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、Amazon SimpleDB 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。有关每个版本的开发工具包附带的受支持服务及其 API 版本的列表，请查看要使用的版本的发行说明。

适用于 Java 的 AWS 开发工具包 2.x

在 https://github.com/aws/aws-sdk-java-v2/ 中了解新的适用于 Java 的 AWS 开发工具包 2.x。它包括许多期待已久的功能，例如插入 HTTP 实施的方法。要了解其用法，请参阅适用于 Java 的 AWS 开发工具包 2.x 开发人员指南。

其他文档和资源

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 API 参考
- Java 开发人员博客
- Java 开发人员论坛
- GitHub:
  - 文档源
  - 文档问题
  - 开发工具包源
  - 开发工具包问题
  - 开发工具包示例
  - Gitter 通道
- AWS 代码示例目录
- @awsforjava (Twitter)
- 发布说明

Eclipse IDE 支持

如果您使用 Eclipse IDE 开发代码，则可使用 AWS Toolkit for Eclipse 将AWS SDK for Java添加到现有 Eclipse 项目或创建新的AWS SDK for Java项目。此工具包还支持创建和上传 Lambda 函数、启动和监控 Amazon EC2 实例、管理 IAM 用户和安全组、AWS CloudFormation 模板编辑器等。

请参阅 AWS Toolkit for Eclipse 用户指南了解完整的文档。

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 Mobile 开发工具包开发人员指南了解完整的文档。

查看开发工具包的修订历史记录

要查看AWS SDK for Java的版本历史记录，包括针对每个开发工具包版本的更改和支持的服务，请参阅开发工具包的发布说明。

构建早期版本开发工具包的 Java 参考文档

适用于 Java 的 AWS 开发工具包 API 参考代表最新版本的开发工具包。如果您使用早期版本的开发工具包，您可能希望访问匹配您使用的版本的开发工具包参考文档。

构建文档的最轻松方式是使用 Apache 的 Maven 构建工具。先下载并安装 Maven（如果您的系统上尚未安装它），然后按照以下说明进行操作来构建参考文档。

构建早期版本开发工具包的参考文档

1. 在 GitHub 上的开发工具包存储库的版本页面上，找到并选择您将使用的开发工具包版本。
2. 选择 `zip`（对于大多数平台，包括 Windows）或 `tar.gz`（对于 Linux, OS X, or Unix）链接以将开发工具包下载到您的计算机上。
3. 将存档提取到本地目录。
4. 在命令行上，导航到将存档提取到的目录，然后键入以下内容。

```
mvn javadoc:javadoc
```

5. 在构建完成后，您将在 `aws-java-sdk/target/site/apidocs/` 目录中找到生成的 HTML 文档。

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 (而不使用根账户凭证) 来提供访问凭证。

> **Note**
>
> 有关iam用户的概述，以及它们对您帐户安全性的重要性，请参阅 身份管理概述: 用户 在 IAM User Guide.

注册 AWS

1. 打开 https://aws.amazon.com/ 并单击注册。
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. 单击侧边栏中的用户以查看您的 IAM 用户。
3. 如果您未设置任何 IAM 用户，则单击 Create New Users (创建新用户) 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. 打开安全凭证选项卡，然后单击创建访问密钥。

   > **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. 在出现的对话框中，单击下载凭证按钮以将凭证文件下载到您的计算机上，或者单击显示用户安全凭证以查看 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，应当在 AWS 共享凭证文件或环境中设置凭证 (p. 6)。

Note

如果您使用 Eclipse IDE，则应当考虑安装 AWS Toolkit for Eclipse 并提供凭证，如 AWS Toolkit for Eclipse User Guide 中的设置 AWS 凭证所述。

设置 AWS SDK for Java

说明如何在您的项目中使用 AWS SDK for Java。

Prerequisites

要使用 AWS SDK for Java，必须拥有：

- 适合的 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)。
- 在您的环境中设置的 AWS 凭证 (访问密钥) 或使用共享 (AWS CLI 和其他开发工具包) 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. 6)。

在您的项目中包括开发工具包

要在项目中包括开发工具包，根据您的编译系统或 IDE 使用以下方法之一：

- Apache Maven – 如果使用 Apache Maven，可以将整个开发工具包（或开发工具包的特定组件）指定为项目的依赖项。有关如何在使用 Maven 时设置开发工具包的详细信息，请参阅将开发工具包与 Apache Maven 一起使用 (p. 9)。
- 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。有关更多信息，请参阅将开发工具包与 Gradle 一起使用 (p. 11)。
- Eclipse IDE – 如果使用 Eclipse IDE，您可能希望安装和使用 AWS Toolkit for Eclipse，它会自动为您下载、安装和更新 Java 开发工具包。有关更多信息和设置说明，请参阅 AWS Toolkit for Eclipse 用户指南。

如果您使用上述方法之一（例如，您使用的是 Maven），则无需下载并安装 AWS JAR 文件（您可以跳过以下部分）。如果要使用不同 IDE、使用 Apache Ant 或任何其他方法构建项目，请按照下一节中的说明下载并提取开发工具包。

下载和解压缩开发工具包

我们建议您为新项目使用预建的最新开发工具包版本，从而针对所有 AWS 服务为您提供最新支持。

Note

有关如何下载和构建开发工具包旧版本的信息，请参阅安装开发工具包的旧版本 (p. 5)。

下载和提取开发工具包的最新版本

1. 从 https://sdk-for-java.amazonwebservices.com/latest/aws-java-sdk.zip 下载开发工具包。
2. 下载开发工具包之后，将内容提取到本地目录中。

开发工具包包含以下目录：

- `documentation` – 包含 API 文档（同时在 Web 上提供：适用于 Java 的 AWS 开发工具包 API 参考）。
- `lib`– 包含开发工具包 `.jar` 文件。
- `samples`– 包含说明如何使用开发工具包的实用示例代码。
- `third-party/lib`– 包含开发工具包使用的第三方库，例如 Apache Commons 日志记录、AspectJ 和 Spring 框架。

要使用开发工具包，将完整路径添加到 `lib`，并将 `third-party` 目录添加到编译文件中的依赖项，然后将它们添加到 java `CLASSPATH` 以运行代码。

安装开发工具包的旧版本

预建表单中仅提供开发工具包的最新版本。不过，可使用 Apache Maven (开源) 构建开发工具包的旧版本。Maven 将一步完成下载所有必需的依赖项、构建和安装开发工具包。有关安装说明和更多信息，请访问 http://maven.apache.org/。

要安装开发工具包的旧版本

1. 转到SDKGithub页面: 适用于Java的AWSSDK(Github).
2. 选择与所需开发工具包的版本号对应的标签。例如：`1.6.10`。
3. 单击 Download Zip 按钮下载选择的开发工具包版本。
4. 将文件解压缩到开发系统中的一个目录中。在很多系统中，可使用自己的图形文件管理器执行该操作，或在终端窗口中使用 `unzip` 实用程序。
5. 在终端窗口中，导航到将开发工具包源文件解压缩的目录。
6. 使用以下命令构建并安装开发工具包 (Maven 需要)：

   ```
   mvn clean install
   ```

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

   ```
   mvn javadoc:javadoc
   ```

   该文档构建到 `target/site/apidocs/` 目录中。

安装 Java 开发环境

AWS SDK for Java要求使用 J2SE Development Kit 6.0 或更高版本。可以从 http://www.oracle.com/technetwork/java/javase/downloads/ 下载最新的 Java 软件。

> **Important**
>
> Java 版本 1.6 (JS2E 6.0) 中没有 SHA256 签名的 SSL 证书的内置支持，而在 2015 年 9 月 30 日以后，与 AWS 的所有 HTTPS 连接都需要该功能。
> Java 版本 1.7 或更高版本包含已更新证书，不受这一问题的影响。

选择 JVM

为了让使用适用于 Java 的 AWS 开发工具包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。

在运行时将 JVM 的 32 位版本与 `-Server` 选项一起使用应可以提供与 64 位 JVM 相当的性能。

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 在许多不同的位置 (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。如果您需要设置用于 EC2 实例的凭证或如果您使用 Eclipse IDE 进行开发，请改为参考以下主题：

- 在使用 EC2 实例时，创建一个 IAM 角色，然后向该角色授予对 EC2 实例的访问权，如使用 IAM 角色授予对 Amazon EC2 上的 AWS 资源的访问权 (p. 60)中所述。
- 使用 AWS Toolkit for Eclipse 设置 Eclipse 中的 AWS 凭证。有关更多信息，请参阅 AWS Toolkit for Eclipse 用户指南中的设置 AWS 凭证。

设置 AWS 凭证

虽然可通过大量方式设置将由 AWS SDK for Java 使用的凭证，但建议使用以下方式：

- 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：
  - `~/.aws/credentials` 上的 Linux, OS X, or Unix
  - Windows 上的 `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。

在使用这些方法之一来设置 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 凭证的其他信息，请参阅使用 AWS 凭证 (p. 22)。

刷新 IMDS 凭证

AWS SDK for Java 支持选择每 1 分钟在后台刷新 IMDS 凭证一次，无论凭证到期时间如何。这可让您更频繁地刷新凭证，并减小未达到 IMDS 影响感知的 AWS 可用性的几率。

```
1. // Refresh credentials using a background thread, automatically every minute. This will
log an error if IMDS is down during
2. // a refresh, but your service calls will continue using the cached credentials until
the credentials are refreshed
3. // again one minute later.
4.
5. InstanceProfileCredentialsProvider credentials =
6.     InstanceProfileCredentialsProvider.createAsyncRefreshingProvider(true);
7.
8. AmazonS3Client.builder()
9.               .withCredentials(credentials)
10.               .build();
11.
12. // This is new: When you are done with the credentials provider, you must close it to
release the background thread.
13. credentials.close();
```

设置 AWS 区域

您应使用适用于 Java 的 AWS 开发工具包设置将用于访问 AWS 服务会的默认 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。要查看每个服务的区域列表，请参阅 Amazon Web Services General Reference 中的区域和终端节点。

> **Note**
>
> 如果您未 选择区域，则默认情况下将使用us-east-1。

您可以使用类似的方法设置凭证以设置默认 AWS 区域：

- 在本地系统上的 AWS 配置文件中设置 AWS 区域，该文件位于：
  - ~/.aws/config 上的 Linux, OS X, or Unix
  - Windows 上的 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。

获取临时凭据 AWS STS

您可以使用 AWS Security Token Service (AWS STS) 获得具有有限权限的临时凭证，用于访问 AWS 服务。

使用 AWS STS 包括三个步骤：

1. 激活区域 (可选)。
2. 从 AWS STS 请求临时安全凭证。
3. 使用凭证访问 AWS 资源。

> **Note**
>
> 激活区域是可选的；默认情况下，临时安全凭证在全局终端节点 sts.amazonaws.com 获取。但为了减少延迟，并在对第一个终端节点的 AWS STS 请求失败的情况下，通过使用其他终端节点来实现请求冗余，可以激活与使用凭证的各服务或应用程序距离更接近的区域。

(可选) 激活并使用 AWS STS 区域

要激活一个区域以用于 AWS STS，请使用 AWS 管理控制台来选择和激活区域。

要激活其他 STS 区域

1. 对于您要在新区域中为其激活 AWS STS 的账户，作为有权执行 IAM 管理任务的 IAM 用户 `"iam:*"` 登录。
2. 打开 IAM 控制台，然后在导航窗格中单击 Account Settings。
3. 展开 STS Regions (STS 区域) 列表，找到要使用的区域，然后单击 Activate (激活)。

在此后，可直接调用与该区域关联的 STS 终端节点。

> **Note**
>
> 有关激活 STS 区域的更多信息和可用 AWS STS 终端节点的列表，请参阅 IAM User Guide 中的在 AWS 区域中激活和停用 AWS STS。

从 AWS STS 请求临时安全凭证

要使用适用于 Java 的 AWS 开发工具包请求临时安全凭证

1. 创建 AWSSecurityTokenServiceClient 对象：

```
AWSSecurityTokenService sts_client = new
 AWSSecurityTokenServiceClientBuilder().standard().withEndpointConfiguration(new
 AwsClientBuilder.EndpointConfiguration("sts-endpoint.amazonaws.com", "signing-
region")).build()
```

   在创建不带参数的客户端 (`AWSSecurityTokenService sts_client = new AWSSecurityTokenServiceClientBuilder().standard().build();`) 时，会使用默认的凭证提供程序链检索凭证。如果您需要，可以提供特定的凭证提供程序：有关更多信息，请参阅“在适用于 Java 的 AWS 开发工具包中提供 AWS 凭证”。
2. 创建 GetSessionTokenRequest 对象，还可以设置临时凭证的有效期 (秒)：

```
GetSessionTokenRequest session_token_request = new GetSessionTokenRequest();
session_token_request.setDurationSeconds(7200); // optional.
```

   对于 IAM 用户，临时凭证的有效期范围是 900 秒 (15 分钟) 到 129600 秒 (36 小时)。如果不指定有效期，则默认使用 43200 秒 (12 小时) 。

   对于根 AWS 账户，临时凭证的有效期范围是 900 到 3600 秒 (1 小时)，如果不指定有效期，则使用默认值 3600 秒。

Important

从安全角度出发，强烈建议 您使用 IAM 用户 而非根账户来进行 AWS 访问。有关更多信息，请参阅 IAM User Guide 中的 IAM 最佳实践。

3. 在 STS 客户端上调用 getSessionToken 以获取会话令牌 (使用 `GetSessionTokenRequest` 对象)：

```
GetSessionTokenResult session_token_result =
    sts_client.getSessionToken(session_token_request);
```

4. 使用调用 `getSessionToken` 的结果获取会话凭证：

```
Credentials session_creds = session_token_result.getCredentials();
```

使用会话凭证只能在 `GetSessionTokenRequest` 对象指定的有效期内进行访问。在凭证过期后，需再次调用 `getSessionToken` 来获取新的会话令牌，才能继续访问 AWS。

使用临时凭证访问 AWS 资源

获得临时安全凭证后，可以按照明确指定凭证 (p. 24)中说明的方法，使用它们对 AWS 服务客户端进行初始化。

例如，使用临时服务凭证创建 S3 客户端：

```
BasicSessionCredentials sessionCredentials = new BasicSessionCredentials(
   session_creds.getAccessKeyId(),
   session_creds.getSecretAccessKey(),
   session_creds.getSessionToken());

AmazonS3 s3 = AmazonS3ClientBuilder.standard()
                        .withCredentials(new
 AWSStaticCredentialsProvider(sessionCredentials))
                        .build();
```

现在可以使用 `AmazonS3` 对象发出 Amazon S3 请求。

有关

有关如何使用临时安全凭证访问 AWS 资源的更多信息，请参阅 IAM User Guide 中以下几节的内容：

- 请求临时安全凭证
- 控制临时安全凭证的权限
- 使用临时安全凭证请求访问 AWS 资源
- 在 AWS 区域中激活和停用 AWS STS

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 软件包

要创建基本的 Maven 软件包，请打开终端 (命令行) 窗口并运行：

```
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软件包命名空间，将 myapp 替换为项目的名称 (这将变为项目的目录名称)。

默认情况下， 使用 quickstart 原型为您创建项目模板，该原型是许多项目的绝佳起点。还提供了更多原型；有关随 打包的原型的列表，请访问 Maven 原型页。可以通过向 `-DarchetypeArtifactId` 命令中添加 `archetype:generate` 参数来选择要使用的特定原型。例如：。

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-webapp \
  -DgroupId=org.example.webapp \
  -DartifactId=mywebapp
```

> **Note**
>
> Maven 入门指南中提供了有关创建和配置 项目的详细信息。

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。从 1.9.0 版开始，可以导入单个组件 (p. 10)或整个开发工具包 (p. 11)。

指定单独的开发工具包模块

要选择单个开发工具包模块，请使用 AWS SDK for Java 的 Maven 材料清单 (BOM)，这将确保您指定的所有模块使用相同版本的开发工具包而且相互兼容。

要使用 BOM，请向应用程序的 `<dependencyManagement>` 文件中添加一个 `pom.xml` 部分，将 `aws-java-sdk-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>com.amazonaws</groupId>
      <artifactId>aws-java-sdk-bom</artifactId>
      <version>1.11.327</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，请访问：https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-bom。您也可以使用此页查看项目 `pom.xml` 文件的 `<dependencies>` 部分中包括的 BOM 管理了哪些模块（依赖项）。

现在，可以从您的应用程序中所使用的开发工具包中选择单个模块。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>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-s3</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-dynamodb</artifactId>
  </dependency>
</dependencies>
```

还可以参考 AWS Code Catalog 来了解要用于给定 AWS 服务的依赖关系。请参阅特定的服务示例下的 POM 文件。例如，如果您想了解 AWS S3 服务的依赖关系，请参阅 GitHub 上的完整示例。（查看 /java/example_code/s3 下的 pom）。

导入所有开发工具包模块

如果您想将整个开发工具包作为一个依赖项拉入，请勿使用 BOM 方法，而只需在 `pom.xml` 中声明该开发工具包，如下所示：

```
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk</artifactId>
    <version>1.11.327</version>
  </dependency>
</dependencies>
```

构建项目。

在设置项目之后，可以使用 Maven 的 `package` 命令进行构建：

```
mvn package
```

这会在 `.jar` 目录中创建 `target` 文件。

使用 Maven 构建开发工具包

可以使用 Apache Maven 从源构建开发工具包。为此，请从 GitHub 下载开发工具包代码，在本地解压缩，然后执行下面的 Maven 命令：

```
mvn clean install
```

将开发工具包与 Gradle 一起使用

要管理 Gradle 项目的开发工具包依赖项，请将AWS SDK for Java的 Maven BOM 导入到 `build.gradle` 文件中。

> **Note**
>
> 在以下示例中，将构建文件中的 1.11.X 替换为AWS SDK for Java的有效版本。可以在适用于 Java 的 AWS 开发工具包 1.11.x 参考中找到最新版本。

Gradle 4.6 或更高版本的项目设置

自 Gradle 4.6 开始，通过声明针对 BOM 的依赖项，便可以使用 Gradle 的经过改进的 POM 支持功能来导入物料清单 (BOM) 文件。

配置适用于 Gradle 4.6 或更高版本的AWS SDK for Java

1. 如果您使用的是 Gradle 5.0 或更高版本，请跳至步骤 2。否则，请在 `settings.gradle` 文件中启用 IMPROVED_POM_SUPPORT 功能。

```
enableFeaturePreview('IMPROVED_POM_SUPPORT')
```

2. 将 BOM 添加到 `build.gradle` 文件的 dependencies 部分

```
...
dependencies {
    implementation platform('com.amazonaws:aws-java-sdk-bom:1.11.X')

    // Declare individual SDK dependencies without version
    ...
}
```

3. 在 dependencies 部分中指定要使用的开发工具包模块。例如，以下内容包括 Amazon Simple Storage Service (Amazon S3) 的依赖项。

```
...
dependencies {
    implementation 'com.amazonaws:aws-java-sdk-s3'
    ...
}
```

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。

以下是包含 Amazon S3 的依赖项的完整 `build.gradle` 文件的示例。

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

apply plugin: 'java'

sourceCompatibility = 1.8

repositories {
  mavenCentral()
}

dependencies {
  implementation platform('com.amazonaws:aws-java-sdk-bom:1.11.X')
  implementation 'com.amazonaws:aws-java-sdk-s3'
  testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

> **Note**
>
> 在前面的示例中，将 Amazon S3 的依赖项替换为您将在项目中使用的 AWS 服务的依赖项。由AWS SDK for Java BOM 管理的模块（依赖项）列在 Maven 中央存储库上 (https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-bom/latest)。

用于 4.6 之前的 Gradle 版本的项目设置

早于 4.6 的 Gradle 版本缺少本机 BOM 支持。要管理项目的AWS SDK for Java依赖项，请使用 Spring 的适用于 Gradle 的依赖项管理插件为开发工具包导入 Maven BOM。

配置用于 4.6 之前的 Gradle 版本的开发工具包

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:1.0.9.RELEASE"
    }
}

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

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

```
dependencyManagement {
    imports {
        mavenBom 'com.amazonaws:aws-java-sdk-bom:1.11.X'
    }
}
```

3. 在 dependencies 部分中指定您将使用的开发工具包模块。例如，以下内容包含 Amazon S3 的依赖项。

```
dependencies {
    compile 'com.amazonaws:aws-java-sdk-s3'
}
```

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。

以下是包含 Amazon S3 的依赖项的完整 `build.gradle` 文件的示例。

```
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:1.0.9.RELEASE"
  }
}

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

```
dependencyManagement {
  imports {
    mavenBom 'com.amazonaws:aws-java-sdk-bom:1.11.X'
  }
}

dependencies {
  compile 'com.amazonaws:aws-java-sdk-s3'
  testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

Note

在前面的示例中，将 Amazon S3 的依赖项替换为您将在项目中使用的 AWS 服务的依赖项。由AWS SDK for Java BOM 管理的模块（依赖项）列在 Maven 中央存储库上 (https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-bom/latest)。

有关使用 BOM 指定开发工具包依赖项的更多信息，请参阅将开发工具包与 Apache Maven 一起使用 (p. 9)。

启用适用于企业支持的 AWS 开发工具包指标

适用于企业支持的 AWS 开发工具包指标 (开发工具包指标) 可让企业客户从其主机和客户端上的 AWS 开发工具包收集指标，这些指标与 AWS 企业支持共享。开发工具包指标 提供的信息有助于加快检测和诊断 AWS 企业支持客户的 AWS 服务连接中发生的问题。

在每个主机上收集遥测数据时，它通过 UDP 中继到 127.0.0.1（也称为 localhost），在此处 CloudWatch 代理会汇总数据并将其发送到 开发工具包指标 服务。因此，要接收指标，需要将 CloudWatch 代理添加到您的实例。

设置开发工具包指标的以下步骤适用于为正在使用AWS SDK for Java的客户端应用程序运行 Amazon Linux 的 Amazon EC2 实例。如果在配置AWS SDK for Java时启用，开发工具包指标也可用于您的生产环境。

要使用 开发工具包指标，请运行最新版本的 CloudWatch 代理。了解如何在 Amazon CloudWatch User Guide 中为开发工具包指标配置 CloudWatch 代理。

要使用AWS SDK for Java设置开发工具包指标，请按照以下说明操作：

1. 使用AWS SDK for Java客户端创建应用程序以使用 AWS 服务。
2. 在 Amazon EC2 实例上或您的本地环境中托管您的项目。
3. 安装并使用最新的 1.x 版本的 AWS SDK for Java。
4. 在 EC2 实例上或您的本地环境中安装和配置 CloudWatch 代理。
5. 授权 开发工具包指标 收集和发送指标。
6. 为适用于 Java 的 AWS 开发工具包启用开发工具包指标 (p. 14)。

有关更多信息，请参阅以下内容：。

- 更新 CloudWatch 代理 (p. 16)
- 禁用开发工具包指标 (p. 17)

为 AWS SDK for Java 启用 开发工具包指标

默认情况下，开发工具包指标 处于关闭状态，并且端口设置为 31000。下面是默认参数。

```
//default values
 [
     'enabled' => false,
     'port' => 31000,
 ]
```

启用 开发工具包指标 与配置凭证以使用 AWS 服务无关。

您可以使用 4 个选项之一启用 开发工具包指标。

- 选项1: 设置环境变量 (p. 15)
- 选项2: 在代码中设置SDK度量标准 (p. 15)
- 选项3: 设置Java系统属性 (p. 15)
- 选项4: AWS 共享配置文件 (p. 16)

选项 1 设置环境变量

如果未设置 `AWS_CSM_ENABLED`，则开发工具包首先会检查 `AWS_PROFILE` 下的环境变量中指定的配置文件，以确定是否已启用 开发工具包指标。默认情况下，该选项设置为 `false`。

要打开开发工具包指标，请将以下内容添加到环境变量中。

```
export AWS_CSM_ENABLED=true
```

其他配置设置 (p. 16)可用。

注意 启用开发工具包指标不会将您的凭证配置为使用 AWS 服务。

选项 2：设置 开发工具包指标 代码

使用 Java 实施，您可以在构建服务客户端时在代码内部设置 开发工具包指标 配置。代码中设置的值将覆盖在下面描述的其他选项中设置的任何配置。

```
CsmConfiguration csmConfig = new CsmConfiguration(true, MY_PORT, MY_CLIENT_ID);
AmazonDynamoDB dynamodb = AmazonDynamoDBClientBuilder.standard()
   .withClientSideMonitoringConfigurationProvider(new
 StaticCsmConfigurationProvider(csmConfig))
   .build();
```

选项 3 设置Java系统属性

如果在环境变量中未找到任何 开发工具包指标 配置，则开发工具包会查看特定 Java 系统属性。

要打开 开发工具包指标，请在执行应用程序时传递以下系统属性标志。

```
-Dcom.amazonaws.sdk.csm.enabled="true"
```

您还可以使用属性对象以编程方式设置值。

```
 Properties props = System.getProperties();
props.setProperty("com.amazonaws.sdk.csm.enabled", "true");
```

其他配置设置 (p. 16)可用。

注意 启用开发工具包指标不会将您的凭证配置为使用 AWS 服务。

选项 4：AWS 共享配置文件

如果在环境变量或 Java 系统属性中未找到任何 开发工具包指标 配置，则开发工具包将查看您的默认 AWS 配置文件字段。如果 `AWS_DEFAULT_PROFILE` 设置为默认值之外的其他值，请更新该配置文件。要启用 开发工具包指标，请将 `csm_enabled` 添加到位于 `~/.aws/config` 的共享配置文件。

```
[default]
csm_enabled = true

[profile aws_csm]
csm_enabled = true
```

其他配置设置 (p. 16)可用。

注意 启用 开发工具包指标 独立于配置凭据以使用AWS服务。您可以使用其他配置文件进行身份验证。

更新 CloudWatch 代理

要更改端口，您需要设置值，然后重新启动当前处于活动状态的所有 AWS 作业。

选项 1 设置环境变量

大多数服务都使用默认端口。但是，如果您的服务需要唯一的端口 ID，请将 AWS_CSM_PORT=[port_number] 添加到主机的环境变量。

```
export AWS_CSM_ENABLED=true
export AWS_CSM_PORT=1234
```

选项 2：设置Java系统属性

大多数服务都使用默认端口。但是，如果您的服务需要唯一的端口 ID，请在执行您的应用程序时指定 -Dcom.amazonaws.sdk.csm.port=[port_number] 系统属性。

```
com.amazonaws.sdk.csm.enabled=true
com.amazonaws.sdk.csm.port=1234
```

选项 3 AWS 共享配置文件

大多数服务都使用默认端口。但是，如果您的服务需要唯一的端口 ID，请将 csm_port = [port_number] 添加到 ~/.aws/config。

```
[default]
csm_enabled = false
csm_port = 1234

[profile aws_csm]
csm_enabled = false
csm_port = 1234
```

重新启动 开发工具包指标

要重新启动作业，请运行以下命令。

```
amazon-cloudwatch-agent-ctl -a stop;
amazon-cloudwatch-agent-ctl -a start;
```

禁用 开发工具包指标

关闭 开发工具包指标，设置 CSM_已启用 至 错误 在您的环境变量中，或者您的AWS共享配置文件位于 `~/.aws/config`。然后重新启动 CloudWatch 代理以使更改生效。

环境变量

```
export AWS_CSM_ENABLED=false
```

AWS 共享配置文件

从位于 `~/.aws/config` 的 AWS 共享配置文件的配置文件中删除 csm_enabled。

> **Note**
>
> 环境变量会覆盖 AWS 共享配置文件。如果已在环境变量中启用 开发工具包指标，则 开发工具包指标 保持启用状态。

```
[default]
csm_enabled = false

[profile aws_csm]
csm_enabled = false
```

要禁用 开发工具包指标，请使用以下命令停止 CloudWatch 代理。

```
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -a stop &&
echo "Done"
```

如果您正在使用其他 CloudWatch 功能，请使用以下命令重新启动 CloudWatch 代理。

```
amazon-cloudwatch-agent-ctl -a start;
```

重新启动 开发工具包指标

要重新启动 开发工具包指标，请运行以下命令。

```
amazon-cloudwatch-agent-ctl -a stop;
amazon-cloudwatch-agent-ctl -a start;
```

开发工具包指标 的定义

您可以使用以下开发工具包指标描述来解释您的结果。通常，这些指标可在常规业务审核期间与您的技术客户经理进行审核。AWS Support 资源和您的技术客户经理应该有权访问开发工具包指标数据以帮助您解决案例，但如果您发现的数据令人困惑或是意外数据，但似乎不会对您的应用程序的性能产生负面影响，那么最好是在预定的业务审核期间审核该数据。

指标	CallCount
Definition	从您的代码向 AWS 服务进行的成功或失败的 API 调用总数。
使用方法	将它用作基准与其他指标（如错误或限制）相关联。

指标	ClientErrorCount
Definition	失败且出现客户端错误（4xx HTTP 响应代码）的 API 调用数量。示例：限制、拒绝访问、S3存储区不存在、参数值无效。
使用方法	除了在某些与限制相关的情况下（例如，由于需要增加限额而发生限制时），此指标可以指明应用程序中需要修复的内容。

指标	ConnectionErrorCount
Definition	由于连接到服务出错而失败的 API 调用数量。这些可能由客户应用程序与 AWS 服务之间的网络问题所导致，包括负载均衡器问题、DNS 故障以及中转提供商问题。在某些情况下，AWS 问题可能导致此错误。
使用方法	使用此指标可确定问题是特定于您的应用程序，还是由基础设施和/或网络所导致。较高的 ConnectionErrorCount 可能也指示 API 调用的超时值较短。

指标	ThrottleCount
Definition	由于 AWS 服务的限制而失败的 API 调用数量。
使用方法	使用此指标来评估应用程序是否已达到限制，以及确定重试和应用程序延迟的原因。请考虑在窗口之上分配调用，而不是批处理调用。

指标	ServerErrorCount
Definition	由于来自 AWS 服务的服务器错误（5xx HTTP 响应代码）而失败的 API 调用数量。这些错误通常由 AWS 服务所导致。
使用方法	确定开发工具包重试或延迟的原因。此指标并不总是指示 AWS 服务出现故障，因为有些 AWS 团队将延迟分类为 HTTP 503 响应。

指标	EndToEndLatency
Definition	应用程序使用 AWS 开发工具包进行调用（含重试）的总时间。换句话说，无论多次尝试后是否成功，或者一旦调用由于不可传递的错误而失败。
使用方法	确定 AWS API 调用对应用程序总体延迟的贡献。网络、防火墙或其他配置设置问题或开发工具包重试导致的延迟可能会导致延迟高于预期值。

使用 AWS SDK for Java

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。

有关特定于服务的编程信息和示例（用于 Amazon EC2、Amazon S3、Amazon SWF 等等），请参阅适用于 Java 的 AWS 开发工具包代码示例 (p. 40)。

主题

AWS开发的最佳实践 AWS SDK for Java

以下最佳实践可帮助您避免在使用 AWS SDK for Java 开发 AWS 应用程序时遇到问题和麻烦。这些最佳实践已按服务分类整理。

Amazon S3

避免 ResetExceptions

当您使用流（通过 AmazonS3 客户端或 TransferManager）将对象上传到 Amazon S3 时，可能遇到网络连接或超时问题。默认情况下，AWS SDK for Java 将尝试通过以下方式尝试重试失败的传输：先在传输开始前标记输入流，然后在重试前重新设置它。

如果流不支持标记和重置操作，则开发工具包将在出现临时故障并支持重试时引发 ResetException。

最佳实践

建议您使用支持标记和重置操作的流。

避免 ResetException 的最可靠方式是使用 File 或 FileInputStream 提供数据，因为它们可由AWS SDK for Java处理，不受标记和重置限制的约束。

如果流不是 FileInputStream 但支持标记和重置操作，您可以使用 RequestClientOptions 的 `setReadLimit` 方法设置标记限制。其默认值为 128KB。将读取限制值设置为比流大小多 1 个字节 的值将可靠地避免 ResetException。

例如，如果流的最大预期大小为 100000 字节，则将读取限制设置为 100001 (100000 + 1) 字节。标记和重置操作将始终适用于 100000 字节或更少的字节。请注意，这可能会导致一些流将该数量的字节缓冲到内存中。

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。推荐的方法是使用服务客户端生成器。

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 AmazonDynamoDB。每个服务接口都有对应的客户端生成器，可用于构建服务接口的实施。DynamoDB 的客户端生成器类名为 AmazonDynamoDBClientBuilder。

获取客户端生成器

要获取客户端生成器的实例，使用下例中所示的静态工厂方法 `standard`。

```
AmazonDynamoDBClientBuilder builder = AmazonDynamoDBClientBuilder.standard();
```

获得生成器以后，可以使用生成器 API 中的多个常用 setter 来自定义客户端的属性。例如，您可以按以下方法设置自定义区域和自定义凭证提供程序。

```
AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.standard()
                        .withRegion(Regions.US_WEST_2)
                        .withCredentials(new ProfileCredentialsProvider("myProfile"))
                        .build();
```

> **Note**
>
> 常用的 `withXXX` 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。创建的客户端不可更改，而且对 `setRegion` 或 `setEndpoint` 的所有调用都会失败。

生成器可以使用相同配置创建多个客户端。在编写应用程序时，请注意生成器可变而且是非线程安全的。

以下代码使用生成器作为客户端实例的工厂。

```
public class DynamoDBClientFactory {
    private final AmazonDynamoDBClientBuilder builder =
        AmazonDynamoDBClientBuilder.standard()
            .withRegion(Regions.US_WEST_2)
            .withCredentials(new ProfileCredentialsProvider("myProfile"));

    public AmazonDynamoDB createClient() {
        return builder.build();
    }
}
```

生成器还显示 ClientConfiguration、RequestMetricCollector 以及自定义 RequestHandler2 列表的常用 setter。

以下给出将覆盖所有可配置属性的完整示例。

```
AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.standard()
```

```
        .withRegion(Regions.US_WEST_2)
        .withCredentials(new ProfileCredentialsProvider("myProfile"))
        .withClientConfiguration(new ClientConfiguration().withRequestTimeout(5000))
        .withMetricsCollector(new MyCustomMetricsCollector())
        .withRequestHandlers(new MyCustomRequestHandler(), new MyOtherCustomRequestHandler)
        .build();
```

创建异步客户端

AWS SDK for Java的每个服务均有异步客户端（Amazon S3 除外），且每个服务都有相应的异步客户端生成器。

使用默认 ExecutorService 创建异步 DynamoDB 客户端

```
AmazonDynamoDBAsync ddbAsync = AmazonDynamoDBAsyncClientBuilder.standard()
        .withRegion(Regions.US_WEST_2)
        .withCredentials(new ProfileCredentialsProvider("myProfile"))
        .build();
```

除了同步（或同步）客户端生成器支持的配置选项之外，异步客户端还可以设置自定义 执行工厂 更改 `ExecutorService` 异步客户端使用。`ExecutorFactory` 是一个功能接口，因此它与Java8λ表达式和方法参考相互操作。

使用自定义执行程序创建异步客户端

```
AmazonDynamoDBAsync ddbAsync = AmazonDynamoDBAsyncClientBuilder.standard()
            .withExecutorFactory(() -> Executors.newFixedThreadPool(10))
            .build();
```

使用 DefaultClient

同步和异步客户端构建器都有另一个名为 `defaultClient`。此方法使用默认提供商链来加载凭据和AWS区域，创建具有默认配置的服务客户端。如果不能根据运行应用程序的环境确定凭证或区域，则对 `defaultClient` 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. 22)和AWS 区域选择 (p. 25)。

创建默认服务客户端

```
AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
```

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，而且为了获得最佳性能，应该将其作为永久对象。每个客户端均有各自的连接池资源。将显式关闭不再需要的客户端，以避免资源泄漏。

要显式关闭客户端，请调用 `shutdown` 方法。在调用 `shutdown` 后，会释放所有客户端资源且客户端不可用。

关闭客户端

```
AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
```

```
ddb.shutdown();
// Client is now unusable
```

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。这些凭证可以是根账户凭证、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 用户而非根账户来进行 AWS 访问。有关更多信息，请参阅 IAM User Guide 中的 IAM 最佳实践。

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 DefaultAWSCredentialsProviderChain 类实施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. 环境变量–`AWS_ACCESS_KEY_ID` 和 `AWS_SECRET_ACCESS_KEY`...TheThethe AWS SDK for Java 使用环境变量LecredEntialsProvider 等级以加载这些凭据。
2. Java系统属性–`aws.accessKeyId` 和 `aws.secretKey`...TheThethe AWS SDK for Java 使用 系统属性凭据提供商 加载这些凭据。
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，也可以使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. 23)。
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。可以指定此值的 IP 地址。
5. 实例配置文件凭证 – 在 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。可以指定此值的 IP 地址。

   > **Note**
   >
   > 仅在未设置 `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` 时使用实例配置文件凭证。有关更多信息，请参阅 EC2ContainerCredentialsProviderWrapper。
6. 来自环境或容器的 Web 身份令牌凭证。

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：

- 要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. 6)。
- 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，请参阅使用 IAM 角色授予对 Amazon EC2 上的 AWS 资源的访问权 (p. 60)。

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 配置文件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，您将使用以下配置文件。

```
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。

> Note
>
> 环境变量优先于系统属性。

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。

您可以指定其他字段，例如 aws_session_token， metadata_service_timeout，和 metadata_service_num_attempts。这些不是与CLI配置的—如果要使用这些文件，您必须手动编辑文件。有关配置文件及其可用字段的更多信息，请参阅 AWS CLI User Guide 中的配置 AWS 命令行界面。

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，应实例化 AWS 服务客户端，但不向生成器明确提供凭证，如下所示。

```
AmazonS3 s3Client = AmazonS3ClientBuilder.standard()
                        .withRegion(Regions.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。

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。以下示例展示使用环境 凭证的具体情况。

```
AmazonS3 s3Client = AmazonS3ClientBuilder.standard()
                        .withCredentials(new EnvironmentVariableCredentialsProvider())
                        .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 中的所有已知实施类。

> 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，方式是使用您自己的可实施 AWSCredentialsProvider 接口的凭证提供程序或通过为 AWSCredentialsProviderChain 类生成子类。

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 BasicAWSCredentials)，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 AWSStaticCredentialsProvider。
3. 使用 AWSStaticCredentialsProvider 配置客户端生成器并构建客户端。

以下是示例：

```
BasicAWSCredentials awsCreds = new BasicAWSCredentials("access_key_id", "secret_key_id");
AmazonS3 s3Client = AmazonS3ClientBuilder.standard()
                        .withCredentials(new AWSStaticCredentialsProvider(awsCreds))
```

```
                        .build();
```

在使用从 STS 获取的临时凭证时 (p. 7)，将创建一个 BasicSessionCredentials 对象，并为该对象传递 STS 提供的凭证和会话令牌。

```
BasicSessionCredentials sessionCredentials = new BasicSessionCredentials(
   session_creds.getAccessKeyId(),
   session_creds.getSecretAccessKey(),
   session_creds.getSessionToken());

AmazonS3 s3 = AmazonS3ClientBuilder.standard()
                        .withCredentials(new
 AWSStaticCredentialsProvider(sessionCredentials))
                        .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. 6)
- 使用 IAM 角色授予对 Amazon EC2 上的 AWS 资源的访问权 (p. 60)

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。

查看 AWS 区域的服务可用性

要确认在一个区域内特定的 AWS 服务是否可用，请对要使用的区域使用 `isServiceSupported` 方法。

```
Region.getRegion(Regions.US_WEST_2)
    .isServiceSupported(AmazonDynamoDB.ENDPOINT_PREFIX);
```

请参阅区域类文档查看可以指定的区域，并使用服务的终端节点前缀进行查询。在服务接口中定义了各服务的终端节点前缀。例如，DynamoDB 的终端节点前缀在 AmazonDynamoDB 中定义。

选择区域

从 AWS SDK for Java 的 1.4 版本开始，您可指定区域名称，然后开发工具包将自动选择适当的终端节点。要自行选择终端节点，请参阅选择特定终端节点 (p. 26)。

要显式设置区域时，我们建议您使用 Regions 枚举。这是所有公开可用区域的枚举。要使用枚举结果中的一个区域创建客户端，请使用以下代码。

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.standard()
                    .withRegion(Regions.US_WEST_2)
                    .build();
```

如果 `Regions` 枚举结果不包含要使用的某个区域，可使用代表该区域名称的字符串。

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.standard()
                    .withRegion("us-west-2")
                    .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

在创建客户端时，通过调用 `withEndpointConfiguration` 方法，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 S3 客户端配置为使用 欧洲（爱尔兰）区域，请使用以下代码。

```
AmazonS3 s3 = AmazonS3ClientBuilder.standard()
     .withEndpointConfiguration(new EndpointConfiguration(
          "https://s3.eu-west-1.amazonaws.com",
          "eu-west-1"))
     .withCredentials(CREDENTIALS_PROVIDER)
     .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

Important

此部分仅适用于使用客户端生成器 (p. 20)访问 AWS 服务。使用客户端构造函数创建的 AWS 客户端不会根据环境自动确定区域，而是使用默认 开发工具包区域 (USEast1)。

在 Amazon EC2 或 Lambda 上运行时，可能需要将客户端配置为与所运行代码使用同一个区域。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。

必须使用客户端生成器，使开发工具包可自动检测代码的运行区域。

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 `defaultClient` 方法：

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();
```

这与使用 `standard` 再加上 `build` 相同。

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.standard()
                    .build();
```

如果您没有使用 `withRegion` 方法明确设置一个区域，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 `withRegion` 或 `setRegion` 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，自定义开发工具包要加载的配置文件。

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. 22)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。

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。

有关使用选中和取消选中的异常的更多信息，请参阅：

- 取消选中的异常 - 争议
- 使用取消选中的异常时的问题
- Java 中取消选中的异常是一个错误（下文会说明我要如何处理这些异常）

AmazonServiceException (和子类)

`AmazonServiceException` 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 `AmazonServiceException` 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 `AmazonServiceException` 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 `AmazonServiceException` 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。

`AmazonServiceException` 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

`AmazonServiceException` 中还包括相关信息，指出请求失败原因是调用方的错误（请求的值非法），还是 AWS 服务的错误（内部服务错误）。

AmazonClientException

AmazonClientException 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 客户端代码出现问题。在一般情况下，AmazonClientException 比 AmazonServiceException 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 AmazonClientException。

异步编程

使用同步或异步方法都可以调用对 AWS 服务的操作。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在响应准备就绪时接收响应。TheThethe AWS SDK for Java 提供两种方式: 未来对象 和 回调方法.

Java Futures

AWS SDK for Java 中的异步方法会返回 Future 对象，其中包含之后 的异步操作的结果。

调用 Future isDone() 方法，确定该服务是否已提供响应对象。当响应准备好时，可以通过调用 Future get() 方法来获取响应对象。在应用程序继续处理其他任务时，可使用该机制定期轮询异步操作的结果。

以下示例演示一个调用 Lambda 函数的异步操作，该操作收到可包含 InvokeResult 对象的 Future。InvokeResult 对象仅在 isDone() 为 true 时可检索到。

```
import com.amazonaws.services.lambda.AWSLambdaAsyncClient;
import com.amazonaws.services.lambda.model.InvokeRequest;
import com.amazonaws.services.lambda.model.InvokeResult;
import java.nio.ByteBuffer;
import java.util.concurrent.Future;
import java.util.concurrent.ExecutionException;

public class InvokeLambdaFunctionAsync
{
    public static void main(String[] args)
    {
        String function_name = "HelloFunction";
        String function_input = "{\"who\":\"AWS SDK for Java\"}";

        AWSLambdaAsync lambda = AWSLambdaAsyncClientBuilder.defaultClient();
        InvokeRequest req = new InvokeRequest()
            .withFunctionName(function_name)
            .withPayload(ByteBuffer.wrap(function_input.getBytes()));

        Future<InvokeResult> future_res = lambda.invokeAsync(req);

        System.out.print("Waiting for future");
        while (future_res.isDone() == false) {
            System.out.print(".");
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                System.err.println("\nThread.sleep() was interrupted!");
                System.exit(1);
            }
        }

        try {
```

```
            InvokeResult res = future_res.get();
            if (res.getStatusCode() == 200) {
                System.out.println("\nLambda function returned:");
                ByteBuffer response_payload = res.getPayload();
                System.out.println(new String(response_payload.array()));
            }
            else {
                System.out.format("Received a non-OK response from AWS: %d\n",
                        res.getStatusCode());
            }
        }
        catch (InterruptedException | ExecutionException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        System.exit(0);
    }
}
```

异步回调

除了使用Java `Future` 对象来监控异步请求的状态，SDK还允许您实施一个使用 东钱德勒 接口。`AsyncHandler` 根据请求完成的方式提供两种调用方法: `onSuccess` 和 `onError`.

回调接口方法的主要优势是它让您无需轮询 `Future` 对象即可确定请求是否已完成。相反，您的代码能够立即开始其下一个活动，并由开发工具包在适当时调用处理程序。

```
import com.amazonaws.services.lambda.AWSLambdaAsync;
import com.amazonaws.services.lambda.AWSLambdaAsyncClientBuilder;
import com.amazonaws.services.lambda.model.InvokeRequest;
import com.amazonaws.services.lambda.model.InvokeResult;
import com.amazonaws.handlers.AsyncHandler;
import java.nio.ByteBuffer;
import java.util.concurrent.Future;

public class InvokeLambdaFunctionCallback
{
    private class AsyncLambdaHandler implements AsyncHandler<InvokeRequest, InvokeResult>
    {
        public void onSuccess(InvokeRequest req, InvokeResult res) {
            System.out.println("\nLambda function returned:");
            ByteBuffer response_payload = res.getPayload();
            System.out.println(new String(response_payload.array()));
            System.exit(0);
        }

        public void onError(Exception e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void main(String[] args)
    {
        String function_name = "HelloFunction";
        String function_input = "{\"who\":\"AWS SDK for Java\"}";

        AWSLambdaAsync lambda = AWSLambdaAsyncClientBuilder.defaultClient();
        InvokeRequest req = new InvokeRequest()
            .withFunctionName(function_name)
            .withPayload(ByteBuffer.wrap(function_input.getBytes()));
```

```
        Future<InvokeResult> future_res = lambda.invokeAsync(req, new
 AsyncLambdaHandler());

        System.out.print("Waiting for async callback");
        while (!future_res.isDone() && !future_res.isCancelled()) {
            // perform some other tasks...
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                System.err.println("Thread.sleep() was interrupted!");
                System.exit(0);
            }
            System.out.print(".");
        }
    }
}
```

最佳实践

回调执行

`AsyncHandler` 的实施在异步客户端拥有的线程池内执行。简短、快速执行的代码在您的 `AsyncHandler` 实施内最适合。如果您的处理程序方法包含长时间运行的代码或阻码，会导致对异步客户端所使用线程池的争用，并阻止客户端执行请求。如果需要从回调开始一种长期运行的任务，请在新的线程或应用程序托管的线程池中让回调运行其任务。

线程池配置

AWS SDK for Java 中的异步客户端提供应当用于大多数应用程序的默认线程池。如果希望加强对线程池管理方式的控制，可以实施自定义 ExecutorService 并将其传递给AWS SDK for Java异步客户端。

例如，您可以提供一个 `ExecutorService` 实施，它使用自定义的 ThreadFactory 控制池中各个线程的命名方式，或者记录有关线程使用的更多信息。

Amazon S3 异步访问

开发工具包中的 TransferManager 类为使用 Amazon S3 提供异步支持。`TransferManager` 管理异步上传和下载、提供传输的详细进度报告并支持对不同事件的回调。

记录AWS SDK for Java调用

AWS SDK for Java使用 Apache Commons Logging 检测，后者是一个抽象层，可实现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。

> **Note**
>
> 本主题主要介绍 Log4j 1.x。Log4j2 不直接支持 Apache Commons Logging，但提供一个适配器，将日志记录调用定向到使用 Apache Commons Logging 界面的 Log 4j2。有关更多信息，请参阅 Log4j2 文档中的 Commons Logging Bridge。

下载 Log4J JAR

要将 Log4j 与开发工具包一起使用，需要从 Apache 网站下载 Log4j JAR。该开发工具包不包括 JAR。将 JAR 文件复制到类路径中的位置。

Log4j 使用配置文件 log4j.properties。配置文件示例如下所示。将该配置文件复制到类路径中的目录中。Log4j JAR 和 log4j.properties 文件不需要在同一目录中。

log4j.properties 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) 以及输出格式等属性。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：

- log4j.logger.com.amazonaws
- log4j.logger.org.apache.http.wire

设置类路径

Log4j JAR 和 log4j.properties 文件都必须位于类路径中。如果您使用 Apache Ant，则在 Ant 文件的 `path` 元素中设置类路径。以下示例显示开发工具包附带的 Amazon S3 示例中 Ant 文件的路径元素。

```
<path id="aws.java.sdk.classpath">
  <fileset dir="../../third-party" includes="**/*.jar"/>
  <fileset dir="../../lib" includes="**/*.jar"/>
  <pathelement location="."/>
</path>
```

如果您使用 Eclipse IDE，可以打开菜单并导航到 Project (项目) | Properties (属性) | Java Build Path (Java 构建路径) 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“com.amazonaws”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 客户端将通过向日志中记录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 log4j.properties 文件将 `rootLogger` 设置为 WARN，也就是包含“com.amazonaws”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 com.amazonaws 记录器明确设置为 WARN。

```
log4j.rootLogger=WARN, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c -  %m%n
# Or you can explicitly enable WARN and ERROR messages for the AWS Java clients
log4j.logger.com.amazonaws=WARN
```

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。如果调用任何服务时失败，可以通过开发工具包中的 Exception 对象来编程访问 AWS 请求 ID，还可以通过“com.amazonaws.request”记录器中的 DEBUG 日志级别进行报告。

以下 log4j.properties 文件用于使用包括 AWS 请求 ID 的请求和响应摘要。

```
log4j.rootLogger=WARN, A1
```

```
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c -  %m%n
# Turn on DEBUG logging in com.amazonaws.request to log
# a summary of requests/responses with AWS request IDs
log4j.logger.com.amazonaws.request=DEBUG
```

以下是日志输出的示例：

```
2009-12-17 09:53:04,269 [main] DEBUG com.amazonaws.request - Sending
Request: POST https://rds.amazonaws.com / Parameters: (MaxRecords: 20,
Action: DescribeEngineDefaultParameters, SignatureMethod: HmacSHA256,
AWSAccessKeyId: ACCESSKEYID, Version: 2009-10-16, SignatureVersion: 2,
Engine: mysql5.1, Timestamp: 2009-12-17T17:53:04.267Z, Signature:
q963XH63Lcovl5Rr71APlzlye99rmWwT9DfuQaNznkD, ) 2009-12-17 09:53:04,464
[main] DEBUG com.amazonaws.request - Received successful response: 200, AWS
Request ID: 694d1242-cee0-c85e-f31f-5dab1ea18bc6 2009-12-17 09:53:04,469
[main] DEBUG com.amazonaws.request - Sending Request: POST
https://rds.amazonaws.com / Parameters: (ResetAllParameters: true, Action:
ResetDBParameterGroup, SignatureMethod: HmacSHA256, DBParameterGroupName:
java-integ-test-param-group-0000000000000, AWSAccessKeyId: ACCESSKEYID,
Version: 2009-10-16, SignatureVersion: 2, Timestamp:
2009-12-17T17:53:04.467Z, Signature:
9WcgfPwTobvLVcpyhbrdN7P7l3uH0oviYQ4yZ+TQjsQ=, )

2009-12-17 09:53:04,646 [main] DEBUG com.amazonaws.request - Received
successful response: 200, AWS Request ID:
694d1242-cee0-c85e-f31f-5dab1ea18bc6
```

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。在生产系统中不应该启用该日志记录，因为写出大的请求（例如，上传到 Amazon S3 的文件）或响应文件会导致应用程序速度明显下降。如果确实需要访问相关信息，可以通过 Apache HttpClient 4 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。

以下 log4j.properties 文件会在 Apache HttpClient 4 中启用完整线路日志记录，只能短时间地打开该文件，否则它会对应用程序的性能造成严重影响。

```
log4j.rootLogger=WARN, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c -  %m%n
# Log all HTTP content (headers, parameters, content, etc)  for
# all requests and responses. Use caution with this since it can
# be very expensive to log such verbose data!
log4j.logger.org.apache.http.wire=DEBUG
```

延迟指标日志记录

如果您正在进行故障排除，并且希望查看诸如哪个进程占用了最多时间或者是服务器还是客户端具有更大延迟等指标，则延迟记录器可能会很有用。将 `com.amazonaws.latency` 记录器设置为 DEBUG 可启用此记录器。

Note

此记录器仅在启用开发工具包指标时才可用。要了解更多有关开发工具包指标软件包的更多信息，请参阅对适用于 Java 的 AWS 开发工具包启用指标 (p. 36)。

```
log4j.rootLogger=WARN, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c -  %m%n
log4j.logger.com.amazonaws.latency=DEBUG
```

以下是日志输出的示例：

```
com.amazonaws.latency - ServiceName=[Amazon S3], StatusCode=[200],
ServiceEndpoint=[https://list-objects-integ-test-test.s3.amazonaws.com],
RequestType=[ListObjectsV2Request], AWSRequestID=[REQUESTID], HttpClientPoolPendingCount=0,
RetryCapacityConsumed=0, HttpClientPoolAvailableCount=0, RequestCount=1,
HttpClientPoolLeasedCount=0, ResponseProcessingTime=[52.154], ClientExecuteTime=[487.041],
HttpClientSendRequestTime=[192.931], HttpRequestTime=[431.652], RequestSigningTime=[0.357],
CredentialsRequestTime=[0.011, 0.001], HttpClientReceiveResponseTime=[146.272]
```

客户端配置

AWS SDK for Java 可用于更改默认客户端配置，当您希望执行以下操作时，这可能非常有用：

- 通过代理连接到 Internet
- 更改 HTTP 传输设置，例如连接超时和请求重试次数
- 指定 TCP 套接字缓冲区大小提示

代理配置

在构造客户端对象时，可以传入可选的 ClientConfiguration 对象以自定义客户端配置。

如果您通过代理服务器连接到 Internet，则将需要通过 `ClientConfiguration` 对象配置代理服务器设置（代理主机、端口和用户名/密码）。

HTTP 传输配置

通过使用 ClientConfiguration 对象配置多个 HTTP 传输选项。有时会添加新选项；要查看可以检索或设置的选项的完整列表，请参阅 AWS SDK for Java API Reference。

> Note
>
> 每个可配置的值都有一个由常量定义的默认值。有关 `ClientConfiguration` 的常量值的列表，请参阅 AWS SDK for Java API Reference 中的常量字段值。

最大连接数

可以通过使用 ClientConfiguration.setMaxConnections 方法来设置允许打开的 HTTP 连接的最大数量。

> Important
>
> 将最大连接数设置为并发事务的数量可避免连接争用和性能不佳。有关默认的最大连接数值，请参阅 AWS SDK for Java API Reference 中的常量字段值。

超时和错误处理

可以设置与 HTTP 连接超时和处理错误相关的选项。

- Connection Timeout

  连接超时是指 HTTP 连接在放弃连接之前等待建立连接的时间长度 (用毫秒表示)。默认值为 10,000 毫秒。

  要亲自设置此值，请使用 ClientConfiguration.setConnectionTimeout 方法。
- Connection Time to Live (TTL)

  默认情况下，开发工具包将尝试尽可能长时间地重用 HTTP 连接。如果因建立连接的服务器已停止服务而失败，则将 TTL 设置为有限值可能会有助于恢复应用程序。例如，将 TTL 设置为 15 分钟可确保您将在 15 分钟内与新服务器重新建立连接，即使您已经与出现问题的服务器建立了连接也是如此。

  要为 HTTP 连接设置 TTL，请使用 ClientConfiguration.setConnectionTTL 方法。
- Maximum Error Retries

  可重试的错误的默认最大重试次数为 3。您可以通过使用 ClientConfiguration.setMaxErrorRetry 方法设置不同的值。

本地地址

要设置 HTTP 客户端将绑定到的本地地址，请使用 ClientConfiguration.setLocalAddress。

TCP 套接字缓冲区大小提示

想要调整低级别 TCP 参数的高级用户可以通过 ClientConfiguration 对象额外设置 TCP 缓冲区大小提示。大多数用户永远不需要调整这些值，这些值是为高级用户提供的。

应用程序的最佳 TCP 缓冲区大小高度依赖网络和操作系统的配置和功能。例如，大多数现代操作系统都为 TCP 缓冲区大小提供了自动调整逻辑，对于需要长时间保持打开状态才能使自动调整功能优化缓冲区大小的 TCP 连接，自动调整逻辑会对连接性能产生很大的影响。

大型缓冲区大小 (例如，2 MB) 将允许操作系统在内存中缓冲更多的数据，而无需远程服务器确认收到该信息，因此，这在网络延迟时间很长时尤其有用。

这仅是一个提示，操作系统可以选择不遵守它。在使用此选项时，用户应当始终检查在操作系统中配置的限值和默认值。大多数操作系统都配置了最大 TCP 缓冲区大小限值，除非您明确提升了最大 TCP 缓冲区大小限值，否则操作系统将不允许您超出此限值。

可以使用许多资源来帮助配置 TCP 缓冲区大小和特定于操作系统的 TCP 设置，其中包括：

- 主机调整

访问控制策略

利用 AWS 访问控制策略，您可以指定对 AWS 资源的精细访问控制。访问控制策略包含一组语句，其形式如下：

账户A 有权执行 行动B 开 资源C 在哪里 条件D 适用。

其中，

- A 是委托人 – 发出访问或修改某个 AWS 资源的请求的 AWS 账户。
- B 是操作 – 访问或修改 AWS 资源的方式，例如，将消息发送到 Amazon SQS 队列或在 Amazon S3 存储桶中存储对象。
- C 是资源 – 委托人希望访问的 AWS 实体，例如 Amazon SQS 队列或存储在 Amazon S3 中的对象。

- D 是一组条件 – 指定何时允许或拒绝委托人访问资源的可选约束。有许多富有表现力的条件，还有一些特定于每项服务的条件。例如，您可以使用日期条件以仅允许在特定时间之后或之前访问资源。

Amazon S3 示例

以下示例说明一个策略，该策略向任何人授予读取某个存储桶中所有对象的权限，但仅允许两个特定的 AWS 账户（除了存储桶所有者的账户之外）将对象上传到该存储桶。

```
Statement allowPublicReadStatement = new Statement(Effect.Allow)
    .withPrincipals(Principal.AllUsers)
    .withActions(S3Actions.GetObject)
    .withResources(new S3ObjectResource(myBucketName, "*"));
Statement allowRestrictedWriteStatement = new Statement(Effect.Allow)
    .withPrincipals(new Principal("123456789"), new Principal("876543210"))
    .withActions(S3Actions.PutObject)
    .withResources(new S3ObjectResource(myBucketName, "*"));

Policy policy = new Policy()
    .withStatements(allowPublicReadStatement, allowRestrictedWriteStatement);

AmazonS3 s3 = AmazonS3ClientBuilder.defaultClient();
s3.setBucketPolicy(myBucketName, policy.toJson());
```

Amazon SQS 示例

策略的一种常见用途是，授权 Amazon SQS 队列接收来自 Amazon SNS 主题的消息。

```
Policy policy = new Policy().withStatements(
    new Statement(Effect.Allow)
        .withPrincipals(Principal.AllUsers)
        .withActions(SQSActions.SendMessage)
        .withConditions(ConditionFactory.newSourceArnCondition(myTopicArn)));

Map queueAttributes = new HashMap();
queueAttributes.put(QueueAttributeName.Policy.toString(), policy.toJson());

AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
sqs.setQueueAttributes(new SetQueueAttributesRequest(myQueueUrl, queueAttributes));
```

Amazon SNS 示例

一些服务提供可用于策略的其他条件。Amazon SNS 提供用于根据主题订阅请求的协议（例如，电子邮件、HTTP、HTTPS、Amazon SQS）和终端节点（例如，电子邮件地址、URL、Amazon SQS ARN）允许或拒绝订阅 SNS 主题的条件。

```
Condition endpointCondition =
    SNSConditionFactory.newEndpointCondition("*@mycompany.com");

Policy policy = new Policy().withStatements(
    new Statement(Effect.Allow)
        .withPrincipals(Principal.AllUsers)
        .withActions(SNSActions.Subscribe)
        .withConditions(endpointCondition));

AmazonSNS sns = AmazonSNSClientBuilder.defaultClient();
sns.setTopicAttributes(
    new SetTopicAttributesRequest(myTopicArn, "Policy", policy.toJson()));
```

设置 DNS 名称查找的 JVM TTL

Java 虚拟机 (JVM) 缓存 DNS 名称查找。当 JVM 将主机名解析为 IP 地址时，它会在指定时间段内 (称为生存时间 (TTL)) 缓存 IP 地址。

由于 AWS 资源使用偶尔变更的 DNS 名称条目，因此建议您为 JVM 配置的 TTL 值不超过 60 秒。这可确保在资源的 IP 地址发生更改时，您的应用程序将能够通过重新查询 DNS 来接收和使用资源的新 IP 地址。

对于一些 Java 配置，将设置 JVM 默认 TTL，以便在重新启动 JVM 之前绝不 刷新 DNS 条目。因此，如果 AWS 资源的 IP 地址在应用程序仍在运行时发生更改，则在您手动重新启动 JVM 并刷新缓存的 IP 信息之前，将无法使用该资源。在此情况下，设置 JVM 的 TTL，以便定期刷新其缓存的 IP 信息是极为重要的。

Note

默认 TTL 是变化的，具体取决于 JVM 的版本以及是否安装安全管理器。许多 JVM 提供的默认 TTL 小于 60 秒。如果您使用此类 JVM 并且未使用安全管理器，则您可以忽略本主题的剩余内容。

如何设置 JVM TTL

要修改 JVM 的 TTL，请设置 networkaddress.cache.ttl 属性值。根据您的需求，使用下列方法之一：

- 全局 (针对所有使用 JVM 的应用程序)。在 `$JAVA_HOME/jre/lib/security/java.security` 文件中设置 `networkaddress.cache.ttl`：

```
networkaddress.cache.ttl=60
```

- 仅针对应用程序，在应用程序的初始化代码中设置 `networkaddress.cache.ttl`：

```
java.security.Security.setProperty("networkaddress.cache.ttl" , "60");
```

启用指标 AWS SDK for Java

AWS SDK for Java可为 CloudWatch 的可视化和监控生成衡量以下各项的指标：

- 访问 AWS 时应用程序的性能
- JVM 与 AWS 结合使用时的性能
- 运行时环境详细信息，例如堆内存、线程数和已打开的文件描述符

Note

AWS SDK Metrics for Enterprise Support 是收集应用程序指标的另一种选择。SDK Metrics 是一项 AWS 服务，可向 Amazon CloudWatch 发布数据并使您能够与 AWS Support 共享指标数据以便进行故障排除。请参阅为企业支持启用 AWS 开发工具包指标 (p. 14)，了解如何为您的应用程序启用 SDK Metrics 服务。

如何启用 AWS SDK for Java 指标生成

AWS SDK for Java指标默认处于禁用状态。要为您的本地开发环境启用此功能，请在启动 JVM 时包括指向您的 AWS 安全凭证文件的系统属性。例如：。

```
-Dcom.amazonaws.sdk.enableDefaultMetrics=credentialFile=/path/aws.properties
```

您需要指定凭证文件的路径，以便开发工具包将收集到的数据点上传到 CloudWatch 供日后分析。

Note

如果您要通过 Amazon EC2 实例元数据服务从 Amazon EC2 实例访问 AWS 服务，则无需指定凭证文件。在这种情况下，您只需要指定以下各项：

```
-Dcom.amazonaws.sdk.enableDefaultMetrics
```

适用于 Java 的开发工具包捕获到的所有指标都位于命名空间 AWSSDK/Java 下，并将上传到 CloudWatch 默认区域 (us-east-1)。要更改该区域，请使用系统属性中的 `cloudwatchRegion` 属性来指定它。例如，要将 CloudWatch 区域设为 us-west-2，请使用：

```
-Dcom.amazonaws.sdk.enableDefaultMetrics=credentialFile=/path/
aws.properties,cloudwatchRegion=us-west-2
```

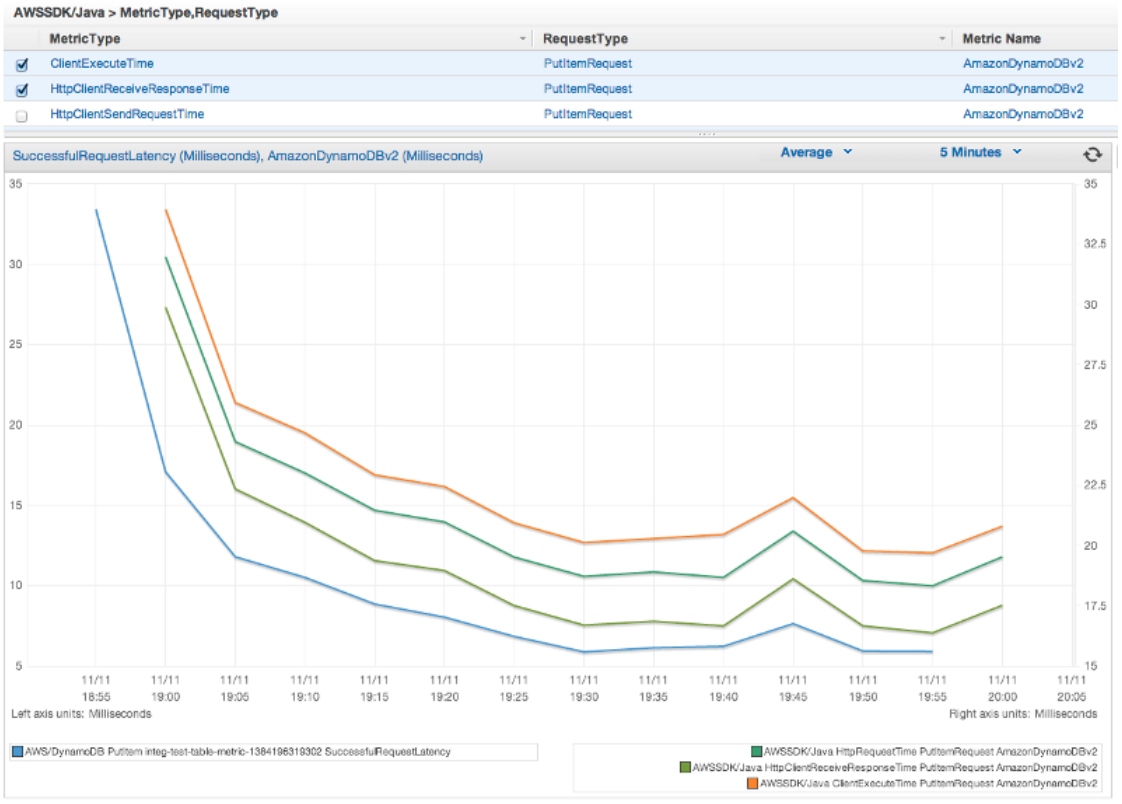
启用该功能后，每次AWS SDK for Java向 AWS 发出服务请求时，都将生成指标数据点，按统计摘要排队，并大约每分钟向 CloudWatch 异步上传一次。指标一旦上传，您就可以使用 AWS 管理控制台将其可视化，并设置潜在问题的警报，如内存泄露、文件描述符泄露等等。

可用指标类型

默认指标组分为三大类：

AWS 请求指标

涵盖诸如 HTTP 请求/响应的延迟、请求数量、异常和重试等领域。

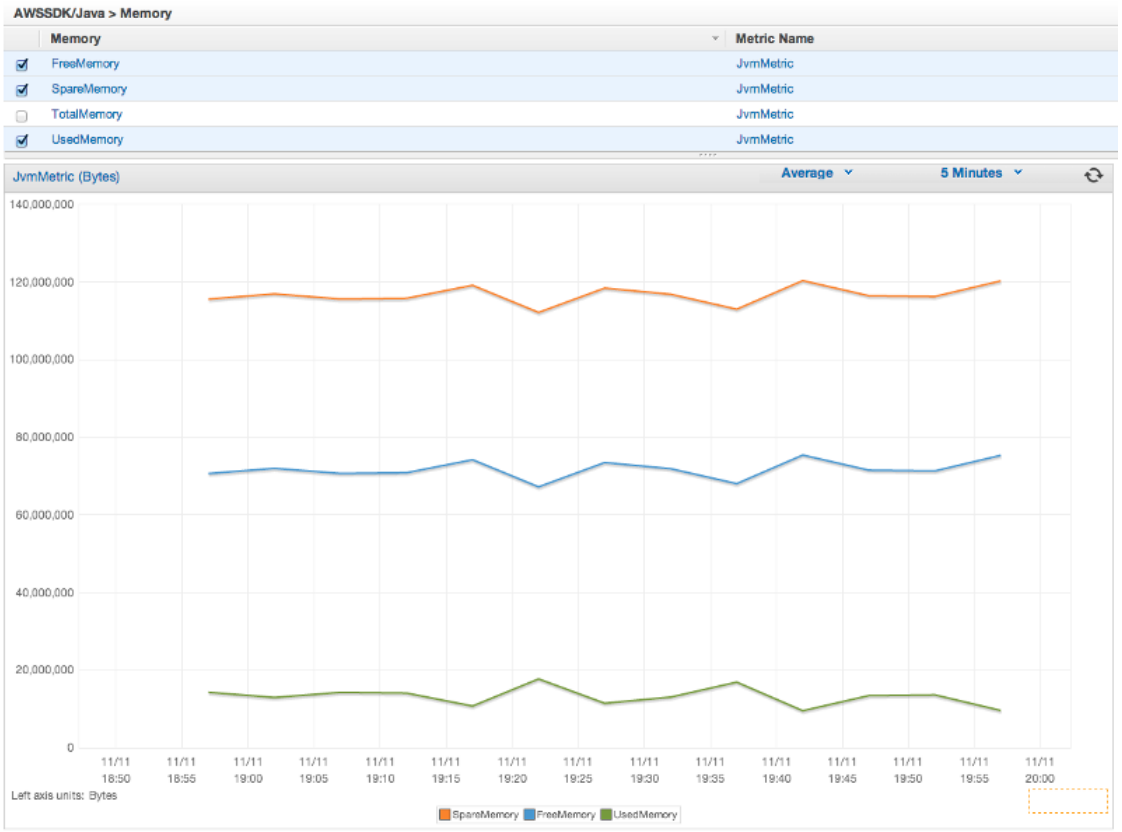


AWS 服务指标

包括特定于 AWS 服务的数据，如 S3 上传和下载的吞吐量和字节数。

机器指标

涵盖运行时环境，包括堆内存、线程数和打开的文件描述符。



如果您想要排除机器指标，请在系统属性中添加 `excludeMachineMetrics`：

```
-Dcom.amazonaws.sdk.enableDefaultMetrics=credentialFile=/path/
aws.properties,excludeMachineMetrics
```

更多信息

- 有关预定义核心指标类型的完整列表，请参阅 amazonaws/metrics package summary。
- 要了解有关使用AWS SDK for Java操作 CloudWatch 的信息，请参阅使用适用于 Java 的 AWS 开发工具包的 CloudWatch 示例 (p. 40)。
- 要了解更多有关性能优化的信息，请参阅优化适用于 Java 的 AWS 开发工具包以提高弹性博客文章。

AWS SDK for Java代码示例

本节提供使用 AWS SDK for Java v1 进行 AWS 服务编程的教程和示例。

您可以在 GitHub 上的 AWS 文档代码示例库中找到这些示例及其他示例的源代码。

要向 AWS 文档团队提请考虑生成新的代码示例，请创建新的请求。该团队正在寻求生成涵盖更多应用场景和使用情形的代码示例，而不仅仅是涵盖个别 API 调用的简单代码片段。有关说明，请参阅 GitHub 上的 Readme 文件中的 Proposing new code examples (提议新代码示例) 一节。

适用于 Java 的 AWS 开发工具包 2.x

在 2018 年，AWS 发布了AWS SDK for Java v2。要查看更多 AWS 示例，请参阅适用于 Java 的 AWS 开发工具包 2.x 开发人员指南。

> Note
>
> 如需可供AWS SDK for Java开发人员使用的更多示例和其他资源，请参阅其他文档和资源 (p. 1)！

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。

> Note
>
> 该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 AmazonCloudWatchClient 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 com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.ListMetricsRequest;
import com.amazonaws.services.cloudwatch.model.ListMetricsResult;
import com.amazonaws.services.cloudwatch.model.Metric;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

ListMetricsRequest request = new ListMetricsRequest()
        .withMetricName(name)
        .withNamespace(namespace);

boolean done = false;

while(!done) {
    ListMetricsResult response = cw.listMetrics(request);

    for(Metric metric : response.getMetrics()) {
        System.out.printf(
            "Retrieved metric %s", metric.getMetricName());
    }

    request.setNextToken(response.getNextToken());

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

调用 `getMetrics` 方法可在 ListMetricsResult 中返回指标。结果可以分页。要检索下一批结果，请在原始请求对象中使用 `ListMetricsResult` 对象的 `getNextToken` 方法的返回值调用 `setNextToken`，并将已修改的请求对象传回对 `listMetrics` 的另一个调用。

更多信息

- Amazon CloudWatch API Reference 中的 ListMetrics。

发布自定义指标数据

许多 AWS 服务在以“AWS/”开头的命名空间中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 AmazonCloudWatchClient 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 com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.Dimension;
import com.amazonaws.services.cloudwatch.model.MetricDatum;
import com.amazonaws.services.cloudwatch.model.PutMetricDataRequest;
import com.amazonaws.services.cloudwatch.model.PutMetricDataResult;
import com.amazonaws.services.cloudwatch.model.StandardUnit;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

Dimension dimension = new Dimension()
    .withName("UNIQUE_PAGES")
    .withValue("URLS");

MetricDatum datum = new MetricDatum()
    .withMetricName("PAGES_VISITED")
    .withUnit(StandardUnit.None)
    .withValue(data_point)
    .withDimensions(dimension);

PutMetricDataRequest request = new PutMetricDataRequest()
    .withNamespace("SITE/TRAFFIC")
    .withMetricData(datum);

PutMetricDataResult response = cw.putMetricData(request);
```

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。

与 CloudWatch 警报

创建警报

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

导入

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.ComparisonOperator;
import com.amazonaws.services.cloudwatch.model.Dimension;
import com.amazonaws.services.cloudwatch.model.PutMetricAlarmRequest;
```

```
import com.amazonaws.services.cloudwatch.model.PutMetricAlarmResult;
import com.amazonaws.services.cloudwatch.model.StandardUnit;
import com.amazonaws.services.cloudwatch.model.Statistic;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

Dimension dimension = new Dimension()
    .withName("InstanceId")
    .withValue(instanceId);

PutMetricAlarmRequest request = new PutMetricAlarmRequest()
    .withAlarmName(alarmName)
    .withComparisonOperator(
        ComparisonOperator.GreaterThanThreshold)
    .withEvaluationPeriods(1)
    .withMetricName("CPUUtilization")
    .withNamespace("AWS/EC2")
    .withPeriod(60)
    .withStatistic(Statistic.Average)
    .withThreshold(70.0)
    .withActionsEnabled(false)
    .withAlarmDescription(
        "Alarm when server CPU utilization exceeds 70%")
    .withUnit(StandardUnit.Seconds)
    .withDimensions(dimension);

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

列出警报

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

导入

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.DescribeAlarmsRequest;
import com.amazonaws.services.cloudwatch.model.DescribeAlarmsResult;
import com.amazonaws.services.cloudwatch.model.MetricAlarm;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

boolean done = false;
DescribeAlarmsRequest request = new DescribeAlarmsRequest();

while(!done) {

    DescribeAlarmsResult response = cw.describeAlarms(request);

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

    request.setNextToken(response.getNextToken());
```

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

警报列表可以通过在 `describeAlarms` 返回的 DescribeAlarmsResult 中调用 `getMetricAlarms` 获得。

结果可以分页。要检索下一批结果，请在原始请求对象中使用 `DescribeAlarmsResult` 对象的 `getNextToken` 方法的返回值调用 `setNextToken`，并将已修改的请求对象传回对 `describeAlarms` 的另一个调用。

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

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 AmazonCloudWatchClient 的 `deleteAlarms` 方法。

导入

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.DeleteAlarmsRequest;
import com.amazonaws.services.cloudwatch.model.DeleteAlarmsResult;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

DeleteAlarmsRequest request = new DeleteAlarmsRequest()
    .withAlarmNames(alarm_name);

DeleteAlarmsResult response = cw.deleteAlarms(request);
```

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**
>
> 通过在创建警报时使用 `setAlarmActions`PutMetricAlarmRequest 的 (p. 42) 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 AmazonCloudWatchClient 的 `enableAlarmActions`。

导入

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.EnableAlarmActionsRequest;
import com.amazonaws.services.cloudwatch.model.EnableAlarmActionsResult;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

EnableAlarmActionsRequest request = new EnableAlarmActionsRequest()
    .withAlarmNames(alarm);

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

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 AmazonCloudWatchClient 的 `disableAlarmActions`。

导入

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.DisableAlarmActionsRequest;
import com.amazonaws.services.cloudwatch.model.DisableAlarmActionsResult;
```

Code

```
final AmazonCloudWatch cw =
    AmazonCloudWatchClientBuilder.defaultClient();

DisableAlarmActionsRequest request = new DisableAlarmActionsRequest()
    .withAlarmNames(alarmName);

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

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 AmazonCloudWatchEventsClientPutEventsRequestEntry`putEvents` 对象 (提供每个事件的详细信息) 的 PutEventsRequest 对象调用 的 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 com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEvents;
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEventsClientBuilder;
import com.amazonaws.services.cloudwatchevents.model.PutEventsRequest;
import com.amazonaws.services.cloudwatchevents.model.PutEventsRequestEntry;
import com.amazonaws.services.cloudwatchevents.model.PutEventsResult;
```

Code

```
final AmazonCloudWatchEvents cwe =
    AmazonCloudWatchEventsClientBuilder.defaultClient();

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

PutEventsRequestEntry request_entry = new PutEventsRequestEntry()
    .withDetail(EVENT_DETAILS)
    .withDetailType("sampleSubmitted")
    .withResources(resource_arn)
    .withSource("aws-sdk-java-cloudwatch-example");

PutEventsRequest request = new PutEventsRequest()
    .withEntries(request_entry);

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

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 AmazonCloudWatchEventsClient 的 `putRule` 方法，可选参数有事件模式、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 com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEvents;
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEventsClientBuilder;
import com.amazonaws.services.cloudwatchevents.model.PutRuleRequest;
import com.amazonaws.services.cloudwatchevents.model.PutRuleResult;
import com.amazonaws.services.cloudwatchevents.model.RuleState;
```

Code

```
final AmazonCloudWatchEvents cwe =
```

```
    AmazonCloudWatchEventsClientBuilder.defaultClient();

PutRuleRequest request = new PutRuleRequest()
    .withName(rule_name)
    .withRoleArn(role_arn)
    .withScheduleExpression("rate(5 minutes)")
    .withState(RuleState.ENABLED);

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

添加目标

目标是触发规则时调用的资源。示例目标包括 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 AmazonCloudWatchEventsClient 的 `putTargets` 方法。

导入

```
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEvents;
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEventsClientBuilder;
import com.amazonaws.services.cloudwatchevents.model.PutTargetsRequest;
import com.amazonaws.services.cloudwatchevents.model.PutTargetsResult;
import com.amazonaws.services.cloudwatchevents.model.Target;
```

Code

```
final AmazonCloudWatchEvents cwe =
    AmazonCloudWatchEventsClientBuilder.defaultClient();

Target target = new Target()
    .withArn(function_arn)
    .withId(target_id);

PutTargetsRequest request = new PutTargetsRequest()
    .withTargets(target)
    .withRule(rule_name);

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

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

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

Note

该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

处理表格 DynamoDB

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）。

  每个关键值都有相关的 数据类型，由 ScalarAttribuTetype 类。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。
- 预置吞吐量值，这些值定义为表保留的读取/写入容量单位数。

  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 DynamoDB 客户端的 `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。仅在创建 DynamoDB 表时定义键表属性。

Note

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

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.CreateTableResult;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.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。

Code

```
CreateTableRequest request = new CreateTableRequest()
    .withAttributeDefinitions(new AttributeDefinition(
            "Name", ScalarAttributeType.S))
    .withKeySchema(new KeySchemaElement("Name", KeyType.HASH))
    .withProvisionedThroughput(new ProvisionedThroughput(
            new Long(10), new Long(10)))
    .withTableName(table_name);

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    CreateTableResult result = ddb.createTable(request);
    System.out.println(result.getTableDescription().getTableName());
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

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。

Code

```
CreateTableRequest request = new CreateTableRequest()
    .withAttributeDefinitions(
          new AttributeDefinition("Language", ScalarAttributeType.S),
          new AttributeDefinition("Greeting", ScalarAttributeType.S))
    .withKeySchema(
          new KeySchemaElement("Language", KeyType.HASH),
          new KeySchemaElement("Greeting", KeyType.RANGE))
    .withProvisionedThroughput(
          new ProvisionedThroughput(new Long(10), new Long(10)))
    .withTableName(table_name);
```

请参阅 GitHub 上的完整示例。

列出表

您可以通过调用 DynamoDB 客户端 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 com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.ListTablesRequest;
import com.amazonaws.services.dynamodbv2.model.ListTablesResult;
```

Code

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

ListTablesRequest request;

boolean more_tables = true;
String last_name = null;

while(more_tables) {
    try {
        if (last_name == null) {
         request = new ListTablesRequest().withLimit(10);
        }
        else {
         request = new ListTablesRequest()
           .withLimit(10)
           .withExclusiveStartTableName(last_name);
        }

        ListTablesResult table_list = ddb.listTables(request);
        List<String> table_names = table_list.getTableNames();

        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 = table_list.getLastEvaluatedTableName();
        if (last_name == null) {
            more_tables = false;
        }
```

默认情况下，每次调用将返回最多 100 个表 — 对返回的 ListTablesResult 对象使用 `getLastEvaluatedTableName` 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 DynamoDB 客户端的 `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 com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughputDescription;
import com.amazonaws.services.dynamodbv2.model.TableDescription;
```

Code

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
```

```
try {
    TableDescription table_info =
       ddb.describeTable(table_name).getTable();

    if (table_info != null) {
        System.out.format("Table name  : %s\n",
              table_info.getTableName());
        System.out.format("Table ARN   : %s\n",
              table_info.getTableArn());
        System.out.format("Status      : %s\n",
              table_info.getTableStatus());
        System.out.format("Item count  : %d\n",
              table_info.getItemCount().longValue());
        System.out.format("Size (bytes): %d\n",
              table_info.getTableSizeBytes().longValue());

        ProvisionedThroughputDescription throughput_info =
           table_info.getProvisionedThroughput();
        System.out.println("Throughput");
        System.out.format("  Read Capacity : %d\n",
              throughput_info.getReadCapacityUnits().longValue());
        System.out.format("  Write Capacity: %d\n",
              throughput_info.getWriteCapacityUnits().longValue());

        List<AttributeDefinition> attributes =
           table_info.getAttributeDefinitions();
        System.out.println("Attributes");
        for (AttributeDefinition a : attributes) {
            System.out.format("  %s (%s)\n",
                  a.getAttributeName(), a.getAttributeType());
        }
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

请参阅 GitHub 上的完整示例。

修改（更新）表

您可以通过调用 DynamoDB 客户端的 `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 com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.AmazonServiceException;
```

Code

```
ProvisionedThroughput table_throughput = new ProvisionedThroughput(
      read_capacity, write_capacity);

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
```

```
try {
    ddb.updateTable(table_name, table_throughput);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

请参阅 GitHub 上的完整示例。

删除表

调用 DynamoDB 客户端的 `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 com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
```

Code

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    ddb.deleteTable(table_name);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon DynamoDB Developer Guide 中的表处理指南
- Amazon DynamoDB Developer Guide 中的在 DynamoDB 中使用表

处理项目 DynamoDB

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 AmazonDynamoDB 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。它返回 GetItemResult 对象。

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

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import java.util.HashMap;
import java.util.Map;
```

Code

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

key_to_get.put("DATABASE_NAME", new AttributeValue(name));

GetItemRequest request = null;
if (projection_expression != null) {
    request = new GetItemRequest()
        .withKey(key_to_get)
        .withTableName(table_name)
        .withProjectionExpression(projection_expression);
} else {
    request = new GetItemRequest()
        .withKey(key_to_get)
        .withTableName(table_name);
}

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    Map<String,AttributeValue> returned_item =
       ddb.getItem(request).getItem();
    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 (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    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的映射。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 com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
```

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.ResourceNotFoundException;
import java.util.ArrayList;
```

Code

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

item_values.put("Name", new AttributeValue(name));

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

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    ddb.putItem(table_name, item_values);
} 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 (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
```

请参阅 GitHub 上的完整示例。

更新表中现有项目

可以使用 AmazonDynamoDB 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 com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeAction;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.AttributeValueUpdate;
import com.amazonaws.services.dynamodbv2.model.ResourceNotFoundException;
import java.util.ArrayList;
```

Code

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

item_key.put("Name", new AttributeValue(name));
```

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

for (String[] field : extra_fields) {
    updated_values.put(field[0], new AttributeValueUpdate(
                new AttributeValue(field[1]), AttributeAction.PUT));
}

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    ddb.updateItem(table_name, item_key, updated_values);
} catch (ResourceNotFoundException e) {
    System.err.println(e.getMessage());
    System.exit(1);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
```

请参阅 GitHub 上的完整示例。

使用 DynamoDBMapper 类

适用于 Java 的 AWS 开发工具包提供了一个 DynamoDBMapper 类，允许您将客户端类映射到 Amazon DynamoDB 表。要使用 DynamoBMapper 类，您可以使用注释定义 DynamoDB 表中的项目与代码中相应的对象实例之间的关系（如下面的代码示例所示）。利用 DynamoDBMapper 类，您能够访问自己的表，执行各种创建、读取、更新和删除 (CRUD) 操作，并执行查询。

> **Note**
>
> DynamoDBMapper 类不允许创建、更新或删除表。

导入

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBRangeKey;
import com.amazonaws.services.dynamodbv2.model.AmazonDynamoDBException;
```

Code

以下Java代码示例显示如何将内容添加到 音乐 表格 Dynamodbmapper 类。将内容添加到表中后，请注意使用 Partition (分区) 和 Sort (排序) 键加载项目。然后 Awards (奖项) 项目会更新。有关创建 Music 表的信息，请参阅 Amazon DynamoDB Developer Guide 中的创建表。

```
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
        MusicItems items = new MusicItems();

        try{
            // Add new content to the Music table
            items.setArtist(artist);
            items.setSongTitle(songTitle);
            items.setAlbumTitle(albumTitle);
            items.setAwards(Integer.parseInt(awards)); //convert to an int

            // Save the item
            DynamoDBMapper mapper = new DynamoDBMapper(client);
```

```
            mapper.save(items);

            // Load an item based on the Partition Key and Sort Key
            // Both values need to be passed to the mapper.load method
            String artistName = artist;
            String songQueryTitle = songTitle;

            // Retrieve the item
            MusicItems itemRetrieved = mapper.load(MusicItems.class, artistName,
songQueryTitle);
            System.out.println("Item retrieved:");
            System.out.println(itemRetrieved);

            // Modify the Award value
            itemRetrieved.setAwards(2);
            mapper.save(itemRetrieved);
            System.out.println("Item updated:");
            System.out.println(itemRetrieved);

            System.out.print("Done");
        } catch (AmazonDynamoDBException e) {
            e.getStackTrace();
        }
    }

    @DynamoDBTable(tableName="Music")
    public static class MusicItems {

        //Set up Data Members that correspond to columns in the Music table
        private String artist;
        private String songTitle;
        private String albumTitle;
        private int awards;

        @DynamoDBHashKey(attributeName="Artist")
        public String getArtist() {
            return this.artist;
        }

        public void setArtist(String artist) {
            this.artist = artist;
        }

        @DynamoDBRangeKey(attributeName="SongTitle")
        public String getSongTitle() {
            return this.songTitle;
        }

        public void setSongTitle(String title) {
            this.songTitle = title;
        }

        @DynamoDBAttribute(attributeName="AlbumTitle")
        public String getAlbumTitle() {
            return this.albumTitle;
        }

        public void setAlbumTitle(String title) {
            this.albumTitle = title;
        }

        @DynamoDBAttribute(attributeName="Awards")
        public int getAwards() {
            return this.awards;
        }
```

```
        public void setAwards(int awards) {
            this.awards = awards;
        }
    }
```

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 的 Amazon EC2示例

此部分提供使用AWS SDK for Java对 Amazon EC2 进行编程的示例。

主题

教程 启动 EC2 实例

本教程演示如何使用适用于 Java 的 AWS 开发工具包启动 EC2 实例。

主题

Prerequisites

在您开始之前，请确保已创建 AWS 账户并且已设置 AWS 凭证。有关更多信息，请参阅入门 (p. 3)。

创建 Amazon EC2 安全组

创建一个安全组作为虚拟防火墙，控制一个或多个 EC2 实例的网络流量。默认情况下，Amazon EC2 将您的实例与不允许入站流量的安全组关联。可以创建允许您的 EC2 实例接受特定流量的安全组。例如，如果需要连接到 Linux 实例，就必须将安全组配置为允许 SSH 流量。您可以使用 Amazon EC2 控制台或AWS SDK for Java创建安全组。

您可以创建在 EC2-Classic 或 EC2-VPC 中使用的安全组。有关 EC2-Classic 和 EC2-VPC 的更多信息，请参阅 Amazon EC2 User Guide for Linux Instances中的支持的平台。

有关使用 Amazon EC2 控制台创建安全组的更多信息，请参阅 Amazon EC2 User Guide for Linux Instances 中的 Amazon EC2 安全组。

创建安全组

1. 创建和初始化 CreateSecurityGroupRequest 实例。使用 withGroupName 方法设置安全组名称，使用 withDescription 方法设置安全组的描述，如下所示：

```
CreateSecurityGroupRequest csgr = new CreateSecurityGroupRequest();
csgr.withGroupName("JavaSecurityGroup").withDescription("My security group");
```

   在您初始化 Amazon EC2 客户端的 AWS 区域内，安全组名称必须是唯一的。必须为安全组的名称和描述使用 US-ASCII 字符。
2. 将请求对象作为参数传递给 createSecurityGroup 方法。该方法返回 CreateSecurityGroupResult 对象，如下所示：

```
CreateSecurityGroupResult createSecurityGroupResult =
    amazonEC2Client.createSecurityGroup(csgr);
```

   如果您尝试创建与现有安全组具有相同名称的安全组，`createSecurityGroup` 引发异常。

默认情况下，新安全组不允许 Amazon EC2 实例的任何入站流量。要允许入站流量，您必须对安全组传入明确地授权。您可以对单个 IP 地址、IP 地址范围、特定协议以及 TCP/UDP 端口的传入进行授权。

对安全组传入进行授权

1. 创建并初始化 IpPermission 实例。使用 withIpv4Ranges 方法可以设置授权传入的 IP 地址范围，使用 withIpProtocol 方法可以设置 IP 协议。使用 withFromPort 和 withToPort 方法可以指定授权传入的端口范围，如下所示：

```
IpPermission ipPermission =
    new IpPermission();

IpRange ipRange1 = new IpRange().withCidrIp("111.111.111.111/32");
IpRange ipRange2 = new IpRange().withCidrIp("150.150.150.150/32");

ipPermission.withIpv4Ranges(Arrays.asList(new IpRange[] {ipRange1, ipRange2}))
            .withIpProtocol("tcp")
            .withFromPort(22)
            .withToPort(22);
```

   必须满足在 `IpPermission` 对象中指定的所有条件，才能允许传入。

   使用 CIDR 表示法指定 IP 地址。如果指定 TCP/UDP 协议，必须提供源端口和目标端口。仅在指定 TCP 或 UDP 时才能授权端口。
2. 创建和初始化 AuthorizeSecurityGroupIngressRequest 实例。使用 `withGroupName` 方法指定安全组名称，并将之前初始化的 `IpPermission` 对象传递给 withIpPermissions 方法，如下所示：

```
AuthorizeSecurityGroupIngressRequest authorizeSecurityGroupIngressRequest =
    new AuthorizeSecurityGroupIngressRequest();

authorizeSecurityGroupIngressRequest.withGroupName("JavaSecurityGroup")
                                    .withIpPermissions(ipPermission);
```

3. 将请求对象传递给 authorizeSecurityGroupIngress 方法，如下所示：

```
amazonEC2Client.authorizeSecurityGroupIngress(authorizeSecurityGroupIngressRequest);
```

如果您使用已授权传入的 IP 地址调用 `authorizeSecurityGroupIngress`，该方法引发异常。创建和初始化新的 `IpPermission` 对象，对不同 IP、端口和协议授权传入，然后调用 `AuthorizeSecurityGroupIngress`。

只要调用 authorizeSecurityGroupIngress 或 authorizeSecurityGroupEgress 方法，一条规则就会添加到安全组中。

创建密钥对

启动 EC2 实例时必须指定密钥对，然后在连接到实例时指定密钥对的私有密钥。您可以创建密钥对，也可以使用在启动其他实例时使用的现有密钥对。有关更多信息，请参阅Amazon EC2 User Guide for Linux Instances中的 Amazon EC2 密钥对。

创建密钥对和保存私有密钥

1. 创建并启动 CreateKeyPairRequest 实例。使用 withKeyName 方法设置密钥对名称，如下所示：

```
CreateKeyPairRequest createKeyPairRequest = new CreateKeyPairRequest();

createKeyPairRequest.withKeyName(keyName);
```

> **Important**
>
> 密钥对名称必须是唯一的。如果您尝试创建的密钥对名称与现有密钥对相同，将引发异常。

2. 将请求对象传送到 createKeyPair 方法。该方法返回 CreateKeyPairResult 实例，如下所示：

```
CreateKeyPairResult createKeyPairResult =
  amazonEC2Client.createKeyPair(createKeyPairRequest);
```

3. 调用结果对象的 getKeyPair 方法，以获取 KeyPair 对象。调用 `KeyPair` 对象的 getKeyMaterial 方法，以获取未加密的 PEM 编码私有密钥，如下所示：

```
KeyPair keyPair = new KeyPair();

keyPair = createKeyPairResult.getKeyPair();

String privateKey = keyPair.getKeyMaterial();
```

运行 Amazon EC2 实例

使用以下过程从同一个 Amazon 系统映像 (AMI) 启动一个或多个具有相同配置的 EC2 实例。创建 EC2 实例后，您可以检查其状态。在您的 EC2 实例运行后，您可以连接这些实例。

启动 Amazon EC2 实例

1. 创建并初始化一个 RunInstancesRequest 实例。确保您指定的 AMI、密钥对和安全组在您创建客户端对象时指定的区域中存在。

```
RunInstancesRequest runInstancesRequest =
   new RunInstancesRequest();
```

```
runInstancesRequest.withImageId("ami-a9d09ed1")
                   .withInstanceType(InstanceType.T1Micro)
                   .withMinCount(1)
                   .withMaxCount(1)
                   .withKeyName("my-key-pair")
                   .withSecurityGroups("my-security-group");
```

withImageId

AMI 的 ID。要了解如何查找 Amazon 提供的公用 AMI 或创建您自己的 AMI，请参阅 Amazon 系统映像 (AMI)。

withInstanceType

与指定的 AMI 兼容的实例类型。有关更多信息，请参阅 Amazon EC2 User Guide for Linux Instances中的实例类型。

withMinCount

要启动的 EC2 实例的最小数量。如果此数量大于 Amazon EC2 可在目标可用区中启动的实例数，则 Amazon EC2 不会启动任何实例。

withMaxCount

要启动的 EC2 实例的最大数量。如果这种情况比例更多 Amazon EC2 可以在目标可用性区域启动， Amazon EC2 启动上述最大可能的实例 `MinCount`...您可以在1之间启动实例类型的最大实例数。有关更多信息，请参阅 Amazon EC2 一般常见问题中的我可以在 Amazon EC2 中运行多少个实例。

withKeyName

EC2 密钥对的名称。如果您在未指定密钥对的情况下启动实例，则无法连接到该实例。有关更多信息，请参阅创建密钥对 (p. 59)。

withSecurityGroups

一个或多个安全组. 有关更多信息，请参阅创建 Amazon EC2 安全组 (p. 57)。

2. 通过将请求对象传递到 runInstances 方法来启动实例。此方法返回一个 RunInstancesResult 对象，如下所示：

```
RunInstancesResult result = amazonEC2Client.runInstances(
                              runInstancesRequest);
```

在您的实例运行后，可使用您的密钥对连接到该实例。有关更多信息，请参阅 Amazon EC2 User Guide for Linux Instances 中的连接到您的 Linux 实例。

使用 IAM 授权访问AWS资源的角色 Amazon EC2

必须使用 AWS 颁发的凭证对发送到 Amazon Web Services (AWS) 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。

本主题介绍如何使用 IAM 角色操作 Amazon EC2 上运行的 Java 开发工具包应用程序。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 EC2 实例配置文件

如果您的应用程序使用默认构造函数创建 AWS 客户端，则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. 在Java系统属性中: `aws.accessKeyId` 和 `aws.secretKey`.

2. 在系统环境变量中: `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. 如果已设置 `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` 环境变量且安全管理器有权访问该变量，则为通过 Amazon EC2 容器服务传递的凭证。
5. 在实例配置文件凭证中，，它存在于与 EC2 实例的 IAM 角色关联的实例元数据中。
6. 来自环境或容器的 Web 身份令牌凭证。

只有在对 Amazon EC2 实例运行应用程序时，默认提供程序链中的 instance profile credentials (实例配置文件证书) 步骤才可用，但在处理 Amazon EC2 实例时，该步骤能够最大限度地简化使用过程并提高安全性。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：。

```
AmazonS3 s3 = AmazonS3ClientBuilder.standard()
              .withCredentials(new InstanceProfileCredentialsProvider(false))
              .build();
```

在使用该方法时，开发工具包在其实例配置文件中，检索与 Amazon EC2 实例的关联 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。

> **Important**
>
> 仅 在以下情况下执行自动凭证刷新：您使用默认客户端构造函数 (它会创建其自身的 `InstanceProfileCredentialsProvider` 作为默认提供程序链的内容) 时；或者您将 `InstanceProfileCredentialsProvider` 实例直接传递给客户端构造函数时。如果您使用其他方法获取或传送实例配置文件凭证，您将负责检查和刷新过期凭证。

如果客户端构造函数使用凭证提供程序链找不到凭证，它会引发 AmazonClientException。

演示 使用EC2实例的IAM角色

以下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 和 Create New Role。
3. 输入角色名称，然后选择下一步。请记住此名称，因为在启动 Amazon EC2 实例时会用到它。
4. 在 Select Role Type 页面上，在 AWS Service Roles 下选择 Amazon EC2。
5. 在 Set Permissions 页面上，在 Select Policy Template 下选择 Amazon S3 Read Only Access，然后选择 Next Step。
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 控制台或AWS SDK for Java，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。

到达核查实例启动页面时，选择编辑实例详细信息。在 IAM 角色中，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。

- 要使用 IAM 角色通过AWS SDK for Java启动 Amazon EC2 实例，请参阅运行 Amazon EC2 实例 (p. 59)。

创建您的应用程序

让我们来构建在 EC2 实例上运行的示例应用程序。首先，创建一个目录来用于保存教程文件 (例如，`GetS3ObjectApp`)。

然后，将 AWS SDK for Java 库复制到新创建的目录中。如果已将AWS SDK for Java下载到 `~/Downloads` 目录中，可以使用以下命令进行复制：

```
cp -r ~/Downloads/aws-java-sdk-{1.7.5}/lib .
cp -r ~/Downloads/aws-java-sdk-{1.7.5}/third-party .
```

打开一个新文件，将其命名为 `GetS3Object.java` 并添加以下代码：

```
import java.io.*;

import com.amazonaws.auth.*;
import com.amazonaws.services.s3.*;
import com.amazonaws.services.s3.model.*;
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;

public class GetS3Object {
  private static String bucketName = "text-content";
  private static String key = "text-object.txt";

  public static void main(String[] args) throws IOException
  {
    AmazonS3 s3Client = AmazonS3ClientBuilder.defaultClient();

    try {
      System.out.println("Downloading an object");
      S3Object s3object = s3Client.getObject(
          new GetObjectRequest(bucketName, key));
      displayTextInputStream(s3object.getObjectContent());
    }
    catch(AmazonServiceException ase) {
      System.err.println("Exception was thrown by the service");
    }
    catch(AmazonClientException ace) {
      System.err.println("Exception was thrown by the client");
    }
  }

  private static void displayTextInputStream(InputStream input) throws IOException
  {
    // Read one text line at a time and display.
    BufferedReader reader = new BufferedReader(new InputStreamReader(input));
    while(true)
    {
      String line = reader.readLine();
      if(line == null) break;
```

```
      System.out.println( "    " + line );
    }
    System.out.println();
  }
}
```

打开一个新文件，将其命名为 `build.xml` 并添加以下行：

```
<project name="Get Amazon S3 Object" default="run" basedir=".">
  <path id="aws.java.sdk.classpath">
    <fileset dir="./lib" includes="**/*.jar"/>
    <fileset dir="./third-party" includes="**/*.jar"/>
    <pathelement location="lib"/>
    <pathelement location="."/>
  </path>

  <target name="build">
  <javac debug="true"
    includeantruntime="false"
    srcdir="."
    destdir="."
    classpathref="aws.java.sdk.classpath"/>
  </target>

  <target name="run" depends="build">
    <java classname="GetS3Object" classpathref="aws.java.sdk.classpath" fork="true"/>
  </target>
</project>
```

构建并运行修改后的程序。请注意，该程序中未存储凭证。因此，除非您已经指定了AWS凭据，否则 `AmazonServiceException`。例如:

```
$ ant
Buildfile: /path/to/my/GetS3ObjectApp/build.xml

build:
  [javac] Compiling 1 source file to /path/to/my/GetS3ObjectApp

run:
   [java] Downloading an object
   [java] AmazonServiceException

BUILD SUCCESSFUL
```

传输已编译的程序到您的 EC2 实例

使用安全复制 (**scp**)，将程序连同AWS SDK for Java库传输到 Amazon EC2 实例。该命令序列与以下序列相似。

```
scp -p -i {my-key-pair}.pem GetS3Object.class ec2-user@{public_dns}:GetS3Object.class
scp -p -i {my-key-pair}.pem build.xml ec2-user@{public_dns}:build.xml
scp -r -p -i {my-key-pair}.pem lib ec2-user@{public_dns}:lib
scp -r -p -i {my-key-pair}.pem third-party ec2-user@{public_dns}:third-party
```

> **Note**
>
> 根据您使用的 Linux 版本，用户名 可能是“ec2-user”、“root”或“ubuntu”。要获取实例的公共DNS名称，请打开 EC2控制台 并寻找 公共DNS 在 描述 选项卡（例如，`ec2-198-51-100-1.compute-1.amazonaws.com`）。

在上述命令中：

- `GetS3Object.class` 是已编译的程序
- `build.xml` 是用于构建和运行您的程序的 Ant 文件
- `lib` 和 `third-party` 目录是AWS SDK for Java中对应的库文件夹。
- `-r` 开关指示 `scp` 应该对AWS SDK for Java版本的 `library` 和 `third-party` 目录中的所有内容以递归方式进行复制。
- `-p` 开关指示 `scp` 在将源文件复制到目标位置时，应保留对应文件的权限。

  Note

  `-p` 开关仅适用于 Linux, OS X, or Unix。如果您从 Windows 中复制文件，可能需要使用以下命令在实例上修复文件权限：

  ```
  chmod -R u+rwx GetS3Object.class build.xml lib third-party
  ```

在 EC2 实例上运行示例程序

要运行程序，请连接到 Amazon EC2 实例。有关更多信息，请参阅 Amazon EC2 User Guide for Linux Instances 中的连接到您的 Linux 实例。

如果 **`ant`** 在您的实例上不可用，请使用以下命令安装它：

```
sudo yum install ant
```

然后使用 `ant` 运行程序，如下所示：

```
ant run
```

该程序会将 Amazon S3 对象的内容写入命令窗口。

教程：Amazon EC2Spot 实例

Overview

与按需实例价格相比，通过 Spot 实例，您可以对未使用的 Amazon Elastic Compute Cloud (Amazon EC2) 容量进行出价（最高达 90%），并在出价高于当前 Spot 价格 时运行您购买的实例。根据供应和需求情况，Amazon EC2 会定期更改 Spot 价格；出价达到或超过 Spot 价格的客户可获得可用的 Spot 实例。就像按需实例和预留实例， Spot 实例为您提供了另一种获得更多计算能力的选择。

Spot 实例可以大幅降低您用于批量处理、科学研究、图像处理、视频编码、数据和 Web 检索、财务分析和测试的 Amazon EC2 成本。除此之外，在不急需容量的情况下， Spot 实例还能让您获得大量的附加容量。

如要使用 Spot 实例，您就需要置入一个 Spot 实例请求，以便指定您愿意支付的每个实例每小时的最高价格；这就是您的竞价。如果您的最高出价超出当前的 Spot 价格，则会满足您的请求，您的实例将会运行，直到您选择终止它们或 Spot 价格增长到高于您的最高价格（以先到者为准）。

请务必记住：

- 您每小时支付的价格通常低于您的出价。随着请求的接收和现有供应的变化，Amazon EC2 会定期调整 Spot 价格。在该期间内，无论每个人的最高出价是否更高，它们支付的 Spot 价格都是相同的。因此，您的支付要低于您的出价，但永远不会支付超过您的出价。

- 如果您正在运行 Spot 实例，而您的出价不再达到或高于当前的 Spot 价格，则您的实例将会终止。这意味着，您要确保工作负载和应用程序足够灵活，以便利用这一机会性的容量。

运行时，Spot 实例的操作方式与其他 Amazon EC2 实例完全相同，而且同其他 Amazon EC2 实例一样，当您不再需要 Spot 实例时可以终止它们。如果终止了实例，您需要为不满一小时的时间付费（与按需或预留实例相同）。不过，如果 Spot 价格超出您的最高价格，且 Amazon EC2 终止了您的实例，则您无需对任何不满一小时的使用时间付费。

本教程介绍如何使用 AWS SDK for Java 执行以下操作。

- 提交一个 Spot 请求
- 判定何时执行该 Spot 请求
- 取消该 Spot 请求
- 终止相关实例

Prerequisites

要使用此指南，您必须已安装 AWS SDK for Java 并且已满足其基本安装先决条件。有关更多信息，请参阅设置适用于 Java 的 AWS 开发工具包 (p. 4)。

步骤 1. 设置您的 凭证

要开始使用此代码示例，您需要设置 AWS 凭证。具体操作说明，请参阅设置用于开发的 AWS 凭证和区域 (p. 6)。

> Note
>
> 建议您使用 IAM 用户凭证来提供这些值。有关更多信息，请参阅注册 AWS 并创建 IAM 用户 (p. 3)。

您既然已配置好了您的设置，现在就可以使用示例中的代码开始了。

步骤 2. 设置安全组

一个安全组可作为一个控制流量进入和流出实例组的防火墙。默认情况下，实例开始运行时没有配置任何安全组，这就意味着，从任何 TCP 端口传入的 IP 流量都将被拒绝。因此，在提交 Spot 请求前，我们会设置一个安全组，以允许必要的网络流量传入。出于本教程的目的，我们将创建一个名为“GettingStarted”的新安全组，以允许从您正在运行的应用程序的 IP 地址传入 Secure Shell (SSH) 流量。要设置一个新的安全组，需要包含或运行下列通过编程的方式来设置安全组的代码示例。

创建 `AmazonEC2` 客户端对象之后，我们会创建一个名为“GettingStarted”的`CreateSecurityGroupRequest` 对象以及对安全组的描述。接下来，我们将调用`ec2.createSecurityGroup` API 来创建安全组。

为访问安全组，我们将使用一个 IP 地址范围（设置为本地计算机子网的 CIDR 表示）创建一个 `ipPermission` 对象，IP 地址的后缀“/10”指明了该指定 IP 地址的子网。我们还为 `ipPermission` 数据元配置了 TCP 协议和端口 22 (SSH)。最后一步是使用我们的安全组名称和 `ec2.authorizeSecurityGroupIngress` 数据元来调用 `ipPermission`。

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Create a new security group.
try {
```

```
    CreateSecurityGroupRequest securityGroupRequest = new
 CreateSecurityGroupRequest("GettingStartedGroup", "Getting Started Security Group");
    ec2.createSecurityGroup(securityGroupRequest);
} catch (AmazonServiceException ase) {
    // Likely this means that the group is already created, so ignore.
    System.out.println(ase.getMessage());
}

String ipAddr = "0.0.0.0/0";

// Get the IP of the current host, so that we can limit the Security
// Group by default to the ip range associated with your subnet.
try {
    InetAddress addr = InetAddress.getLocalHost();

    // Get IP Address
    ipAddr = addr.getHostAddress()+"/10";
} catch (UnknownHostException e) {
}

// Create a range that you would like to populate.
ArrayList<String> ipRanges = new ArrayList<String>();
ipRanges.add(ipAddr);

// Open up port 22 for TCP traffic to the associated IP
// from above (e.g. ssh traffic).
ArrayList<IpPermission> ipPermissions = new ArrayList<IpPermission> ();
IpPermission ipPermission = new IpPermission();
ipPermission.setIpProtocol("tcp");
ipPermission.setFromPort(new Integer(22));
ipPermission.setToPort(new Integer(22));
ipPermission.setIpRanges(ipRanges);
ipPermissions.add(ipPermission);

try {
    // Authorize the ports to the used.
    AuthorizeSecurityGroupIngressRequest ingressRequest =
        new AuthorizeSecurityGroupIngressRequest("GettingStartedGroup",ipPermissions);
    ec2.authorizeSecurityGroupIngress(ingressRequest);
} catch (AmazonServiceException ase) {
    // Ignore because this likely means the zone has
    // already been authorized.
    System.out.println(ase.getMessage());
}
```

请注意，要创建一个新的安全组，您只需要运行一次此应用程序。

您还可以使用 AWS Toolkit for Eclipse 创建安全组。有关更多信息，请参阅通过 AWS Explorer 管理安全组。

步骤 3 提交您的 Spot 请求

为了提交一个 Spot 请求，您首先需要确定该实例类型，Amazon 系统映像 (AMI)，和您要使用的最高出价。还须包括我们先前配置好的安全组，这样一来，如果需要的话，您就可以登录到该实例中了。

有几个实例类型可供选择；请转到 Amazon EC2 实例类型获取完整列表。在本教程中，我们将使用最便宜的实例类型 t1.micro。下一步是确定我们想用的 AMI 类型。在本教程中，我们使用的是最新版的 Amazon Linux AMI，即 ami-a9d09ed1。最新的 AMI 可能会随时间而改变，但您始终可以通过执行以下步骤来确定最新版的 AMI：

1. 打开 Amazon EC2 控制台。
2. 选择 Launch Instance (启动实例) 按钮。

3. 第一个窗口将显示可用的 AMI。每个 AMI 标题旁边都列出了 AMI ID。或者，您也可以使用 `DescribeImages` API，但该命令的使用不在本教程的范围之内。

有很多方法可以竞价 Spot 实例，如要大致了解各种方法，您应当观看对 Spot 实例出价视频。然而，为了入门，我们将介绍三种常见的策略：确保成本低于按需定价的竞价；基于所得计算值的竞价；以便尽可能快地获取计算能力的竞价。

- 降低成本至低于按需实例您需要进行花费数小时或数天的批处理工作。然而，您可以灵活调整启动和完成时间。您希望看到是否以较低的成本完成了按需实例。您可以通过使用 AWS 管理控制台或 Amazon EC2 API 来检查各类型实例的 Spot 价格记录。如需更多信息，请转到来查看 Spot 价格记录。在您分析了给定可用区内所需实例类型的价格记录之后，您有两种可供选择的方法进行竞价：
  - 您可以在现货价格范围（这仍然低于按需定价）的上端竞价，预测您单次现货请求很有可能会达成，并运行足够的连续计算时间来完成此项工作。
  - 或者，您可以通过按需实例价格的百分比形式，指定您愿意为 Spot 实例支付的金额，并计划将持久请求期间启动的许多实例结合起来。如果超过指定价格，则 Spot 实例将终止。（在本教程之后我们会介绍如何自动运行该任务。）
- 支付不超过该结果的值您需要进行数据处理工作。您将会对该工作的结果有一个很好的了解，以便于能够让您知道在计算成本方面它们的价值。当您分析了实例类型的 Spot 价格记录之后，选择一个计算时间成本不高于该工作结果成本的竞价。由于 Spot 价格的波动，该价格可能会达到或低于您的竞价，所以您要创建一个持久出价，并允许它间歇运行。
- 快速获取计算容量您对附加容量有一个无法预料的短期需求，该容量不能通过按需实例获取。当您分析了实例类型的 Spot 价格记录之后，您出价高于历史最高价格，以便提供一个高的能很快执行实例的可能性，并继续计算，直到完成实例。

在选择竞价之后，您可以请求一个 Spot 实例。考虑到本教程的目的，我们将以按需定价来出价 (0.03 US)，以便能最大化执行出价的机率。您可以通过进入 Amazon EC2 定价页面来确定可用实例的类型和这些实例的按需价格。当 Spot 实例在运行时，您将支付实例运行期间生效的 Spot 价格。Spot 实例的价格由 Amazon EC2 设置，并根据 Spot 实例容量的长期供求趋势逐步调整。您还可以指定您愿意为 Spot 实例支付的金额作为按需实例价格的百分比。要请求 Spot 实例，您只需使用先前选择的参数来构建请求。首先，我们创建一个`RequestSpotInstanceRequest`数据元。数据元的请求需要要启动的实例数量及其竞价。此外，您还需要设置`LaunchSpecification`该请求，其中包括实例类型、AMI ID，和要使用的安全组。在填写好该请求后，您可以调用该数据元上的`requestSpotInstances`方法`AmazonEC2Client`。以下示例演示了如何请求一个 Spot 实例。

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Setup the specifications of the launch. This includes the
// instance type (e.g. t1.micro) and the latest Amazon Linux
// AMI id available. Note, you should always use the latest
// Amazon Linux AMI id or another of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-a9d09ed1");
launchSpecification.setInstanceType(InstanceType.T1Micro);

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specifications to the request.
```

```
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(requestRequest);
```

此代码的运行将启动一个新的 Spot 实例请求。还有其他可用来配置 Spot 请求的选择。如需了解更多信息，请访问 教程: AdvancedAmazonEC2现货请求管理 (p. 71) 或 请求情况 分类 AWS SDK for Java API Reference.

Note

您需为任何已启动的 Spot 实例付费，因此，请确保您取消了任何请求并终止了任何已启动的实例，以便减少所有相关费用。

步骤 4. 确定您 Spot 请求的状态

下一步是，要一直等到在进行最后一步之前、现货请求达到“活跃”状态时再创建代码。为了确定 Spot 请求的状态，我们轮询了 describeSpotInstanceRequests方法来确定要监视的 Spot 请求 ID 的状态。

第 2 步中创建的请求 ID 内嵌在该`requestSpotInstances`请求响应中。以下示例代码显示了如何从`requestSpotInstances`响应中收集请求 ID 和如何用它们填写一个`ArrayList`。

```
// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(requestRequest);
List<SpotInstanceRequest> requestResponses = requestResult.getSpotInstanceRequests();

// Setup an arraylist to collect all of the request ids we want to
// watch hit the running state.
ArrayList<String> spotInstanceRequestIds = new ArrayList<String>();

// Add all of the request ids to the hashset, so we can determine when they hit the
// active state.
for (SpotInstanceRequest requestResponse : requestResponses) {
    System.out.println("Created Spot Request:
 "+requestResponse.getSpotInstanceRequestId());
    spotInstanceRequestIds.add(requestResponse.getSpotInstanceRequestId());
}
```

为哦了监控您的请求 ID，请调用`describeSpotInstanceRequests`方法来确定该请求的状态。然后循环，直到该请求不处于“打开”的状态。请注意，我们监控的是“打开”这一状态，而不是“活跃”状态，因为如果请求参数有问题，该请求可以直接“关闭”。以下代码示例提供了如何完成此项任务的详细信息。

```
// Create a variable that will track whether there are any
// requests still in the open state.
boolean anyOpen;

do {
    // Create the describeRequest object with all of the request ids
    // to monitor (e.g. that we started).
    DescribeSpotInstanceRequestsRequest describeRequest = new
 DescribeSpotInstanceRequestsRequest();
    describeRequest.setSpotInstanceRequestIds(spotInstanceRequestIds);

    // Initialize the anyOpen variable to false - which assumes there
    // are no requests open unless we find one that is still open.
    anyOpen=false;

    try {
        // Retrieve all of the requests we want to monitor.
        DescribeSpotInstanceRequestsResult describeResult =
 ec2.describeSpotInstanceRequests(describeRequest);
```

```
        List<SpotInstanceRequest> describeResponses =
 describeResult.getSpotInstanceRequests();

        // Look through each request and determine if they are all in
        // the active state.
        for (SpotInstanceRequest describeResponse : describeResponses) {
            // If the state is open, it hasn't changed since we attempted
            // to request it. There is the potential for it to transition
            // almost immediately to closed or cancelled so we compare
            // against open instead of active.
        if (describeResponse.getState().equals("open")) {
            anyOpen = true;
            break;
        }
    }
} catch (AmazonServiceException e) {
      // If we have an exception, ensure we don't break out of
      // the loop. This prevents the scenario where there was
      // blip on the wire.
      anyOpen = true;
    }

    try {
        // Sleep for 60 seconds.
        Thread.sleep(60*1000);
    } catch (Exception e) {
        // Do nothing because it woke up early.
    }
} while (anyOpen);
```

运行此代码后， Spot 实例请求会完成或失败，如果失败，将输出一个错误提示到屏幕上。在任一情况下，我们都可以进行下一步，以便清理任何已活跃请求并终止任何正在运行的实例。

步骤 5. 清理您的 Spot 请求和实例

最后，我们需要清理请求和实例。重要的是，要取消所有未完成的请求并终止所有实例。只取消请求不会终止您的实例，这意味着您需要继续为它们支付费用。如果您终止了实例，那么 Spot 请求可能会被取消，但在某些情况下，例如，如果您使用的是持久出价，那么终止实例则不足以阻止请求重新执行。因此，最好的做法是取消所有已活跃出价并终止所有正在运行的实例。

以下代码演示了如何取消您的请求。

```
try {
    // Cancel requests.
    CancelSpotInstanceRequestsRequest cancelRequest =
       new CancelSpotInstanceRequestsRequest(spotInstanceRequestIds);
    ec2.cancelSpotInstanceRequests(cancelRequest);
} catch (AmazonServiceException e) {
    // Write out any exceptions that may have occurred.
    System.out.println("Error cancelling instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

要终止所有挂起的实例，您需要实例 ID 和启动它们的请求。以下代码示例采用了原代码来监控这些实例，并增加了一个存储这些实例 ID 和相关联的`ArrayList`响应的`describeInstance`。

```
// Create a variable that will track whether there are any requests
// still in the open state.
boolean anyOpen;
```

```
// Initialize variables.
ArrayList<String> instanceIds = new ArrayList<String>();

do {
   // Create the describeRequest with all of the request ids to
   // monitor (e.g. that we started).
   DescribeSpotInstanceRequestsRequest describeRequest = new
 DescribeSpotInstanceRequestsRequest();
   describeRequest.setSpotInstanceRequestIds(spotInstanceRequestIds);

   // Initialize the anyOpen variable to false, which assumes there
   // are no requests open unless we find one that is still open.
   anyOpen = false;

   try {
         // Retrieve all of the requests we want to monitor.
         DescribeSpotInstanceRequestsResult describeResult =
            ec2.describeSpotInstanceRequests(describeRequest);

         List<SpotInstanceRequest> describeResponses =
            describeResult.getSpotInstanceRequests();

         // Look through each request and determine if they are all
         // in the active state.
         for (SpotInstanceRequest describeResponse : describeResponses) {
           // If the state is open, it hasn't changed since we
           // attempted to request it. There is the potential for
           // it to transition almost immediately to closed or
           // cancelled so we compare against open instead of active.
           if (describeResponse.getState().equals("open")) {
              anyOpen = true; break;
           }
           // Add the instance id to the list we will
           // eventually terminate.
           instanceIds.add(describeResponse.getInstanceId());
         }
   } catch (AmazonServiceException e) {
      // If we have an exception, ensure we don't break out
      // of the loop. This prevents the scenario where there
      // was blip on the wire.
      anyOpen = true;
   }

    try {
        // Sleep for 60 seconds.
        Thread.sleep(60*1000);
    } catch (Exception e) {
        // Do nothing because it woke up early.
    }
} while (anyOpen);
```

使用存储在`ArrayList`中的实例 ID，通过使用以下代码片段来终止任何正在运行的实例。

```
try {
    // Terminate instances.
    TerminateInstancesRequest terminateRequest = new
 TerminateInstancesRequest(instanceIds);
    ec2.terminateInstances(terminateRequest);
} catch (AmazonServiceException e) {
    // Write out any exceptions that may have occurred.
    System.out.println("Error terminating instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
```

```
}
```

综述

为了将所有内容组合在一起，我们提供了一个更加面向数据元的方法，该方法结合了上文所示步骤：初始化 EC2 客户端，提交 Spot 请求，确定何时 Spot 请求不再处于开放状态，并清理所有延迟的 Spot 请求和相关实例。我们建立一个执行这些操作的类别，命名为`Requests`。

我们还创建了一个 `GettingStartedApp` 类，为我们执行高级函数调用提供主要方法。具体地，我们对之前所述的数据元`Requests`进行初始化。提交 Spot 实例请求。然后等待 Spot 请求达到“有效”状态。最后，清理这些请求和实例。

可在 GitHub 查看和下载此示例的完整源代码。

恭喜您！您已经完成了用 AWS SDK for Java 开发 Spot 实例软件的入门教程。

后续步骤

继续 教程: AdvancedAmazonEC2现货请求管理 (p. 71).

教程 高级 Amazon EC2 现货请求管理

Amazon EC2 Spot 实例允许您对未使用的 Amazon EC2 容量出价，并在出价高于当前 Spot 价格 的期间运行此类实例。Amazon EC2 基于供给和需求定期更改 Spot 价格。有关 Spot 实例的更多信息，请参阅 Amazon EC2 User Guide for Linux Instances中的 Spot 实例。

Prerequisites

要使用此指南，您必须已安装 AWS SDK for Java 并且已满足其基本安装先决条件。有关更多信息，请参阅设置适用于 Java 的 AWS 开发工具包 (p. 4)。

设置您的凭证

要开始使用此代码示例，您需要设置 AWS 凭证。具体操作说明，请参阅设置用于开发的 AWS 凭证和区域 (p. 6)。

> Note
>
> 建议您使用 IAM 用户凭证来提供这些值。有关更多信息，请参阅注册 AWS 并创建 IAM 用户 (p. 3)。

您既然已配置好了您的设置，现在就可以使用示例中的代码开始了。

设置安全组

一个安全组可作为一个控制流量进入和流出实例组的防火墙。默认情况下，实例开始运行时没有配置任何安全组，这就意味着，从任何 TCP 端口传入的 IP 流量都将被拒绝。因此，在提交 Spot 请求前，我们会设置一个安全组，以允许必要的网络流量传入。出于本教程的目的，我们将创建一个名为“GettingStarted”的新安全组，以允许从您正在运行的应用程序的 IP 地址传入 Secure Shell (SSH) 流量。要设置一个新的安全组，需要包含或运行下列通过编程的方式来设置安全组的代码示例。

创建 `AmazonEC2` 客户端对象之后，我们会创建一个名为“GettingStarted”的`CreateSecurityGroupRequest` 对象以及对安全组的描述。接下来，我们将调用`ec2.createSecurityGroup` API 来创建安全组。

为访问安全组，我们将使用一个 IP 地址范围（设置为本地计算机子网的 CIDR 表示）创建一个 `ipPermission` 对象，IP 地址的后缀“/10”指明了该指定 IP 地址的子网。我们还为 `ipPermission` 数据元配置了 TCP 协议和端口 22 (SSH)。最后一步是使用我们的安全组名称和 `ec2.authorizeSecurityGroupIngress` 数据元来调用 `ipPermission`。

（以下代码与我们在第一个教程中使用的代码相同。）

```
// Create the AmazonEC2Client object so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.standard()
                    .withCredentials(credentials)
                    .build();

// Create a new security group.
try {
    CreateSecurityGroupRequest securityGroupRequest =
        new CreateSecurityGroupRequest("GettingStartedGroup",
        "Getting Started Security Group");
    ec2.createSecurityGroup(securityGroupRequest);
} catch (AmazonServiceException ase) {
    // Likely this means that the group is already created, so ignore.
    System.out.println(ase.getMessage());
}

String ipAddr = "0.0.0.0/0";

// Get the IP of the current host, so that we can limit the Security Group
// by default to the ip range associated with your subnet.
try {
    // Get IP Address
    InetAddress addr = InetAddress.getLocalHost();
    ipAddr = addr.getHostAddress()+"/10";
}
catch (UnknownHostException e) {
    // Fail here...
}

// Create a range that you would like to populate.
ArrayList<String> ipRanges = new ArrayList<String>();
ipRanges.add(ipAddr);

// Open up port 22 for TCP traffic to the associated IP from
// above (e.g. ssh traffic).
ArrayList<IpPermission> ipPermissions = new ArrayList<IpPermission> ();
IpPermission ipPermission = new IpPermission();
ipPermission.setIpProtocol("tcp");
ipPermission.setFromPort(new Integer(22));
ipPermission.setToPort(new Integer(22));
ipPermission.setIpRanges(ipRanges);
ipPermissions.add(ipPermission);

try {
    // Authorize the ports to the used.
    AuthorizeSecurityGroupIngressRequest ingressRequest =
        new AuthorizeSecurityGroupIngressRequest(
            "GettingStartedGroup",ipPermissions);
    ec2.authorizeSecurityGroupIngress(ingressRequest);
}
catch (AmazonServiceException ase) {
    // Ignore because this likely means the zone has already
    // been authorized.
    System.out.println(ase.getMessage());
}
```

您可以在 `advanced.CreateSecurityGroupApp.java` 代码示例中查看整个代码示例。请注意，要创建一个新的安全组，您只需要运行一次此应用程序。

> **Note**
>
> 您还可以使用 AWS Toolkit for Eclipse 创建安全组。有关更多信息，请参阅 AWS Toolkit for Eclipse User Guide 中的通过 AWS Explorer 管理安全组。

详细 Spot 实例请求创建选项

正如我们在 教程: AmazonEC2Spot实例 (p. 64)，您需要使用实例类型、亚马逊机器映像(AMI)和最大投标价格来创建请求。

让我们从创建 `RequestSpotInstanceRequest` 对象开始。请求数据元需要您所需的实例数量和竞标价格。此外，我们需要为请求设置 `LaunchSpecification`，包括实例类型、AMI ID 和您需要使用的安全组。填入请求后，我们将在 `requestSpotInstances` 数据元上调用 `AmazonEC2Client` 方法。以下是如何申请 Spot 实例的一个示例。

（以下代码与我们在第一个教程中使用的代码相同。）

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set up the specifications of the launch. This includes the
// instance type (e.g. t1.micro) and the latest Amazon Linux
// AMI id available. Note, you should always use the latest
// Amazon Linux AMI id or another of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-a9d09ed1");
launchSpecification.setInstanceType(InstanceType.T1Micro);

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

持久性请求和一次性请求

建立一个 Spot 请求时，您可以指定几个可选参数。首先是您的请求是一次性的还是持久性的。默认情况下，一般是一次性请求。一次性请求可以只执行一次，请求实例终止后，请求将被关闭。同一请求中没有 Spot 实例运行的任何时候，持久性请求都被视为已完成。要指定请求的类型，您只需要设置 Spot 请求的类型。您可以使用以下代码完成设置。

```
// Retrieves the credentials from an AWSCredentials.properties file.
AWSCredentials credentials = null;
try {
    credentials = new PropertiesCredentials(
        GettingStartedApp.class.getResourceAsStream("AwsCredentials.properties"));
}
catch (IOException e1) {
    System.out.println(
        "Credentials were not properly entered into AwsCredentials.properties.");
    System.out.println(e1.getMessage());
    System.exit(-1);
}
```

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest =
    new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set the type of the bid to persistent.
requestRequest.setType("persistent");

// Set up the specifications of the launch. This includes the
// instance type (e.g. t1.micro) and the latest Amazon Linux
// AMI id available. Note, you should always use the latest
// Amazon Linux AMI id or another of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-a9d09ed1");
launchSpecification.setInstanceType(InstanceType.T1Micro);

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

限制请求的持续时间

您还可以有选择地指定您的请求持续有效的时长。您可以指定有效期开始和结束的时间。默认情况下，从创建那一刻开始，系统将默认执行 Spot 请求，直到该请求完成或被取消。然而，如果您有需要，您可以限制有效期。以下代码显示了如何指定有效期的示例。

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set the valid start time to be two minutes from now.
Calendar cal = Calendar.getInstance();
cal.add(Calendar.MINUTE, 2);
requestRequest.setValidFrom(cal.getTime());

// Set the valid end time to be two minutes and two hours from now.
cal.add(Calendar.HOUR, 2);
requestRequest.setValidUntil(cal.getTime());

// Set up the specifications of the launch. This includes
// the instance type (e.g. t1.micro)

// and the latest Amazon Linux AMI id available.
// Note, you should always use the latest Amazon
```

```
// Linux AMI id or another of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-a9d09ed1");
launchSpecification.setInstanceType("t1.micro");

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(requestRequest);
```

将您的 Amazon EC2 Spot 实例请求分组

您可以选择几种不同方法为您的 Spot 实例请求分组。我们来看看使用启动组、可用区组和置放组的好处。

如果您想要确保您的 Spot 实例全部一起启动和终止，您可以选择利用启动组。启动组是将一系列竞价分在一组的标签。启动组内的所有实例都一起启动和终止。请注意，如果启动组内的实例已经完成了，不能保证同一启动组新启动的实例也随之完成。以下代码示例显示了如何设置启动组的示例。

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 5 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(5));

// Set the launch group.
requestRequest.setLaunchGroup("ADVANCED-DEMO-LAUNCH-GROUP");

// Set up the specifications of the launch. This includes
// the instance type (e.g. t1.micro) and the latest Amazon Linux
// AMI id available. Note, you should always use the latest
// Amazon Linux AMI id or another of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-a9d09ed1");
launchSpecification.setInstanceType(InstanceType.T1Micro);

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

如果您要确保请求中的所有实例在同一可用区内启动，而对具体哪个可用区并没有要求，您可以利用可用区组。可用区域组是将一系列同一可用区域内的实例分在一组的标签。共享同一可用区域组并同时完成的所有实例将在同一可用区域内开始运行。以下是如何设置可用区域组的一个示例。

```
// Create the AmazonEC2 client so we can call various APIs.
```

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 5 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(5));

// Set the availability zone group.
requestRequest.setAvailabilityZoneGroup("ADVANCED-DEMO-AZ-GROUP");

// Set up the specifications of the launch.  This includes the instance
// type (e.g.  t1.micro) and the latest Amazon Linux AMI id available.
// Note, you should always use the latest Amazon Linux AMI id or another
// of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-a9d09ed1");
launchSpecification.setInstanceType(InstanceType.T1Micro);

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

您可以为您的 Spot 实例指定一个可用区域。以下代码示例为您显示如何设置可用区域。

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set up the specifications of the launch. This includes the instance
// type (e.g. t1.micro) and the latest Amazon Linux AMI id available.
// Note, you should always use the latest Amazon Linux AMI id or another
// of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-a9d09ed1");
launchSpecification.setInstanceType(InstanceType.T1Micro);

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Set up the availability zone to use. Note we could retrieve the
// availability zones using the ec2.describeAvailabilityZones() API. For
// this demo we will just use us-east-1a.
SpotPlacement placement = new SpotPlacement("us-east-1b");
launchSpecification.setPlacement(placement);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);
```

```
// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

最后，如果您使用的是高性能计算 (HPC) Spot 实例（例如，集群计算实例或集群 GPU 实例），则您可以指定一个置放群组。置放组为您提供更低的延迟和实例之间的高带宽连接。以下是如何设置置放组的一个示例。

```
// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set up the specifications of the launch. This includes the instance
// type (e.g. t1.micro) and the latest Amazon Linux AMI id available.
// Note, you should always use the latest Amazon Linux AMI id or another
// of your choosing.

LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-a9d09ed1");
launchSpecification.setInstanceType(InstanceType.T1Micro);

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Set up the placement group to use with whatever name you desire.
// For this demo we will just use "ADVANCED-DEMO-PLACEMENT-GROUP".
SpotPlacement placement = new SpotPlacement();
placement.setGroupName("ADVANCED-DEMO-PLACEMENT-GROUP");
launchSpecification.setPlacement(placement);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

本节所显示的所有参数都是可选的。还必须意识到，这些参数中的大多数（您的竞价是一次性还是永久性这个参数除外）都能降低实现出价的可能性。因此，只有当您需要的时候才利用这些选择，这很重要。所有前面的代码示例被组合成一个长代码示例，可在 `com.amazonaws.codesamples.advanced.InlineGettingStartedCodeSampleApp.java` 类别中找到。

中断或终止发生后，如何持久保存一个根分区

管理 Spot 实例中断最简单的方法之一是确保定期将您的数据点校验到 Amazon Elastic Block Store (Amazon EBS) 卷。通过定期执行点校验，如果发生中断，您只会丢失上一次点校验后创建的数据（假设中间没有执行其他非幂等操作）。为了使这个过程变得更容易，您可以配置您的 Spot 请求，以确保中断或终止发生时您的根分区不会被删除。在以下如何实现此方案的示例中，我们插入了新代码。

在添加的代码中，我们创建了一个 `BlockDeviceMapping` 对象，并设置与其相关联的 Elastic Block Storage (EBS) 到 EBS 对象，之前我们已对此 EBS 对象进行配置，如果 Spot 实例被终止，此对象不会 (`not`) 随之删除。然后，我们将此 `BlockDeviceMapping` 添加到启动说明中所包含的映射的 ArrayList。

```
// Retrieves the credentials from an AWSCredentials.properties file.
AWSCredentials credentials = null;
try {
    credentials = new PropertiesCredentials(
        GettingStartedApp.class.getResourceAsStream("AwsCredentials.properties"));
}
catch (IOException e1) {
    System.out.println(
        "Credentials were not properly entered into AwsCredentials.properties.");
    System.out.println(e1.getMessage());
    System.exit(-1);
}

// Create the AmazonEC2 client so we can call various APIs.
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

// Initializes a Spot Instance Request
RequestSpotInstancesRequest requestRequest = new RequestSpotInstancesRequest();

// Request 1 x t1.micro instance with a bid price of $0.03.
requestRequest.setSpotPrice("0.03");
requestRequest.setInstanceCount(Integer.valueOf(1));

// Set up the specifications of the launch. This includes the instance
// type (e.g. t1.micro) and the latest Amazon Linux AMI id available.
// Note, you should always use the latest Amazon Linux AMI id or another
// of your choosing.
LaunchSpecification launchSpecification = new LaunchSpecification();
launchSpecification.setImageId("ami-a9d09ed1");
launchSpecification.setInstanceType(InstanceType.T1Micro);

// Add the security group to the request.
ArrayList<String> securityGroups = new ArrayList<String>();
securityGroups.add("GettingStartedGroup");
launchSpecification.setSecurityGroups(securityGroups);

// Create the block device mapping to describe the root partition.
BlockDeviceMapping blockDeviceMapping = new BlockDeviceMapping();
blockDeviceMapping.setDeviceName("/dev/sda1");

// Set the delete on termination flag to false.
EbsBlockDevice ebs = new EbsBlockDevice();
ebs.setDeleteOnTermination(Boolean.FALSE);
blockDeviceMapping.setEbs(ebs);

// Add the block device mapping to the block list.
ArrayList<BlockDeviceMapping> blockList = new ArrayList<BlockDeviceMapping>();
blockList.add(blockDeviceMapping);

// Set the block device mapping configuration in the launch specifications.
launchSpecification.setBlockDeviceMappings(blockList);

// Add the launch specification.
requestRequest.setLaunchSpecification(launchSpecification);

// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult =
    ec2.requestSpotInstances(requestRequest);
```

如果您想在启动时重新连接该卷到您的实例中，也可以使用数据块存储设备映射设置。另外，如果您连接了一个非根分区，则可以指定您希望在 Spot 实例恢复后连接到该实例的 Amazon EBS 卷。要实现此功能，您只需指定 `EbsBlockDevice` 数据元中的快照 ID 和 `BlockDeviceMapping` 数据元中的替代设备名称。通过利用数据块存储设备映射，可以让引导实例变得更加容易。

使用根分区来对您的关键数据执行点校验是管理您的实例可能性中断的好方法。有关更多管理可能性中断的方法，请参阅“Managing Interruption”视频。

如何标记您的 Spot 请求和实例

为 EC2 资源添加标签能简化您的云基础设施管理。某种形式的元数据、标记可用于创建用户友好型名称、增强搜索能力，并改善多个用户之间的协作。您也可以使用标记来自动化脚本和部分进程。要阅读有关为 Amazon EC2 资源添加标签的更多信息，请转至 Amazon EC2 User Guide for Linux Instances 中的使用标签。

标记 请求

要为您的 Spot 请求添加标签，您需要在提交请求之后 为它们添加标签。来自 `requestSpotInstances()` 的返回值将为您提供一个 RequestSpotInstancesResult 对象，此对象可用于获取 Spot 请求 ID 以便添加标签：

```
// Call the RequestSpotInstance API.
RequestSpotInstancesResult requestResult = ec2.requestSpotInstances(requestRequest);
List<SpotInstanceRequest> requestResponses = requestResult.getSpotInstanceRequests();

// A list of request IDs to tag
ArrayList<String> spotInstanceRequestIds = new ArrayList<String>();

// Add the request ids to the hashset, so we can determine when they hit the
// active state.
for (SpotInstanceRequest requestResponse : requestResponses) {
    System.out.println("Created Spot Request:
 "+requestResponse.getSpotInstanceRequestId());
    spotInstanceRequestIds.add(requestResponse.getSpotInstanceRequestId());
}
```

在获得 ID 后，您可以通过将请求 ID 添加到 CreateTagsRequest 并调用 EC2 客户端的 `createTags()` 方法来为请求添加标签：

```
// The list of tags to create
ArrayList<Tag> requestTags = new ArrayList<Tag>();
requestTags.add(new Tag("keyname1","value1"));

// Create the tag request
CreateTagsRequest createTagsRequest_requests = new CreateTagsRequest();
createTagsRequest_requests.setResources(spotInstanceRequestIds);
createTagsRequest_requests.setTags(requestTags);

// Tag the spot request
try {
    ec2.createTags(createTagsRequest_requests);
}
catch (AmazonServiceException e) {
    System.out.println("Error terminating instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

标记实例

与 Spot 请求本身类似，您只能在创建实例后为其添加标签，此操作将在满足 Spot 请求后 (不再处于打开 状态) 立即执行。

您可以通过使用 `describeSpotInstanceRequests()`DescribeSpotInstanceRequestsRequest 对象调用 EC2 客户端的 方法来检查请求的状态。返回的 DescribeSpotInstanceRequestsResult 对象包含 SpotInstanceRequest 对象的列表，可使用这些对象查询 Spot 请求的状态并在其不再处于打开 状态后获取其实例 ID。

在 Spot 请求不在处于打开状态后，您可以通过调用其 `getInstanceId()` 方法从 `SpotInstanceRequest` 对象检索其实例 ID。

```
boolean anyOpen; // tracks whether any requests are still open

// a list of instances to tag.
ArrayList<String> instanceIds = new ArrayList<String>();

do {
    DescribeSpotInstanceRequestsRequest describeRequest =
        new DescribeSpotInstanceRequestsRequest();
    describeRequest.setSpotInstanceRequestIds(spotInstanceRequestIds);

    anyOpen=false; // assume no requests are still open

    try {
        // Get the requests to monitor
        DescribeSpotInstanceRequestsResult describeResult =
            ec2.describeSpotInstanceRequests(describeRequest);

        List<SpotInstanceRequest> describeResponses =
            describeResult.getSpotInstanceRequests();

        // are any requests open?
        for (SpotInstanceRequest describeResponse : describeResponses) {
                if (describeResponse.getState().equals("open")) {
                    anyOpen = true;
                    break;
                }
                // get the corresponding instance ID of the spot request
                instanceIds.add(describeResponse.getInstanceId());
        }
    }
    catch (AmazonServiceException e) {
        // Don't break the loop due to an exception (it may be a temporary issue)
        anyOpen = true;
    }

    try {
        Thread.sleep(60*1000); // sleep 60s.
    }
    catch (Exception e) {
        // Do nothing if the thread woke up early.
    }
} while (anyOpen);
```

现在，您可以为返回的实例添加标签：

```
// Create a list of tags to create
ArrayList<Tag> instanceTags = new ArrayList<Tag>();
instanceTags.add(new Tag("keyname1","value1"));

// Create the tag request
CreateTagsRequest createTagsRequest_instances = new CreateTagsRequest();
createTagsRequest_instances.setResources(instanceIds);
createTagsRequest_instances.setTags(instanceTags);

// Tag the instance
```

```
try {
    ec2.createTags(createTagsRequest_instances);
}
catch (AmazonServiceException e) {
    // Write out any exceptions that may have occurred.
    System.out.println("Error terminating instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

取消 Spot 请求并终止实例

取消 Spot 请求

要取消 Spot 实例请求，请使用 CancelSpotInstanceRequestsRequest 对象调用 EC2 客户端上的 `cancelSpotInstanceRequests`。

```
try {
    CancelSpotInstanceRequestsRequest cancelRequest = new
 CancelSpotInstanceRequestsRequest(spotInstanceRequestIds);
    ec2.cancelSpotInstanceRequests(cancelRequest);
} catch (AmazonServiceException e) {
    System.out.println("Error cancelling instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

终止 Spot 实例

您可以通过将任何正在运行的 Spot 实例的 ID 传递到 EC2 客户端的 `terminateInstances()` 方法来终止该实例。

```
try {
    TerminateInstancesRequest terminateRequest = new
 TerminateInstancesRequest(instanceIds);
    ec2.terminateInstances(terminateRequest);
} catch (AmazonServiceException e) {
    System.out.println("Error terminating instances");
    System.out.println("Caught Exception: " + e.getMessage());
    System.out.println("Reponse Status Code: " + e.getStatusCode());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Request ID: " + e.getRequestId());
}
```

综述

综述起来，我们提供一种更加以数据元为导向的方法，将此教程中所示步骤结合到一个易于使用的类别。我们将执行这些操作的一个被称为 `Requests` 的类别实例化。我们还创建了一个 `GettingStartedApp` 类，为我们执行高级函数调用提供主要方法。

可在 GitHub 查看和下载此示例的完整源代码。

恭喜您！您已经学完了“高级请求功能”教程，了解如何使用 AWS SDK for Java 开发 Spot 实例软件。

管理 Amazon EC2实例

创建 实例

要创建新 Amazon EC2 实例，请调用 AmazonEC2Client 的 `runInstances` 方法，并为它提供 RunInstancesRequest，其中包含要使用的 Amazon 系统映像 (AMI) 和一个实例类型。

导入

```
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.InstanceType;
import com.amazonaws.services.ec2.model.RunInstancesRequest;
import com.amazonaws.services.ec2.model.RunInstancesResult;
import com.amazonaws.services.ec2.model.Tag;
```

Code

```
RunInstancesRequest run_request = new RunInstancesRequest()
    .withImageId(ami_id)
    .withInstanceType(InstanceType.T1Micro)
    .withMaxCount(1)
    .withMinCount(1);

RunInstancesResult run_response = ec2.runInstances(run_request);

String reservation_id =
 run_response.getReservation().getInstances().get(0).getInstanceId();
```

请参阅完整示例。

启动 实例

要启动 Amazon EC2 实例，请调用 AmazonEC2Client 的 `startInstances` 方法，并为它提供 StartInstancesRequest，其中包含要启动实例的 ID。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.StartInstancesRequest;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

StartInstancesRequest request = new StartInstancesRequest()
    .withInstanceIds(instance_id);

ec2.startInstances(request);
```

请参阅完整示例。

停止 实例

要停止 Amazon EC2 实例，请调用 AmazonEC2Client 的 `stopInstances` 方法，并为它提供 StopInstancesRequest，其中包含要停止的实例的 ID。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.StopInstancesRequest;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

StopInstancesRequest request = new StopInstancesRequest()
    .withInstanceIds(instance_id);

ec2.stopInstances(request);
```

请参阅完整示例。

重启 实例

要重启 Amazon EC2 实例，请调用 AmazonEC2Client 的 `rebootInstances` 方法，并为它提供 RebootInstancesRequest，其中包含要重启实例的 ID。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.RebootInstancesRequest;
import com.amazonaws.services.ec2.model.RebootInstancesResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

RebootInstancesRequest request = new RebootInstancesRequest()
    .withInstanceIds(instance_id);

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

请参阅完整示例。

描述 实例

要列出您的实例，您需要创建 DescribeInstancesRequest 并调用 AmazonEC2Client 的 `describeInstances` 方法。它将返回 DescribeInstancesResult 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，您必须首先在每个返回的 Reservation 对象上调用 `DescribeInstancesResult` 类的 `getReservations' method, and then call :methodname:`getInstances`。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DescribeInstancesRequest;
import com.amazonaws.services.ec2.model.DescribeInstancesResult;
import com.amazonaws.services.ec2.model.Instance;
import com.amazonaws.services.ec2.model.Reservation;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();
boolean done = false;

DescribeInstancesRequest request = new DescribeInstancesRequest();
while(!done) {
    DescribeInstancesResult response = ec2.describeInstances(request);

    for(Reservation reservation : response.getReservations()) {
        for(Instance instance : reservation.getInstances()) {
            System.out.printf(
                "Found instance with id %s, " +
                "AMI %s, " +
                "type %s, " +
                "state %s " +
                "and monitoring state %s",
                instance.getInstanceId(),
                instance.getImageId(),
                instance.getInstanceType(),
                instance.getState().getName(),
                instance.getMonitoring().getState());
        }
    }

    request.setNextToken(response.getNextToken());

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

结果将分页；您可以获取更多结果，方式是：将从结果对象的 `getNextToken` 方法返回的值传递到您的原始请求对象的 `setNextToken` 方法，然后在下一个 `describeInstances` 调用中使用相同的请求对象。

请参阅完整示例。

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，您必须用要监控的实例的 ID 创建一个 MonitorInstancesRequest，并将其传递给 AmazonEC2Client 的 `monitorInstances` 方法。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.MonitorInstancesRequest;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

MonitorInstancesRequest request = new MonitorInstancesRequest()
        .withInstanceIds(instance_id);

ec2.monitorInstances(request);
```

请参阅完整示例。

停止实例监控

要停止监控实例，您必须用要停止监控的 实例的 ID 创建一个 UnmonitorInstancesRequest，并将其传递给 AmazonEC2Client 的 `unmonitorInstances` 方法。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.UnmonitorInstancesRequest;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

UnmonitorInstancesRequest request = new UnmonitorInstancesRequest()
    .withInstanceIds(instance_id);

ec2.unmonitorInstances(request);
```

请参阅完整示例。

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 MonitorInstances
- Amazon EC2 API Reference 中的 UnmonitorInstances

在 Amazon EC2 中使用弹性 IP 地址

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 AmazonEC2Client 的 `allocateAddress` 方法。

返回的 AllocateAddressResult 包含一个分配 ID，您可以用它来将地址与实例关联，在 AssociateAddressRequest 中将分配 ID 和实例 ID 传递给 AmazonEC2Client 的 `associateAddress` 方法。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.AllocateAddressRequest;
import com.amazonaws.services.ec2.model.AllocateAddressResult;
import com.amazonaws.services.ec2.model.AssociateAddressRequest;
import com.amazonaws.services.ec2.model.AssociateAddressResult;
import com.amazonaws.services.ec2.model.DomainType;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

AllocateAddressRequest allocate_request = new AllocateAddressRequest()
    .withDomain(DomainType.Vpc);

AllocateAddressResult allocate_response =
    ec2.allocateAddress(allocate_request);

String allocation_id = allocate_response.getAllocationId();

AssociateAddressRequest associate_request =
    new AssociateAddressRequest()
        .withInstanceId(instance_id)
        .withAllocationId(allocation_id);

AssociateAddressResult associate_response =
    ec2.associateAddress(associate_request);
```

请参阅完整示例。

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 AmazonEC2Client 的 `describeAddresses` 方法。它会返回 DescribeAddressesResult，您可以使用它来获取在账户中代表弹性 IP 地址的 Address 对象的列表。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.Address;
import com.amazonaws.services.ec2.model.DescribeAddressesResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

DescribeAddressesResult response = ec2.describeAddresses();

for(Address address : response.getAddresses()) {
    System.out.printf(
            "Found address with public IP %s, " +
            "domain %s, " +
            "allocation id %s " +
            "and NIC id %s",
            address.getPublicIp(),
            address.getDomain(),
            address.getAllocationId(),
            address.getNetworkInterfaceId());
}
```

请参阅完整示例。

释放弹性 IP 地址

要释放弹性 IP 地址，请调用 AmazonEC2Client 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 com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.ReleaseAddressRequest;
import com.amazonaws.services.ec2.model.ReleaseAddressResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

ReleaseAddressRequest request = new ReleaseAddressRequest()
    .withAllocationId(alloc_id);

ReleaseAddressResult 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。如果您尝试释放已释放的弹性 IP 地址，且该地址已分配到另一个 AWS 账户，您会收到 AuthFailure 错误。

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 AmazonEC2Client 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

使用区域和可用区域

描述区域

要列出您账户的可用区域，请调用 AmazonEC2Client 的 `describeRegions` 方法。该方法返回 DescribeRegionsResult。调用返回对象的 `getRegions` 方法，获取表示各个区域的 Region 对象的列表。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DescribeRegionsResult;
import com.amazonaws.services.ec2.model.Region;
import com.amazonaws.services.ec2.model.AvailabilityZone;
import com.amazonaws.services.ec2.model.DescribeAvailabilityZonesResult;
```

Code

```
DescribeRegionsResult regions_response = ec2.describeRegions();

for(Region region : regions_response.getRegions()) {
    System.out.printf(
        "Found region %s " +
```

```
        "with endpoint %s",
        region.getRegionName(),
        region.getEndpoint());
}
```

请参阅完整示例。

描述可用区

要列出您账户的每个可用区，请调用 AmazonEC2Client 的 `describeAvailabilityZones` 方法。该方法返回 DescribeAvailabilityZonesResult。调用其 `getAvailabilityZones` 方法，获取表示各个可用区的 AvailabilityZone 对象的列表。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DescribeRegionsResult;
import com.amazonaws.services.ec2.model.Region;
import com.amazonaws.services.ec2.model.AvailabilityZone;
import com.amazonaws.services.ec2.model.DescribeAvailabilityZonesResult;
```

Code

```
DescribeAvailabilityZonesResult zones_response =
    ec2.describeAvailabilityZones();

for(AvailabilityZone zone : zones_response.getAvailabilityZones()) {
    System.out.printf(
        "Found availability zone %s " +
        "with status %s " +
        "in region %s",
        zone.getZoneName(),
        zone.getState(),
        zone.getRegionName());
}
```

请参阅完整示例。

描述账户

要描述您的帐户，请调用 AmazonEC2Client 的 `describeAccountAttributes` 方法。此方法返回 DescribeAccountAttributesResult 对象。调用此对象的 `getAccountAttributes` 方法以获取 AccountAttribute 对象的列表。您可以遍历该列表来检索 AccountAttribute 对象。

您可以通过调用 AccountAttribute 对象的 `getAttributeValues` 方法来获取您账户的属性值。此方法返回 AccountAttributeValue 对象的列表。您可以遍历第二个列表来显示属性的值（请参阅以下代码示例）。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.AccountAttributeValue;
import com.amazonaws.services.ec2.model.DescribeAccountAttributesResult;
import com.amazonaws.services.ec2.model.AccountAttribute;
import java.util.List;
import java.util.ListIterator;
```

Code

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

try{
    DescribeAccountAttributesResult accountResults = ec2.describeAccountAttributes();
    List<AccountAttribute> accountList = accountResults.getAccountAttributes();

    for (ListIterator iter = accountList.listIterator(); iter.hasNext(); ) {

        AccountAttribute attribute = (AccountAttribute) iter.next();
        System.out.print("\n The name of the attribute is "+attribute.getAttributeName());
        List<AccountAttributeValue> values = attribute.getAttributeValues();

         //iterate through the attribute values
        for (ListIterator iterVals = values.listIterator(); iterVals.hasNext(); ) {
            AccountAttributeValue myValue = (AccountAttributeValue) iterVals.next();
            System.out.print("\n The value of the attribute is
 "+myValue.getAttributeValue());
        }
    }
    System.out.print("Done");
}
catch (Exception e)
{
    e.getStackTrace();
}
```

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

创建密钥对

要创建密钥对，请使用包含密钥名称的 CreateKeyPairRequest 调用 AmazonEC2Client 的 `createKeyPair` 方法。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.CreateKeyPairRequest;
import com.amazonaws.services.ec2.model.CreateKeyPairResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

CreateKeyPairRequest request = new CreateKeyPairRequest()
    .withKeyName(key_name);

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

请参阅完整示例。

描述密钥对

要列出您的密钥对或获取其相关信息，请调用 AmazonEC2Client 的 `describeKeyPairs` 方法。它返回 DescribeKeyPairsResult，您可以通过调用其 `getKeyPairs` 方法来访问密钥对的列表，该方法返回一个 KeyPairInfo 对象的列表。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DescribeKeyPairsResult;
import com.amazonaws.services.ec2.model.KeyPairInfo;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

DescribeKeyPairsResult response = ec2.describeKeyPairs();

for(KeyPairInfo key_pair : response.getKeyPairs()) {
    System.out.printf(
        "Found key pair with name %s " +
        "and fingerprint %s",
        key_pair.getKeyName(),
        key_pair.getKeyFingerprint());
}
```

请参阅完整示例。

删除密钥对

要删除密钥对，请调用 AmazonEC2Client 的 `deleteKeyPair` 方法，将其传递给一个包含要删除的密钥对名称的 DeleteKeyPairRequest。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DeleteKeyPairRequest;
import com.amazonaws.services.ec2.model.DeleteKeyPairResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

DeleteKeyPairRequest request = new DeleteKeyPairRequest()
    .withKeyName(key_name);

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

请参阅完整示例。

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

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 AmazonEC2Client 的 `createSecurityGroup` 方法。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.CreateSecurityGroupRequest;
import com.amazonaws.services.ec2.model.CreateSecurityGroupResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

CreateSecurityGroupRequest create_request = new
    CreateSecurityGroupRequest()
        .withGroupName(group_name)
        .withDescription(group_desc)
        .withVpcId(vpc_id);

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

请参阅完整示例。

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 AmazonEC2Client 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 com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.CreateSecurityGroupRequest;
import com.amazonaws.services.ec2.model.CreateSecurityGroupResult;
```

Code

```
IpRange ip_range = new IpRange()
    .withCidrIp("0.0.0.0/0");

IpPermission ip_perm = new IpPermission()
    .withIpProtocol("tcp")
    .withToPort(80)
```

```
    .withFromPort(80)
    .withIpv4Ranges(ip_range);

IpPermission ip_perm2 = new IpPermission()
    .withIpProtocol("tcp")
    .withToPort(22)
    .withFromPort(22)
    .withIpv4Ranges(ip_range);

AuthorizeSecurityGroupIngressRequest auth_request = new
    AuthorizeSecurityGroupIngressRequest()
        .withGroupName(group_name)
        .withIpPermissions(ip_perm, ip_perm2);

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

要向安全组添加出口规则，请在 AuthorizeSecurityGroupEgressRequest 中向 AmazonEC2Client 的 `authorizeSecurityGroupEgress` 方法提供相似的数据。

请参阅完整示例。

描述安全组

要描述您的安全组或获取相关信息，请调用 AmazonEC2Client 的 `describeSecurityGroups` 方法。它会返回 DescribeSecurityGroupsResult，您可以通过调用其 `getSecurityGroups` 方法来访问安全组的列表，该方法返回一个 SecurityGroupInfo 对象的列表。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DescribeSecurityGroupsRequest;
import com.amazonaws.services.ec2.model.DescribeSecurityGroupsResult;
```

Code

```

final String USAGE =
    "To run this example, supply a group id\n" +
    "Ex: DescribeSecurityGroups <group-id>\n";

if (args.length != 1) {
    System.out.println(USAGE);
    System.exit(1);
}

String group_id = args[0];
```

请参阅完整示例。

删除安全组

要删除安全组，请调用 AmazonEC2Client 的 `deleteSecurityGroup` 方法，将其传递给一个包含要删除的安全组 ID 的 DeleteSecurityGroupRequest。

导入

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
```

```
import com.amazonaws.services.ec2.model.DeleteSecurityGroupRequest;
import com.amazonaws.services.ec2.model.DeleteSecurityGroupResult;
```

Code

```
final AmazonEC2 ec2 = AmazonEC2ClientBuilder.defaultClient();

DeleteSecurityGroupRequest request = new DeleteSecurityGroupRequest()
    .withGroupId(group_id);

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

请参阅完整示例。

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

本节提供编程示例 iam 使用 适用于Java的AWSSDK.

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。

> **Note**
>
> 该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

创建访问密钥

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

`CreateAccessKeyRequest` 有两个构造函数，一个需要用户名，另一个不带参数。如果您使用不带参数的版本，则必须使用 `withUserName` setter 设置用户名，然后再将其传递给 `createAccessKey` 方法。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreateAccessKeyRequest;
import com.amazonaws.services.identitymanagement.model.CreateAccessKeyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

CreateAccessKeyRequest request = new CreateAccessKeyRequest()
    .withUserName(user);

CreateAccessKeyResult 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，其中包含要列出其密钥的用户名，并将该对象传递给 AmazonIdentityManagementClient 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 com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.AccessKeyMetadata;
import com.amazonaws.services.identitymanagement.model.ListAccessKeysRequest;
import com.amazonaws.services.identitymanagement.model.ListAccessKeysResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

boolean done = false;
ListAccessKeysRequest request = new ListAccessKeysRequest()
        .withUserName(username);

while (!done) {

    ListAccessKeysResult response = iam.listAccessKeys(request);

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

    request.setMarker(response.getMarker());

    if (!response.getIsTruncated()) {
        done = true;
```

```
    }
}
```

`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 ListAccessKeysResult 对象中的 `getIsTruncated`，以查看该查询返回的结果是否少于可用结果。如果是这样，则在 `ListAccessKeysRequest` 上调用 `setMarker` 并将其传递回 `listAccessKeys` 的后续调用。

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 ID（可使用 GetAccessKeyLastUsedRequest 对象传入，也可直接传给直接接收访问密钥 ID 的重载）调用 AmazonIdentityManagementClient 的 `getAccessKeyLastUsed` 方法。

然后，您可以使用返回的 GetAccessKeyLastUsedResult 对象检索上次使用密钥的时间。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.GetAccessKeyLastUsedRequest;
import com.amazonaws.services.identitymanagement.model.GetAccessKeyLastUsedResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

GetAccessKeyLastUsedRequest request = new GetAccessKeyLastUsedRequest()
    .withAccessKeyId(access_id);

GetAccessKeyLastUsedResult response = iam.getAccessKeyLastUsed(request);

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

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，方式是创建 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 AmazonIdentityManagementClient 的 `updateAccessKey` 方法。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.UpdateAccessKeyRequest;
import com.amazonaws.services.identitymanagement.model.UpdateAccessKeyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();
```

```
UpdateAccessKeyRequest request = new UpdateAccessKeyRequest()
    .withAccessKeyId(access_id)
    .withUserName(username)
    .withStatus(status);

UpdateAccessKeyResult 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 AmazonIdentityManagementClient 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，请改用 updateAccessKey (p. 95) 方法。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteAccessKeyRequest;
import com.amazonaws.services.identitymanagement.model.DeleteAccessKeyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

DeleteAccessKeyRequest request = new DeleteAccessKeyRequest()
    .withAccessKeyId(access_key)
    .withUserName(username);

DeleteAccessKeyResult 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

创建用户

通过向 AmazonIdentityManagementClient 的 `createUser` 方法提供用户名来创建新 IAM 用户，用户名可直接提供，也可以使用包含用户名的 CreateUserRequest 对象。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreateUserRequest;
import com.amazonaws.services.identitymanagement.model.CreateUserResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

CreateUserRequest request = new CreateUserRequest()
    .withUserName(username);

CreateUserResult 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 AmazonIdentityManagementClient 的 `listUsers` 方法。您可以通过在返回的 ListUsersResponse 对象上调用 `getUsers` 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 `getIsTruncated` 方法查看更多可检索的结果。如果返回 `true`，则调用请求对象的 `setMarker()` 方法，并为其传递响应对象的 `getMarker()` 方法的返回值。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListUsersRequest;
import com.amazonaws.services.identitymanagement.model.ListUsersResult;
import com.amazonaws.services.identitymanagement.model.User;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

boolean done = false;
ListUsersRequest request = new ListUsersRequest();

while(!done) {
    ListUsersResult response = iam.listUsers(request);

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

    request.setMarker(response.getMarker());

    if(!response.getIsTruncated()) {
        done = true;
    }
}
```

请参阅 GitHub 上的完整示例。

更新用户

要更新用户，请调用 AmazonIdentityManagementClient 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 com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.UpdateUserRequest;
import com.amazonaws.services.identitymanagement.model.UpdateUserResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

UpdateUserRequest request = new UpdateUserRequest()
    .withUserName(cur_name)
    .withNewUserName(new_name);

UpdateUserResult 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 AmazonIdentityManagementClient 的 `deleteUser` 请求，该对象中设置了要删除的用户名。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteConflictException;
import com.amazonaws.services.identitymanagement.model.DeleteUserRequest;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

DeleteUserRequest request = new DeleteUserRequest()
    .withUserName(username);

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 帐户别名

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 AmazonIdentityManagementClient 的 `createAccountAlias` 方法。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreateAccountAliasRequest;
import com.amazonaws.services.identitymanagement.model.CreateAccountAliasResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

CreateAccountAliasRequest request = new CreateAccountAliasRequest()
    .withAccountAlias(alias);

CreateAccountAliasResult 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 AmazonIdentityManagementClient 的 `listAccountAliases` 方法。

> **Note**
>
> 返回的 ListAccountAliasesResponse 支持与其他AWS SDK for Java list 方法相同的 `getIsTruncated` 和 `getMarker` 方法，但 AWS 账户只能有一个 账户别名。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListAccountAliasesResult;
```

code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

ListAccountAliasesResult response = iam.listAccountAliases();

for (String alias : response.getAccountAliases()) {
    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 AmazonIdentityManagementClient 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 com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteAccountAliasRequest;
import com.amazonaws.services.identitymanagement.model.DeleteAccountAliasResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

DeleteAccountAliasRequest request = new DeleteAccountAliasRequest()
    .withAccountAlias(alias);

DeleteAccountAliasResult response = iam.deleteAccountAlias(request);
```

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

创建策略

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

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
```

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreatePolicyRequest;
import com.amazonaws.services.identitymanagement.model.CreatePolicyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

CreatePolicyRequest request = new CreatePolicyRequest()
    .withPolicyName(policy_name)
    .withPolicyDocument(POLICY_DOCUMENT);

CreatePolicyResult response = iam.createPolicy(request);
```

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 AmazonIdentityManagementClient 的 `getPolicy` 方法，并在 GetPolicyRequest 对象中提供策略的 ARN。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.GetPolicyRequest;
import com.amazonaws.services.identitymanagement.model.GetPolicyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();
```

```
GetPolicyRequest request = new GetPolicyRequest()
    .withPolicyArn(policy_arn);

GetPolicyResult 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 AmazonIdentityManagementClient 的 `attachRolePolicy` 方法，并在 AttachRolePolicyRequest 中为其提供角色名称和策略 ARN。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.AttachRolePolicyRequest;
import com.amazonaws.services.identitymanagement.model.AttachedPolicy;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

AttachRolePolicyRequest attach_request =
    new AttachRolePolicyRequest()
        .withRoleName(role_name)
        .withPolicyArn(POLICY_ARN);

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 AmazonIdentityManagementClient 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 ListAttachedRolePoliciesResult 对象中调用 `getAttachedPolicies` 来获取所附加策略的列表。如果 `ListAttachedRolePoliciesResult` 对象的 `getIsTruncated` 方法返回 `true`，调用 `ListAttachedRolePoliciesRequest` 对象的 `setMarker` 方法并使用其再次调用 `listAttachedRolePolicies` 来获取下一批结果，则结果可能被截断。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListAttachedRolePoliciesRequest;
import com.amazonaws.services.identitymanagement.model.ListAttachedRolePoliciesResult;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
```

Code

```
final AmazonIdentityManagement iam =
```

```
    AmazonIdentityManagementClientBuilder.defaultClient();

ListAttachedRolePoliciesRequest request =
    new ListAttachedRolePoliciesRequest()
        .withRoleName(role_name);

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

boolean done = false;

while(!done) {
    ListAttachedRolePoliciesResult response =
        iam.listAttachedRolePolicies(request);

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

    if(!response.getIsTruncated()) {
        done = true;
    }
    request.setMarker(response.getMarker());
}
```

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 AmazonIdentityManagementClient 的 `detachRolePolicy` 方法，并在 DetachRolePolicyRequest 中为其提供角色名称和策略 ARN。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DetachRolePolicyRequest;
import com.amazonaws.services.identitymanagement.model.DetachRolePolicyResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

DetachRolePolicyRequest request = new DetachRolePolicyRequest()
    .withRoleName(role_name)
    .withPolicyArn(policy_arn);

DetachRolePolicyResult 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

要在 AWS 上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 来预配置、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 AmazonIdentityManagementClient 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 com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.GetServerCertificateRequest;
import com.amazonaws.services.identitymanagement.model.GetServerCertificateResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

GetServerCertificateRequest request = new GetServerCertificateRequest()
            .withServerCertificateName(cert_name);

GetServerCertificateResult 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 AmazonIdentityManagementClient 的 `listServerCertificates` 方法。它返回 ListServerCertificatesResult。

调用返回的 `ListServerCertificateResult` 对象的 `getServerCertificateMetadataList` 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 `ListServerCertificateResult` 对象的 `getIsTruncated` 方法返回 `true`，调用 `ListServerCertificatesRequest` 对象的 `setMarker` 方法并使用其再次调用 `listServerCertificates` 来获取下一批结果，则结果可能被截断。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListServerCertificatesRequest;
```

```
import com.amazonaws.services.identitymanagement.model.ListServerCertificatesResult;
import com.amazonaws.services.identitymanagement.model.ServerCertificateMetadata;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

boolean done = false;
ListServerCertificatesRequest request =
        new ListServerCertificatesRequest();

while(!done) {

    ListServerCertificatesResult response =
        iam.listServerCertificates(request);

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

    request.setMarker(response.getMarker());

    if(!response.getIsTruncated()) {
        done = true;
    }
}
```

请参阅 GitHub 上的完整示例。

更新服务器证书

您可以通过调用 AmazonIdentityManagementClient 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 com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.UpdateServerCertificateRequest;
import com.amazonaws.services.identitymanagement.model.UpdateServerCertificateResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

UpdateServerCertificateRequest request =
    new UpdateServerCertificateRequest()
        .withServerCertificateName(cur_name)
        .withNewServerCertificateName(new_name);

UpdateServerCertificateResult 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 AmazonIdentityManagementClient 的 `deleteServerCertificate` 方法。

导入

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteServerCertificateRequest;
import com.amazonaws.services.identitymanagement.model.DeleteServerCertificateResult;
```

Code

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

DeleteServerCertificateRequest request =
    new DeleteServerCertificateRequest()
        .withServerCertificateName(cert_name);

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

请参阅 GitHub 上的完整示例。

更多信息

- IAM User Guide 中的使用服务器证书
- 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

Lambda示例使用 AWS SDK for Java

此部分提供使用 AWS SDK for Java对 Lambda 进行编程的示例。

> **Note**
>
> 该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

调用、列表和删除Lambda功能

此部分提供了使用适用于 Java 的 AWS 开发工具包对 Lambda 服务客户端进行编程的示例。要了解如何创建 Lambda 函数，请参阅如何创建 AWS Lambda 函数。

主题

调用 Lambda 函数

可以通过创建 AWSLambda 对并调用其 `invoke` 方法来调用 Lambda 函数。创建 InvokeRequest 对象可指定其他信息，例如函数名称和要传递给 Lambda 函数的负载。函数名称显示为 arn:aws:lambda:us-west-2:555556330391:function:HelloFunction。可以通过查看 AWS 控制台中的函数来检索值。

要将负载数据传递给函数，请调用 InvokeRequest 对象的 `withPayload` 方法并指定 JSON 格式的字符串，如以下代码示例中所示。

导入

```
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.lambda.AWSLambda;
import com.amazonaws.services.lambda.AWSLambdaClientBuilder;
import com.amazonaws.services.lambda.model.InvokeRequest;
import com.amazonaws.services.lambda.model.InvokeResult;
import com.amazonaws.services.lambda.model.ServiceException;

import java.nio.charset.StandardCharsets;
```

Code

以下代码示例演示如何调用 Lambda 函数。

```
        String functionName = args[0];

        InvokeRequest invokeRequest = new InvokeRequest()
                .withFunctionName(functionName)
                .withPayload("{\n" +
                        " \"Hello \": \"Paris\",\n" +
                        " \"countryCode\": \"FR\"\n" +
                        "}");
        InvokeResult invokeResult = null;

        try {
            AWSLambda awsLambda = AWSLambdaClientBuilder.standard()
                    .withCredentials(new ProfileCredentialsProvider())
                    .withRegion(Regions.US_WEST_2).build();

            invokeResult = awsLambda.invoke(invokeRequest);

            String ans = new String(invokeResult.getPayload().array(),
 StandardCharsets.UTF_8);

            //write out the return value
            System.out.println(ans);

        } catch (ServiceException e) {
            System.out.println(e);
        }

        System.out.println(invokeResult.getStatusCode());
```

请参阅 Github 上的完整示例。

列出 Lambda 函数

构建一个 AWSLambda 对象并调用其 `listFunctions` 方法。此方法返回一个 ListFunctionsResult 对象。可以调用此对象的 `getFunctions` 方法来返回 FunctionConfiguration 对象的列表。可以遍历该列表来检索有关函数的信息。例如，以下 Java 代码示例说明如何获取每个函数名称。

导入

```
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.lambda.AWSLambda;
import com.amazonaws.services.lambda.AWSLambdaClientBuilder;
import com.amazonaws.services.lambda.model.FunctionConfiguration;
import com.amazonaws.services.lambda.model.ListFunctionsResult;
import com.amazonaws.services.lambda.model.ServiceException;
import java.util.Iterator;
import java.util.List;
```

Code

以下 Java 代码示例演示如何检索 Lambda 函数名称的列表。

```
        ListFunctionsResult functionResult = null;

        try {
            AWSLambda awsLambda = AWSLambdaClientBuilder.standard()
                    .withCredentials(new ProfileCredentialsProvider())
                    .withRegion(Regions.US_WEST_2).build();

            functionResult = awsLambda.listFunctions();

            List<FunctionConfiguration> list = functionResult.getFunctions();

            for (Iterator iter = list.iterator(); iter.hasNext(); ) {
                FunctionConfiguration config = (FunctionConfiguration)iter.next();

                System.out.println("The function name is "+config.getFunctionName());
            }

        } catch (ServiceException e) {
            System.out.println(e);
        }
```

请参阅 Github 上的完整示例。

删除 Lambda 函数

构建一个 AWSLambda 对象并调用其 `deleteFunction` 方法。创建一个 DeleteFunctionRequest 对象并将该对象传递给 `deleteFunction` 方法。此对象包含要删除的函数的名称等信息。函数名称显示为 arn:aws:lambda:us-west-2:555556330391:function:HelloFunction。可以通过查看 AWS 控制台中的函数来检索值。

导入

```
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.lambda.AWSLambda;
import com.amazonaws.services.lambda.AWSLambdaClientBuilder;
import com.amazonaws.services.lambda.model.ServiceException;
import com.amazonaws.services.lambda.model.DeleteFunctionRequest;
```

Code

以下 Java 代码演示如何删除 Lambda 函数。

```
        String functionName = args[0];
        try {
            AWSLambda awsLambda = AWSLambdaClientBuilder.standard()
                    .withCredentials(new ProfileCredentialsProvider())
                    .withRegion(Regions.US_WEST_2).build();

            DeleteFunctionRequest delFunc = new DeleteFunctionRequest();
            delFunc.withFunctionName(functionName);

            //Delete the functiom
            awsLambda.deleteFunction(delFunc);
            System.out.println("The function is deleted");

        } catch (ServiceException e) {
            System.out.println(e);
        }
```

请参阅 Github 上的完整示例。

使用 AWS SDK for Java 的 Amazon Pinpoint示例

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

> Note
>
> 该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

创建和删除应用程序 Amazon Pinpoint

应用程序是您在其中为不同应用程序定义受众并通过定制消息吸引此受众的 Amazon Pinpoint 项目。此页中的示例演示如何创建新的应用程序或删除现有应用程序。

创建 () 应用程序

通过向 CreateAppRequest 对象提供应用程序名称，然后将该对象传递到 AmazonPinpointClient 的 `createApp` 方法，在 Amazon Pinpoint 中创建新的应用程序。

导入

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
import com.amazonaws.services.pinpoint.model.CreateAppRequest;
```

```
import com.amazonaws.services.pinpoint.model.CreateAppResult;
import com.amazonaws.services.pinpoint.model.CreateApplicationRequest;
```

Code

```
CreateApplicationRequest appRequest = new CreateApplicationRequest()
  .withName(appName);

CreateAppRequest request = new CreateAppRequest();
request.withCreateApplicationRequest(appRequest);
CreateAppResult result = pinpoint.createApp(request);
```

请参阅 GitHub 上的完整示例。

删除应用程序

要删除应用程序，请使用 DeleteAppRequest 对象调用 AmazonPinpointClient 的 `deleteApp` 请求，该对象中设置了要删除的应用程序名称。

导入

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
```

Code

```
DeleteAppRequest deleteRequest = new DeleteAppRequest()
  .withApplicationId(appID);

pinpoint.deleteApp(deleteRequest);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon Pinpoint API Reference 中的应用程序
- Amazon Pinpoint API Reference 中的应用程序

创建端点 Amazon Pinpoint

终端节点唯一地标识可以使用 Amazon Pinpoint 向其发送推送通知的用户设备。如果您的应用程序启用了 Amazon Pinpoint 支持，则在新用户打开应用程序时，应用程序自动向 Amazon Pinpoint 注册终端节点。以下示例演示如何以编程方式添加新的终端节点。

创建终端节点 ()

通过在 EndpointRequest 对象中提供终端节点数据，在 Amazon Pinpoint 中创建新的终端节点。

导入

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
import com.amazonaws.services.pinpoint.model.UpdateEndpointRequest;
import com.amazonaws.services.pinpoint.model.UpdateEndpointResult;
```

```
import com.amazonaws.services.pinpoint.model.EndpointDemographic;
import com.amazonaws.services.pinpoint.model.EndpointLocation;
import com.amazonaws.services.pinpoint.model.EndpointRequest;
import com.amazonaws.services.pinpoint.model.EndpointResponse;
import com.amazonaws.services.pinpoint.model.EndpointUser;
import com.amazonaws.services.pinpoint.model.GetEndpointRequest;
import com.amazonaws.services.pinpoint.model.GetEndpointResult;
```

Code

```
HashMap<String, List<String>> customAttributes = new HashMap<>();
List<String> favoriteTeams = new ArrayList<>();
favoriteTeams.add("Lakers");
favoriteTeams.add("Warriors");
customAttributes.put("team", favoriteTeams);


EndpointDemographic demographic = new EndpointDemographic()
        .withAppVersion("1.0")
        .withMake("apple")
        .withModel("iPhone")
        .withModelVersion("7")
        .withPlatform("ios")
        .withPlatformVersion("10.1.1")
        .withTimezone("America/Los_Angeles");

EndpointLocation location = new EndpointLocation()
        .withCity("Los Angeles")
        .withCountry("US")
        .withLatitude(34.0)
        .withLongitude(-118.2)
        .withPostalCode("90068")
        .withRegion("CA");

Map<String,Double> metrics = new HashMap<>();
metrics.put("health", 100.00);
metrics.put("luck", 75.00);

EndpointUser user = new EndpointUser()
        .withUserId(UUID.randomUUID().toString());

DateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm'Z'"); // Quoted "Z" to indicate
 UTC, no timezone offset
String nowAsISO = df.format(new Date());

EndpointRequest endpointRequest = new EndpointRequest()
        .withAddress(UUID.randomUUID().toString())
        .withAttributes(customAttributes)
        .withChannelType("APNS")
        .withDemographic(demographic)
        .withEffectiveDate(nowAsISO)
        .withLocation(location)
        .withMetrics(metrics)
        .withOptOut("NONE")
        .withRequestId(UUID.randomUUID().toString())
        .withUser(user);
```

然后使用该 EndpointRequest 对象创建 UpdateEndpointRequest 对象。最后，将 UpdateEndpointRequest 对象传递到 AmazonPinpointClient 的 `updateEndpoint` 方法。

Code

```
UpdateEndpointRequest updateEndpointRequest = new UpdateEndpointRequest()
```

```
        .withApplicationId(appId)
        .withEndpointId(endpointId)
        .withEndpointRequest(endpointRequest);

UpdateEndpointResult updateEndpointResponse = client.updateEndpoint(updateEndpointRequest);
System.out.println("Update Endpoint Response: " + updateEndpointResponse.getMessageBody());
```

请参阅 GitHub 上的完整示例。

更多信息

- 在 Amazon Pinpoint Developer Guide 中添加终端节点
- Amazon Pinpoint API Reference 中的终端节点

在 Amazon Pinpoint

用户分段表示基于共同的特征（例如用户最近什么时候打开了您的应用程序或他们使用哪个设备）的用户子集。以下示例演示如何定义用户分段。

创建分段

通过在 SegmentDimensions 对象中定义分段的维度，在 Amazon Pinpoint 中创建新的分段。

导入

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
import com.amazonaws.services.pinpoint.model.CreateSegmentRequest;
import com.amazonaws.services.pinpoint.model.CreateSegmentResult;
import com.amazonaws.services.pinpoint.model.AttributeDimension;
import com.amazonaws.services.pinpoint.model.AttributeType;
import com.amazonaws.services.pinpoint.model.RecencyDimension;
import com.amazonaws.services.pinpoint.model.SegmentBehaviors;
import com.amazonaws.services.pinpoint.model.SegmentDemographics;
import com.amazonaws.services.pinpoint.model.SegmentDimensions;
import com.amazonaws.services.pinpoint.model.SegmentLocation;
import com.amazonaws.services.pinpoint.model.SegmentResponse;
import com.amazonaws.services.pinpoint.model.WriteSegmentRequest;
```

Code

```
Pinpoint pinpoint =
 AmazonPinpointClientBuilder.standard().withRegion(Regions.US_EAST_1).build();
Map<String, AttributeDimension> segmentAttributes = new HashMap<>();
segmentAttributes.put("Team", new
 AttributeDimension().withAttributeType(AttributeType.INCLUSIVE).withValues("Lakers"));

SegmentBehaviors segmentBehaviors = new SegmentBehaviors();
SegmentDemographics segmentDemographics = new SegmentDemographics();
SegmentLocation segmentLocation = new SegmentLocation();

RecencyDimension recencyDimension = new RecencyDimension();
recencyDimension.withDuration("DAY_30").withRecencyType("ACTIVE");
segmentBehaviors.setRecency(recencyDimension);

SegmentDimensions dimensions = new SegmentDimensions()
        .withAttributes(segmentAttributes)
        .withBehavior(segmentBehaviors)
```

```
        .withDemographic(segmentDemographics)
        .withLocation(segmentLocation);
```

下一个设置 分段维度 对象 写入请求，这又用于创建 创建请求 对象。最后，将 CreateSegmentRequest 对象传递到 AmazonPinpointClient 的 `createSegment` 方法。

Code

```
WriteSegmentRequest writeSegmentRequest = new WriteSegmentRequest()
        .withName("MySegment").withDimensions(dimensions);

CreateSegmentRequest createSegmentRequest = new CreateSegmentRequest()
        .withApplicationId(appId).withWriteSegmentRequest(writeSegmentRequest);

CreateSegmentResult createSegmentResult = client.createSegment(createSegmentRequest);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon Pinpoint User Guide 中的 Amazon Pinpoint 分段
- 在 Amazon Pinpoint Developer Guide 中创建分段
- Amazon Pinpoint API Reference 中的分段
- Amazon Pinpoint API Reference 中的分段

在 Amazon Pinpoint

您可以使用市场活动来帮助增加应用程序与用户之间的互动。您可以创建市场活动，通过定制消息或特殊促销吸引特定的用户分段。此示例演示如何创建新的标准市场活动，以向特定的分段发送自定义推送消息。

创建市场活动

创建新的市场活动之前，您必须定义计划和消息，并在 WriteCampaignRequest 对象中设置这些值。

导入

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
import com.amazonaws.services.pinpoint.model.CreateCampaignRequest;
import com.amazonaws.services.pinpoint.model.CreateCampaignResult;
import com.amazonaws.services.pinpoint.model.Action;
import com.amazonaws.services.pinpoint.model.CampaignResponse;
import com.amazonaws.services.pinpoint.model.Message;
import com.amazonaws.services.pinpoint.model.MessageConfiguration;
import com.amazonaws.services.pinpoint.model.Schedule;
import com.amazonaws.services.pinpoint.model.WriteCampaignRequest;
```

Code

```
Schedule schedule = new Schedule()
        .withStartTime("IMMEDIATE");

Message defaultMessage = new Message()
        .withAction(Action.OPEN_APP)
```

```
        .withBody("My message body.")
        .withTitle("My message title.");

MessageConfiguration messageConfiguration = new MessageConfiguration()
        .withDefaultMessage(defaultMessage);

WriteCampaignRequest request = new WriteCampaignRequest()
        .withDescription("My description.")
        .withSchedule(schedule)
        .withSegmentId(segmentId)
        .withName("MyCampaign")
        .withMessageConfiguration(messageConfiguration);
```

然后通过将具有市场活动配置的 WriteCampaignRequest 提供给 CreateCampaignRequest 对象，在 Amazon Pinpoint 中创建新的市场活动。最后，将 CreateCampaignRequest 对象传递到 AmazonPinpointClient 的 `createCampaign` 方法。

Code

```
CreateCampaignRequest createCampaignRequest = new CreateCampaignRequest()
        .withApplicationId(appId).withWriteCampaignRequest(request);

CreateCampaignResult result = client.createCampaign(createCampaignRequest);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon Pinpoint User Guide 中的 Amazon Pinpoint 市场活动
- 在 Amazon Pinpoint Developer Guide 中创建市场活动
- Amazon Pinpoint API Reference 中的市场活动
- Amazon Pinpoint API Reference 中的市场活动
- Amazon Pinpoint API Reference 中的市场活动
- Amazon Pinpoint API Reference 中的市场活动版本
- Amazon Pinpoint API Reference 中的市场活动版本

更新信道 Amazon Pinpoint

渠道定义您可将消息传递到的平台类型。此示例演示如何使用 APN 渠道发送消息。

更新渠道

通过提供应用程序 ID 以及您希望更新的渠道类型的请求对象，在 Amazon Pinpoint 中启用渠道。此示例将更新 APN 渠道，这需要 APNSChannelRequest 对象。请在 UpdateApnsChannelRequest 中进行设置并将该对象传递到 AmazonPinpointClient 的 `updateApnsChannel` 方法。

导入

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
import com.amazonaws.services.pinpoint.model.APNSChannelRequest;
import com.amazonaws.services.pinpoint.model.APNSChannelResponse;
import com.amazonaws.services.pinpoint.model.GetApnsChannelRequest;
import com.amazonaws.services.pinpoint.model.GetApnsChannelResult;
import com.amazonaws.services.pinpoint.model.UpdateApnsChannelRequest;
```

```
import com.amazonaws.services.pinpoint.model.UpdateApnsChannelResult;
```

Code

```
APNSChannelRequest request = new APNSChannelRequest()
  .withEnabled(enabled);

UpdateApnsChannelRequest updateRequest = new UpdateApnsChannelRequest()
  .withAPNSChannelRequest(request)
  .withApplicationId(appId);
UpdateApnsChannelResult result = client.updateApnsChannel(updateRequest);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon Pinpoint User Guide 中的 Amazon Pinpoint 渠道
- Amazon Pinpoint API Reference 中的 ADM 渠道
- Amazon Pinpoint API Reference 中的 APN 渠道
- Amazon Pinpoint API Reference 中的 APN 沙盒渠道
- Amazon Pinpoint API Reference 中的 APN VoIP 渠道
- Amazon Pinpoint API Reference 中的 APN VoIP 沙盒渠道
- Amazon Pinpoint API Reference 中的 Baidu 渠道
- Amazon Pinpoint API Reference 中的电子邮件渠道
- Amazon Pinpoint API Reference 中的 GCM 渠道
- Amazon Pinpoint API Reference 中的 SMS 渠道

使用 AWS SDK for Java 的 Amazon S3示例

本节提供 Amazon S3 编程 (使用用于 Java 的 AWS 开发工具包) 的示例。

> **Note**
>
> 该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、列表和删除 Amazon S3 桶

Amazon S3 中的每个对象（文件）必须放入存储桶，它代表对象的集合（容器）。每个存储桶使用必须唯一的键 (名称) 命名。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

这些代码示例假定您了解使用适用于 Java 的 AWS 开发工具包 (p. 19)中的内容，并且已使用设置用于开发的 AWS 凭证和区域 (p. 6)中的信息配置默认 AWS 凭证。

创建存储桶

使用 AmazonS3 客户端的 `createBucket` 方法。会返回新的存储桶。如果存储桶已存在，`createBucket` 方法将引发异常。

Note

要尝试创建一个具有相同名称的存储桶来检查存储桶是否已存在，请调用 `doesBucketExist` 方法。如果存储桶存在，它将返回 `true`，否则将返回 `false`。

导入

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AmazonS3Exception;
import com.amazonaws.services.s3.model.Bucket;

import java.util.List;
```

Code

```
if (s3.doesBucketExistV2(bucket_name)) {
    System.out.format("Bucket %s already exists.\n", bucket_name);
    b = getBucket(bucket_name);
} else {
    try {
        b = s3.createBucket(bucket_name);
    } catch (AmazonS3Exception e) {
        System.err.println(e.getErrorMessage());
    }
}
return b;
```

请参阅 GitHub 上的完整示例。

列出存储桶

使用 AmazonS3 客户端的 `listBucket` 方法。如果成功，会返回存储桶的列表。

导入

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
```

```
import com.amazonaws.services.s3.model.Bucket;

import java.util.List;
```

Code

```
List<Bucket> buckets = s3.listBuckets();
System.out.println("Your Amazon S3 buckets are:");
for (Bucket b : buckets) {
    System.out.println("* " + b.getName());
}
```

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，否则会导致错误。如果您的存储桶受版本控制，则必须同时删除与该存储桶关联的所有受版本控制对象。

> **Note**
>
> 完整示例中依次包含上述每个步骤，提供用于删除 Amazon S3 存储桶及其内容的完整解决方案。

主题

删除不受版本控制的存储桶之前先删除其中的对象

使用 AmazonS3 客户端的 `listObjects` 方法来检索对象列表，并使用 `deleteObject` 删除每个对象。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.*;

import java.util.Iterator;
```

Code

```
System.out.println(" - removing objects from bucket");
ObjectListing object_listing = s3.listObjects(bucket_name);
while (true) {
    for (Iterator<?> iterator =
         object_listing.getObjectSummaries().iterator();
         iterator.hasNext(); ) {
        S3ObjectSummary summary = (S3ObjectSummary) iterator.next();
        s3.deleteObject(bucket_name, summary.getKey());
    }

    // more object_listing to retrieve?
    if (object_listing.isTruncated()) {
        object_listing = s3.listNextBatchOfObjects(object_listing);
```

```
    } else {
        break;
    }
}
```

请参阅 GitHub 上的完整示例。

删除受版本控制的存储桶之前先删除其中的对象

如果您使用受版本控制的存储桶，还需要先删除存储桶中存储的所有受版本控制对象，然后才能删除存储桶。

使用在删除存储桶中的对象时所用的类似方法，通过使用 AmazonS3 客户端的 `listVersions` 方法列出所有受版本控制的对象，然后使用 `deleteVersion` 删除各个对象。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.*;

import java.util.Iterator;
```

Code

```
System.out.println(" - removing versions from bucket");
VersionListing version_listing = s3.listVersions(
        new ListVersionsRequest().withBucketName(bucket_name));
while (true) {
    for (Iterator<?> iterator =
         version_listing.getVersionSummaries().iterator();
         iterator.hasNext(); ) {
        S3VersionSummary vs = (S3VersionSummary) iterator.next();
        s3.deleteVersion(
                bucket_name, vs.getKey(), vs.getVersionId());
    }

    if (version_listing.isTruncated()) {
        version_listing = s3.listNextBatchOfVersions(
                version_listing);
    } else {
        break;
    }
}
```

请参阅 GitHub 上的完整示例。

删除空存储桶

在删除存储桶中的对象（包括所有受版本控制的对象）后，就可以使用 AmazonS3 客户端的 `deleteBucket` 方法删除存储桶本身。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
```

```
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.*;

import java.util.Iterator;
```

Code

```
System.out.println(" OK, bucket ready to delete!");
s3.deleteBucket(bucket_name);
```

请参阅 GitHub 上的完整示例。

执行操作 Amazon S3 对象

Amazon S3 对象表示一个文件 或数据集合。每个对象必须驻留在一个存储桶 (p. 115)中。

> **Note**
>
> 这些代码示例假定您了解使用适用于 Java 的 AWS 开发工具包 (p. 19)中的内容，并且已使用设置用于开发的 AWS 凭证和区域 (p. 6)中的信息配置默认 AWS 凭证。

主题

上传对象

使用 AmazonS3 客户端的 `putObject` 方法，并为其提供存储桶名称、键名称和要上传的文件。存储桶必须存在，否则将出现错误。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
```

Code

```
System.out.format("Uploading %s to S3 bucket %s...\n", file_path, bucket_name);
final AmazonS3 s3 =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    s3.putObject(bucket_name, key_name, new File(file_path));
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
```

请参阅 GitHub 上的完整示例。

列出对象

要获取存储桶中的对象列表，请使用 AmazonS3 客户端的 `listObjects` 方法，并为其提供存储桶名称。

`listObjects` 方法返回一个 ObjectListing 对象，该对象提供有关存储桶中对象的信息。要列出对象名称（键），可使用 `getObjectSummaries` 方法获取 S3ObjectSummary 对象的列表，其中每个对象均表示存储桶中的一个对象。然后调用其 `getKey` 方法以检索对象名称。

导入

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.ListObjectsV2Result;
import com.amazonaws.services.s3.model.S3ObjectSummary;
```

Code

```
System.out.format("Objects in S3 bucket %s:\n", bucket_name);
final AmazonS3 s3 =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
ListObjectsV2Result result = s3.listObjectsV2(bucket_name);
List<S3ObjectSummary> objects = result.getObjectSummaries();
for (S3ObjectSummary os : objects) {
    System.out.println("* " + os.getKey());
}
```

请参阅 GitHub 上的完整示例。

下载对象

使用 AmazonS3 客户端的 `getObject` 方法，并向其传递要下载的存储桶和对象的名称。如果成功，此方法将返回一个 S3Object。指定的存储桶和对象键必须存在，否则将出现错误。

您可以通过致电 `getObjectContent` 在 `S3Object`。这将返回 S3ObjectinputStream 作为标准Java的行为 `InputStream` 对象。

以下示例从 S3 下载一个对象，然后将该对象的内容保存到一个文件（使用与对象键相同的名称）：

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.S3Object;
import com.amazonaws.services.s3.model.S3ObjectInputStream;

import java.io.File;
```

Code

```
System.out.format("Downloading %s from S3 bucket %s...\n", key_name, bucket_name);
final AmazonS3 s3 =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    S3Object o = s3.getObject(bucket_name, key_name);
```

```
    S3ObjectInputStream s3is = o.getObjectContent();
    FileOutputStream fos = new FileOutputStream(new File(key_name));
    byte[] read_buf = new byte[1024];
    int read_len = 0;
    while ((read_len = s3is.read(read_buf)) > 0) {
        fos.write(read_buf, 0, read_len);
    }
    s3is.close();
    fos.close();
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
} catch (FileNotFoundException e) {
    System.err.println(e.getMessage());
    System.exit(1);
} catch (IOException e) {
    System.err.println(e.getMessage());
    System.exit(1);
```

请参阅 GitHub 上的完整示例。

复制、移动或重命名对象

您可以使用 AmazonS3 客户端的 `copyObject` 方法将对象从一个存储桶复制到另一个存储桶。它采用要从中复制的存储桶的名称、要复制的对象以及目标存储桶名称。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
```

Code

```
    s3.copyObject(from_bucket, object_key, to_bucket, object_key);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
System.out.println("Done!");
```

请参阅 GitHub 上的完整示例。

> **Note**
>
> 您可以将 `copyObject` 与 deleteObject (p. 121) 配合使用来移动或重命名对象，方式是先将对象复制到新名称 (您可以使用与源和目标相同的存储桶)，然后从对象的旧位置删除对象。

删除数据元

使用 AmazonS3 客户端的 `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，否则将出现错误。

导入

```
import com.amazonaws.AmazonServiceException;
```

```
import com.amazonaws.regions.Regions;
```

Code

```
final AmazonS3 s3 =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    s3.deleteObject(bucket_name, object_key);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

请参阅 GitHub 上的完整示例。

一次性删除多个对象

利用 AmazonS3 客户端的 `deleteObjects` 方法，您可以从同一存储桶中删除多个对象，方式是将这些对象的名称传递到 DeleteObjectRequest `withKeys` 方法。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
```

Code

```
final AmazonS3 s3 =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    DeleteObjectsRequest dor = new DeleteObjectsRequest(bucket_name)
            .withKeys(object_keys);
    s3.deleteObjects(dor);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
```

请参阅 GitHub 上的完整示例。

管理 Amazon S3 存储桶和对象的访问权限

您可以为 Amazon S3 存储桶和对象使用访问控制列表 (ACL)，以实现对 Amazon S3 资源的精细控制。

> **Note**
>
> 这些代码示例假定您了解使用适用于 Java 的 AWS 开发工具包 (p. 19)中的内容，并且已使用设置用于开发的 AWS 凭证和区域 (p. 6)中的信息配置默认 AWS 凭证。

获取存储桶的访问控制列表

要获取存储桶的当前 ACL，请调用 AmazonS3 的 `getBucketAcl` 方法，将存储桶名称传递给它以进行查询。此方法将返回 AccessControlList 对象。要获取列表中的每个访问授权，请调用其 `getGrantsAsList` 方法，这会返回一个包含 Grant 对象的标准 Java 列表。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.Grant;
```

Code

```
final AmazonS3 s3 =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    AccessControlList acl = s3.getBucketAcl(bucket_name);
    List<Grant> grants = acl.getGrantsAsList();
    for (Grant grant : grants) {
        System.out.format("  %s: %s\n", grant.getGrantee().getIdentifier(),
                grant.getPermission().toString());
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

请参阅 GitHub 上的完整示例。

设置存储桶的访问控制列表

要添加或修改存储桶对 ACL 的权限，请调用 AmazonS3 的 `setBucketAcl` 方法。这需要一个 AccessControlList 对象，它包含被授权者和要设置的访问级别的列表。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.EmailAddressGrantee;
```

Code

```
final AmazonS3 s3 =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    // get the current ACL
    AccessControlList acl = s3.getBucketAcl(bucket_name);
    // set access for the grantee
    EmailAddressGrantee grantee = new EmailAddressGrantee(email);
    Permission permission = Permission.valueOf(access);
    acl.grantPermission(grantee, permission);
    s3.setBucketAcl(bucket_name, acl);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

Note

您可以使用 Grantee 类直接提供被授权者的唯一标识符，也可以使用 EmailAddressGrantee 类通过电子邮件设置被授权者，这里采用后者。

请参阅 GitHub 上的完整示例。

获取对象的访问控制列表

要获取对象的当前 ACL，请调用 AmazonS3 的 `getObjectAcl` 方法，将存储桶名称和对象名称传递给它以进行查询。和 `getBucketAcl` 类似，此方法将返回一个 AccessControlList 对象，您可以用它来检查每个 Grant。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.Grant;
```

Code

```
try {
    AccessControlList acl = s3.getObjectAcl(bucket_name, object_key);
    List<Grant> grants = acl.getGrantsAsList();
    for (Grant grant : grants) {
        System.out.format("  %s: %s\n", grant.getGrantee().getIdentifier(),
                grant.getPermission().toString());
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

请参阅 GitHub 上的完整示例。

设置对象的访问控制列表

要添加或修改对象对 ACL 的权限，请调用 AmazonS3 的 `setObjectAcl` 方法。这需要一个 AccessControlList 对象，它包含被授权者和要设置的访问级别的列表。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.EmailAddressGrantee;
```

Code

```
    try {
        // get the current ACL
        AccessControlList acl = s3.getObjectAcl(bucket_name, object_key);
```

```
        // set access for the grantee
        EmailAddressGrantee grantee = new EmailAddressGrantee(email);
        Permission permission = Permission.valueOf(access);
        acl.grantPermission(grantee, permission);
        s3.setObjectAcl(bucket_name, object_key, acl);
    } catch (AmazonServiceException e) {
        System.err.println(e.getErrorMessage());
        System.exit(1);
    }
}
```

Note

您可以使用 Grantee 类直接提供被授权者的唯一标识符，也可以使用 EmailAddressGrantee 类通过电子邮件设置被授权者，这里采用后者。

请参阅 GitHub 上的完整示例。

更多信息

- Amazon S3 API Reference 中的 GET Bucket acl
- Amazon S3 API Reference 中的 PUT Bucket acl
- Amazon S3 API Reference 中的 GET 对象 ACL
- Amazon S3 API Reference 中的 PUT Object acl

管理访问 Amazon S3 使用桶策略的存储区

您可以设置、获取或删除存储桶策略 来管理对 Amazon S3 存储桶的访问。

设置存储桶策略

您可以通过以下方式为特定的 S3 存储桶设置存储桶策略：

- 调用 AmazonS3 客户端的 `setBucketPolicy` 并为其提供 SetBucketPolicyRequest
- 使用接收存储桶名称和策略文本 (JSON 格式) 的 `setBucketPolicy` 重载直接设置策略

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.policy.Policy;
import com.amazonaws.auth.policy.Principal;
```

Code

```
    s3.setBucketPolicy(bucket_name, policy_text);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

使用策略类生成或验证策略

为 `setBucketPolicy` 提供存储桶策略时，您可以执行以下操作：

- 使用 JSON 格式的文本字符串直接指定策略
- 使用 Policy 类构建策略

使用 `Policy` 类，您不必担心如何正确设置文本字符串的格式。要从 `Policy` 类获取 JSON 策略文本，请使用其 `toJson` 方法。

导入

```
import com.amazonaws.auth.policy.Resource;
import com.amazonaws.auth.policy.Statement;
import com.amazonaws.auth.policy.actions.S3Actions;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
```

Code

```
        new Statement(Statement.Effect.Allow)
                .withPrincipals(Principal.AllUsers)
                .withActions(S3Actions.GetObject)
                .withResources(new Resource(
                        "arn:aws:s3:::" + bucket_name + "/*")));
return bucket_policy.toJson();
```

`Policy` 类还提供 `fromJson` 方法，它会尝试使用传入的 JSON 字符串构建策略。该方法会验证文本以确保可以转换为有效策略结构，如果策略文本无效，就会失败并引发 `IllegalArgumentException`。

```
Policy bucket_policy = null;
try {
    bucket_policy = Policy.fromJson(file_text.toString());
} catch (IllegalArgumentException e) {
    System.out.format("Invalid policy text in file: \"%s\"",
            policy_file);
    System.out.println(e.getMessage());
}
```

您可以使用此方法，提前验证您从文件读入或通过其他方法得到的策略。

请参阅 GitHub 上的完整示例。

获取存储桶策略

要检索 Amazon S3 存储桶的策略，请调用 AmazonS3 客户端的 `getBucketPolicy` 方法，将存储桶的名称传递给它以获取策略。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
```

Code

```
  try {
```

```
        BucketPolicy bucket_policy = s3.getBucketPolicy(bucket_name);
        policy_text = bucket_policy.getPolicyText();
    } catch (AmazonServiceException e) {
        System.err.println(e.getErrorMessage());
        System.exit(1);
    }
```

如果指定的存储桶不存在、您没有访问该存储桶的权限或者其中不包含存储桶策略，会引发 `AmazonServiceException`。

请参阅 GitHub 上的完整示例。

删除存储桶策略

要删除存储桶策略，调用 AmazonS3 客户端的 `deleteBucketPolicy`，并为其提供存储桶名称。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
```

Code

```
    try {
        s3.deleteBucketPolicy(bucket_name);
    } catch (AmazonServiceException e) {
        System.err.println(e.getErrorMessage());
        System.exit(1);
    }
```

即使存储桶中还没有策略，该方法也会成功。如果您指定的存储桶名称不存在，或者您没有访问该存储桶的权限，会引发 `AmazonServiceException`。

请参阅 GitHub 上的完整示例。

更多信息

- Amazon S3 Developer Guide 中的访问策略语言概述
- Amazon S3 Developer Guide中的存储桶策略示例

使用转让经理 Amazon S3 操作

您可以使用 AWS SDK for JavaTransferManager 类可靠地将文件从本地环境传输到 Amazon S3 并将对象从一个 S3 位置复制到另一个 S3 位置。`TransferManager` 可获取传输进度，并且能够暂停或恢复上传和下载。

> **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

这些代码示例假定您了解使用适用于 Java 的 AWS 开发工具包 (p. 19)中的内容，并且已使用设置用于开发的 AWS 凭证和区域 (p. 6)中的信息配置默认 AWS 凭证。

上传文件和目录

TransferManager 可将文件、文件列表和目录上传到您之前创建 (p. 116)的任何 Amazon S3 存储桶。

主题

- 上传单个文件 (p. 128)
- 上传文件列表 (p. 128)
- 上传目录 (p. 129)

上传单个文件

调用 TransferManager`upload` 方法，提供 Amazon S3 存储桶名称、键（对象）名称和代表要上传的文件的标准 Java File 对象。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.MultipleFileUpload;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;
import com.amazonaws.services.s3.transfer.Upload;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
```

Code

```
File f = new File(file_path);
TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
try {
    Upload xfer = xfer_mgr.upload(bucket_name, key_name, f);
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    //  or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

`upload` 方法立即返回值，为您提供一个 `Upload` 对象，用于检查传输状态或等待传输完成。

请参阅等待传输完成 (p. 132)以了解有关在调用 TransferManager 的 `shutdownNow` 方法之前使用 `waitForCompletion` 成功完成传输的信息。在等待传输完成时，您可以轮询或侦听有关其状态和进度的更新。有关更多信息，请参阅获取传输状态和进度 (p. 133)。

请参阅 GitHub 上的完整示例。

上传文件列表

要通过一次操作上传多个文件，请调用 TransferManager 的 `uploadFileList` 方法，并为其提供：

- Amazon S3 存储桶名称。
- 一个键前缀，它将添加到创建的对象名称的前面 (将对象放置到的存储桶中的路径)
- 一个 File 对象，此对象表示将从中创建文件路径的相对目录
- 一个 List 对象，包含一组要上传的 File 对象

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.MultipleFileUpload;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;
import com.amazonaws.services.s3.transfer.Upload;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
```

Code

```
ArrayList<File> files = new ArrayList<File>();
for (String path : file_paths) {
    files.add(new File(path));
}

TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
try {
    MultipleFileUpload xfer = xfer_mgr.uploadFileList(bucket_name,
            key_prefix, new File("."), files);
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    // or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

请参阅等待传输完成 (p. 132)以了解有关在调用 TransferManager 的 `shutdownNow` 方法之前使用 `waitForCompletion` 成功完成传输的信息。在等待传输完成时，您可以轮询或侦听有关其状态和进度的更新。有关更多信息，请参阅获取传输状态和进度 (p. 133)。

可使用由 `uploadFileList` 返回的 MultipleFileUpload 对象来查询传输状态或进度。有关更多信息，请参阅轮询传输的当前进度 (p. 133)和使用 ProgressListener 获取传输进度 (p. 134)。

您也可以使用 `MultipleFileUpload` 的 `getSubTransfers` 方法为要传输的每个文件获取单个 `Upload` 对象。有关更多信息，请参阅获取子传输的进度 (p. 135)。

请参阅 GitHub 上的完整示例。

上传目录

可使用 TransferManager 的 `uploadDirectory` 方法通过用于以递归方式复制子目录中的文件的选项来上传整个文件目录。您提供一个 Amazon S3 存储桶名称、一个 S3 键前缀、一个表示要复制的本地目录的 File 对象和一个 `boolean` 值，该值指示您是否需要以递归方式复制子目录 (true 或 false) 。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.MultipleFileUpload;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;
import com.amazonaws.services.s3.transfer.Upload;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
```

Code

```
TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
try {
    MultipleFileUpload xfer = xfer_mgr.uploadDirectory(bucket_name,
            key_prefix, new File(dir_path), recursive);
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    // or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

请参阅等待传输完成 (p. 132)以了解有关在调用 TransferManager 的 `shutdownNow` 方法之前使用 `waitForCompletion` 成功完成传输的信息。在等待传输完成时，您可以轮询或侦听有关其状态和进度的更新。有关更多信息，请参阅获取传输状态和进度 (p. 133)。

可使用由 `uploadFileList` 返回的 MultipleFileUpload 对象来查询传输状态或进度。有关更多信息，请参阅轮询传输的当前进度 (p. 133)和使用 ProgressListener 获取传输进度 (p. 134)。

您也可以使用 `MultipleFileUpload` 的 `getSubTransfers` 方法为要传输的每个文件获取单个 `Upload` 对象。有关更多信息，请参阅获取子传输的进度 (p. 135)。

请参阅 GitHub 上的完整示例。

下载文件或目录

使用 TransferManager 类从 Amazon S3 下载单个文件（Amazon S3 对象）或目录（一个 Amazon S3 存储桶名称，后跟对象前缀）。

主题

- 下载单个文件 (p. 130)
- 下载目录 (p. 131)

下载单个文件

使用 TransferManager 的 `download` 方法，并为其提供包含要下载的对象的 Amazon S3 存储桶名称、键（对象）名称和一个 File 对象（该对象表示要在本地系统上创建的文件）。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.Download;
```

```
import com.amazonaws.services.s3.transfer.MultipleFileDownload;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;

import java.io.File;
```

Code

```
File f = new File(file_path);
TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
try {
    Download xfer = xfer_mgr.download(bucket_name, key_name, f);
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    // or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

请参阅等待传输完成 (p. 132)以了解有关在调用 TransferManager 的 `shutdownNow` 方法之前使用 `waitForCompletion` 成功完成传输的信息。在等待传输完成时，您可以轮询或侦听有关其状态和进度的更新。有关更多信息，请参阅获取传输状态和进度 (p. 133)。

请参阅 GitHub 上的完整示例。

下载目录

要从 Amazon S3 下载一组共享一个公共键前缀的文件（类似于文件系统上的目录），可使用 TransferManager 的 `downloadDirectory` 方法。该方法需要包含要下载的对象的 Amazon S3 存储桶名称、所有对象共享的对象前缀和一个 File 对象（此对象表示要将文件下载到的本地系统目录）。如果指定目录尚不存在，将创建此目录。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.Download;
import com.amazonaws.services.s3.transfer.MultipleFileDownload;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;

import java.io.File;
```

Code

```
TransferManager xfer_mgr = TransferManagerBuilder.standard().build();

try {
    MultipleFileDownload xfer = xfer_mgr.downloadDirectory(
            bucket_name, key_prefix, new File(dir_path));
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    // or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
```

```
}
xfer_mgr.shutdownNow();
```

请参阅等待传输完成 (p. 132)以了解有关在调用 TransferManager 的 `shutdownNow` 方法之前使用 `waitForCompletion` 成功完成传输的信息。在等待传输完成时，您可以轮询或侦听有关其状态和进度的更新。有关更多信息，请参阅获取传输状态和进度 (p. 133)。

请参阅 GitHub 上的完整示例。

复制对象

要将对象从一个 S3 存储桶复制到另一个 S3 存储桶，可使用 TransferManager 的 `copy` 方法。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.Copy;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;
```

Code

```
System.out.println("Copying s3 object: " + from_key);
System.out.println("      from bucket: " + from_bucket);
System.out.println("     to s3 object: " + to_key);
System.out.println("        in bucket: " + to_bucket);

TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
try {
    Copy xfer = xfer_mgr.copy(from_bucket, from_key, to_bucket, to_key);
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    // or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

请参阅 GitHub 上的完整示例。

请等待传输完成。

如果可以在传输完成前阻止您的应用程序（或线程），则可使用 Transfer 接口的 `waitForCompletion` 方法来阻止它，直至传输完成或出现异常。

```
try {
    xfer.waitForCompletion();
} catch (AmazonServiceException e) {
    System.err.println("Amazon service error: " + e.getMessage());
    System.exit(1);
} catch (AmazonClientException e) {
    System.err.println("Amazon client error: " + e.getMessage());
    System.exit(1);
} catch (InterruptedException e) {
    System.err.println("Transfer interrupted: " + e.getMessage());
    System.exit(1);
```

```
}
```

如果您在调用 `waitForCompletion` 之前 轮询事件、在单独线程上实施轮询机制或使用 ProgressListener 异步接收进度更新，则可获取传输进度。

请参阅 GitHub 上的完整示例。

获取传输状态和进度

TransferManager 的 `upload*`、`download*` 和 `copy` 方法所返回的每个类均返回以下某个类的实例，具体取决于它是单文件操作还是多文件操作。

课程	返回方
复制	`copy`
下载	`download`
MultipleFileDownload	`downloadDirectory`
上传	`upload`
MultipleFileUpload	`uploadFileList`, `uploadDirectory`

所有这些类别都实施 转移 接口。`Transfer` 提供了各种有用的方法，如获取传输进度、暂停或恢复传输以及获取传输的当前或最终状态等。

主题

- 轮询传输的当前进度 (p. 133)
- 使用 ProgressListener 获取传输进度 (p. 134)
- 获取子传输的进度 (p. 135)

轮询传输的当前进度

此循环打印传输的进度，在其运行时检查其当前进度，然后在传输完成时打印最终状态。

导入

```
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.event.ProgressEvent;
import com.amazonaws.event.ProgressListener;
import com.amazonaws.services.s3.transfer.*;
import com.amazonaws.services.s3.transfer.Transfer.TransferState;

import java.io.File;
import java.util.ArrayList;
import java.util.Collection;
```

Code

```
// print the transfer's human-readable description
System.out.println(xfer.getDescription());
// print an empty progress bar...
```

```
printProgressBar(0.0);
// update the progress bar while the xfer is ongoing.
do {
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        return;
    }
    // Note: so_far and total aren't used, they're just for
    // documentation purposes.
    TransferProgress progress = xfer.getProgress();
    long so_far = progress.getBytesTransferred();
    long total = progress.getTotalBytesToTransfer();
    double pct = progress.getPercentTransferred();
    eraseProgressBar();
    printProgressBar(pct);
} while (xfer.isDone() == false);
// print the final state of the transfer.
TransferState xfer_state = xfer.getState();
System.out.println(": " + xfer_state);
```

请参阅 GitHub 上的完整示例。

使用 ProgressListener 获取传输进度

您可以附加 Progresslistener 使用 转移 接口 `addProgressListener` 方法。

ProgressListener 只需要一个方法，即 `progressChanged`，此方法将采用 ProgressEvent 对象。您可以使用此对象获取操作的总字节数 (通过调用其 `getBytes` 方法) 和目前已传输的字节数 (通过调用 `getBytesTransferred`)。

导入

```
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.event.ProgressEvent;
import com.amazonaws.event.ProgressListener;
import com.amazonaws.services.s3.transfer.*;
import com.amazonaws.services.s3.transfer.Transfer.TransferState;

import java.io.File;
import java.util.ArrayList;
import java.util.Collection;
```

Code

```
File f = new File(file_path);
TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
try {
    Upload u = xfer_mgr.upload(bucket_name, key_name, f);
    // print an empty progress bar...
    printProgressBar(0.0);
    u.addProgressListener(new ProgressListener() {
        public void progressChanged(ProgressEvent e) {
            double pct = e.getBytesTransferred() * 100.0 / e.getBytes();
            eraseProgressBar();
            printProgressBar(pct);
        }
    });
    // block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(u);
```

```
    // print the final state of the transfer.
    TransferState xfer_state = u.getState();
    System.out.println(": " + xfer_state);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

请参阅 GitHub 上的完整示例。

获取子传输的进度

MultipleFileUpload 类可通过调用其 `getSubTransfers` 方法来返回有关其子传输的信息。它将返回Upload 对象的不可修改的 Collection，并单独提供每个子传输的传输状态和进度。

导入

```
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.event.ProgressEvent;
import com.amazonaws.event.ProgressListener;
import com.amazonaws.services.s3.transfer.*;
import com.amazonaws.services.s3.transfer.Transfer.TransferState;

import java.io.File;
import java.util.ArrayList;
import java.util.Collection;
```

Code

```
Collection<? extends Upload> sub_xfers = new ArrayList<Upload>();
sub_xfers = multi_upload.getSubTransfers();

do {
    System.out.println("\nSubtransfer progress:\n");
    for (Upload u : sub_xfers) {
        System.out.println("  " + u.getDescription());
        if (u.isDone()) {
            TransferState xfer_state = u.getState();
            System.out.println("  " + xfer_state);
        } else {
            TransferProgress progress = u.getProgress();
            double pct = progress.getPercentTransferred();
            printProgressBar(pct);
            System.out.println();
        }
    }

    // wait a bit before the next update.
    try {
        Thread.sleep(200);
    } catch (InterruptedException e) {
        return;
    }
} while (multi_upload.isDone() == false);
// print the final state of the transfer.
TransferState xfer_state = multi_upload.getState();
System.out.println("\nMultipleFileUpload " + xfer_state);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon S3 Developer Guide 中的 对象键

配置 Amazon S3 网站

您可以配置 Amazon S3 存储桶，使其具有与网站类似的行为。要执行此操作，您需要设置其网站配置。

Note

这些代码示例假定您了解使用适用于 Java 的 AWS 开发工具包 (p. 19)中的内容，并且已使用设置用于开发的 AWS 凭证和区域 (p. 6)中的信息配置默认 AWS 凭证。

设置存储桶的网站配置

要设置 Amazon S3 存储桶的网站配置，请使用要设置其配置的存储桶名称，以及包含存储桶网站配置的 BucketWebsiteConfiguration 对象，来调用 AmazonS3 的 `setWebsiteConfiguration` 方法。

设置索引文档是必需的；所有其他参数都是可选的。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.BucketWebsiteConfiguration;
```

Code

```
    String bucket_name, String index_doc, String error_doc) {
BucketWebsiteConfiguration website_config = null;

if (index_doc == null) {
    website_config = new BucketWebsiteConfiguration();
} else if (error_doc == null) {
    website_config = new BucketWebsiteConfiguration(index_doc);
} else {
    website_config = new BucketWebsiteConfiguration(index_doc, error_doc);
}

final AmazonS3 s3 =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    s3.setBucketWebsiteConfiguration(bucket_name, website_config);
} catch (AmazonServiceException e) {
    System.out.format(
            "Failed to set website configuration for bucket '%s'!\n",
            bucket_name);
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

Note

设置网站配置不会修改您的存储桶的访问权限。要使您的文件在 Web 上可见，您还需要设置一个存储桶策略，允许对存储桶中文件的公共读取访问权限。有关更多信息，请参阅使用存储桶策略管理对 Amazon S3 存储桶的访问 (p. 125)。

请参阅 GitHub 上的完整示例。

获取存储桶的网站配置

要获取 Amazon S3 存储桶的网站配置，请使用要获取其配置的存储桶的名称来调用 AmazonS3 客户端的 `getWebsiteConfiguration` 方法。

将以 BucketWebsiteConfiguration 对象的形式返回配置。如果该存储桶没有网站配置，则会返回 `null`。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.BucketWebsiteConfiguration;
```

Code

```
final AmazonS3 s3 =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    BucketWebsiteConfiguration config =
            s3.getBucketWebsiteConfiguration(bucket_name);
    if (config == null) {
        System.out.println("No website configuration found!");
    } else {
        System.out.format("Index document: %s\n",
                config.getIndexDocumentSuffix());
        System.out.format("Error document: %s\n",
                config.getErrorDocument());
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.out.println("Failed to get website configuration!");
    System.exit(1);
}
```

请参阅 GitHub 上的完整示例。

删除存储桶的网站配置

要删除 Amazon S3 存储桶的网站配置，请使用要从中删除配置的存储桶的名称来调用 AmazonS3 的 `deleteWebsiteConfiguration` 方法。

导入

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
```

Code

```
final AmazonS3 s3 =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

```
try {
    s3.deleteBucketWebsiteConfiguration(bucket_name);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.out.println("Failed to delete website configuration!");
    System.exit(1);
}
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon S3 API Reference 中的 PUT 存储桶网站
- Amazon S3 API Reference 中的 GET 存储桶网站
- Amazon S3 API Reference 中的 DELETE 存储桶网站

使用 Amazon S3 客户端加密

使用 Amazon S3 加密客户端加密数据是您可以用于为存储在 Amazon S3 中的敏感信息提供一层额外保护的一种方法。此部分中的示例演示如何为您的应用程序创建和配置 Amazon S3 加密客户端。如果您是首次使用加密，请参阅 AWS KMS Developer Guide 中的加密基础知识，大致了解加密术语和加密算法。

> **Note**
>
> 这些代码示例假定您了解使用适用于 Java 的 AWS 开发工具包 (p. 19)中的内容，并且已使用设置用于开发的 AWS 凭证和区域 (p. 6)中的信息配置默认 AWS 凭证。

主题

- Amazon S3 客户端加密配合客户端主密钥 (p. 138)
- Amazon S3 客户端加密配合 AWS KMS 托管密钥 (p. 142)

要了解有关所有 AWS 开发工具包的加密支持的信息，请参阅 Amazon Web Services General Reference 中的 Amazon S3 客户端加密的 AWS 开发工具包支持。

Amazon S3 客户端加密配合客户端主密钥

以下示例使用 AmazonS3EncryptionClientBuilder 类创建启用客户端加密的 Amazon S3 客户端。启用后，您使用此客户端上传到 Amazon S3 的任何对象都将加密。您使用此客户端从 Amazon S3 获取的任何对象都将自动解密。

> **Note**
>
> 以下示例演示如何配合使用 Amazon S3 客户端加密和客户托管的客户端主密钥。要了解如何配合使用加密和 AWS KMS 托管密钥，请参阅 Amazon S3 客户端加密配合 AWS KMS 托管密钥 (p. 142)。

启用客户端 Amazon S3 加密时，您可以从三种加密模式中进行选择：仅加密、经身份验证和经严格身份验证。以下部分说明了如何启用每种类型。要了解每种模式使用哪种算法，请参阅 CryptoMode 定义。

必需的导入

为这些示例导入以下类。

导入

```
import com.amazonaws.regions.Region;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.AmazonS3Encryption;
import com.amazonaws.services.s3.AmazonS3EncryptionClientBuilder;
import com.amazonaws.services.s3.model.*;

import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
```

仅加密模式

如果未指定 `CryptoMode`，则默认为仅加密模式。要启用加密，您必须将一个密钥传递到 EncryptionMaterials 构造函数。以下示例使用 `KeyGenerator` Java 类生成对称私有密钥。

Code

```
public void encryptionOnly_RangeGet_CustomerManagedKey() throws NoSuchAlgorithmException {
    SecretKey secretKey = KeyGenerator.getInstance("AES").generateKey();
    AmazonS3Encryption s3Encryption = AmazonS3EncryptionClientBuilder
            .standard()
            .withRegion(Regions.US_WEST_2)
            .withCryptoConfiguration(new CryptoConfiguration(CryptoMode.EncryptionOnly))
            .withEncryptionMaterials(new StaticEncryptionMaterialsProvider(new
 EncryptionMaterials(secretKey)))
            .build();

    s3Encryption.putObject(BUCKET_NAME, ENCRYPTED_KEY, "some contents");
    System.out.println(s3Encryption.getObject(new GetObjectRequest(BUCKET_NAME,
 ENCRYPTED_KEY)
            .withRange(0, 2)));
}
```

要使用非对称密钥或密钥对，只需将该密钥对传递到同一 EncryptionMaterials 类即可。以下示例使用 `KeyPairGenerator` 类生成对称密钥对。

Code

```
KeyPair keyPair = KeyPairGenerator.getInstance("RSA").generateKeyPair();
AmazonS3Encryption s3Encryption = AmazonS3EncryptionClientBuilder
        .standard()
        .withRegion(Regions.US_WEST_2)
        .withCryptoConfiguration(new CryptoConfiguration(CryptoMode.EncryptionOnly))
        .withEncryptionMaterials(new StaticEncryptionMaterialsProvider(new
 EncryptionMaterials(keyPair)))
        .build();

AmazonS3 s3NonEncrypt =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

对 Amazon S3 加密客户端调用 `putObject` 方法以上传对象。

Code

```
s3Encryption.putObject(BUCKET_NAME, ENCRYPTED_KEY, "some contents");
```

```
s3NonEncrypt.putObject(BUCKET_NAME, NON_ENCRYPTED_KEY, "some other contents");
```

您可以使用同一个客户端检索该对象。此示例调用 `getObjectAsString` 方法以检索存储的字符串。

Code

```
System.out.println(s3Encryption.getObjectAsString(BUCKET_NAME, ENCRYPTED_KEY));
System.out.println(s3Encryption.getObjectAsString(BUCKET_NAME, NON_ENCRYPTED_KEY));
```

请参阅 GitHub 上的完整示例。

经身份验证加密模式

使用 `AuthenticatedEncryption` 模式时，在加密期间会应用改进的密钥包装算法。在此模式下解密时，该算法会验证已解密对象的完整性，如果检查失败，则引发异常。有关经身份验证加密模式工作原理的更多详细信息，请参阅 Amazon S3 客户端经身份验证加密博客文章。

> **Note**
>
> 要使用客户端经身份验证加密，您必须将最新的 Bouncy Castle jar 文件加入应用程序的类路径中。

要启用此模式，请在 `withCryptoConfiguration` 方法中指定 `AuthenticatedEncryption` 值。

Code

```
SecretKey secretKey = KeyGenerator.getInstance("AES").generateKey();
AmazonS3Encryption s3Encryption = AmazonS3EncryptionClientBuilder
        .standard()
        .withRegion(Regions.US_WEST_2)
        .withCryptoConfiguration(new
 CryptoConfiguration(CryptoMode.AuthenticatedEncryption))
        .withEncryptionMaterials(new StaticEncryptionMaterialsProvider(new
 EncryptionMaterials(secretKey)))
        .build();

AmazonS3 s3NonEncrypt =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

`AuthenticatedEncryption` 模式可检索未加密对象以及使用 `EncryptionOnly` 模式加密的对象。以下示例显示检索未加密对象的 Amazon S3 加密客户端。

Code

```
public void authenticatedEncryption_CustomerManagedKey() throws NoSuchAlgorithmException {
    SecretKey secretKey = KeyGenerator.getInstance("AES").generateKey();
    AmazonS3Encryption s3Encryption = AmazonS3EncryptionClientBuilder
            .standard()
            .withRegion(Regions.US_WEST_2)
            .withCryptoConfiguration(new
 CryptoConfiguration(CryptoMode.AuthenticatedEncryption))
            .withEncryptionMaterials(new StaticEncryptionMaterialsProvider(new
 EncryptionMaterials(secretKey)))
            .build();

    AmazonS3 s3NonEncrypt =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();

    s3Encryption.putObject(BUCKET_NAME, ENCRYPTED_KEY, "some contents");
    s3NonEncrypt.putObject(BUCKET_NAME, NON_ENCRYPTED_KEY, "some other contents");
    System.out.println(s3Encryption.getObjectAsString(BUCKET_NAME, ENCRYPTED_KEY));
    System.out.println(s3Encryption.getObjectAsString(BUCKET_NAME, NON_ENCRYPTED_KEY));
```

```
}
```

请参阅 GitHub 上的完整示例。

经严格身份验证加密

要启用此模式，请在 `withCryptoConfiguration` 方法中指定 `StrictAuthenticatedEncryption` 值。

> **Note**
>
> 要使用客户端经身份验证加密，您必须将最新的 Bouncy Castle jar 文件加入应用程序的类路径中。

Code

```
SecretKey secretKey = KeyGenerator.getInstance("AES").generateKey();
AmazonS3Encryption s3Encryption = AmazonS3EncryptionClientBuilder
        .standard()
        .withRegion(Regions.US_WEST_2)
        .withCryptoConfiguration(new
 CryptoConfiguration(CryptoMode.StrictAuthenticatedEncryption))
        .withEncryptionMaterials(new StaticEncryptionMaterialsProvider(new
 EncryptionMaterials(secretKey)))
        .build();

AmazonS3 s3NonEncrypt =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

在 `StrictAuthenticatedEncryption` 模式下，当 Amazon S3 客户端检索到未使用经身份验证模式加密的对象时，会引发异常。

Code

```
public void strictAuthenticatedEncryption_CustomerManagedKey() throws
 NoSuchAlgorithmException {
    SecretKey secretKey = KeyGenerator.getInstance("AES").generateKey();
    AmazonS3Encryption s3Encryption = AmazonS3EncryptionClientBuilder
            .standard()
            .withRegion(Regions.US_WEST_2)
            .withCryptoConfiguration(new
 CryptoConfiguration(CryptoMode.StrictAuthenticatedEncryption))
            .withEncryptionMaterials(new StaticEncryptionMaterialsProvider(new
 EncryptionMaterials(secretKey)))
            .build();

    AmazonS3 s3NonEncrypt =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();

    s3Encryption.putObject(BUCKET_NAME, ENCRYPTED_KEY, "some contents");
    s3NonEncrypt.putObject(BUCKET_NAME, NON_ENCRYPTED_KEY, "some other contents");
    System.out.println(s3Encryption.getObjectAsString(BUCKET_NAME, ENCRYPTED_KEY));
    try {
        s3Encryption.getObjectAsString(BUCKET_NAME, NON_ENCRYPTED_KEY);
    } catch (SecurityException e) {
        // Strict authenticated encryption will throw an exception if an object is not
 encrypted with AES/GCM
        System.err.println(NON_ENCRYPTED_KEY + " was not encrypted with AES/GCM");
    }
}
```

请参阅 GitHub 上的完整示例。

Amazon S3 客户端加密配合 AWS KMS 托管密钥

以下示例使用 AmazonS3EncryptionClientBuilder 类创建启用客户端加密的 Amazon S3 客户端。配置后，您使用此客户端上传到 Amazon S3 的任何对象都将加密。您使用此客户端从 Amazon S3 获取的任何对象都将自动解密。

> **Note**
>
> 以下示例演示如何配合使用 Amazon S3 客户端加密和 AWS KMS 托管密钥。要了解如何配合使用加密和您自己的密钥，请参阅 Amazon S3 客户端加密配合客户端主密钥 (p. 138)。

启用客户端 Amazon S3 加密时，您可以从三种加密模式中进行选择：仅加密、经身份验证和经严格身份验证。以下部分说明了如何启用每种类型。要了解每种模式使用哪种算法，请参阅 CryptoMode 定义。

必需的导入

为这些示例导入以下类。

导入

```
import com.amazonaws.regions.Region;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.AmazonS3Encryption;
import com.amazonaws.services.s3.AmazonS3EncryptionClientBuilder;
import com.amazonaws.services.s3.model.*;

import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
```

仅加密模式

如果未指定 `CryptoMode`，则默认为仅加密模式。要使用 AWS KMS 托管密钥进行加密，请将 AWS KMS 密钥 ID 或别名传递到 KMSEncryptionMaterialsProvider 构造函数。

Code

```
AmazonS3Encryption s3Encryption = AmazonS3EncryptionClientBuilder
        .standard()
        .withRegion(Regions.US_WEST_2)
        .withCryptoConfiguration(new
 CryptoConfiguration(CryptoMode.EncryptionOnly).withAwsKmsRegion(Region.getRegion(Regions.US_WEST_2)))
        // Can either be Key ID or alias (prefixed with 'alias/')
        .withEncryptionMaterials(new KMSEncryptionMaterialsProvider("alias/s3-kms-key"))
        .build();

AmazonS3 s3NonEncrypt =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

对 Amazon S3 加密客户端调用 `putObject` 方法以上传对象。

Code

```
s3Encryption.putObject(BUCKET_NAME, ENCRYPTED_KEY, "some contents");
s3NonEncrypt.putObject(BUCKET_NAME, NON_ENCRYPTED_KEY, "some other contents");
```

您可以使用同一个客户端检索该对象。此示例调用 `getObjectAsString` 方法以检索存储的字符串。

Code

```
System.out.println(s3Encryption.getObjectAsString(BUCKET_NAME, ENCRYPTED_KEY));
System.out.println(s3Encryption.getObjectAsString(BUCKET_NAME, NON_ENCRYPTED_KEY));
```

请参阅 GitHub 上的完整示例。

经身份验证加密模式

使用 `AuthenticatedEncryption` 模式时，在加密期间会应用改进的密钥包装算法。在此模式下解密时，该算法会验证已解密对象的完整性，如果检查失败，则引发异常。有关经身份验证加密模式工作原理的更多详细信息，请参阅 Amazon S3 客户端经身份验证加密博客文章。

> **Note**
>
> 要使用客户端经身份验证加密，您必须将最新的 Bouncy Castle jar 文件加入应用程序的类路径中。

要启用此模式，请在 `withCryptoConfiguration` 方法中指定 `AuthenticatedEncryption` 值。

Code

```
AmazonS3Encryption s3Encryption = AmazonS3EncryptionClientBuilder
        .standard()
        .withRegion(Regions.US_WEST_2)
        .withCryptoConfiguration(new
 CryptoConfiguration(CryptoMode.AuthenticatedEncryption).withAwsKmsRegion(Region.getRegion(Regions.US_W
        // Can either be Key ID or alias (prefixed with 'alias/')
        .withEncryptionMaterials(new KMSEncryptionMaterialsProvider("alias/s3-kms-key"))
        .build();

AmazonS3 s3NonEncrypt =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

`AuthenticatedEncryption` 模式可检索未加密对象以及使用 `EncryptionOnly` 模式加密的对象。以下示例显示检索未加密对象的 Amazon S3 加密客户端。

Code

```
s3Encryption.putObject(BUCKET_NAME, ENCRYPTED_KEY, "some contents");
s3NonEncrypt.putObject(BUCKET_NAME, NON_ENCRYPTED_KEY, "some other contents");
System.out.println(s3Encryption.getObjectAsString(BUCKET_NAME, ENCRYPTED_KEY));
System.out.println(s3Encryption.getObjectAsString(BUCKET_NAME, NON_ENCRYPTED_KEY));
```

请参阅 GitHub 上的完整示例。

经严格身份验证加密

要启用此模式，请在 `withCryptoConfiguration` 方法中指定 `StrictAuthenticatedEncryption` 值。

> **Note**
>
> 要使用客户端经身份验证加密，您必须将最新的 Bouncy Castle jar 文件加入应用程序的类路径中。

Code

```
AmazonS3Encryption s3Encryption = AmazonS3EncryptionClientBuilder
        .standard()
```

```
        .withRegion(Regions.US_WEST_2)
        .withCryptoConfiguration(new
 CryptoConfiguration(CryptoMode.StrictAuthenticatedEncryption).withAwsKmsRegion(Region.getRegion(Region
        // Can either be Key ID or alias (prefixed with 'alias/')
        .withEncryptionMaterials(new KMSEncryptionMaterialsProvider("alias/s3-kms-key"))
        .build();

AmazonS3 s3NonEncrypt =
 AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

对 Amazon S3 加密客户端调用 `putObject` 方法以上传对象。

Code

```
s3Encryption.putObject(BUCKET_NAME, ENCRYPTED_KEY, "some contents");
s3NonEncrypt.putObject(BUCKET_NAME, NON_ENCRYPTED_KEY, "some other contents");
```

在 `StrictAuthenticatedEncryption` 模式下，当 Amazon S3 客户端检索到未使用经身份验证模式加密的对象时，会引发异常。

Code

```
try {
    s3Encryption.getObjectAsString(BUCKET_NAME, NON_ENCRYPTED_KEY);
} catch (SecurityException e) {
    // Strict authenticated encryption will throw an exception if an object is not
 encrypted with AES/GCM
    System.err.println(NON_ENCRYPTED_KEY + " was not encrypted with AES/GCM");
}
```

请参阅 GitHub 上的完整示例。

使用 AWS SDK for Java 的 Amazon SQS示例

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

> **Note**
>
> 该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

与 Amazon SQS 消息队列

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

请使用 AmazonSQS 客户端的 `createQueue` 方法，它提供一个描述队列参数的 CreateQueueRequest 对象。

导入

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.AmazonSQSException;
import com.amazonaws.services.sqs.model.CreateQueueRequest;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
CreateQueueRequest create_request = new CreateQueueRequest(QUEUE_NAME)
        .addAttributesEntry("DelaySeconds", "60")
        .addAttributesEntry("MessageRetentionPeriod", "86400");

try {
    sqs.createQueue(create_request);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}
```

您可以使用 `createQueue` 的简化形式，这只需要队列名称即可创建标准队列。

```
sqs.createQueue("MyQueue" + new Date().getTime());
```

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，可调用 AmazonSQS 客户端的 `listQueues` 方法。

导入

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.ListQueuesResult;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
ListQueuesResult lq_result = sqs.listQueues();
System.out.println("Your SQS Queue URLs:");
for (String url : lq_result.getQueueUrls()) {
    System.out.println(url);
}
```

使用 `listQueues` 重载 (不带任何参数) 将返回所有队列。您可以通过向其传递一个 `ListQueuesRequest` 对象来筛选返回的结果。

导入

```
import com.amazonaws.services.sqs.AmazonSQS;
```

```
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.ListQueuesRequest;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
String name_prefix = "Queue";
lq_result = sqs.listQueues(new ListQueuesRequest(name_prefix));
System.out.println("Queue URLs with prefix: " + name_prefix);
for (String url : lq_result.getQueueUrls()) {
    System.out.println(url);
}
```

请参阅 GitHub 上的完整示例。

获取队列的 URL

调用 AmazonSQS 客户端的 `getQueueUrl` 方法。

导入

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
String queue_url = sqs.getQueueUrl(QUEUE_NAME).getQueueUrl();
```

请参阅 GitHub 上的完整示例。

删除队列

向 AmazonSQS 客户端的 `deleteQueue` 方法提供队列的 URL (p. 146)。

导入

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
sqs.deleteQueue(queue_url);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon SQS Developer Guide 中的 Amazon SQS 队列的工作方式
- 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

本主题描述了如何发送、接收和删除 Amazon SQS 消息。始终使用 SQS 队列 (p. 144)发送消息。

发送消息

通过调用 AmazonSQS 客户端的 `sendMessage` 方法，将单个消息添加到 Amazon SQS 队列。提供包含该队列的 URL、消息正文和可选延迟值 (以秒为单位) 的 SendMessageRequest (p. 146) 对象。

导入

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.SendMessageRequest;
```

Code

```
SendMessageRequest send_msg_request = new SendMessageRequest()
        .withQueueUrl(queueUrl)
        .withMessageBody("hello world")
        .withDelaySeconds(5);
sqs.sendMessage(send_msg_request);
```

请参阅 GitHub 上的完整示例。

一次性发送多条消息

您可以在一个请求中发送多条消息。要发送多条消息，可使用 AmazonSQS 客户端的 `sendMessageBatch` 方法，此方法采用 SendMessageBatchRequest，后者包含队列 URL 和要发送的消息列表（每条消息对应一个 SendMessageBatchRequestEntry）。您也可以为每条消息设置一个可选延迟值。

导入

```
import com.amazonaws.services.sqs.model.SendMessageBatchRequest;
import com.amazonaws.services.sqs.model.SendMessageBatchRequestEntry;
```

Code

```
SendMessageBatchRequest send_batch_request = new SendMessageBatchRequest()
        .withQueueUrl(queueUrl)
        .withEntries(
                new SendMessageBatchRequestEntry(
                        "msg_1", "Hello from message 1"),
                new SendMessageBatchRequestEntry(
                        "msg_2", "Hello from message 2")
                        .withDelaySeconds(10));
sqs.sendMessageBatch(send_batch_request);
```

请参阅 GitHub 上的完整示例。

接收消息

可通过调用 AmazonSQS 客户端的 `receiveMessage` 方法（这将为其传递队列的 URL）来检索当前位于队列中的任何消息。消息将作为一系列 Message 对象返回。

导入

```
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.AmazonSQSException;
import com.amazonaws.services.sqs.model.SendMessageBatchRequest;
```

Code

```
List<Message> messages = sqs.receiveMessage(queueUrl).getMessages();
```

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 AmazonSQS 客户端的 `deleteMessage` 方法来从队列中删除消息。

Code

```
for (Message m : messages) {
    sqs.deleteMessage(queueUrl, m.getReceiptHandle());
}
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon SQS Developer Guide 中的 Amazon SQS 队列的工作方式
- 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

启用长时间轮询 Amazon SQS 消息队列

默认情况下，Amazon SQS 使用短轮询，此时仅查询服务器的一个子集（基于加权随机分布），以确定是否有任何消息可包含在响应中。

长轮询有助于降低使用 Amazon SQS 的费用，它可在答复发送到 Amazon SQS 队列的 ReceiveMessage 请求时，减少因没有消息可返回而造成的空响应数，还可消除假的空响应。

> **Note**
>
> 您可以设置 1 到 20 秒的长轮询频率。

创建队列时启用长轮询

要在创建 Amazon SQS 队列时启用长轮询，请设置 CreateQueueRequest 对象的 `ReceiveMessageWaitTimeSeconds` 属性，然后再调用 AmazonSQS 类的 `createQueue` 方法。

导入

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.AmazonSQSException;
import com.amazonaws.services.sqs.model.CreateQueueRequest;
```

Code

```
final AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();

// Enable long polling when creating a queue
CreateQueueRequest create_request = new CreateQueueRequest()
        .withQueueName(queue_name)
        .addAttributesEntry("ReceiveMessageWaitTimeSeconds", "20");

try {
    sqs.createQueue(create_request);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}
```

请参阅 GitHub 上的完整示例。

在现有队列上启用长轮询

除了在创建队列时启用长轮询之外，您也可以通过在 SetQueueAttributesRequest 上设置 `ReceiveMessageWaitTimeSeconds`，然后再调用 AmazonSQS 类的 `setQueueAttributes` 方法，在现有队列上启用长轮询。

导入

```
import com.amazonaws.services.sqs.model.SetQueueAttributesRequest;
```

Code

```
SetQueueAttributesRequest set_attrs_request = new SetQueueAttributesRequest()
        .withQueueUrl(queue_url)
        .addAttributesEntry("ReceiveMessageWaitTimeSeconds", "20");
sqs.setQueueAttributes(set_attrs_request);
```

请参阅 GitHub 上的完整示例。

在接收消息时启用长轮询

您可以在接收消息时启用长轮询，方法是在您提供给 AmazonSQS 类的 `receiveMessage` 方法的 ReceiveMessageRequest 中设置等待时间（以秒为单位）。

> **Note**
>
> 您应确保 AWS 客户端的请求超时时间大于最大长轮询时间 (20 秒)，以确保您的 `receiveMessage` 请求在等待下一轮询事件时不会超时！

导入

```
import com.amazonaws.services.sqs.model.ReceiveMessageRequest;
```

Code

```
ReceiveMessageRequest receive_request = new ReceiveMessageRequest()
        .withQueueUrl(queue_url)
        .withWaitTimeSeconds(20);
sqs.receiveMessage(receive_request);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon SQS Developer Guide 中的 Amazon SQS 长轮询
- Amazon SQS API Reference 中的 CreateQueue
- Amazon SQS API Reference 中的 ReceiveMessage
- Amazon SQS API Reference 中的 SetQueueAttributes

设置可见性超时 Amazon SQS

为了确保消息接收，在 Amazon SQS 中收到的消息会保留在队列中，直到被删除。在指定的可见性超时时间后，已接收但未删除的消息将可以在后续请求中使用，以帮助防止在对消息进行处理和删除之前重复接收消息。

> Note
>
> 使用标准队列时，可见性超时无法保证不会接收消息两次。如果您使用的是标准队列，请确保您的代码能够处理多次收到同一条消息的情况。

为单个消息设置消息可见性超时

当您收到消息时，您可以通过在 ChangeMessageVisibilityRequest 中将消息的接收句柄传递到 AmazonSQS 类的 `changeMessageVisibility` 方法来修改其可见性超时。

导入

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();

// Get the receipt handle for the first message in the queue.
String receipt = sqs.receiveMessage(queue_url)
                    .getMessages()
                    .get(0)
                    .getReceiptHandle();

sqs.changeMessageVisibility(queue_url, receipt, timeout);
```

请参阅 GitHub 上的完整示例。

一次性为多条消息设置的消息可见性超时

要一次性设置多条消息的可见性超时，请创建 ChangeMessageVisibilityBatchRequestEntry 对象的列表，每个对象包含唯一的 ID 和接收句柄。然后将该列表传递给 Amazon SQS 客户端类的 `changeMessageVisibilityBatch` 方法。

导入

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.ChangeMessageVisibilityBatchRequestEntry;
import java.util.ArrayList;
import java.util.List;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();

List<ChangeMessageVisibilityBatchRequestEntry> entries =
    new ArrayList<ChangeMessageVisibilityBatchRequestEntry>();

entries.add(new ChangeMessageVisibilityBatchRequestEntry(
            "unique_id_msg1",
            sqs.receiveMessage(queue_url)
               .getMessages()
               .get(0)
               .getReceiptHandle())
        .withVisibilityTimeout(timeout));

entries.add(new ChangeMessageVisibilityBatchRequestEntry(
            "unique_id_msg2",
            sqs.receiveMessage(queue_url)
               .getMessages()
               .get(0)
               .getReceiptHandle())
        .withVisibilityTimeout(timeout + 200));

sqs.changeMessageVisibilityBatch(queue_url, entries);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon SQS Developer Guide 中的可见性超时
- Amazon SQS API Reference 中的 SetQueueAttributes
- Amazon SQS API Reference 中的 GetQueueAttributes
- Amazon SQS API Reference 中的 ReceiveMessage
- Amazon SQS API Reference 中的 ChangeMessageVisibility
- Amazon SQS API Reference 中的 ChangeMessageVisibilityBatch

在 Amazon SQS 中使用死信队列

Amazon SQS 支持死信队列。死信队列是其他（源）队列可将其作为无法成功处理的消息的目标的队列。您可以在死信队列中留出和隔离这些消息以确定其处理失败的原因。

创建死信队列

死信队列的创建方式与常规队列相同，但有以下限制：

- 死信队列必须与源队列属于相同的队列类型 (FIFO 或标准)。
- 死信队列必须与源队列使用相同的 AWS 账户和区域创建。

在这里，我们创建两个相同的 Amazon SQS 队列，其中一个用作死信队列：

导入

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.AmazonSQSException;
```

Code

```
final AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();

// Create source queue
try {
    sqs.createQueue(src_queue_name);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}

// Create dead-letter queue
try {
    sqs.createQueue(dl_queue_name);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}
```

请参阅 GitHub 上的完整示例。

为源队列指定死信队列

要指定死信队列，您必须先创建一个重新驱动策略，然后在队列属性中设置该策略。重新驱动策略以 JSON 指定，并指定死信队列的 ARN，以及在向死信队列发送消息之前，允许接收但不处理消息的最大次数。

要为源队列设置重新驱动策略，请使用 SetQueueAttributesRequest 对象调用 AmazonSQS 类的 `setQueueAttributes` 方法，并使用您的 JSON 重新驱动策略为该对象设置 `RedrivePolicy` 属性。

导入

```
import com.amazonaws.services.sqs.model.GetQueueAttributesRequest;
import com.amazonaws.services.sqs.model.GetQueueAttributesResult;
import com.amazonaws.services.sqs.model.SetQueueAttributesRequest;
```

Code

```
String dl_queue_url = sqs.getQueueUrl(dl_queue_name)
                         .getQueueUrl();

GetQueueAttributesResult queue_attrs = sqs.getQueueAttributes(
        new GetQueueAttributesRequest(dl_queue_url)
            .withAttributeNames("QueueArn"));

String dl_queue_arn = queue_attrs.getAttributes().get("QueueArn");

// Set dead letter queue with redrive policy on source queue.
String src_queue_url = sqs.getQueueUrl(src_queue_name)
                          .getQueueUrl();

SetQueueAttributesRequest request = new SetQueueAttributesRequest()
        .withQueueUrl(src_queue_url)
        .addAttributesEntry("RedrivePolicy",
                "{\"maxReceiveCount\":\"5\", \"deadLetterTargetArn\":\""
                + dl_queue_arn + "\"}");

sqs.setQueueAttributes(request);
```

请参阅 GitHub 上的完整示例。

更多信息

- Amazon SQS Developer Guide 中的使用 Amazon SQS 死信队列
- Amazon SQS API Reference 中的 SetQueueAttributes

使用 AWS SDK for Java 的 Amazon SWF示例

Amazon SWF 是一项工作流管理服务，可帮助开发人员构建和扩展分布式工作流，这些工作流可具有包含活动、子工作流或 Lambda 任务的并行或顺序步骤。

通过AWS SDK for Java使用 Amazon SWF 有两种方法：使用 SWF client 对象或者使用 AWS Flow Framework for Java。AWS Flow Framework for Java 的初始配置难度更大，因为它使用了大量批注并依赖其他库，例如 AspectJ 和 Spring Framework。但对于大型项目或复杂项目，您可使用 AWS Flow Framework for Java 来节省编写代码的时间。有关更多信息，请参阅适用于 Java 的 AWS Flow Framework 开发人员指南。

此部分提供了直接使用AWS SDK for Java客户端来为 Amazon SWF 编程的示例。

主题

Amazon SWF 基本知识

这些是通过 Amazon SWF 使用AWS SDK for Java的一般模式。这意味着它主要用于参考。有关更完整的介绍性教程，请参阅构建简单 Amazon SWF 应用程序 (p. 154)。

Dependencies

基本 Amazon SWF 应用程序将需要AWS SDK for Java附带的以下依赖项：

- aws-java-sdk-1.11.*.jar
- commons-logging-1.1.*.jar
- httpclient-4.3.*.jar
- httpcore-4.3.*.jar
- jackson-annotations-2.5.*.jar
- jackson-core-2.5.*.jar
- jackson-databind-2.5.*.jar
- joda-time-2.8.*.jar

> **Note**
>
> 虽然这些程序包的版本号将因您拥有的开发工具包版本而异，但开发工具包附带的版本已经过兼容性测试，并且您应使用这些版本。

AWS Flow Framework for Java 应用程序需要其他设置和 其他依赖项。有关使用框架的更多信息，请参阅 AWS Flow Framework for Java 开发人员指南。

Imports

通常，您可以将以下导入用于代码开发：

```
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.*;
```

不过，好的做法是仅导入您所需的类。您可能最终会在 `com.amazonaws.services.simpleworkflow.model` 工作区中指定特定的类：

```
import com.amazonaws.services.simpleworkflow.model.PollForActivityTaskRequest;
import com.amazonaws.services.simpleworkflow.model.RespondActivityTaskCompletedRequest;
import com.amazonaws.services.simpleworkflow.model.RespondActivityTaskFailedRequest;
import com.amazonaws.services.simpleworkflow.model.TaskList;
```

如果您使用 AWS Flow Framework for Java，则将从 `com.amazonaws.services.simpleworkflow.flow` 工作区导入类。例如：。

```
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.flow.ActivityWorker;
```

> **Note**
>
> 除了AWS SDK for Java的基本要求外，AWS Flow Framework for Java 有额外要求。有关更多信息，请参阅适用于 Java 的 AWS Flow Framework 开发人员指南。

使用 SWF 客户端类

您通过 AmazonSimpleWorkflowClient 或 AmazonSimpleWorkflowAsyncClient 类与 Amazon SWF 进行基本交互。二者之间的主要差异是，`*AsyncClient` 类返回 Future 对象以进行并发 (异步) 编程。

```
AmazonSimpleWorkflowClient swf = AmazonSimpleWorkflowClientBuilder.defaultClient();
```

建立简单 Amazon SWF 应用

此主题将向您介绍使用AWS SDK for Java，编写 Amazon SWF 应用程序，并在此过程中提供了一些重要概念。

关于示例

示例项目将创建带有一个活动的工作流程，接受通过 AWS 云传递的工作流程数据（在 HelloWorld 的传统中，这应该是要问候的某个人的名称）并在响应中输出问候语。

虽然表面上看起来这非常简单，不过 Amazon SWF 应用程序由多个协同工作的部件组成：

- 一个域，用作工作流执行数据的逻辑容器。
- 一个或多个工作流程，它们表示代码组件，这些组件定义工作流程的活动和子工作流程执行的逻辑顺序。
- 一个工作流工作线程，也称为决策程序，轮询决策任务并在响应中计划活动或子工作流。
- 一个或多个活动，每个活动表示工作流中的一个工作单元。
- 一个活动工作线程，轮询活动任务并在响应中运行活动方法。
- 一个或多个任务列表，这是由 Amazon SWF 维护的队列，用于发布请求到工作流和活动工作线程。任务列表上用于工作流工作线程的任务称为决策任务。用于活动工作线程的任务称为活动任务。

- 一个工作流启动程序，用于开始工作流的执行。

在后台，Amazon SWF 协调这些组件的操作，协调从 AWS 云的传输，在它们之间传递数据，处理超时和检测信号通知，以及记录工作流执行历史记录。

Prerequisites

开发环境

此教程中使用的开发环境包括：

- 适用于 Java 的 AWS 开发工具包。
- Apache Maven (3.3.1)。
- JDK 1.7 或更高版本。本教程使用 JDK 1.8.0 开发和测试。
- 一个适用的 Java 文本编辑器 (由您选择)。

> **Note**
>
> 如果您使用的编译系统不是 Maven，则仍可以使用适用于您环境的相应步骤创建项目，使用此处提供的概念来跟踪。入门 (p. 3)中提供了在不同编译系统中配置和使用AWS SDK for Java的更多信息。
> 与此类似，但需要更多工作，此处列出的步骤也可以使用支持 Amazon SWF 的任意 AWS 开发工具包实施。

所有必需的外部依赖项包括在 AWS SDK for Java 中，因此无需下载其他内容。

AWS 访问

要访问 Amazon Web Services (AWS)，您必须具有活动 AWS 账户。有关注册 AWS 和创建 IAM 用户的信息（建议通过使用根账户凭证），请参阅注册 AWS 并创建 IAM 用户 (p. 3)。

本教程使用终端 (命令行) 运行示例代码，要求您具有能够访问开发工具包的 AWS 凭证和配置。最简单的方法是使用环境变量 `AWS_ACCESS_KEY_ID` 和 `AWS_SECRET_ACCESS_KEY`。您还应设置 `AWS_REGION` 要使用的地区。

例如，在 Linux, OS X, or Unix 上，使用以下方法设置变量：

```
export AWS_ACCESS_KEY_ID=your_access_key_id
export AWS_SECRET_ACCESS_KEY=your_secret_access_key
export AWS_REGION=us-east-1
```

要在 Windows 上设置这些变量，请使用以下命令：

```
set AWS_ACCESS_KEY_ID=your_access_key_id
set AWS_SECRET_ACCESS_KEY=your_secret_access_key
set AWS_REGION=us-east-1
```

> **Important**
>
> 将此处显示的示例值替换为您自己的访问密钥、秘密访问密钥和区域信息。
> 有关为开发工具包配置凭证的更多信息，请参阅设置用于开发的 AWS 凭证和区域 (p. 6)。

创建 SWF 项目

1. 使用 Maven 启动新项目：

```
mvn archetype:generate -DartifactId=helloswf \
-DgroupId=aws.example.helloswf -DinteractiveMode=false
```

这将创建具有标准 maven 项目结构的新项目：

```
helloswf
### pom.xml
### src
    ### main
    #   ### java
    #       ### aws
    #           ### example
    #               ### helloswf
    #                   ### App.java
    ### test
        ### ...
```

您可以忽略或删除 `test` 目录及其中包含的所有内容，我们不会将其用于此教程。您还可以删除 `App.java`，因为我们将使用新类来替换它。

2. 编辑项目的 `pom.xml` 文件，通过将 aws-java-sdk-simpleworkflow 模块的依赖项添加到 `<dependencies>` 块中来添加该模块。

```
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-simpleworkflow</artifactId>
    <version>1.11.245</version>
  </dependency>
</dependencies>
```

3. 确保 Maven 使用 JDK 1.7+ 支持构建您的项目。将以下内容添加到您项目 (在 `<dependencies>` 块之前或之后) 的 `pom.xml` 中：

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.6.1</version>
      <configuration>
          <source>1.8</source>
          <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

编码项目

示例项目包括四个独立的应用程序，我们将逐个查看：

- HelloTypes.java — 包含项目的域、活动和工作流程类型数据，与其他组件共享。它还处理这些类型在 SWF 中的注册。
- ActivityWorker.java — 包含活动工作线程，将轮询活动任务并在响应中运行活动。
- WorkflowWorker.java — 包含工作流工作线程（决策程序），将轮询决策任务并计划新活动。
- WorkflowStarter.java — 包含工作流启动程序，将启动新的工作流执行，这将导致 SWF 开始生成决策和工作流任务供工作线程使用。

所有源文件的常见步骤

您创建的用于托管 Java 类的所有文件都有几个共同点。出于时间考虑，这些步骤在每次添加新文件到项目时是隐含的：

1. 在项目的 `src/main/java/example/swf/hello/` 目录中创建文件。
2. 添加 `package` 声明到每个文件的开头用于声明其命名空间。示例项目使用：

```
package aws.example.helloswf;
```

3. 添加 AmazonSimpleWorkflowClient 类以及 `com.amazonaws.services.simpleworkflow.model` 命名空间中多个类的 `import` 声明。为了简化操作，我们使用：

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.*;
```

注册域、工作流程和活动类型

我们将开始创建一个新的可执行类别，`HelloTypes.java`。此文件将包含您工作流程的不同部分的共享数据，例如活动和工作流类型的名称和版本、域名和任务列表名称。

1. 打开文本编辑器并创建文件 `HelloTypes.java`，添加程序包声明并根据通用步骤 (p. 157)导入。
2. 声明 `HelloTypes` 类并向其提供值，以供注册的活动和工作流类型使用：

```
    public static final String DOMAIN = "HelloDomain";
    public static final String TASKLIST = "HelloTasklist";
    public static final String WORKFLOW = "HelloWorkflow";
    public static final String WORKFLOW_VERSION = "1.0";
    public static final String ACTIVITY = "HelloActivity";
    public static final String ACTIVITY_VERSION = "1.0";
```

   这些值将在代码中使用。

3. 在字符串声明之后，创建 AmazonSimpleWorkflowClient 类的实例。这是由AWS SDK for Java向 Amazon SWF 方法提供的基本接口。

```
private static final AmazonSimpleWorkflow swf =

 AmazonSimpleWorkflowClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

4. 添加新函数以注册到 SWF 域。域 是多种相关 SWF 活动和工作流类型的逻辑容器。SWF 组件只有在位于同一个域中时才能彼此通信。

```
    try {
        System.out.println("** Registering the domain '" + DOMAIN + "'.");
        swf.registerDomain(new RegisterDomainRequest()
            .withName(DOMAIN)
            .withWorkflowExecutionRetentionPeriodInDays("1"));
    } catch (DomainAlreadyExistsException e) {
        System.out.println("** Domain already exists!");
    }
```

   在注册域时，您需要提供名称（不含 `:`、`/`、`|`、控制字符或文本字符串“arn”的 1 – 256 个字符组合）以及保留期，这是在工作流程执行完成后，Amazon SWF 保留工作流程执行历史记录数据的天数。最长的工作流执行保留期为 90 天。有关更多信息，请参阅 RegisterDomainRequest。

如果具有该名称的域已存在，则将引发 DomainAlreadyExistsException。因为我们并不关注是否已经创建了域，因此可以忽略此异常。

Note

此代码演示了使用AWS SDK for Java方法时的一个通用模式，方法的数据由 `simpleworkflow.model` 命名空间中的类提供，该命名空间使用可链接的 `.with*` 方法实例化和填充。

5. 添加函数以注册新活动类型。活动 表示工作流中的一个工作单元。

```
try {
    System.out.println("** Registering the activity type '" + ACTIVITY +
        "-" + ACTIVITY_VERSION + "'.");
    swf.registerActivityType(new RegisterActivityTypeRequest()
        .withDomain(DOMAIN)
        .withName(ACTIVITY)
        .withVersion(ACTIVITY_VERSION)
        .withDefaultTaskList(new TaskList().withName(TASKLIST))
        .withDefaultTaskScheduleToStartTimeout("30")
        .withDefaultTaskStartToCloseTimeout("600")
        .withDefaultTaskScheduleToCloseTimeout("630")
        .withDefaultTaskHeartbeatTimeout("10"));
} catch (TypeAlreadyExistsException e) {
    System.out.println("** Activity type already exists!");
}
```

活动类型由名称和版本标识，它们在所注册到的域中用于将活动与任何其他活动区分开。活动还包含多种可选参数，例如用于从 SWF 接收任务和数据的默认任务列表，以及您可用来对活动各个部分执行所用时长施加限制的不同超时。有关更多信息，请参阅 RegisterActivityTypeRequest。

Note

所有超时值以秒 为单位指定。有关超时如何影响工作流执行的完整说明，请参阅 Amazon SWF 超时类型。

如果您尝试注册的活动类型已存在，则将引发 TypeAlreadyExistsException。

6. 添加函数以注册新工作流类型。工作流程 也称为决策程序，表示工作流程执行的逻辑。

```
try {
    System.out.println("** Registering the workflow type '" + WORKFLOW +
        "-" + WORKFLOW_VERSION + "'.");
    swf.registerWorkflowType(new RegisterWorkflowTypeRequest()
        .withDomain(DOMAIN)
        .withName(WORKFLOW)
        .withVersion(WORKFLOW_VERSION)
        .withDefaultChildPolicy(ChildPolicy.TERMINATE)
        .withDefaultTaskList(new TaskList().withName(TASKLIST))
        .withDefaultTaskStartToCloseTimeout("30"));
} catch (TypeAlreadyExistsException e) {
    System.out.println("** Workflow type already exists!");
}
```

与活动类型类似，工作流类型由名称 和版本 标识，也具有可配置的超时。有关更多信息，请参阅 RegisterWorkflowTypeRequest。

如果您尝试注册的工作流程类型已存在，则将引发 TypeAlreadyExistsException。

7. 最后，请通过向类提供 `main` 方法确保其可执行，这反过来会注册域、活动类型和工作流类型：

```
registerDomain();
```

```
    registerWorkflowType();
    registerActivityType();
```

现在，您可以编译 (p. 164)并运行 (p. 165)应用程序来运行注册脚本，或者继续对活动和工作流工作线程编写代码。注册了域、工作流程和活动之后，您无需重新运行此步骤 — 这些内容将保留，直至您自行弃用它们。

实施活动工作线程

活动 是工作流中的基本工作单元。工作流提供逻辑、要运行的计划活动 (或要采取的其他操作) 来响应决策任务。典型的工作流通常包含多种活动，可以同步、异步或者以两种方式结合运行。

活动工作线程 是一段代码，轮询由 Amazon SWF 生成的活动任务来响应工作流决策。在收到活动任务时，它将运行对应的活动并将成功/失败响应返回到工作流。

我们将实施驱动单个活动的简单活动工作线程。

1. 打开文本编辑器并创建文件 `ActivityWorker.java`，添加程序包声明并根据通用步骤 (p. 157)导入。

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.*;
```

2. 向文件中添加 `ActivityWorker` 类，并向其提供数据成员以保存用来与 Amazon SWF 交互的 SWF 客户端：

```
    private static final AmazonSimpleWorkflow swf =

 AmazonSimpleWorkflowClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

3. 添加将用作活动的方法：

```
private static String sayHello(String input) throws Throwable {
    return "Hello, " + input + "!";
}
```

该活动就是获取字符串，将其组合到问候语中，然后返回结果。虽然此活动有很小的可能性会引发异常，但最好的做法是将活动设计为在出现问题时会引发错误。

4. 添加我们将用作活动任务轮询方法的 `main` 方法。我们首先添加一些代码来轮询任务列表中的活动任务：

```
        System.out.println("Polling for an activity task from the tasklist '"
                + HelloTypes.TASKLIST + "' in the domain '" +
                HelloTypes.DOMAIN + "'.");

        ActivityTask task = swf.pollForActivityTask(
            new PollForActivityTaskRequest()
                .withDomain(HelloTypes.DOMAIN)
                .withTaskList(
                    new TaskList().withName(HelloTypes.TASKLIST)));

        String task_token = task.getTaskToken();
```

活动通过调用 Amazon SWF 客户端的 `pollForActivityTask` 方法从 SWF 接收任务，指定在传入的 PollForActivityTaskRequest 中使用的域和任务列表。

一旦收到任务，我们将通过调用任务的 `getTaskToken` 方法来检索它的唯一标识符。

5. 接下来，写入一些代码来处理传入的任务。将以下内容添加到您的 `main` 方法，就在轮询任务和检索其任务令牌代码的后方。

```
    if (task_token != null) {
        String result = null;
        Throwable error = null;

        try {
            System.out.println("Executing the activity task with input '" +
                    task.getInput() + "'.");
            result = sayHello(task.getInput());
        } catch (Throwable th) {
            error = th;
        }

        if (error == null) {
            System.out.println("The activity task succeeded with result '"
                    + result + "'.");
            swf.respondActivityTaskCompleted(
                new RespondActivityTaskCompletedRequest()
                    .withTaskToken(task_token)
                    .withResult(result));
        } else {
            System.out.println("The activity task failed with the error '"
                    + error.getClass().getSimpleName() + "'.");
            swf.respondActivityTaskFailed(
                new RespondActivityTaskFailedRequest()
                    .withTaskToken(task_token)
                    .withReason(error.getClass().getSimpleName())
                    .withDetails(error.getMessage()));
        }
    }
```

如果任务令牌不是 `null`，则我们可以开始运行活动方法 (`sayHello`)，只要它具有随任务发送的输入数据。

如果任务成功（未生成错误），则工作线程使用包含任务令牌和活动结果数据的 RespondActivityTaskCompletedRequest 对象调用 SWF 客户端的 `respondActivityTaskCompleted` 方法，以此来响应 SWF。

另一方面，如果任务失败，则我们通过调用带有 RespondActivityTaskFailedRequest 对象的 `respondActivityTaskFailed` 方法，向其传递任务令牌和有关错误的信息，以此来响应。

> **Note**
>
> 如果终止，此活动不会正常关闭。虽然这超出了本教程的范围，不过在相关主题适当地关闭活动和工作流工作线程 (p. 169)中提供了此活动工作线程的替代实施方法。

实施工作流工作线程

您的工作流逻辑位于称为工作流工作线程的代码块中。工作流工作线程在工作流类型注册到的默认任务列表上，轮询域中 Amazon SWF 发送的决策任务。

工作流工作线程接收任务时，它会做出某种类型的决策 (通常为是否计划新活动) 并采取相应操作 (例如计划活动)。

1. 打开文本编辑器并创建文件 `WorkflowWorker.java`，添加程序包声明并根据通用步骤 (p. 157)导入。
2. 将一些额外的导入添加到文件中：

```
import com.amazonaws.regions.Regions;
```

```
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.*;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
```

3. 声明 `WorkflowWorker` 类，创建用于访问 SWF 方法的 AmazonSimpleWorkflowClient 类的实例。

```
    private static final AmazonSimpleWorkflow swf =

 AmazonSimpleWorkflowClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

4. 添加 `main` 方法。该方法持续循环，使用 SWF 客户端的 `pollForDecisionTask` 方法轮询决策任务。PollForDecisionTaskRequest 提供详细信息。

```
    PollForDecisionTaskRequest task_request =
        new PollForDecisionTaskRequest()
            .withDomain(HelloTypes.DOMAIN)
            .withTaskList(new TaskList().withName(HelloTypes.TASKLIST));

    while (true) {
        System.out.println(
                "Polling for a decision task from the tasklist '" +
                HelloTypes.TASKLIST + "' in the domain '" +
                HelloTypes.DOMAIN + "'.");

        DecisionTask task = swf.pollForDecisionTask(task_request);

        String taskToken = task.getTaskToken();
        if (taskToken != null) {
            try {
                executeDecisionTask(taskToken, task.getEvents());
            } catch (Throwable th) {
                th.printStackTrace();
            }
        }
    }
```

在收到任务之后，我们调用其 `getTaskToken` 方法，这会返回可用于标识任务的字符串。如果返回的令牌不是 `null`，则我们在 `executeDecisionTask` 方法中进一步处理它，向它传递随任务发送的任务令牌以及 HistoryEvent 对象的列表。

5. 添加 `executeDecisionTask` 方法，获取任务令牌 (`String`) 和 `HistoryEvent` 列表。

```
    List<Decision> decisions = new ArrayList<Decision>();
    String workflow_input = null;
    int scheduled_activities = 0;
    int open_activities = 0;
    boolean activity_completed = false;
    String result = null;
```

我们还可以设置一些数据成员来跟踪内容，例如：

- 用于报告任务处理结果的决策对象列表。
- 用于保存由“WorkflowExecutionStarted”事件提供的工作流程输入的字符串
- 已计划和打开 (正在运行) 活动的计数，用于避免再次计划已经计划或者当前正在运行的相同活动。
- 用于指示活动已完成的布尔值。
- 用于保存活动结果的字符串，以将其作为我们的工作流结果返回。

6. 接下来，添加一些代码到 `executeDecisionTask`，基于 `HistoryEvent` 方法报告的事件类型处理随任务发送的 `getEventType` 对象。

```
System.out.println("Executing the decision task for the history events: [");
for (HistoryEvent event : events) {
    System.out.println("  " + event);
    switch(event.getEventType()) {
        case "WorkflowExecutionStarted":
            workflow_input =
                event.getWorkflowExecutionStartedEventAttributes()
                     .getInput();
            break;
        case "ActivityTaskScheduled":
            scheduled_activities++;
            break;
        case "ScheduleActivityTaskFailed":
            scheduled_activities--;
            break;
        case "ActivityTaskStarted":
            scheduled_activities--;
            open_activities++;
            break;
        case "ActivityTaskCompleted":
            open_activities--;
            activity_completed = true;
            result = event.getActivityTaskCompletedEventAttributes()
                          .getResult();
            break;
        case "ActivityTaskFailed":
            open_activities--;
            break;
        case "ActivityTaskTimedOut":
            open_activities--;
            break;
    }
}
System.out.println("]");
```

对于我们的工作流，我们最感兴趣的是：

- “WorkflowExecutionStarted”事件，这指示工作流程执行已启动（通常意味着您应该运行工作流程中的第一个活动），并且这提供了初始输入（提供到工作流程中）。在这种情况下，这是我们问候语的名称部分，因此将其保存在字符串中以在计划活动运行时使用。
- “ActivityTaskCompleted”事件在计划的活动完成后立即发送。事件数据还包括已完成活动的返回值。因为我们仅有一个活动，我们将使用该值作为整个工作流程的结果。

其他事件类型在工作流需要时可以使用。有关各个事件类型的信息，请参阅 HistoryEvent 类说明。

> **Note**
>
> `switch` 语句中的字符串在 Java 7 中引入。如果您使用的是 Java 的较早版本，则可以使用 EventType 类将 `history_event.getType()` 返回的 `String` 转换为枚举值，然后可在需要时将其转换回 `String`：
>
> ```
> EventType et = EventType.fromValue(event.getEventType());
> ```

7. 在 `switch` 语句之后，添加更多代码，根据所收到的任务采用合适的决策 进行响应。

```
if (activity_completed) {
    decisions.add(
        new Decision()
            .withDecisionType(DecisionType.CompleteWorkflowExecution)
            .withCompleteWorkflowExecutionDecisionAttributes(
                new CompleteWorkflowExecutionDecisionAttributes()
```

```
                    .withResult(result)));
} else {
    if (open_activities == 0 && scheduled_activities == 0) {

        ScheduleActivityTaskDecisionAttributes attrs =
            new ScheduleActivityTaskDecisionAttributes()
                .withActivityType(new ActivityType()
                    .withName(HelloTypes.ACTIVITY)
                    .withVersion(HelloTypes.ACTIVITY_VERSION))
                .withActivityId(UUID.randomUUID().toString())
                .withInput(workflow_input);

        decisions.add(
                new Decision()
                    .withDecisionType(DecisionType.ScheduleActivityTask)
                    .withScheduleActivityTaskDecisionAttributes(attrs));
    } else {
        // an instance of HelloActivity is already scheduled or running. Do nothing,
 another
        // task will be scheduled once the activity completes, fails or times out
    }
}

System.out.println("Exiting the decision task with the decisions " + decisions);
```

- 如果尚未计划活动，我们使用 `ScheduleActivityTask` 决策进行响应，这在 ScheduleActivityTaskDecisionAttributes 结构中提供关于 Amazon SWF 接下来应计划的活动的信息，也包括 Amazon SWF 应发送到活动的任何数据。
- 如果活动已完成，则我们将考虑完成的整个工作流，并使用 `CompletedWorkflowExecution` 决策进行响应，填入 CompleteWorkflowExecutionDecisionAttributes 结构以提供有关已完成工作流的详细信息。在这种情况下，我们将返回活动的结果。

在任何一种情况下，决策信息将添加到在方法顶部声明的 `Decision` 列表。

8. 返回在处理任务时收集的 `Decision` 对象列表来完成决策任务。在我们所编写的 `executeDecisionTask` 方法尾部添加此代码：

```
swf.respondDecisionTaskCompleted(
    new RespondDecisionTaskCompletedRequest()
        .withTaskToken(taskToken)
        .withDecisions(decisions));
```

SWF 客户端的 `respondDecisionTaskCompleted` 方法获取标识任务的任务令牌以及 `Decision` 对象列表。

实施工作流启动程序

最后，我们将编写一些代码用于启动工作流程执行。

1. 打开文本编辑器并创建文件 `WorkflowStarter.java`，添加程序包声明并根据通用步骤 (p. 157)导入。
2. 添加 `WorkflowStarter` 类：

```
package aws.example.helloswf;


import com.amazonaws.regions.Regions;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.*;
```

```
public class WorkflowStarter {
    private static final AmazonSimpleWorkflow swf =

 AmazonSimpleWorkflowClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
    public static final String WORKFLOW_EXECUTION = "HelloWorldWorkflowExecution";

    public static void main(String[] args) {
        String workflow_input = "Amazon SWF";
        if (args.length > 0) {
            workflow_input = args[0];
        }

        System.out.println("Starting the workflow execution '" + WORKFLOW_EXECUTION +
                "' with input '" + workflow_input + "'.");

        WorkflowType wf_type = new WorkflowType()
            .withName(HelloTypes.WORKFLOW)
            .withVersion(HelloTypes.WORKFLOW_VERSION);

        Run run = swf.startWorkflowExecution(new StartWorkflowExecutionRequest()
            .withDomain(HelloTypes.DOMAIN)
            .withWorkflowType(wf_type)
            .withWorkflowId(WORKFLOW_EXECUTION)
            .withInput(workflow_input)
            .withExecutionStartToCloseTimeout("90"));

        System.out.println("Workflow execution started with the run id '" +
                run.getRunId() + "'.");
    }
}
```

`WorkflowStarter` 类包含一个方法 `main`，它获取命令行上传递的可选参数作为工作流的输入数据。

SWF 客户端方法 `startWorkflowExecution`，获取 StartWorkflowExecutionRequest 对象作为输入。此处，除了指定要运行的域和工作流类型之外，我们提供了：

- 便于阅读的工作流执行名称
- 工作流输入数据 (我们的示例中在命令行上提供)
- 超时值，以秒为单位，表示整个工作流运行所应使用的时长。

`startWorkflowExecution` 返回的运行对象提供了运行 ID，这是用于在 Amazon SWF 的工作流程执行历史记录中标识此特定工作流程执行的值。

> **Note**
>
> 运行 ID 由 Amazon SWF 生成，不同于 您在启动工作流执行时传入的工作流执行名称。

编译示例

要使用 Maven 编译示例项目，请转到 `helloswf` 目录并键入：

```
mvn package
```

生成的 `helloswf-1.0.jar` 将在 `target` 目录中生成。

运行示例

示例包括四个独立的可执行类，彼此独立运行。

Note

如果您使用的是 Linux, OS X, or Unix 系统，您可以在单个终端窗口中将它们全部逐个运行。如果您运行的是 Windows，则应该打开两个额外的命令行实例并分别导航到 `helloswf` 目录。

设置 Java 类路径

虽然 Maven 已经为您处理了依赖项来运行示例，您仍需要在 Java 类路径上提供 AWS 开发工具包库及其依赖项。您可以将 `CLASSPATH` 环境变量设置为 AWS 开发工具包库的位置，以及开发工具包中包括必要依赖项的 `third-party/lib` 目录：

```
export CLASSPATH='target/helloswf-1.0.jar:/path/to/sdk/lib/*:/path/to/sdk/third-party/lib/
*'
java example.swf.hello.HelloTypes
```

或者使用 **`java`** 命令的 `-cp` 选项在运行各个应用程序时设置类路径。

```
java -cp target/helloswf-1.0.jar:/path/to/sdk/lib/*:/path/to/sdk/third-party/lib/* \
  example.swf.hello.HelloTypes
```

您使用的样式由您决定。如果您在编译代码时没有问题，但在尝试运行示例时遇到一系列“NoClassDefFound”错误，则可能是因为类路径设置不正确。

注册域、工作流程和活动类型

在运行工作线程和工作流程启动程序之前，您需要注册域以及工作流程和活动类型。执行此操作的代码在注册域、工作流和活动类型 (p. 157)中实施。

在编译之后，如果您已设置 CLASSPATH (p. 165)，则可以通过执行以下命令运行注册代码：

```
    echo 'Supply the name of one of the example classes as an argument.'
```

启动活动和工作流工作线程

现在类型已注册，您可以启动活动和工作流工作线程。它们将持续运行并轮询任务，直至终止，因此您应该在单独终端窗口中运行它们，或者，如果您在 Linux, OS X, or Unix 上运行它们，则可以使用 `&` 运算符来使得它们中的每一个在运行时生成单独进程。

```
    echo 'If there are arguments to the class, put them in quotes after the class name.'
    exit 1
```

如果您在单独窗口中运行这些命令，则忽略每一行最后的 `&` 运算符。

启动工作流执行

现在正在轮询您的活动和工作流工作线程，您可以启动工作流执行。此进程将运行直至工作流返回已完成状态。您应在新终端窗口中运行它 (除非您使用 `&` 运算符将工作线程作为新生成的进程运行)。

```
fi
```

Note

如果您要提供自己的输入数据 (这将首先传递到工作流，然后传递到活动)，则将其添加到命令行中。例如：。

```
echo "## Running $className..."
```

一旦开始工作流执行，您应该开始查看这两种工作线程以及工作流执行本身提供的输出。工作流最终完成之后，其输出将显示在屏幕上。

此示例的完整源代码

您可以在 Github 的 aws-java-developer-guide 存储库中浏览此示例的完整源代码。

有关

- 如果在工作流轮询仍在进行时关闭此处提供的工作线程，则它们会导致任务丢失。要了解如何适当地关闭工作线程，请参阅适当地关闭活动和工作流工作线程 (p. 169)。
- 如需了解有关 Amazon SWF 的更多信息，请访问 Amazon SWF 主页或查看 Amazon SWF 开发人员指南。
- 您可以使用 AWS Flow Framework for Java，使用注释以更讲究的 Java 样式编写更复杂的工作流。要了解更多信息，请参阅适用于 Java 的 AWS Flow Framework 开发人员指南。

Lambda 任务

另一个方法是与 Amazon SWF 活动一起使用，就是使用 Lambda 函数代表工作流中的工作单元，并按照安排活动的相似方法安排它们。

本主题主要介绍如何使用AWS SDK for Java实施 Amazon SWF Lambda 任务。有关 Lambda 任务的更多概要性信息，请参阅 Amazon SWF Developer Guide 中的 AWS Lambda 任务。

设置跨服务 IAM 角色以运行 Lambda 函数

在 Amazon SWF 能够运行您的 Lambda 函数前，需要设置一个 IAM 角色并授予 Amazon SWF 权限，让它代表您运行 Lambda 函数。有关如何完成该操作的完整信息，请参阅AWS Lambda 任务。

在注册将使用 Lambda 任务的工作流程时，将需要此 IAM 角色的 Amazon Resource Name (ARN)。

创建 Lambda 函数

您可以使用包括 Java 在内的多种不同语言编写 Lambda 函数。有关如何编写、部署和使用 Lambda 函数的完整信息，请参阅 AWS Lambda 开发人员指南。

> **Note**
>
> 使用哪种语言编写 Lambda 函数并不重要，无论使用哪种语言编写工作流程代码，所有 Amazon SWF 工作流程都可以安排和运行您的函数。Amazon SWF 处理运行函数和传入传出数据的详细信息。

下面是一个简单的 Lambda 函数，它可以用于代替构建简单的 Amazon SWF 应用程序 (p. 154)中的活动。

- 该版本使用 JavaScript 编写，使用 AWS 管理控制台可以直接输入。

```
exports.handler = function(event, context) {
    context.succeed("Hello, " + event.who + "!");
};
```

- 以下是使用 Java 编写的相同函数，您同样可以在 Lambda 上部署和运行它：

```
package example.swf.hellolambda;
```

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.util.json.JSONException;
import com.amazonaws.util.json.JSONObject;

public class SwfHelloLambdaFunction implements RequestHandler<Object, Object> {
    @Override
    public Object handleRequest(Object input, Context context) {
        String who = "Amazon SWF";
        if (input != null) {
            JSONObject jso = null;
            try {
                jso = new JSONObject(input.toString());
                who = jso.getString("who");
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }
        return ("Hello, " + who + "!");
    }
}
```

Note

要了解有关将 Java 函数部署到 Lambda 的更多信息，请参阅 AWS Lambda Developer Guide 中的创建部署程序包 (Java)。您可能还希望查看标题为使用 Java 编写 LAM 函数的编程模型的章节。

Lambda 函数使用 event 或 input 对象作为第一个参数，使用 context 对象作为第二个参数，提供有关运行 Lambda 函数的请求的相关信息。该特定函数要求使用 JSON 提供输入，并将 `who` 字段设置为用于创建问候语的名称。

注册用于 Lambda 的工作流

对于预定 Lambda 函数的工作流，必须提供 IAM 角色的名称，由其为 Amazon SWF 提供调用 Lambda 函数的权限。您可以在工作流注册期间，使用 RegisterWorkflowTypeRequest 的 `withDefaultLambdaRole` 或 `setDefaultLambdaRole` 方法完成该设置。

```
System.out.println("** Registering the workflow type '" + WORKFLOW + "-" + WORKFLOW_VERSION
        + "'.");
try {
    swf.registerWorkflowType(new RegisterWorkflowTypeRequest()
        .withDomain(DOMAIN)
        .withName(WORKFLOW)
        .withDefaultLambdaRole(lambda_role_arn)
        .withVersion(WORKFLOW_VERSION)
        .withDefaultChildPolicy(ChildPolicy.TERMINATE)
        .withDefaultTaskList(new TaskList().withName(TASKLIST))
        .withDefaultTaskStartToCloseTimeout("30"));
}
catch (TypeAlreadyExistsException e) {
```

调度 Lambda 任务

调度 Lambda 任务与调度活动相似。使用 DecisionType`ScheduleLambdaFunction` 和 ScheduleLambdaFunctionDecisionAttributes 提供 Decision。

```
running_functions == 0 && scheduled_functions == 0) {
AWSLambda lam = AWSLambdaClientBuilder.defaultClient();
GetFunctionConfigurationResult function_config =
    lam.getFunctionConfiguration(
            new GetFunctionConfigurationRequest()
                .withFunctionName("HelloFunction"));
String function_arn = function_config.getFunctionArn();

ScheduleLambdaFunctionDecisionAttributes attrs =
    new ScheduleLambdaFunctionDecisionAttributes()
        .withId("HelloFunction (Lambda task example)")
        .withName(function_arn)
        .withInput(workflow_input);

decisions.add(
```

在 `ScheduleLambdaFuntionDecisionAttributes` 中，必须提供 name，这是要调用的 Lambda 函数的 ARN；还必须提供 id，这是用于在历史记录日志中标识 Lambda 函数的 Amazon SWF 的名称。

还可以为 Lambda 函数提供可选的 input 并设置它的 start to close timeout 值，这是在生成 `LambdaFunctionTimedOut` 事件之前允许 Lambda 函数运行的秒数。

> **Note**
>
> 在给出函数名称后，该代码使用 AWSLambdaClient 检索 Lambda 函数的 ARN。您可以使用该方法，以避免您的代码中包含完整 ARN 的硬编码 (包括 AWS 账户 ID)。

在决策程序中处理 Lambda 函数事件

Lambda 任务将生成一系列事件，您可以在工作流工作人员的决策任务轮询中采取行动，与您的 Lambda 任务， 事件类型 诸如 `LambdaFunctionScheduled`， `LambdaFunctionStarted`，和 `LambdaFunctionCompleted`。如果 Lambda 功能失败，或者运行时间比设置超时值要长，您将收到 `LambdaFunctionFailed` 或 `LambdaFunctionTimedOut` 事件类型。

```
boolean function_completed = false;
String result = null;

System.out.println("Executing the decision task for the history events: [");
for (HistoryEvent event : events) {
    System.out.println("  " + event);
    EventType event_type = EventType.fromValue(event.getEventType());
    switch(event_type) {
    case WorkflowExecutionStarted:
        workflow_input =
            event.getWorkflowExecutionStartedEventAttributes()
                 .getInput();
        break;
    case LambdaFunctionScheduled:
        scheduled_functions++;
        break;
    case ScheduleLambdaFunctionFailed:
        scheduled_functions--;
        break;
    case LambdaFunctionStarted:
        scheduled_functions--;
        running_functions++;
        break;
    case LambdaFunctionCompleted:
        running_functions--;
        function_completed = true;
        result = event.getLambdaFunctionCompletedEventAttributes()
                      .getResult();
```

```
        break;
    case LambdaFunctionFailed:
        running_functions--;
        break;
    case LambdaFunctionTimedOut:
        running_functions--;
        break;
```

从您的 Lambda 函数接收输出

在接收 `LambdaFunctionCompleted` EventType 时，可以检索 Lambda 函数的返回值，方法是先对 HistoryEvent 调用 `getLambdaFunctionCompletedEventAttributes` 以获取 LambdaFunctionCompletedEventAttributes 对象，然后调用其 `getResult` 方法以检索 Lambda 函数的输出：

```
 LambdaFunctionCompleted:
running_functions--;
```

此示例的完整源代码

您可以浏览 完整来源:github:`<awsdocs/aws-java-developer-guide/tree/master/doc_source/snippets/helloswf_lambda/> 在此示例中，在 AWS-Java-Developer-Guide 资料库。

适当地关闭活动和工作流工作线程

构建简单的 Amazon SWF 应用程序 (p. 154)主题中介绍实施包括注册应用程序、活动、工作流工作线程以及工作流启动程序的简单工作流应用程序的整个过程。

工作线程类设计为持续运行，轮询 Amazon SWF 发送的任务，以便运行活动或返回决策。完成轮询请求后，Amazon SWF 会记录轮询器，并尝试为其分配任务。

如果工作流工作线程在长轮询过程中终止，Amazon SWF 可能仍然会尝试向终止的工作线程发送任务，导致该任务丢失 (直至该任务超时)。

解决上述情况的一个方法是等待所有长轮询请求返回，然后再终止工作线程。

在该主题中，我们会使用 Java 的关闭挂钩来重写 `helloswf` 中的活动工作线程，以尝试适当地关闭活动工作线程。

以下是完整的代码：

```
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

import com.amazonaws.regions.Regions;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.ActivityTask;
import com.amazonaws.services.simpleworkflow.model.PollForActivityTaskRequest;
import com.amazonaws.services.simpleworkflow.model.RespondActivityTaskCompletedRequest;
import com.amazonaws.services.simpleworkflow.model.RespondActivityTaskFailedRequest;
import com.amazonaws.services.simpleworkflow.model.TaskList;

public class ActivityWorkerWithGracefulShutdown {

    private static final AmazonSimpleWorkflow swf =

 AmazonSimpleWorkflowClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
    private static CountDownLatch waitForTermination = new CountDownLatch(1);
```

```
private static volatile boolean terminate = false;

private static String executeActivityTask(String input) throws Throwable {
    return "Hello, " + input + "!";
}

public static void main(String[] args) {
    Runtime.getRuntime().addShutdownHook(new Thread() {
        @Override
        public void run() {
            try {
                terminate = true;
                System.out.println("Waiting for the current poll request" +
                        " to return before shutting down.");
                waitForTermination.await(60, TimeUnit.SECONDS);
            }
            catch (InterruptedException e) {
                // ignore
            }
        }
    });
    try {
        pollAndExecute();
    }
    finally {
        waitForTermination.countDown();
    }
}

public static void pollAndExecute() {
    while (!terminate) {
        System.out.println("Polling for an activity task from the tasklist '"
                + HelloTypes.TASKLIST + "' in the domain '" +
                HelloTypes.DOMAIN + "'.");

        ActivityTask task = swf.pollForActivityTask(new PollForActivityTaskRequest()
            .withDomain(HelloTypes.DOMAIN)
            .withTaskList(new TaskList().withName(HelloTypes.TASKLIST)));

        String taskToken = task.getTaskToken();

        if (taskToken != null) {
            String result = null;
            Throwable error = null;

            try {
                System.out.println("Executing the activity task with input '"
                        + task.getInput() + "'.");
                result = executeActivityTask(task.getInput());
            }
            catch (Throwable th) {
                error = th;
            }

            if (error == null) {
                System.out.println("The activity task succeeded with result '"
                        + result + "'.");
                swf.respondActivityTaskCompleted(
                    new RespondActivityTaskCompletedRequest()
                        .withTaskToken(taskToken)
                        .withResult(result));
            }
            else {
                System.out.println("The activity task failed with the error '"
                        + error.getClass().getSimpleName() + "'.");
                swf.respondActivityTaskFailed(
```

```
                        new RespondActivityTaskFailedRequest()
                            .withTaskToken(taskToken)
                            .withReason(error.getClass().getSimpleName())
                            .withDetails(error.getMessage()));
                }
            }
        }
    }
}
```

在该版本中，原始版本的 `main` 功能中的轮询代码已被移至其自己的 `pollAndExecute` 方法中。

现在，`main` 功能使用 CountDownLatch 以及关闭挂钩，实现了在收到终止线程的请求后，让线程等待最多 60 秒才允许线程关闭。

注册域

Amazon SWF 中的每个工作流和活动均需包含一个运行的域。

注册 Amazon SWF 域

1. 创建新的 RegisterDomainRequest 对象，并至少为该对象提供域名和工作流执行保留期 (这两个参数是必需的)。
2. 呼叫 AmazonSimpleWorkFlowClient.Registerdomain 使用 RegisterdomainRequest 对象。
3. 如果您请求的域已存在（在此情况下，通常不需要任何操作），则将捕获 DomainAlreadyExistsException。

以下代码演示了此过程：

```
public void register_swf_domain(AmazonSimpleWorkflowClient swf, String name)
{
    RegisterDomainRequest request = new RegisterDomainRequest().withName(name);
    request.setWorkflowExecutionRetentionPeriodInDays("10");
    try
    {
        swf.registerDomain(request);
    }
    catch (DomainAlreadyExistsException e)
    {
        System.out.println("Domain already exists!");
    }
}
```

列出域

您可以按照注册类型，列出与账户和 AWS 区域关联的 Amazon SWF 域。

列出 Amazon SWF 域

1. 创建 ListDomainsRequest 对象，然后指定目标域的注册状态 — （必填项）。
2. 呼叫 AmazonSimpleWorkFlowClient.Listdomains 与 ListdoMainRequest 对象。结果在 DomainInfos 对象中提供。
3. 对返回的对象调用 getDomainInfos，以获取 DomainInfo 对象的列表。
4. 呼叫 getname 每个 Domaininfo 要获得其名称的对象。

以下代码演示了此过程：

```
public void list_swf_domains(AmazonSimpleWorkflowClient swf)
{
    ListDomainsRequest request = new ListDomainsRequest();
    request.setRegistrationStatus("REGISTERED");
    DomainInfos domains = swf.listDomains(request);
    System.out.println("Current Domains:");
    for (DomainInfo di : domains.getDomainInfos())
    {
        System.out.println(" * " + di.getName());
    }
}
```

SDK中包含的代码样本

AWS SDK for Java 附带代码示例，这些示例在可构建且可运行的程序中演示了该开发工具包的许多功能。您可以使用 AWS SDK for Java 学习或修改这些程序以实施您自己的 AWS 解决方案。

如何获取示例

AWS SDK for Java代码示例在开发工具包的 samples 目录中提供。如果您已使用设置适用于 Java 的 AWS 开发工具包 (p. 4)中的信息下载并安装开发工具包，则您的系统中已包含示例。

您也可以在AWS SDK for Java GitHub 存储库中查看最新示例（位于 src/samples 目录中）。

使用命令行构建并运行示例

示例包含 Ant 构建脚本，以便您从命令行轻松构建和运行这些脚本。每个示例还包含一个 HTML 格式的 README 文件，此文件包含每个示例特定的信息。

> Note
>
> 如果您浏览 GitHub 上的代码示例，请在查看示例的 README.html 文件时单击源代码显示中的 Raw (原始) 按钮。在原始模式中，HTML 将在浏览器中按预期方式呈现。

Prerequisites

在运行任意AWS SDK for Java示例之前，您需要在环境中或使用 AWS CLI 设置 AWS 凭证，如设置用于开发的 AWS 凭证和区域 (p. 6)中所述。这些示例使用默认凭证提供程序链 (如果可能)。因此，您可以通过此方式设置凭证以消除将 AWS 凭证插入源代码目录中的文件 (可能无意中签入并公开共享这些凭证) 的有风险的实践。

运行示例

从命令行运行示例

1. 对包含示例代码的目录所做的更改。例如，如果您在 AWS 开发工具包下载的根目录中，并且希望运行 `AwsConsoleApp` 示例，则可键入：

   ```
   cd samples/AwsConsoleApp
   ```

2. 使用 Ant 构建和运行示例。默认构建目标将执行这两项操作，您只需输入：

   ```
   ant
   ```

该示例将信息打印到标准输出 — 例如：

```
===========================================

Welcome to the AWS Java SDK!

===========================================
You have access to 4 Availability Zones.

You have 0 Amazon EC2 instance(s) running.

You have 13 Amazon SimpleDB domain(s) containing a total of 62 items.

You have 23 Amazon S3 bucket(s), containing 44 objects with a total size of 154767691
 bytes.
```

使用 Eclipse IDE 构建并运行示例

如果您使用 AWS Toolkit for Eclipse，也可以基于AWS SDK for Java在 Eclipse 中启动新项目或将该开发工具包添加到现有 Java 项目。

Prerequisites

在安装 AWS Toolkit for Eclipse 后，建议您使用安全凭证配置此工具包。您可以随时通过以下方式执行此操作：从 Eclipse 中的 Window 菜单选择 Preferences，然后选择 AWS Toolkit 部分。

运行示例

使用 AWS Toolkit for Eclipse 运行示例

1. 打开 Eclipse。
2. 创建新的 AWS Java 项目。在 Eclipse 中的 File 菜单上，选择 New，然后单击 Project。New Project 向导随即打开。
3. 展开 AWS 类别，然后选择 AWS Java Project。
4. 选择 Next (下一步)。项目设置页面随即显示。
5. 在 Project Name 框中输入名称。适用于 Java 的 AWS 开发工具包示例组显示该开发工具包中可用的示例，如前所述。
6. 通过选中每个复选框，选择要包含在项目中的示例。
7. 输入您的 AWS 凭证。如果您已使用您的凭证配置 AWS Toolkit for Eclipse，则将自动填入该凭证。
8. 选择 Finish。这将创建项目并将其添加到 Project Explorer。

运行项目

1. 选择要运行的示例 `.java` 文件。例如，对于 Amazon S3 示例，选择 `S3Sample.java`。
2. 从 Run 菜单中选择 Run。

将开发工具包添加到现有项目

1. 右键单击 Project Explorer 中的项目，指向 Build Path，然后选择 Add Libraries。
2. 选择 AWS Java SDK，然后选择 Next，并按照其余的屏幕说明执行操作。

this AWS Product or Service 的安全性

Amazon Web Services (AWS) 的云安全性的优先级最高。作为 AWS 客户，您将从专为满足大多数安全敏感型组织的要求而打造的数据中心和网络架构中受益。安全性是 AWS 和您的共同责任。责任共担模式将其描述为云的安全性和云中的安全性。

云的安全性 – AWS 负责保护运行 AWS 云中提供的所有服务的基础设施，并向您提供可安全使用的服务。我们的安全责任是 AWS 中的最高优先级，作为 AWS 合规性计划的一部分，我们安全性的有效性由第三方审计员定期进行测试和验证。

云中的安全性 – 您的责任由您所使用的 AWS 服务以及其他因素决定，包括您数据的敏感性、您组织的要求以及适用的法律法规。

主题

this AWS Product or Service 中的数据保护

This AWS product or service 符合责任共担模式，此模式包含适用于数据保护的法规和准则。Amazon Web Services (AWS) 负责保护运行所有 AWS 服务的全球基础设施。AWS 保持对此基础设施上托管的数据的控制，包括用于处理客户内容和个人数据的安全配置控制。充当数据控制者或数据处理者的 AWS 客户和 APN 合作伙伴对他们在 AWS 云中放置的任何个人数据承担责任。

出于数据保护的目的，我们建议您保护 AWS 账户凭证并使用 AWS Identity and Access Management (IAM) 设置单个用户账户，以便仅向每个用户提供履行其工作职责所需的权限。我们还建议您通过以下方式保护您的数据：

- 对每个账户使用 Multi-Factor Authentication (MFA)。
- 使用 SSL/TLS 与 AWS 资源进行通信。要使用最低 TLS 版本 1.2，请参阅强制实施 TLS 1.2。
- 使用 AWS CloudTrail 设置 API 和用户活动日志记录。
- 使用 AWS 加密解决方案以及 AWS 服务中的所有默认安全控制。
- 使用高级托管安全服务（例如 Amazon Macie），它有助于发现和保护存储在 Amazon S3 中的个人数据。

我们强烈建议您切勿将敏感的可识别信息（例如您客户的账号）放入自由格式字段（例如 Name (名称) 字段）。这包括使用控制台、API、AWS CLI 或 AWS 开发工具包处理 this AWS product or service 或其他 AWS 服务时。您输入到 this AWS product or service 或其他服务中的任何数据都可能被选取以包含在诊断日志中。当您向外部服务器提供 URL 时，请勿在 URL 中包含凭证信息来验证您对该服务器的请求。

有关数据保护的更多信息，请参阅 AWS共同责任模式和GDPR 博客帖子 AWS安全博客.

适用于TLS1.2的AWSSDKforJava支持

以下信息仅适用于JavaSSL实施（Java的AWSSDK中的默认SSL实施）。如果您正在使用不同的SSL实施，请参阅具体SSL实施，了解如何执行TLS版本。

Java的TLS支持

TLS1.2支持从Java7开始。

如何检查TLS版本

要检查您的Java虚拟机(JVM)中支持TLS版本，您可以使用以下代码。

```
System*.out.println(*Arrays*.toString(*SSLContext*.getDefault().getSupportedSSLParameters().getProtocol
```

要查看SSL握握操作以及使用哪个版本的TLS，您可以使用系统属性 javax.net.debug.

```
java app.jar -Djavax.net.debug=ssl
```

如何设置TLS版本

适用于 Java 的 AWS 开发工具包 1.x

- ApacheHTTP客户端: SDK始终喜欢TLS1.2（如果平台支持）。

适用于 Java 的 AWS 开发工具包 2.x

- ApacheHttpClient: SDK始终喜欢TLS1.2（如果平台支持）。
- URLHTTPConnectionClient: 要仅强制执行TLS1.2，您可以使用此Java命令。

```
java app.jar -Djdk.tls.client.protocols=TLSv1.2
```

或使用本代码。

```
System.setProperty("jdk.tls.client.protocols", "TLSv1.2");
```

- NetTynioHttpClient: NetTY的SDK依赖关系为TLS1.2（如果平台支持）。

此AWS产品或服务的身份和访问管理

AWS Identity and Access Management (IAM) 是一项 Amazon Web Services (AWS) 服务，可以帮助管理员安全地控制对 AWS 资源的访问。IAM 管理员控制谁可以通过身份验证（登录）和授权（具有权限）以使用 AWS 服务中的资源。IAM 是一个可以免费使用的 AWS 服务。

要使用此 AWS 产品或服务访问 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。

有关使用 IAM 的详细信息，请参阅 AWS Identity and Access Management。

有关iam用户的概述，以及它们对您帐户安全性的重要性，请参阅 AWS安全凭据 在 AmazonWebServices一般参考.

此 AWS 产品或服务通过它支持的特定 Amazon Web Services (AWS) 服务遵循责任共担模式。有关 AWS 服务安全性信息，请参阅 AWS 服务安全性文档页面和合规性计划规定的 AWS 合规性工作范围内的 AWS 服务。

此AWS产品或服务的合规验证

此 AWS 产品或服务通过它支持的特定 Amazon Web Services (AWS) 服务遵循责任共担模式。有关 AWS 服务安全性信息，请参阅 AWS 服务安全性文档页面和合规性计划规定的 AWS 合规性工作范围内的 AWS 服务。

作为多个 AWS 合规性计划的一部分，第三方审计员将评估 AWS 服务的安全性和合规性。其中包括 SOC、PCI、FedRAMP、HIPAA 及其他。AWS 在合规性计划范围内的 AWS 服务中提供特定合规性计划范围内经常更新的 AWS 服务列表。

第三方审计报告可供您使用 AWS Artifact 进行下载。有关更多信息，请参阅下载 AWS Artifact 中的报告。

有关 AWS 合规性计划的更多信息，请参阅 AWS 合规性计划。

您在使用此 AWS 产品或服务访问 AWS 服务时的合规性责任由您数据的敏感性、您组织的合规性目标以及适用的法律法规决定。如果您对 AWS 服务的使用需遵守 HIPAA、PCI 或 FedRAMP 等标准，AWS 将提供以下有用资源：

- 安全性与合规性快速入门指南 – 部署指南讨论架构注意事项，并提供在 AWS 上部署侧重安全性和合规性的基准环境的步骤。
- 《设计符合 HIPAA 安全性和合规性要求的架构》白皮书 – 此白皮书描述公司如何使用 AWS 创建符合 HIPAA 标准的应用程序。
- AWS 合规性资源 这是业务手册和指南集合，可能适用于您所在的行业和位置。
- AWS Config 一项服务，可评估您的资源配置对内部实践、行业指南和法规的遵循情况。
- AWS Security Hub – AWS 中安全状态的全面视图，可帮助您检查是否符合安全行业标准和最佳实践。

此AWS产品或服务的弹性

Amazon Web Services (AWS) 全球基础设施是围绕 AWS 区域和可用区而构建的。

AWS 区域提供多个在物理上独立且隔离的可用区，这些可用区通过延迟低、吞吐量高且冗余性高的网络连接在一起。

利用可用区，您可以设计和操作在可用区之间无中断地自动实现故障转移的应用程序和数据库。与传统的单个或多个数据中心基础设施相比，可用区具有更高的可用性、容错性和可扩展性。

有关 AWS 区域和可用区的更多信息，请参阅 AWS 全球基础设施。

此 AWS 产品或服务通过它支持的特定 Amazon Web Services (AWS) 服务遵循责任共担模式。有关 AWS 服务安全性信息，请参阅 AWS 服务安全性文档页面和合规性计划规定的 AWS 合规性工作范围内的 AWS 服务。

此AWS产品或服务的基础设施安全

此 AWS 产品或服务通过它支持的特定 Amazon Web Services (AWS) 服务遵循责任共担模式。有关 AWS 服务安全性信息，请参阅 AWS 服务安全性文档页面和合规性计划规定的 AWS 合规性工作范围内的 AWS 服务。

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。

本文件建立于: 2020年6月27日

2018 年 3 月 22 日

删除了在 DynamoDB 中管理 Tomcat 会话的示例，因为该工具不再受到支持。

2017 年 11 月 2 日

添加了适用于 Amazon S3 加密客户端，包括新主题: 使用AmazonS3客户端加密 (p. 138) 和 带AWSKMS管理密钥的AmazonS3客户端加密 (p. 142) 和 带客户端主密钥的AmazonS3客户端加密 (p. 138).

2017 年 8 月 14 日

对 使用AWSSDKforJava的AmazonS3示例 (p. 115) 部分，包括新主题: 管理Buckets和对象的AmazonS3访问权限 (p. 122) 和 将AmazonS3桶配置为网站 (p. 136).

2017 年 4 月 4 日

新增主题为适用于 Java 的 AWS 开发工具包启用指标 (p. 36)，描述如何为AWS SDK for Java生成应用程序和开发工具包性能指标。

2017 年 4 月 3 日

新增 CloudWatch 示例到 用于Java的AWSSDK的CloudWatch示例 (p. 40) 部分: 从CloudWatch获取度量 (p. 40)， 发布自定义度量标准数据 (p. 41)， 使用CloudWatch警报 (p. 42)， 在CloudWatch中使用警报操作 (p. 44)，和 向CloudWatch发送事件 (p. 45)

2017 年 3 月 27 日

添加更多 Amazon EC2 示例到 使用AWSSDKforJava的AmazonEC2示例 (p. 57) 部分: 管理AmazonEC2实例 (p. 82)， 在AmazonEC2中使用弹性IP地址 (p. 85)， 使用区域和可用区域 (p. 87)， 使用AmazonEC2密钥对 (p. 89)，和 与AmazonEC2的安全团队合作 (p. 91).

2017 年 3 月 21 日

新增了一组 IAM 示例到 使用AWSSDKforJava的IAM示例 (p. 93) 部分: 管理iam访问密钥 (p. 93)， 管理iam用户 (p. 96)， 使用iam帐户别名 (p. 99)， 使用IAM政策 (p. 100)，和 使用iam服务器证书 (p. 104)

2017 年 3 月 13 日

将三个新主题添加到 Amazon SQS 部分: 启用AmazonSQS消息队列的长时间轮询 (p. 148)， 设置AmazonSQS中的可见性超时 (p. 150)，和 在AmazonSQS中使用死字母队列 (p. 151).

2017 年 1 月 26 日

在Amazon S3使用适用于 Java 的 AWS 开发工具包部分增加了新的 (p. 127)主题 (使用TransferManager 执行 Amazon S3 操作 (p. 19)) 和新主题使用适用于 Java 的 AWS 开发工具包进行AWS 开发的最佳实践 (p. 19)。

2017 年 1 月 16 日

增加了一个新的 Amazon S3 主题使用存储桶策略管理对 Amazon S3 存储桶的访问 (p. 125)和两个新的 Amazon SQS 主题使用 Amazon SQS 消息队列 (p. 144)和发送、接收和删除 Amazon SQS 消息 (p. 147)。

2016 年 12 月 16 日

添加了新示例主题 DynamoDB: 使用DynamodB中的表格 (p. 48) 和 使用DynamodB中的项目 (p. 52).

2016 年 9 月 26 日

高级部分中的主题已移入使用适用于 Java 的 AWS 开发工具包 (p. 19)中，因为这些主题实际上是有关使用开发工具包的中心主题。

2016 年 8 月 25 日

已将一个新的创建服务客户端 (p. 20)主题添加到使用适用于 Java 的 AWS 开发工具包 (p. 19)中，该主题说明如何使用客户端构建器更轻松地创建 AWS 服务客户端。

适用于 Java 的 AWS 开发工具包示例 (p. 40)部分已由 S3 的新示例 (p. 115)进行更新，这些示例由包含完整示例代码的 GitHub 上的存储库提供支持。

2016 年 02 月 5 日

已将一个新的异步编程 (p. 28)主题添加到使用适用于 Java 的 AWS 开发工具包 (p. 19)部分，此主题说明如何使用返回 `Future` 对象或采用 `AsyncHandler` 的异步客户端方法。

2016 年 4 月 26 日

已删除 SSL 证书要求 主题，因为该主题不再相关。2015 版本中已弃用对 SHA-1 签名证书的支持，并且已删除包含测试脚本的站点。

2016 年 3 月 14 日

将新主题添加到 Amazon SWF 部分: Lambda任务 (p. 166)，描述如何实施 Amazon SWF 呼叫工作流 Lambda 作为使用传统的替代方案的任务 Amazon SWF 活动。

2016 年 3 月 4 日

使用适用于 Java 的 AWS 开发工具包的 Amazon SWF 示例 (p. 153)部分已由新内容进行更新：

- Amazon SWF 基础知识 (p. 153) – 提供有关如何在项目中包含 SWF 的基本信息。
- 构建简单 Amazon SWF 应用程序 (p. 154) – 一个新的教程，此教程为初次使用 Amazon SWF 的 Java 开发人员提供分步指南。
- 适当地关闭活动和工作流程工作线程 (p. 169) – 说明如何使用 Java 的并发类适当地关闭 Amazon SWF 工作线程类。

2016 年 2 月 23 日

AWS SDK for Java Developer Guide的来源已移至 aws-java-developer-guide。

2015 年 12 月 28 日

设置 DNS 名称查找的 JVM TTL (p. 36) 已从高级移入使用适用于 Java 的 AWS 开发工具包 (p. 19)，并且为清楚起见，已重新编写此主题。

已使用有关如何在项目中包含开发工具包的物料清单 (BOM) 的信息更新将开发工具包与 Apache Maven 一起使用 (p. 9)。

2015 年 8 月 4 日

SL 证书要求是入门 (p. 3)部分中的一个新主题，此主题说明 AWS 如何移至 SSL 连接的 SHA256 签名证书，以及如何修复 1.6 版和以前的 Java 环境以使用这些证书（自 2015 年 9 月 30 日起，需要 这些证书才能访问 AWS）。

> **Note**
>
> Java 1.7+ 已能够使用 SHA256 签名证书。

2014 年 14 月 5 日

已对简介 (p. 1)和入门 (p. 3)内容进行大量修订以支持新的指南结构，并且现在包含有关如何设置用于开发的 AWS 凭证和区域 (p. 6)的指南。

对代码示例 (p. 172)的讨论已移至其在其他文档和资源 (p. 1)部分中所对应的主题。

有关如何查看开发工具包修订历史记录 (p. 2)的信息已移入简介部分。

2014 年 5 月 9 日

简化了AWS SDK for Java文档的总体结构，并更新了入门 (p. 3)和其他文档和资源 (p. 1)主题。

添加了新主题：

- 使用 AWS 凭证 (p. 22) – 讨论可用于指定要与AWS SDK for Java配合使用的凭证的各种方式。
- 使用 IAM 角色授予对 Amazon EC2 上的 AWS 资源的访问权 (p. 60) – 提供了有关如何安全地为 EC2 实例上运行的应用程序指定凭证的信息。

2013 年 9 月 9 日

此文档历史记录 主题跟踪对AWS SDK for Java Developer Guide所做的更改。旨在与发行说明历史记录一起提供。

如果我们为英文版本指南提供翻译，那么如果存在任何冲突，将以英文版本指南为准。在提供翻译时使用机器翻译。