, returned_item.get(key).toString());
        }
    } else {
        System.out.format("No item found with the key %s!\n", name);
    }
} catch (DynamoDbException e) {
    System.err.println(e.errorMessage());
    System.exit(1);
}
```

请参阅 GitHub 上的完整示例。

在表中添加新项目

创建表示项目属性的键值对映射。其中必须包括表的主键字段的值。如果主键标识的项目已存在，那么其字段将通过该请求更新。

> **Note**
>
> 如果您的账户和区域没有该已命名的表，则将引发 ResourceNotFoundException 异常。

导入

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
```

代码

```
HashMap<String,AttributeValue> item_values =
    new HashMap<String,AttributeValue>();

item_values.put("Name", AttributeValue.builder().s(name).build());

for (String[] field : extra_fields) {
    item_values.put(field[0], AttributeValue.builder().s(field[1]).build());
}

DynamoDbClient ddb = DynamoDbClient.create();
PutItemRequest request = PutItemRequest.builder()
  .tableName(table_name)
  .item(item_values)
  .build();

try {
    ddb.putItem(request);
} catch (ResourceNotFoundException e) {
    System.err.format("Error: The table \"%s\" can't be found.\n", table_name);
    System.err.println("Be sure that it exists and that you've typed its name correctly!");
    System.exit(1);
} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
System.out.println("Done!");
```

请参阅 GitHub 上的完整示例。

更新表中现有项目

可以使用 DynamoDBClient 的 `updateItem` 方法，通过提供要更新的表名称、主键值和字段映射，更新表中已有项目的属性。

> **Note**
>
> 如果您的账户和区域没有该已命名的表，或者不存在传入的主键标识的项目，会导致 ResourceNotFoundException 异常。

导入

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.AttributeAction;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.AttributeValueUpdate;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
```

代码

```
HashMap<String,AttributeValue> item_key =
   new HashMap<String,AttributeValue>();

item_key.put("Name", AttributeValue.builder().s(name).build());

HashMap<String,AttributeValueUpdate> updated_values =
    new HashMap<String,AttributeValueUpdate>();
```

```
for (String[] field : extra_fields) {
    updated_values.put(field[0], AttributeValueUpdate.builder()
      .value(AttributeValue.builder().s(field[1]).build())
      .action(AttributeAction.PUT)
      .build());
}

UpdateItemRequest request = UpdateItemRequest.builder()
  .tableName(table_name)
  .key(item_key)
  .attributeUpdates(updated_values)
  .build();

DynamoDbClient ddb = DynamoDbClient.create();

try {
    ddb.updateItem(request);
} catch (ResourceNotFoundException e) {
    System.err.println(e.getMessage());
    System.exit(1);
} catch (DynamoDbException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
System.out.println("Done!");
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon DynamoDB Developer Guide 中的项目处理准则
- Amazon DynamoDB Developer Guide 中的使用 DynamoDB 中的项目

使用 AWS SDK for Java (Developer Preview) 的 Amazon EC2 示例

此部分提供使用适用于 Java 的 AWS 开发工具包 2.0 对 Amazon EC2 进行编程的示例。

> **Important**
>
> 这是一个预览版，不建议用于生产环境。

主题

管理 Amazon EC2 实例

> **Important**
>
> 这是一个预览版，不建议用于生产环境。

创建实例

要创建新 Amazon EC2 实例，请调用 EC2Client 的 `runInstances` 方法，并为它提供 RunInstancesRequest，其中包含要使用的 Amazon Machine Image (AMI) 和一个实例类型。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.InstanceType;
import software.amazon.awssdk.services.ec2.model.RunInstancesRequest;
import software.amazon.awssdk.services.ec2.model.RunInstancesResponse;
import software.amazon.awssdk.services.ec2.model.Tag;
import software.amazon.awssdk.services.ec2.model.CreateTagsRequest;
```

代码

```
EC2Client ec2 = EC2Client.create();

RunInstancesRequest run_request = RunInstancesRequest.builder()
    .imageId(ami_id)
    .instanceType(InstanceType.T1_MICRO)
    .maxCount(1)
    .minCount(1)
    .build();

RunInstancesResponse response = ec2.runInstances(run_request);

String instance_id = response.reservation().reservationId();

Tag tag = Tag.builder()
    .key("Name")
    .value(name)
    .build();

CreateTagsRequest tag_request = CreateTagsRequest.builder()
    .tags(tag)
    .build();

try {
 ec2.createTags(tag_request);

    System.out.printf(
        "Successfully started EC2 instance %s based on AMI %s",
        instance_id, ami_id);
}
catch (EC2Exception e) {
 System.err.println(e.errorMessage());
 System.exit(1);
}
```

请参阅 GitHub 上的完整示例。

启动实例

要启动 Amazon EC2 实例，请调用 EC2Client 的 `startInstances` 方法，并为它提供 StartInstancesRequest，其中包含要启动实例的 ID。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.StartInstancesRequest;
```

代码

```
EC2Client ec2 = EC2Client.create();

StartInstancesRequest request = StartInstancesRequest.builder()
    .instanceIds(instance_id).build();

ec2.startInstances(request);
```

请参阅 GitHub 上的完整示例。

停止实例

要停止 Amazon EC2 实例，请调用 EC2Client 的 `stopInstances` 方法，并为它提供 StopInstancesRequest，其中包含要停止实例的 ID。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.StopInstancesRequest;
```

代码

```
EC2Client ec2 = EC2Client.create();

StopInstancesRequest request = StopInstancesRequest.builder()
    .instanceIds(instance_id).build();

ec2.stopInstances(request);
```

请参阅 GitHub 上的完整示例。

重启实例

要重启 Amazon EC2 实例，请调用 EC2Client 的 `rebootInstances` 方法，并为它提供 RebootInstancesRequest，其中包含要重启实例的 ID。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.RebootInstancesRequest;
import software.amazon.awssdk.services.ec2.model.RebootInstancesResponse;
```

代码

```
EC2Client ec2 = EC2Client.create();

RebootInstancesRequest request = RebootInstancesRequest.builder()
    .instanceIds(instance_id).build();

RebootInstancesResponse response = ec2.rebootInstances(request);
```

请参阅 GitHub 上的完整示例。

描述实例

要列出您的实例，您需要创建 DescribeInstancesRequest 并调用 EC2Client 的 `describeInstances` 方法。该方法将返回 DescribeInstancesResponse 对象，您可以用它来列出您的账户和区域的 Amazon EC2 实例。

实例按预留进行分组。每个预留对应对启动实例的 `startInstances` 的调用。要列出您的实例，您必须先调用 `DescribeInstancesResponse` 类的 `reservations` 方法，然后在每个返回的 Reservation 对象上调用 `instances`。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.DescribeInstancesRequest;
import software.amazon.awssdk.services.ec2.model.DescribeInstancesResponse;
import software.amazon.awssdk.services.ec2.model.Instance;
import software.amazon.awssdk.services.ec2.model.Reservation;
```

代码

```
DescribeInstancesRequest request = DescribeInstancesRequest.builder().build();

while(!done) {
    DescribeInstancesResponse response = ec2.describeInstances(request);

    for(Reservation reservation : response.reservations()) {
        for(Instance instance : reservation.instances()) {
            System.out.printf(
                "Found reservation with id %s, " +
                "AMI %s, " +
                "type %s, " +
                "state %s " +
                "and monitoring state %s",
                instance.instanceId(),
                instance.imageId(),
                instance.instanceType(),
                instance.state().name(),
                instance.monitoring().state());
            System.out.println("");
        }
    }

    if(response.nextToken() == null) {
        done = true;
    }
}
```

结果将分页；您可以获取更多结果，方法是将从结果对象的 `nextToken` 方法返回的值传递到新请求对象的 `nextToken` 方法，然后在下一个 `describeInstances` 调用中使用新请求对象。

请参阅 GitHub 上的完整示例。

监控实例

您可以监控 Amazon EC2 实例的各方面，例如 CPU 和网络利用率、可用内存和剩余磁盘空间。要了解有关实例监控的更多信息，请参阅 Amazon EC2 User Guide for Linux Instances 中的监控 Amazon EC2。

要开始监控实例，您必须用要监控实例的 ID 创建一个 MonitorInstancesRequest，并将其传递给 EC2Client 的 `monitorInstances` 方法。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.MonitorInstancesRequest;
```

代码

```
EC2Client ec2 = EC2Client.create();

MonitorInstancesRequest request = MonitorInstancesRequest.builder()
        .instanceIds(instance_id).build();

ec2.monitorInstances(request);
```

请参阅 GitHub 上的完整示例。

停止实例监控

要停止监控实例，您必须用要停止监控实例的 ID 创建一个 UnmonitorInstancesRequest，并将其传递给 EC2Client 的 `unmonitorInstances` 方法。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.UnmonitorInstancesRequest;
```

代码

```
EC2Client ec2 = EC2Client.create();

UnmonitorInstancesRequest request = UnmonitorInstancesRequest.builder()
    .instanceIds(instance_id).build();

ec2.unmonitorInstances(request);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon EC2 API Reference 中的 RunInstances
- Amazon EC2 API Reference 中的 DescribeInstances
- Amazon EC2 API Reference 中的 StartInstances
- Amazon EC2 API Reference 中的 StopInstances
- Amazon EC2 API Reference 中的 RebootInstances
- Amazon EC2 API Reference 中的 DescribeInstances
- Amazon EC2 API Reference 中的 MonitorInstances
- Amazon EC2 API Reference 中的 UnmonitorInstances

在 Amazon EC2 中使用弹性 IP 地址

Important

这是一个预览版，不建议用于生产环境。

分配弹性 IP 地址

要使用弹性 IP 地址，您应首先向您的账户分配这样一个地址，然后将其与您的实例或网络接口关联。

要分配弹性 IP 地址，请使用包含网络类型 (经典 EC2 或 VPC) 的 AllocateAddressRequest 对象调用 EC2Client 的 `allocateAddress` 方法。

返回的 AllocateAddressResponse 包含一个分配 ID，可用于将地址与实例关联，方法是将 AssociateAddressRequest 中的分配 ID 和实例 ID 传递给 EC2Client 的 `associateAddress` 方法。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.AllocateAddressRequest;
import software.amazon.awssdk.services.ec2.model.AllocateAddressResponse;
import software.amazon.awssdk.services.ec2.model.AssociateAddressRequest;
import software.amazon.awssdk.services.ec2.model.AssociateAddressResponse;
import software.amazon.awssdk.services.ec2.model.DomainType;
```

代码

```
EC2Client ec2 = EC2Client.create();

AllocateAddressRequest allocate_request = AllocateAddressRequest.builder()
    .domain(DomainType.VPC)
    .build();

AllocateAddressResponse allocate_response =
    ec2.allocateAddress(allocate_request);

String allocation_id = allocate_response.allocationId();

AssociateAddressRequest associate_request =
    AssociateAddressRequest.builder()
        .instanceId(instance_id)
        .allocationId(allocation_id)
        .build();

AssociateAddressResponse associate_response =
    ec2.associateAddress(associate_request);

System.out.printf(
    "Successfully associated Elastic IP address %s " +
    "with instance %s",
    associate_response.associationId(),
    instance_id);
```

请参阅 GitHub 上的完整示例。

描述弹性 IP 地址

要列出分配到您的账户的弹性 IP 地址，请调用 EC2Client 的 `describeAddresses` 方法。它将返回 DescribeAddressesResponse，可用于获取账户中代表弹性 IP 地址的 Address 对象的列表。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.Address;
import software.amazon.awssdk.services.ec2.model.DescribeAddressesResponse;
```

代码

```
EC2Client ec2 = EC2Client.create();

DescribeAddressesResponse response = ec2.describeAddresses();

for(Address address : response.addresses()) {
    System.out.printf(
            "Found address with public IP %s, " +
            "domain %s, " +
            "allocation id %s " +
            "and NIC id %s",
            address.publicIp(),
            address.domain(),
            address.allocationId(),
            address.networkInterfaceId());
}
```

请参阅 GitHub 上的完整示例。

释放弹性 IP 地址

要释放弹性 IP 地址，请调用 EC2Client 的 `releaseAddress` 方法，向其传递 ReleaseAddressRequest，包含您要释放的弹性 IP 地址的分配 ID。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.ReleaseAddressRequest;
import software.amazon.awssdk.services.ec2.model.ReleaseAddressResponse;
```

代码

```
EC2Client ec2 = EC2Client.create();

ReleaseAddressRequest request = ReleaseAddressRequest.builder()
    .allocationId(alloc_id).build();

ReleaseAddressResponse response = ec2.releaseAddress(request);
```

在释放弹性 IP 地址后，它将回到 AWS IP 地址池，您此后不能再使用该地址。请务必更新您的 DNS 记录和通过该地址进行通信的任何服务器或设备。

如果您使用的是 EC2-Classic 或默认 VPC，则释放弹性 IP 地址会自动断开该地址与任何实例的关联。要在不释放的情况下取消关联弹性 IP 地址，请使用 EC2Client 的 `disassociateAddress` 方法。

如果您使用的是非默认 VPC，则必须使用 `disassociateAddress` 取消弹性 IP 地址的关联，然后再尝试释放它。否则，Amazon EC2 会返回错误 (InvalidIPAddress.InUse)。

请参阅 GitHub 上的完整示例。

更多信息

- Amazon EC2 User Guide for Linux Instances 中的弹性 IP 地址
- Amazon EC2 API Reference 中的 AllocateAddress
- Amazon EC2 API Reference 中的 DescribeAddresses
- Amazon EC2 API Reference 中的 ReleaseAddress

使用区域和可用区

Important

这是一个预览版，不建议用于生产环境。

描述区域

要列出账户的可用区域，请调用 EC2Client 的 `describeRegions` 方法。该方法返回 DescribeRegionsResponse。调用返回对象的 `regions` 方法，获取表示各个区域的 Region 对象的列表。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.DescribeRegionsResponse;
import software.amazon.awssdk.services.ec2.model.Region;
```

代码

```
EC2Client ec2 = EC2Client.create();

DescribeRegionsResponse regions_response = ec2.describeRegions();

for(Region region : regions_response.regions()) {
    System.out.printf(
        "Found region %s " +
        "with endpoint %s",
        region.regionName(),
        region.endpoint());
    System.out.println();
```

请参阅 GitHub 上的完整示例。

描述可用区

要列出账户的每个可用区域，请调用 EC2Client 的 `describeAvailabilityZones` 方法。该方法返回 DescribeAvailabilityZonesResponse。调用其 `availabilityZones` 方法，获取表示各个可用区的 AvailabilityZone 对象的列表。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.AvailabilityZone;
import software.amazon.awssdk.services.ec2.model.DescribeAvailabilityZonesResponse;
```

代码

```
EC2Client ec2 = EC2Client.create();

DescribeAvailabilityZonesResponse zones_response =
    ec2.describeAvailabilityZones();

for(AvailabilityZone zone : zones_response.availabilityZones()) {
    System.out.printf(
        "Found availability zone %s " +
        "with status %s " +
        "in region %s",
        zone.zoneName(),
```

```
        zone.state(),
        zone.regionName());
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon EC2 User Guide for Linux Instances 中的区域和可用区
- Amazon EC2 API Reference 中的 DescribeRegions
- Amazon EC2 API Reference 中的 DescribeAvailabilityZones

使用 Amazon EC2 密钥对

> Important
>
> 这是一个预览版，不建议用于生产环境。

创建密钥对

要创建密钥对，请使用包含密钥名称的 CreateKeyPairRequest 调用 EC2Client 的 `createKeyPair` 方法。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.CreateKeyPairRequest;
import software.amazon.awssdk.services.ec2.model.CreateKeyPairResponse;
```

代码

```
EC2Client ec2 = EC2Client.create();

CreateKeyPairRequest request = CreateKeyPairRequest.builder()
    .keyName(key_name).build();

CreateKeyPairResponse response = ec2.createKeyPair(request);
```

请参阅 GitHub 上的完整示例。

描述密钥对

要列出您的密钥对或获取相关信息，请调用 EC2Client 的 `describeKeyPairs` 方法。该方法返回 DescribeKeyPairsResponse，使用它后，您可以通过调用其 `keyPairs` 方法（返回一个 KeyPairInfo 对象的列表）来访问密钥对的列表。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.DescribeKeyPairsResponse;
import software.amazon.awssdk.services.ec2.model.KeyPairInfo;
```

代码

```
EC2Client ec2 = EC2Client.create();

DescribeKeyPairsResponse response = ec2.describeKeyPairs();
```

```
for(KeyPairInfo key_pair : response.keyPairs()) {
    System.out.printf(
        "Found key pair with name %s " +
        "and fingerprint %s",
        key_pair.keyName(),
        key_pair.keyFingerprint());
    System.out.println("");
}
```

请参阅 GitHub 上的完整示例。

删除密钥对

要删除密钥对，请调用 EC2Client 的 `deleteKeyPair` 方法，将其传递给一个包含要删除密钥对名称的 DeleteKeyPairRequest。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.DeleteKeyPairRequest;
import software.amazon.awssdk.services.ec2.model.DeleteKeyPairResponse;
```

代码

```
EC2Client ec2 = EC2Client.create();

DeleteKeyPairRequest request = DeleteKeyPairRequest.builder()
    .keyName(key_name)
    .build();

DeleteKeyPairResponse response = ec2.deleteKeyPair(request);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon EC2 User Guide for Linux Instances 中的 Amazon EC2 密钥对
- Amazon EC2 API Reference 中的 CreateKeyPair
- Amazon EC2 API Reference 中的 DescribeKeyPairs
- Amazon EC2 API Reference 中的 DeleteKeyPair

在 Amazon EC2 中使用安全组

> Important
>
> 这是一个预览版，不建议用于生产环境。

正在创建安全组

要创建安全组，请使用包含密钥名称的 CreateSecurityGroupRequest 调用 EC2Client 的 `createSecurityGroup` 方法。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
```

```
import software.amazon.awssdk.services.ec2.model.CreateSecurityGroupRequest;
import software.amazon.awssdk.services.ec2.model.CreateSecurityGroupResponse;
```

代码

```
EC2Client ec2 = EC2Client.create();

CreateSecurityGroupRequest create_request = CreateSecurityGroupRequest.builder()
        .groupName(group_name)
        .description(group_desc)
        .vpcId(vpc_id)
        .build();

CreateSecurityGroupResponse create_response =
    ec2.createSecurityGroup(create_request);
```

请参阅 GitHub 上的完整示例。

配置安全组

安全组可以控制对 Amazon EC2 实例的入站 (入口) 流量和出站 (出口) 流量。

要向安全组添加入口规则，请使用 EC2Client 的 `authorizeSecurityGroupIngress` 方法，提供安全组的名称和您想要在 AuthorizeSecurityGroupIngressRequest 对象中分配给安全组的访问规则 (IpPermission)。以下示例介绍如何将 IP 权限添加到安全组。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.AuthorizeSecurityGroupIngressRequest;
import software.amazon.awssdk.services.ec2.model.AuthorizeSecurityGroupIngressResponse;
import software.amazon.awssdk.services.ec2.model.IpPermission;
import software.amazon.awssdk.services.ec2.model.IpRange;
```

代码

```
EC2Client ec2 = EC2Client.create();
IpRange ip_range = IpRange.builder()
    .cidrIp("0.0.0.0/0").build();

IpPermission ip_perm = IpPermission.builder()
    .ipProtocol("tcp")
    .toPort(80)
    .fromPort(80)
    .ipv4Ranges(ip_range)
    .build();

IpPermission ip_perm2 = IpPermission.builder()
    .ipProtocol("tcp")
    .toPort(22)
    .fromPort(22)
    .ipv4Ranges(ip_range)
    .build();

AuthorizeSecurityGroupIngressRequest auth_request =
    AuthorizeSecurityGroupIngressRequest.builder()
        .groupName(group_name)
        .ipPermissions(ip_perm, ip_perm2)
        .build();
```

```
AuthorizeSecurityGroupIngressResponse auth_response =
    ec2.authorizeSecurityGroupIngress(auth_request);
```

要向安全组添加出口规则，请在 AuthorizeSecurityGroupEgressRequest 中向 EC2Client 的 `authorizeSecurityGroupEgress` 方法提供相似的数据。

请参阅 GitHub 上的完整示例。

描述安全组

要描述您的安全组或获取相关信息，请调用 EC2Client 的 `describeSecurityGroups` 方法。该方法返回 DescribeSecurityGroupsResponse，使用它后，您可以通过调用其 `securityGroups` 方法（返回一个 SecurityGroup 对象的列表）来访问安全组的列表。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.DescribeSecurityGroupsRequest;
import software.amazon.awssdk.services.ec2.model.DescribeSecurityGroupsResponse;
import software.amazon.awssdk.services.ec2.model.SecurityGroup;
```

代码

```
EC2Client ec2 = EC2Client.create();

DescribeSecurityGroupsRequest request =
    DescribeSecurityGroupsRequest.builder()
        .groupIds(group_id).build();

DescribeSecurityGroupsResponse response =
    ec2.describeSecurityGroups(request);

for(SecurityGroup group : response.securityGroups()) {
    System.out.printf(
        "Found security group with id %s, " +
        "vpc id %s " +
        "and description %s",
        group.groupId(),
        group.vpcId(),
        group.description());
}
```

请参阅 GitHub 上的完整示例。

正在删除安全组

要删除安全组，请调用 EC2Client 的 `deleteSecurityGroup` 方法，将其传递给一个包含要删除安全组 ID 的 DeleteSecurityGroupRequest。

导入

```
import software.amazon.awssdk.services.ec2.EC2Client;
import software.amazon.awssdk.services.ec2.model.DeleteSecurityGroupRequest;
import software.amazon.awssdk.services.ec2.model.DeleteSecurityGroupResponse;
```

代码

```
EC2Client ec2 = EC2Client.create();
```

```
DeleteSecurityGroupRequest request = DeleteSecurityGroupRequest.builder()
    .groupId(group_id)
    .build();

DeleteSecurityGroupResponse response = ec2.deleteSecurityGroup(request);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon EC2 User Guide for Linux Instances 中的 Amazon EC2 安全组
- Amazon EC2 User Guide for Linux Instances 中的为您的 Linux 实例授权入站流量
- Amazon EC2 API Reference 中的 CreateSecurityGroup
- Amazon EC2 API Reference 中的 DescribeSecurityGroups
- Amazon EC2 API Reference 中的 DeleteSecurityGroup
- Amazon EC2 API Reference 中的 AuthorizeSecurityGroupIngress

使用 AWS SDK for Java 的 IAM 示例

本节提供 IAM 编程 (使用适用于 Java 的 AWS 开发工具包) 的示例。

> **Important**
>
> 这是一个预览版，不建议用于生产环境。

AWS Identity and Access Management (IAM) 使您能够安全地控制用户对 AWS 服务和资源的访问权限。您可以使用 IAM 创建和管理 AWS 用户和组，并使用各种权限来允许或拒绝他们对 AWS 资源的访问。有关 IAM 的完整说明，请访问 IAM 用户指南。

以下示例仅包含演示每种方法所需的代码。完整的示例代码在 GitHub 上提供。您可以从中下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行它们。

主题

管理 IAM 访问密钥

> **Important**
>
> 这是一个预览版，不建议用于生产环境。

创建访问密钥

要创建 IAM 访问密钥，请使用 CreateAccessKeyRequest 对象调用 IAMClient 的 `createAccessKey` 方法。

> **Note**
>
> 由于 IAM 是一项全局服务，因此，您必须将区域设置为 AWS_GLOBAL 才能使 IAMClient 调用生效。

导入

```
import software.amazon.awssdk.services.iam.model.CreateAccessKeyRequest;
import software.amazon.awssdk.services.iam.model.CreateAccessKeyResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

CreateAccessKeyRequest request = CreateAccessKeyRequest.builder()
    .userName(user).build();

CreateAccessKeyResponse response = iam.createAccessKey(request);
```

请参阅 GitHub 上的完整示例。

列出访问密钥

要列出指定用户的访问密钥，请创建一个 ListAccessKeysRequest 对象，其中包含要列出密钥的用户名，并将该对象传递给 IAMClient 的 `listAccessKeys` 方法。

> **Note**
>
> 如果您未向 `listAccessKeys` 提供用户名，则它将尝试列出与签署该请求的 AWS 账户相关联的访问密钥。

导入

```
import software.amazon.awssdk.services.iam.model.AccessKeyMetadata;
import software.amazon.awssdk.services.iam.model.ListAccessKeysRequest;
import software.amazon.awssdk.services.iam.model.ListAccessKeysResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

boolean done = false;
String new_marker = null;

while (!done) {
    ListAccessKeysResponse response;

 if(new_marker == null) {
   ListAccessKeysRequest request = ListAccessKeysRequest.builder()
                .userName(username).build();
         response = iam.listAccessKeys(request);
 }
 else {
  ListAccessKeysRequest request = ListAccessKeysRequest.builder()
             .userName(username)
             .marker(new_marker).build();
```

```
  response = iam.listAccessKeys(request);
 }

    for (AccessKeyMetadata metadata :
            response.accessKeyMetadata()) {
        System.out.format("Retrieved access key %s",
                metadata.accessKeyId());
    }

    if (!response.isTruncated()) {
        done = true;
    }
    else {
     new_marker = response.marker();
    }
}
```

`listAccessKeys` 的结果分页显示 (默认情况下，每个调用最多返回 100 个记录)。您可以调用返回的 ListAccessKeysResponse 对象中的 `isTruncated` 以查看该查询返回的结果是否少于可用结果。如果是，则调用 `ListAccessKeysResponse` 中的 `marker` 并在创建新请求时使用它。在下次调用 `listAccessKeys` 时使用该新请求。

请参阅 GitHub 上的完整示例。

检索上次使用访问密钥的时间

要获取上次使用访问密钥的时间，请使用访问密钥的 ID（可通过 GetAccessKeyLastUsedRequest 对象传入）调用 IAMClient 的 `getAccessKeyLastUsed` 方法。

随后，您可以使用返回的 GetAccessKeyLastUsedResponse 对象来检索密钥的上次使用时间。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
import software.amazon.awssdk.services.iam.model.GetAccessKeyLastUsedRequest;
import software.amazon.awssdk.services.iam.model.GetAccessKeyLastUsedResponse;
```

代码

```

Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

GetAccessKeyLastUsedRequest request = GetAccessKeyLastUsedRequest.builder()
    .accessKeyId(access_id).build();

GetAccessKeyLastUsedResponse response = iam.getAccessKeyLastUsed(request);

System.out.println("Access key was last used at: " +
        response.accessKeyLastUsed().lastUsedDate());
```

请参阅 GitHub 上的完整示例。

激活或停用访问密钥

您可以激活或停用访问密钥，方法是创建 UpdateAccessKeyRequest 对象，提供访问密钥 ID、用户名（可选）和所需状态，然后将请求对象传递给 IAMClient 的 `updateAccessKey` 方法。

导入

```
import software.amazon.awssdk.services.iam.model.StatusType;
import software.amazon.awssdk.services.iam.model.UpdateAccessKeyRequest;
import software.amazon.awssdk.services.iam.model.UpdateAccessKeyResponse;

import software.amazon.awssdk.regions.Region;
```

代码

```
String username = args[0];
String access_id = args[1];
String status = args[2];

StatusType statusType;

if (status.toLowerCase().equalsIgnoreCase("active")) {
 statusType = StatusType.ACTIVE;
}
else if (status.toLowerCase().equalsIgnoreCase("inactive")) {
 statusType = StatusType.INACTIVE;
}
else {
 statusType = StatusType.UNKNOWN_TO_SDK_VERSION;
}

Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

UpdateAccessKeyRequest request = UpdateAccessKeyRequest.builder()
    .accessKeyId(access_id)
    .userName(username)
    .status(statusType)
    .build();

UpdateAccessKeyResponse response = iam.updateAccessKey(request);
```

请参阅 GitHub 上的完整示例。

删除访问密钥

要永久删除访问密钥，请调用 IAMClient 的 `deleteKey` 方法，并为它提供 DeleteAccessKeyRequest，其中包含访问密钥的 ID 和用户名。

> **Note**
>
> 密钥在删除后无法再检索或使用。要临时停用密钥，使其可以稍后再次激活，请改为使用 updateAccessKey (p. 64) 方法。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
import software.amazon.awssdk.services.iam.model.DeleteAccessKeyRequest;
import software.amazon.awssdk.services.iam.model.DeleteAccessKeyResponse;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

DeleteAccessKeyRequest request = DeleteAccessKeyRequest.builder()
```

```
    .accessKeyId(access_key)
    .userName(username).build();

DeleteAccessKeyResponse response = iam.deleteAccessKey(request);
```

请参阅 GitHub 上的完整示例。

更多信息

- IAM API Reference 中的 CreateAccessKey
- IAM API Reference 中的 ListAccessKeys
- IAM API Reference 中的 GetAccessKeyLastUsed
- IAM API Reference 中的 UpdateAccessKey
- IAM API Reference 中的 DeleteAccessKey

管理 IAM 用户

> Important
>
> 这是一个预览版，不建议用于生产环境。

创建用户

使用包含用户名的 CreateUserRequest 对象向 IAMClient 的 `createUser` 方法提供用户名，从而创建新 IAM 用户。

导入

```
import software.amazon.awssdk.services.iam.model.CreateUserRequest;
import software.amazon.awssdk.services.iam.model.CreateUserResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

CreateUserRequest request = CreateUserRequest.builder()
    .userName(username).build();

CreateUserResponse response = iam.createUser(request);
```

请参阅 GitHub 上的完整示例。

列出用户

要列出您账户中的 IAM 用户，请创建新的 ListUsersRequest 并将其传递给 IAMClient 的 `listUsers` 方法。您可以通过在返回的 ListUsersResponse 对象上调用 `users` 来检索用户列表。

`listUsers` 返回的用户列表已分页。您可以通过调用响应对象的 `isTruncated` 方法查看更多可检索的结果。如果它返回 `true`，则调用响应对象的 `marker()` 方法。使用标记值创建新的请求对象。然后使用新请求再次调用 `listUsers` 方法。

导入

```
import software.amazon.awssdk.services.iam.model.ListUsersRequest;
import software.amazon.awssdk.services.iam.model.ListUsersResponse;
import software.amazon.awssdk.services.iam.model.User;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

boolean done = false;
String new_marker = null;

while(!done) {
    ListUsersResponse response;

    if (new_marker == null) {
        ListUsersRequest request = ListUsersRequest.builder().build();
        response = iam.listUsers(request);
    }
    else {
        ListUsersRequest request = ListUsersRequest.builder()
          .marker(new_marker).build();
        response = iam.listUsers(request);
    }

    for(User user : response.users()) {
        System.out.format("Retrieved user %s", user.userName());
    }

    if(!response.isTruncated()) {
        done = true;
    }
    else {
     new_marker = response.marker();
    }
}
```

请参阅 GitHub 上的完整示例。

更新用户

要更新用户，请调用 IAMClient 对象的 `updateUser` 方法，该方法采用 UpdateUserRequest 对象，您可以使用它更改用户的名称或路径。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
import software.amazon.awssdk.services.iam.model.UpdateUserRequest;
import software.amazon.awssdk.services.iam.model.UpdateUserResponse;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

UpdateUserRequest request = UpdateUserRequest.builder()
    .userName(cur_name)
```

```
    .newUserName(new_name).build();

UpdateUserResponse response = iam.updateUser(request);
```

请参阅 GitHub 上的完整示例。

删除用户

要删除用户，请使用 UpdateUserRequest 对象调用 IAMClient 的 `deleteUser` 请求，该对象中设置了要删除的用户名。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
import software.amazon.awssdk.services.iam.model.DeleteConflictException;
import software.amazon.awssdk.services.iam.model.DeleteUserRequest;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

DeleteUserRequest request = DeleteUserRequest.builder()
  .userName(username).build();

try {
    iam.deleteUser(request);
} catch (DeleteConflictException e) {
    System.out.println("Unable to delete user. Verify user is not" +
            " associated with any resources");
    throw e;
}
```

请参阅 GitHub 上的完整示例。

更多信息

- IAM User Guide 中的 IAM 用户
- IAM User Guide 中的管理 IAM 用户
- IAM API Reference 中的 CreateUser
- IAM API Reference 中的 ListUsers
- IAM API Reference 中的 UpdateUser
- IAM API Reference 中的 DeleteUser

使用 IAM 账户别名

> **Important**
>
> 这是一个预览版，不建议用于生产环境。

如果您希望在登录页面的 URL 用贵公司名称 (或其他友好标识) 取代您的 AWS 账户 ID，可以为您的 AWS 账户创建一个别名。

> **Note**
>
> AWS 的每个账户支持一个账户别名。

创建账户别名

要创建账户别名，请使用包含别名的 CreateAccountAliasRequest 对象调用 IAMClient 的 `createAccountAlias` 方法。

导入

```
import software.amazon.awssdk.services.iam.model.CreateAccountAliasRequest;
import software.amazon.awssdk.services.iam.model.CreateAccountAliasResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

CreateAccountAliasRequest request = CreateAccountAliasRequest.builder()
    .accountAlias(alias).build();

CreateAccountAliasResponse response = iam.createAccountAlias(request);
```

请参阅 GitHub 上的完整示例。

列出账户别名

要列出您的账户别名 (如果有)，请调用 IAMClient 的 `listAccountAliases` 方法。

> **Note**
>
> 返回的 ListAccountAliasesResponse 支持与其他 AWS SDK for Java列出方法相同的 `isTruncated` 和 `marker` 方法，但 AWS 账户只能有一个账户别名。

导入

```
import software.amazon.awssdk.services.iam.model.ListAccountAliasesResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

code

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

ListAccountAliasesResponse response = iam.listAccountAliases();

for (String alias : response.accountAliases()) {
    System.out.printf("Retrieved account alias %s", alias);
}
```

请参阅 GitHub 上的完整示例。

删除账户别名

要删除您的账户别名，请调用 IAMClient 的 `deleteAccountAlias` 方法。在删除账户别名时，您必须使用 DeleteAccountAliasRequest 对象提供其名称。

导入

```
import software.amazon.awssdk.services.iam.model.DeleteAccountAliasRequest;
import software.amazon.awssdk.services.iam.model.DeleteAccountAliasResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

DeleteAccountAliasRequest request = DeleteAccountAliasRequest.builder()
    .accountAlias(alias).build();

DeleteAccountAliasResponse response = iam.deleteAccountAlias(request);

System.out.println("Successfully deleted account alias " + alias);
```

请参阅 GitHub 上的完整示例。

更多信息

- IAM User Guide 中的 AWS 账户 ID 及其别名
- IAM API Reference 中的 CreateAccountAlias
- IAM API Reference 中的 ListAccountAliases
- IAM API Reference 中的 DeleteAccountAlias

使用 IAM 策略

> **Important**
>
> 这是一个预览版，不建议用于生产环境。

创建策略

要创建新策略，请在 CreatePolicyRequest 中向 IAMClient 的 `createPolicy` 方法提供策略名称和 JSON 格式的策略文档。

导入

```
import software.amazon.awssdk.services.iam.model.CreatePolicyRequest;
import software.amazon.awssdk.services.iam.model.CreatePolicyResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

CreatePolicyRequest request = CreatePolicyRequest.builder()
    .policyName(policy_name)
```

```
    .policyDocument(POLICY_DOCUMENT).build();

CreatePolicyResponse response = iam.createPolicy(request);

System.out.println("Successfully created policy: " +
        response.policy().policyName());
```

IAM 策略文档是使用明确语法的 JSON 字符串。下面的示例中提供了向 DynamoDB 发出特定请求的访问权。

```
public static final String POLICY_DOCUMENT =
    "{" +
    "  \"Version\": \"2012-10-17\"," +
    "  \"Statement\": [" +
    "    {" +
    "        \"Effect\": \"Allow\"," +
    "        \"Action\": \"logs:CreateLogGroup\"," +
    "        \"Resource\": \"%s\"" +
    "    }," +
    "    {" +
    "        \"Effect\": \"Allow\"," +
    "        \"Action\": [" +
    "            \"dynamodb:DeleteItem\"," +
    "            \"dynamodb:GetItem\"," +
    "            \"dynamodb:PutItem\"," +
    "            \"dynamodb:Scan\"," +
    "            \"dynamodb:UpdateItem\"" +
    "        ]," +
    "        \"Resource\": \"RESOURCE_ARN\"" +
    "    }" +
    "   ]" +
    "}";
```

请参阅 GitHub 上的完整示例。

获取策略

要检索现有策略，请调用 IAMClient 的 `getPolicy` 方法，并在 GetPolicyRequest 对象中提供策略的 ARN。

导入

```
import software.amazon.awssdk.services.iam.model.GetPolicyRequest;
import software.amazon.awssdk.services.iam.model.GetPolicyResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

GetPolicyRequest request = GetPolicyRequest.builder()
    .policyArn(policy_arn).build();

GetPolicyResponse response = iam.getPolicy(request);
```

请参阅 GitHub 上的完整示例。

附加角色策略

您可以将策略附加到 IAM 角色，方式是调用 IAMClient 的 `attachRolePolicy` 方法，并在 AttachRolePolicyRequest 中为其提供角色名称和策略 ARN。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
import software.amazon.awssdk.services.iam.model.AttachRolePolicyRequest;
import software.amazon.awssdk.services.iam.model.AttachedPolicy;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();
AttachRolePolicyRequest attach_request =
    AttachRolePolicyRequest.builder()
        .roleName(role_name)
        .policyArn(POLICY_ARN).build();

iam.attachRolePolicy(attach_request);
```

请参阅 GitHub 上的完整示例。

列出附加的角色策略

通过调用 IAMClient 的 `listAttachedRolePolicies` 方法列出角色中附加的策略。这需要 ListAttachedRolePoliciesRequest 对象，它包含要列出策略的角色名称。

在返回的 ListAttachedRolePoliciesResponse 对象上调用 `getAttachedPolicies` 来获取所附加策略的列表。结果可能被截断；如果 `ListAttachedRolePoliciesResponse` 对象的 `isTruncated` 方法返回了 `true`，请调用 `ListAttachedRolePoliciesResponse` 对象的 `marker` 方法。使用返回的标记创建新请求并使用该请求再次调用 `listAttachedRolePolicies` 以获取下一批结果。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
import software.amazon.awssdk.services.iam.model.ListAttachedRolePoliciesRequest;
import software.amazon.awssdk.services.iam.model.ListAttachedRolePoliciesResponse;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

List<AttachedPolicy> matching_policies = new ArrayList<>();

boolean done = false;
String new_marker = null;

while(!done) {

 ListAttachedRolePoliciesResponse response;

 if (new_marker == null) {
  ListAttachedRolePoliciesRequest request =
                ListAttachedRolePoliciesRequest.builder()
                    .roleName(role_name).build();
```

```
  response = iam.listAttachedRolePolicies(request);
 }
 else {
  ListAttachedRolePoliciesRequest request =
                ListAttachedRolePoliciesRequest.builder()
                    .roleName(role_name)
                    .marker(new_marker).build();
  response = iam.listAttachedRolePolicies(request);
 }

    matching_policies.addAll(
            response.attachedPolicies()
                    .stream()
                    .filter(p -> p.policyName().equals(role_name))
                    .collect(Collectors.toList()));

    if(!response.isTruncated()) {
        done = true;
    }
    else {
     new_marker = response.marker();
    }
}

if (matching_policies.size() > 0) {
    System.out.println(role_name +
            " policy is already attached to this role.");
    return;
}
```

请参阅 GitHub 上的完整示例。

分离角色策略

要从角色分离策略，请调用 IAMClient 的 `detachRolePolicy` 方法，并在 DetachRolePolicyRequest 中为其提供角色名称和策略 ARN。

导入

```
import software.amazon.awssdk.services.iam.model.DetachRolePolicyRequest;
import software.amazon.awssdk.services.iam.model.DetachRolePolicyResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

DetachRolePolicyRequest request = DetachRolePolicyRequest.builder()
    .roleName(role_name)
    .policyArn(policy_arn).build();

DetachRolePolicyResponse response = iam.detachRolePolicy(request);
```

请参阅 GitHub 上的完整示例。

更多信息

- IAM User Guide 中的 IAM 策略概述。

- IAM User Guide 中的 AWS IAM 策略参考。
- IAM API Reference 中的 CreatePolicy
- IAM API Reference 中的 GetPolicy
- IAM API Reference 中的 AttachRolePolicy
- IAM API Reference 中的 ListAttachedRolePolicies
- IAM API Reference 中的 DetachRolePolicy

使用 IAM 服务器证书

Important

这是一个预览版，不建议用于生产环境。

要在 AWS 中启用与您的网站或应用程序的 HTTPS 连接，您需要 SSL/TLS 服务器证书。您可以使用 AWS Certificate Manager 提供的服务器证书或您从外部提供程序获得的服务器证书。

我们建议您使用 ACM 来预置、管理和部署您的服务器证书。利用 ACM，您可以申请证书，将其部署到 AWS 资源，然后让 ACM 为您处理证书续订事宜。ACM 提供的证书是免费的。有关 ACM 的更多信息，请参阅 ACM 用户指南。

获取服务器证书

您可以通过调用 IAMClient 的 `getServerCertificate` 方法检索服务器证书，将包含证书名称的 GetServerCertificateRequest 传递给它。

导入

```
import software.amazon.awssdk.services.iam.model.GetServerCertificateRequest;
import software.amazon.awssdk.services.iam.model.GetServerCertificateResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

GetServerCertificateRequest request = GetServerCertificateRequest.builder()
            .serverCertificateName(cert_name).build();

GetServerCertificateResponse response = iam.getServerCertificate(request);
```

请参阅 GitHub 上的完整示例。

列出服务器证书

要列出您的服务器证书，请使用 ListServerCertificatesRequest 调用 IAMClient 的 `listServerCertificates` 方法。它返回 ListServerCertificatesResponse。

调用返回的 `ListServerCertificateResponse` 对象的 `serverCertificateMetadataList` 方法获取 ServerCertificateMetadata 对象的列表，您可以用它来获取关于每个证书的信息。

如果 `ListServerCertificateResponse` 对象的 `isTruncated` 方法返回了 `true`，调用 `ListServerCertificatesResponse` 对象的 `marker` 方法并使用标记创建一个新请求，则结果可能被截断。使用该新请求重新调用 `listServerCertificates` 以获取下一批结果。

导入

```
import software.amazon.awssdk.services.iam.model.ListServerCertificatesRequest;
import software.amazon.awssdk.services.iam.model.ListServerCertificatesResponse;
import software.amazon.awssdk.services.iam.model.ServerCertificateMetadata;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

boolean done = false;
String new_marker = null;

while(!done) {
 ListServerCertificatesResponse response;

 if (new_marker == null) {
        ListServerCertificatesRequest request =
          ListServerCertificatesRequest.builder().build();
        response = iam.listServerCertificates(request);
 }
 else {
        ListServerCertificatesRequest request =
          ListServerCertificatesRequest.builder()
          .marker(new_marker).build();
        response = iam.listServerCertificates(request);
 }

    for(ServerCertificateMetadata metadata :
            response.serverCertificateMetadataList()) {
        System.out.printf("Retrieved server certificate %s",
                metadata.serverCertificateName());
    }

    if(!response.isTruncated()) {
        done = true;
    }
    else {
     new_marker = response.marker();
    }
}
```

请参阅 GitHub 上的完整示例。

更新服务器证书

您可以通过调用 IAMClient 的 `updateServerCertificate` 方法更新服务器证书的名称或路径。这需要通过服务器证书的当前名称以及要使用的新名称或新路径来设置 UpdateServerCertificateRequest 对象。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
import software.amazon.awssdk.services.iam.model.UpdateServerCertificateRequest;
import software.amazon.awssdk.services.iam.model.UpdateServerCertificateResponse;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

UpdateServerCertificateRequest request =
    UpdateServerCertificateRequest.builder()
        .serverCertificateName(cur_name)
        .newServerCertificateName(new_name)
        .build();

UpdateServerCertificateResponse response =
    iam.updateServerCertificate(request);
```

请参阅 GitHub 上的完整示例。

删除服务器证书

要删除服务器证书，请使用包含证书名称的 DeleteServerCertificateRequest 调用 IAMClient 的 `deleteServerCertificate` 方法。

导入

```
import software.amazon.awssdk.services.iam.model.DeleteServerCertificateRequest;
import software.amazon.awssdk.services.iam.model.DeleteServerCertificateResponse;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.iam.IAMClient;
```

代码

```
Region region = Region.AWS_GLOBAL;
IAMClient iam = IAMClient.builder().region(region).build();

DeleteServerCertificateRequest request =
    DeleteServerCertificateRequest.builder()
        .serverCertificateName(cert_name).build();

DeleteServerCertificateResponse response =
    iam.deleteServerCertificate(request);
```

请参阅 GitHub 上的完整示例。

更多信息

- 在 IAM User Guide 中使用服务器证书
- IAM API Reference 中的 GetServerCertificate
- IAM API Reference 中的 ListServerCertificates
- IAM API Reference 中的 UpdateServerCertificate
- IAM API Reference 中的 DeleteServerCertificate
- ACM 用户指南

使用 AWS SDK for Java 的 Kinesis 示例

此部分提供了使用适用于 Java 的 AWS 开发工具包 2.0 对 Amazon Kinesis 进行编程的示例。

有关 Kinesis 的更多信息，请参阅 Amazon Kinesis 开发人员指南。

以下示例仅包含演示每种方法所需的代码。完整的示例代码在 GitHub 上提供。您可以从中下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行它们。

主题

订阅 Amazon Kinesis 数据流

以下示例向您演示如何使用 `subscribeToShard` 方法检索和处理 Amazon Kinesis Data Streams 中的数据。Kinesis Data Streams 现在采用增强的扇出功能和低延迟 HTTP/2 数据检索 API，同时让开发人员能够更易于在相同的 Kinesis 数据流上运行多个低延迟、高性能的应用程序。

设置

首先，创建一个异步 Kinesis 客户端和一个 SubscribeToShardRequest 对象。这些对象用于以下每个示例中以订阅 Kinesis 事件。

导入

```
import java.net.URI;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.function.Supplier;

import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;
import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.core.async.SdkPublisher;
import software.amazon.awssdk.http.Protocol;
import software.amazon.awssdk.http.SdkHttpConfigurationOption;
import software.amazon.awssdk.http.nio.netty.NettyNioAsyncHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.ShardIteratorType;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardEvent;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardEventStream;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardRequest;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardResponse;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardResponseHandler;
import software.amazon.awssdk.utils.AttributeMap;
```

代码

```

KinesisAsyncClient client = KinesisAsyncClient.create();

SubscribeToShardRequest request = SubscribeToShardRequest.builder()
        .consumerARN(CONSUMER_ARN)
        .shardId("shardId-000000000000")
        .startingPosition(s -> s.type(ShardIteratorType.LATEST)).build();
```

使用 Builder 接口

您可以使用 `builder` 方法来简化创建 SubscribeToShardResponseHandler 的过程。

使用生成器，您可以通过方法调用设置每个生命周期回调，而非实施完整的接口。

代码

```
private static CompletableFuture<Void> responseHandlerBuilder(KinesisAsyncClient client,
 SubscribeToShardRequest request) {
    SubscribeToShardResponseHandler responseHandler = SubscribeToShardResponseHandler
        .builder()
        .onError(t -> System.err.println("Error during stream - " + t.getMessage()))
        .onComplete(() -> System.out.println("All records stream successfully"))
        // Must supply some type of subscriber
        .subscriber(e -> System.out.println("Received event - " + e))
        .build();
    return client.subscribeToShard(request, responseHandler);
}
```

要对发布者进行更多控制，您可以使用 `publisherTransformer` 方法自定义发布者。

代码

```
private static CompletableFuture<Void>
 responseHandlerBuilder_PublisherTransformer(KinesisAsyncClient client,
 SubscribeToShardRequest request) {
    SubscribeToShardResponseHandler responseHandler = SubscribeToShardResponseHandler
        .builder()
        .onError(t -> System.err.println("Error during stream - " + t.getMessage()))
        .publisherTransformer(p -> p.filter(e -> e instanceof
 SubscribeToShardEvent).limit(100))
        .subscriber(e -> System.out.println("Received event - " + e))
        .build();
    return client.subscribeToShard(request, responseHandler);
}
```

请参阅 GitHub 上的完整示例。

使用自定义响应处理程序

要完全控制订阅者和发布者，请实施 SubscribeToShardResponseHandler 接口。

在本示例中，您实施 `onEventStream` 方法，此方法允许您对发布者具有完全访问权限。此示例演示如何将发布者转换为事件记录以供订阅者打印。

代码

```
private static CompletableFuture<Void> responseHandlerBuilder_Classic(KinesisAsyncClient
 client, SubscribeToShardRequest request) {
    SubscribeToShardResponseHandler responseHandler = new SubscribeToShardResponseHandler()
 {

        @Override
        public void responseReceived(SubscribeToShardResponse response) {
            System.out.println("Receieved initial response");
        }

        @Override
        public void onEventStream(SdkPublisher<SubscribeToShardEventStream> publisher) {
            publisher
                // Filter to only SubscribeToShardEvents
                .filter(SubscribeToShardEvent.class)
                // Flat map into a publisher of just records
                .flatMapIterable(SubscribeToShardEvent::records)
                // Limit to 1000 total records
                .limit(1000)
                // Batch records into lists of 25
                .buffer(25)
```

```
                // Print out each record batch
                .subscribe(batch -> System.out.println("Record Batch - " + batch));
        }

        @Override
        public void complete() {
            System.out.println("All records stream successfully");
        }

        @Override
        public void exceptionOccurred(Throwable throwable) {
            System.err.println("Error during stream - " + throwable.getMessage());
        }

    };
    return client.subscribeToShard(request, responseHandler);
}
```

请参阅 GitHub 上的完整示例。

使用 Visitor 接口

您可以使用 Visitor 对象订阅您有兴趣观看的特定事件。

代码

```
private static CompletableFuture<Void>
 responseHandlerBuilder_VisitorBuilder(KinesisAsyncClient client, SubscribeToShardRequest
 request) {
    SubscribeToShardResponseHandler.Visitor visitor =
 SubscribeToShardResponseHandler.Visitor
        .builder()
        .onSubscribeToShardEvent(e -> System.out.println("Received subscribe to shard event
 " + e))
        .build();
    SubscribeToShardResponseHandler responseHandler = SubscribeToShardResponseHandler
        .builder()
        .onError(t -> System.err.println("Error during stream - " + t.getMessage()))
        .subscriber(visitor)
        .build();
    return client.subscribeToShard(request, responseHandler);
}
```

请参阅 GitHub 上的完整示例。

使用自定义订阅者

您也可以实施您自己的自定义订阅者以订阅流。

此代码段显示了一个示例订阅者。

代码

```
private static class MySubscriber implements Subscriber<SubscribeToShardEventStream> {

    private Subscription subscription;
    private AtomicInteger eventCount = new AtomicInteger(0);

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
```

```
        this.subscription.request(1);
    }

    @Override
    public void onNext(SubscribeToShardEventStream shardSubscriptionEventStream) {
        System.out.println("Received event " + shardSubscriptionEventStream);
        if (eventCount.incrementAndGet() >= 100) {
            // You can cancel the subscription at any time if you wish to stop receiving
 events.
            subscription.cancel();
        }
        subscription.request(1);
    }

    @Override
    public void onError(Throwable throwable) {
        System.err.println("Error occurred while stream - " + throwable.getMessage());
    }

    @Override
    public void onComplete() {
        System.out.println("Finished streaming all events");
    }
}
```

您可以将自定义订阅者传递到 `subscribe` 方法，这与预览示例类似。下面的代码段显示了此示例。

代码

```
private static CompletableFuture<Void> responseHandlerBuilder_Subscriber(KinesisAsyncClient
 client, SubscribeToShardRequest request) {
    SubscribeToShardResponseHandler responseHandler = SubscribeToShardResponseHandler
        .builder()
        .onError(t -> System.err.println("Error during stream - " + t.getMessage()))
        .subscriber(MySubscriber::new)
        .build();
    return client.subscribeToShard(request, responseHandler);
}
```

请参阅 GitHub 上的完整示例。

使用第三方库

您可以使用其他第三方库，而不是实现自定义订阅者。本示例演示了如何使用 RxJava 实现，但您可以使用实现反应式流接口的任何库。有关该库的更多信息，请参阅 Github 上的 RxJava 维基页面。

要使用该库，请将其作为依赖项添加。如果您使用 Maven，示例将显示要使用的 POM 代码段。

POM 条目

```
    <dependency>
      <groupId>io.reactivex.rxjava2</groupId>
      <artifactId>rxjava</artifactId>
      <version>2.1.14</version>
    </dependency>
```

导入

```
import java.net.URI;
import java.util.concurrent.CompletableFuture;
```

```
import io.reactivex.Flowable;
import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.core.async.SdkPublisher;
import software.amazon.awssdk.http.Protocol;
import software.amazon.awssdk.http.SdkHttpConfigurationOption;
import software.amazon.awssdk.http.nio.netty.NettyNioAsyncHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.ShardIteratorType;
import software.amazon.awssdk.services.kinesis.model.StartingPosition;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardEvent;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardRequest;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardResponseHandler;
import software.amazon.awssdk.utils.AttributeMap;
```

此示例在 `onEventStream` 生命周期方法中使用 RxJava。这样，您就对发布者具有完全访问权限，这可用于创建 Rx Flowable。

代码

```
SubscribeToShardResponseHandler responseHandler = SubscribeToShardResponseHandler
    .builder()
    .onError(t -> System.err.println("Error during stream - " + t.getMessage()))
    .onEventStream(p -> Flowable.fromPublisher(p)
                                .ofType(SubscribeToShardEvent.class)
                                .flatMapIterable(SubscribeToShardEvent::records)
                                .limit(1000)
                                .buffer(25)
                                .subscribe(e -> System.out.println("Record batch = " + e)))
    .build();
```

您也可以使用带有 `Flowable` 发布者的 `publisherTransformer` 方法。您必须使 `Flowable` 发布者适应 SdkPublisher，如以下示例所示。

代码

```
SubscribeToShardResponseHandler responseHandler = SubscribeToShardResponseHandler
    .builder()
    .onError(t -> System.err.println("Error during stream - " + t.getMessage()))
    .publisherTransformer(p -> SdkPublisher.adapt(Flowable.fromPublisher(p).limit(100)))
    .build();
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon Kinesis API Reference 中的 SubscribeToShardEvent
- Amazon Kinesis API Reference 中的 SubscribeToShard

检索分页结果

当响应对象太大而无法在单个响应中返回时，很多 AWS 操作都会返回分页的结果。在适用于 Java 的 AWS 开发工具包 1.0 中，响应包含了您必须使用以检索下一页结果的令牌。适用于 Java 的 AWS 开发工具包 2.0 中新增了自动分页方法，该方法可进行多个服务调用以自动为您获取下一页结果。您只需编写处理结果的代码。此外，这两种类型的方法还具有同步和异步版本。有关异步客户端的更多详细信息，请参阅异步编程 (p. 16)。

以下示例使用 Amazon S3 和 Amazon DynamoDB 操作来演示从分页响应中检索数据的各种方法。

Note

这些代码段假定您了解使用适用于 Java 的 AWS 开发工具包 2.0 开发人员预览 (p. 11)中的内容，并且已使用设置用于开发的 AWS 凭证和区域 (p. 5)中的信息配置默认 AWS 凭证。

同步分页

这些示例使用同步分页方法来列出 Amazon S3 存储桶中的对象。

迭代页面

构建一个 ListObjectsV2Request 并提供一个存储桶名称。您可以选择性地提供要一次性检索的最大密钥数。将其传递到 S3Client 的 `listObjectsV2Paginator` 方法。此方法将返回 ListObjectsV2Iterable 对象，该对象是 ListObjectsV2Response 类的 `Iterable`。

第一个示例演示如何使用分页工具对象，借助 `stream` 方法迭代所有响应页面。您可以直接流式通过响应页面，将响应流转换为 S3Object 内容的流，然后处理 Amazon S3 对象的内容。

导入

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.ListObjectsV2Request;
import software.amazon.awssdk.services.s3.paginators.ListObjectsV2Iterable;
```

代码

```
// Build the list objects request
ListObjectsV2Request listReq = ListObjectsV2Request.builder()
        .bucket(bucket)
        .maxKeys(1)
        .build();

ListObjectsV2Iterable listRes = s3.listObjectsV2Paginator(listReq);
// Process response pages
listRes.stream()
         .flatMap(r -> r.contents().stream())
         .forEach(content -> System.out.println(" Key: " + content.key() + " size = " +
 content.size()));
```

请参阅 GitHub 上的完整示例。

迭代对象

以下示例演示了迭代响应中返回的对象（而不是响应的页面）的方法。

使用流

在响应内容上使用 `stream` 方法来迭代分页项目集合。

代码

```
// Helper method to work with paginated collection of items directly
listRes.contents().stream()
         .forEach(content -> System.out.println(" Key: " + content.key() + " size = " +
 content.size()));
```

请参阅 GitHub 上的完整示例。

使用 For 循环

使用标准 `for` 循环迭代响应的内容。

代码

```
// Use simple for loop if stream is not necessary
for (S3Object content : listRes.contents()) {
    System.out.println(" Key: " + content.key() + " size = " + content.size());
}
```

请参阅 GitHub 上的完整示例。

手动分页

如果您的使用案例需要手动分页，则手动分页仍然可用。对后续请求使用响应对象中的下一个令牌。下面是使用 `while` 循环的示例。

代码

```
// Use manual pagination
ListObjectsV2Request listObjectsReqManual = ListObjectsV2Request.builder()
        .bucket(bucket)
        .maxKeys(1)
        .build();

boolean done = false;
while (!done) {
    ListObjectsV2Response listObjResponse = s3.listObjectsV2(listObjectsReqManual);
    for (S3Object content : listObjResponse.contents()) {
        System.out.println(content.key());
    }

    if (listObjResponse.nextContinuationToken() == null) {
        done = true;
    }

    listObjectsReqManual = listObjectsReqManual.toBuilder()
            .continuationToken(listObjResponse.nextContinuationToken())
            .build();
}
```

请参阅 GitHub 上的完整示例。

异步分页

这些示例使用异步分页方法来列出 DynamoDB 中的表。异步编程 (p. 16)主题中提供了手动分页示例。

迭代表名称页面

首先，创建一个异步 DynamoDB 客户端。然后，调用 `listTablesPaginator` 方法来获取 ListTablesPublisher。这是反应式流 Publisher 接口的实现。要了解有关反应式流模型的更多信息，请参阅反应式流 Github 存储库。

在 ListTablesPublisher 上调用 `subscribe` 方法并传递订阅者实现。在本示例中，订阅者有一个 `onNext` 方法，该方法一次从发布者请求一个项目。这是在检索到所有页面之前重复调用的方法。`onSubscribe` 方法

将调用 Subscription.request 方法来对来自发布者的数据启动请求。必须调用此方法以开始从发布者获取数据。如果检索数据时出现错误，将触发 onError 方法。最后，在 onComplete 方法在请求所有页面后调用。

使用订阅者

导入

```
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;

import software.amazon.awssdk.core.async.SdkPublisher;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.ListTablesRequest;
import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;
import software.amazon.awssdk.services.dynamodb.paginators.ListTablesPublisher;
```

代码

```
final DynamoDbAsyncClient asyncClient = DynamoDbAsyncClient.create();

ListTablesRequest listTablesRequest = ListTablesRequest.builder().limit(3).build();
ListTablesPublisher publisher = asyncClient.listTablesPaginator(listTablesRequest);

// A Subscription represents a one-to-one life-cycle of a Subscriber subscribing to a
 Publisher.
publisher.subscribe(new Subscriber<ListTablesResponse>() {
    // Maintain a reference to the subscription object, which is required to request data
 from the publisher
    private Subscription subscription;

    @Override
    public void onSubscribe(Subscription s) {
        subscription = s;
        // Request method should be called to demand data. Here we request a single page
        subscription.request(1);
    }

    @Override
    public void onNext(ListTablesResponse response) {
        response.tableNames().forEach(System.out::println);
        // Once you process the current page, call the request method to signal that you
 are ready for next page
        subscription.request(1);
    }

    @Override
    public void onError(Throwable t) {
        // Called when an error has occurred while processing the requests
    }

    @Override
    public void onComplete() {
        // This indicates all the results are delivered and there are no more pages left
    }
});
```

请参阅 GitHub 上的完整示例。

使用 For 循环

使用 `for` 循环，针对创建新订阅者可能开销过高的简单使用案例迭代页面。响应发布者对象有一个 `forEach` 帮助程序方法来达到此目的。

代码

```
// Use a for-loop for simple use cases
CompletableFuture<Void> future = publisher.forEach(response -> response.tableNames()

 .forEach(System.out::println));

future.get();
```

请参阅 GitHub 上的完整示例。

迭代表名称

以下示例演示了迭代响应中返回的对象（而不是响应的页面）的方法。与同步结果类似，异步结果类具有一个与基础项目集合交互的方法。此便捷方法的返回类型是一个可用于跨所有页面请求项目的发布者。

使用订阅者

代码

```
// Creates a default client with credentials and regions loaded from the environment
final DynamoDbAsyncClient asyncClient = DynamoDbAsyncClient.create();

ListTablesRequest listTablesRequest = ListTablesRequest.builder().limit(3).build();
ListTablesPublisher listTablesPublisher =
 asyncClient.listTablesPaginator(listTablesRequest);
SdkPublisher<String> publisher = listTablesPublisher.tableNames();
// Use subscriber
publisher.subscribe(new Subscriber<String>() {
    private Subscription subscription;

    @Override
    public void onSubscribe(Subscription s) {
        subscription = s;
        subscription.request(1);
    }

    @Override
    public void onNext(String tableName) {
        System.out.println(tableName);
        subscription.request(1);
    }

    @Override
    public void onError(Throwable t) { }

    @Override
    public void onComplete() { }
});
```

请参阅 GitHub 上的完整示例。

使用 For 循环

使用 `forEach` 便捷方法迭代结果。

代码

```
// Use forEach
CompletableFuture<Void> future = publisher.forEach(System.out::println);
future.get();
```

请参阅 GitHub 上的完整示例。

使用第三方库

您可以使用其他第三方库，而不是实现自定义订阅者。本示例演示了如何使用 RxJava 实现，但实现反应式流接口的任何库都可以使用。有关该库的更多信息，请参阅 Github 上的 RxJava 维基页面。

要使用该库，请将其作为依赖项添加。如果使用了 Maven，示例将显示要使用的 POM 代码段。

POM 条目

```
    <dependency>
       <groupId>io.reactivex.rxjava2</groupId>
       <artifactId>rxjava</artifactId>
       <version>2.1.9</version>
   </dependency>
```

导入

```
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.ListTablesRequest;
import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;
import software.amazon.awssdk.services.dynamodb.paginators.ListTablesPublisher;
import io.reactivex.Flowable;
```

代码

```
DynamoDbAsyncClient asyncClient = DynamoDbAsyncClient.create();
ListTablesPublisher publisher = asyncClient.listTablesPaginator(ListTablesRequest.builder()
                                                                .build());

// The Flowable class has many helper methods that work with any reactive streams
 compatible publisher implementation
List<String> tables = Flowable.fromPublisher(publisher)
                              .flatMapIterable(ListTablesResponse::tableNames)
                              .toList()
                              .blockingGet();
System.out.println(tables);
```

文档历史记录

本主题介绍 AWS SDK for Java Developer Guide 在其发展历程中的重要更改。

文档创建时间：2018 年 8 月 2 日

2018 年 4 月 5 日

增加了自动分页主题。

2017 年 12 月 29 日

增加了 IAM、Amazon EC2、Cloudwatch 和 DynamoDB 的示例主题

2017 年 8 月 7 日

增加了 S3 的 getobject 示例。

2017 年 8 月 4 日

增加了异步主题。

2017 年 6 月 28 日

发布了新的开发工具包版本 2.0.