aws

版本 1.x 开发人员指南

适用于 Java 的 AWS SDK 1.x



适用于 Java 的 AWS SDK 1.x: 版本 1.x 开发人员指南

Table of Contents

..... viii
适用于 Java 的 AWS SDK 1.x 1
发布了 SDK 的版本 2 1
其他文档和资源 1
Eclipse IDE 支持 2
开发适用于 Android 的应用程序 2
查看开发工具包的修订历史记录 2
构建早期版本开发工具包的 Java 参考文档 2
入门 3
基本设置 3
概述 3
AWS 访问门户的登录能力 4
设置共享配置文件 4
安装 Java 开发环境 6
如何获得 适用于 Java 的 AWS SDK 6
先决条件 6
使用构建工具 6
下载预构建的 jar 6
从源代码构建 7
使用构建工具 8
将 SDK 与 Apache Maven 结合使用 8
将 SDK 与 Gradle 结合使用 11
临时凭证和区域 14
配置临时凭证 15
刷新 IMDS 凭证 15
设置 AWS 区域 16
使用 适用于 Java 的 AWS SDK 18
最佳实践适用于 AWS 与... 一起发展 适用于 Java 的 AWS SDK 18
S3 18
创建服务客户端 19
获取客户端生成器 19
创建异步客户端 20
使用 DefaultClient 21
客户端生命周期 21

提供临时凭证 22
使用默认凭证提供程序链 22
指定凭证提供程序或提供程序链 25
明确指定临时凭证 26
更多信息 26
AWS 区域 Selection 26
查看区域的服务可用性 26
选择区域 27
选择特定终端节点 27
根据环境自动确定区域 28
异常处理 29
为什么使用未选中的异常？ 29
AmazonServiceException （和子类） 30
AmazonClientException 30
异步编程 30
Java Futures 31
异步回调 32
最佳实践 34
日志记录 适用于 Java 的 AWS SDK 呼叫 34
下载 Log4J JAR 35
设置类路径 35
Service-Specific 错误和警告 35
Request/Response 摘要记录 36
详细线路日志记录 37
延迟指标日志记录 37
客户端配置 38
代理配置 38
HTTP 传输配置 38
TCP 套接字缓冲区大小提示 40
访问控制策略 40
Amazon S3 示例 41
Amazon SQS 示例 41
Amazon SNS 示例 41
为 DNS 名称查找设置 JVM TTL 42
如何设置 JVM TTL 42
为启用指标 适用于 Java 的 AWS SDK 44

如何启用 Java SDK 指标生成 44
可用指标类型 45
更多信息 48
代码示例 49
适用于 Java 的 AWS SDK 2.x 49
Amazon CloudWatch 示例 49
从 CloudWatch 获取指标 50
发布自定义指标数据 51
使用 CloudWatch 警报 53
在 CloudWatch 中使用警报操作 56
将 事件发送到 CloudWatch 57
Amazon DynamoDB 示例 60
使用基于 AWS 账户的端点 60
处理 DynamoDB 中的表 61
处理 DynamoDB 中的项目 68
Amazon EC2 示例 74
教程：启动 EC2 实例 75
使用 IAM 角色授予对 Amazon EC2 上的 AWS 资源的访问权 79
教程：Amazon EC2 竞价型实例 85
教程：高级 Amazon EC2 竞价型实例请求管理 95
管理 Amazon EC2实例 111
在 Amazon EC2 中使用弹性 IP 地址 116
使用区域和可用区 119
使用 Amazon EC2 密钥对 122
在 Amazon EC2 中使用安全组 124
AWS Identity and Access Management (IAM) 示例 127
管理 IAM 访问密钥 128
管理 IAM 用户 132
使用 IAM 账户别名 135
使用 IAM 策略 138
使用 IAM 服务器证书 142
Amazon Lambda 示例 146
服务操作 146
Amazon Pinpoint 示例 150
在 Amazon Pinpoint 中创建和删除应用程序 150
在 Amazon Pinpoint 中创建端点 152

在 Amazon Pinpoint 中创建分段 154
在 Amazon Pinpoint 中创建市场活动 156
在 Amazon Pinpoint 中更新渠道 157
Amazon S3 示例 158
创建、列出和删除 Amazon S3 桶 159
在 Amazon S3 对象上执行操作 164
管理对桶和对象的 Amazon S3 访问权限 169
使用桶策略管理对 Amazon S3 桶的访问 172
使用 TransferManager 执行 Amazon S3 操作 176
将 Amazon S3 桶配置为网站 188
使用 Amazon S3 客户端加密 191
Amazon SQS 示例 197
使用 Amazon SQS 消息队列 197
发送、接收和删除 Amazon SQS 消息 200
为 Amazon SQS 消息队列启用长轮询 202
在 Amazon SQS 中设置可见性超时 205
在 Amazon SQS 中使用死信队列 207
Amazon SWF 示例 209
SWF 基本知识 210
构建简单 Amazon SWF 应用程序 211
Lambda 任务 229
适当地关闭活动和工作流工作线程 233
注册域 236
列出域 236
SDK 中包含的代码示例 237
如何获取示例 237
使用命令行构建并运行示例 237
使用 Eclipse IDE 构建并运行示例 239
安全性 240
数据保护 240
强制实施最低 TLS 版本 241
如何查看 TLS 版本 241
强制实施最低 TLS 版本 242
身份和访问管理 242
受众 242
使用身份进行身份验证 243

使用策略管理访问 244
操作方法 AWS 服务 使用 IAM 245
故障排除 AWS 身份和访问权限 245
合规性验证 247
恢复能力 247
基础设施安全性 248
S3 加密客户端迁移 248
先决条件 248
迁移概述 249
更新现有客户端以读取新格式 249
将加密和解密客户端迁移到 V2 250
其他示例 252
OpenPGP 密钥 254
当前密钥 254
以前的密钥 259
文档历史记录 266

适用于 Java 的 AWS SDK 1.x于2025年12月31日达到支撑终止状态。我们建议您迁移到 AWS SDK for Java 2.x 以继续获得新功能、可用性改进和安全更新。

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。

开发人员指南 – 适用于 Java 的 AWS SDK 1.x

适用于 Java 的 AWS SDK 为 AWS 服务提供了 Java API。利用此开发工具包，您可以轻松构建使用 Amazon S3、Amazon EC2、DynamoDB 等的 Java 应用程序。我们将定期向适用于 Java 的 AWS SDK 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。

发布了 SDK 的版本 2

请访问 https://github.com/aws/aws-sdk-java-v2/，了解新的适用于 Java 的 AWS SDK 2.x。它包括许多期待已久的功能，例如插入 HTTP 实施的方法。要开始使用，请参阅《适用于 Java 的 AWS SDK 2.x 开发人员指南》。

其他文档和资源

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

- 适用于 Java 的 AWS SDK 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 频道
- 这些区域有：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 将适用于 Java 的 AWS SDK 添加到现有 Eclipse 项目或创建新的适用于 Java 的 AWS SDK 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。

有关完整文档，请参阅《AWS Toolkit for Eclipse User Guide》。

开发适用于 Android 的应用程序

对于 Android 开发人员，Amazon Web Services 发布了专用于 Android 开发的 SDK：Amplify Android（适用于 Android 的 AWS Mobile SDK）。

查看开发工具包的修订历史记录

要查看适用于 Java 的 AWS SDK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 SDK API Reference 提供 SDK 版本 1.x 最新构建版的信息。如果您使用 1.x 版本的早期构建版本，您可能希望访问与您使用的版本相匹配的 SDK 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。

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、macOS 或 Unix）链接以将 SDK 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 适用于 Java 的 AWS SDK 的信息。

主题

- 要使用的基本设置 AWS 服务
- 如何获得 适用于 Java 的 AWS SDK
- 使用构建工具
- 设置 AWS 临时证书和 AWS 区域 用于开发

要使用的基本设置 AWS 服务

概述

要成功开发 AWS 服务 使用访问的应用程序 适用于 Java 的 AWS SDK，需要满足以下条件：

- 您必须能够登录 AWS IAM Identity Center中提供的 AWS 访问门户。
- 为软件开发工具包配置的 IAM 角色的权限必须允许访问您的应用程序所需的权限。 AWS 服务与PowerUserAccess AWS 托管策略关联的权限足以满足大多数开发需求。
- 包含以下元素的开发环境：
  - 通过以下方式设置的共享配置文件：
    - 该`config`文件包含一个默认配置文件，该配置文件指定 AWS 区域.
    - `credentials` 文件包含作为默认配置文件一部分的临时凭证。
  - 一个合适的 Java 安装。
  - 一种构建自动化工具，例如 Maven 或 Gradle。
  - 用于处理代码的文本编辑器。
  - （可选，但建议使用）IDE（集成开发环境），例如 IntelliJ ID EA、Eclipse 或。NetBeans

    使用 IDE 时，还可以集成 AWS Toolkit，以便更轻松地使用 AWS 服务。AWS Toolkit for IntelliJ 和 AWS Toolkit for Eclipse 是两个可用于 Java 开发的工具包。

> **Important**
>
> 本设置部分中的说明假设您或组织使用 IAM Identity Center。如果您的组织使用独立于 IAM Identity Center 运行的外部身份提供商，请了解如何获取临时凭证以供适用于 Java 的 SDK 使用。按照以下说明向 `~/.aws/credentials` 文件添加临时凭证。
> 如果您的身份提供商自动向 `~/.aws/credentials` 文件添加临时凭证，请确保配置文件名称为 `[default]`，这样您就无需向 SDK 或 AWS CLI提供配置文件名称。

AWS 访问门户的登录能力

AWS 访问门户是您手动登录 IAM 身份中心的网址。URL 的格式为 `d-xxxxxxxxxx.awsapps.com/start` 或 *`your_subdomain`*`.awsapps.com/start`。

如果您不熟悉 AWS 访问门户，请按照 AWS SDKs 和工具参考指南中 IAM Identity Center 身份验证主题步骤 1 中的账户访问指南进行操作。请勿执行步骤 2，因为 适用于 Java 的 AWS SDK 1.x 不支持步骤 2 所描述的 SDK 的自动令牌刷新和自动检索临时证书。

设置共享配置文件

共享的配置文件位于您的开发工作站上，包含所有 AWS SDKs 和 AWS Command Line Interface (CLI) 使用的基本设置。共享配置文件可以包含许多设置，但本说明用于设置使用 SDK 所需的基本元素。

设置共享 `config` 文件

以下示例展示了共享 `config` 文件的内容。

```
[default]
region=us-east-1
output=json
```

出于开发目的，请使用离你计划运行代码的地方 AWS 区域 最近的。有关可在 `config` 文件中使用的区域代码的列表，请参阅 Amazon Web Services 一般参考 指南。输出格式的 `json` 设置是几个可能的值之一。

按照此部分中的指导创建 `config` 文件。

为 SDK 设置临时凭证

通过访问门户 AWS 访问 AWS 账户 和 IAM 角色后，请使用临时证书配置您的开发环境以供开发工具包访问。

使用临时凭证设置本地 **credentials** 文件的步骤

1. 创建共享 credentials 文件。
2. 在 credentials 文件中，粘贴以下占位符文本，直到粘贴有效的临时凭证为止。

```
[default]
aws_access_key_id=<value from AWS access portal>
aws_secret_access_key=<value from AWS access portal>
aws_session_token=<value from AWS access portal>
```

3. 保存该文件。该 ~/.aws/credentials 文件现在应该存在于您的本地开发系统上。此文件包含 [默认] 配置文件，如果未指定特定的命名配置文件，则适用于 Java 的 SDK 将使用该配置文件。
4. 登录 AWS 访问门户。
5. 按照手动刷新凭证标题下的说明从 AWS 访问门户复制 IAM 角色证书。
   a. 对于链接的说明中的步骤 4，选择可授予访问权限以满足您的开发需求的 IAM 角色名称。此角色的名称通常类似于PowerUserAccess或开发人员。
   b. 对于步骤 7，选择将配置文件手动添加到您的 AWS 凭证文件选项并复制内容。
6. 将复制的凭证粘贴到您的本地 credentials 文件中，并移除所有已粘贴的配置文件名称。您的文件应类似于以下内容：

```
[default]
aws_access_key_id=AKIAIOSFODNN7EXAMPLE
aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
aws_session_token=IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZVERYLONG
```

7. 保存 credentials 文件

当适用于 Java 的 SDK 创建服务客户端时，它将访问这些临时凭证并将它们用于每个请求。在步骤 5a 中选择的 IAM 角色的设置决定了临时凭证的有效时间。最长持续时间为 12 小时。

在临时凭证过期后，重复步骤 4 到 7。

安装 Java 开发环境

适用于 Java 的 AWS SDK V1 需要 Java 7 JDK 或更高版本，并且支持所有 Java LTS（长期支持）JDK 版本。如果您使用 SDK 版本 1.12.767 或更早版本，则可以使用 Java 7，但是如果您使用 SDK 版本 1.12.768 或更高版本，则需要 Java 8。Maven 中央存储库列出了适用于 Java 的 SDK 的最新版本。

适用于 Java 的 AWS SDK 它可与 Oracle Java SE 开发套件以及 A mazon Corret to、Red Hat OpenJDK 和 Adoptium 等开放式 Java 开发套件 (OpenJDK) 的发行版配合使用。

如何获得 适用于 Java 的 AWS SDK

先决条件

要使用 适用于 Java 的 AWS SDK，您必须具备：

- 您必须能够登录 AWS IAM Identity Center中提供的 AWS 访问门户。
- 一个合适的 Java 安装。
- 在您的本地共享 `credentials` 文件中设置的临时凭证。

有关如何进行设置以使用适用于 Java 的 SDK 的说明，请参阅the section called “基本设置”主题。

使用构建工具管理适用于 Java 的 SDK 的依赖项（推荐）

建议在项目中使用 Apache Maven 或 Gradle 来访问适用于 Java 的 SDK 所需的依赖项。 本部分说明如何使用这些工具。

下载并解压缩 SDK（不推荐）

建议使用构建工具来访问项目的 SDK，但您也可以下载最新版 SDK 的预构建 jar。

> **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。

1. 从 https://sdk-for-java.amazonwebservices.com/latest/aws-java-sdk.zip 下载软件开发工具包。

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 SDK API Reference）。
- `lib` – 包含 SDK `.jar` 文件。
- `samples` – 包含说明如何使用 SDK 的实用示例代码。
- `third-party/lib` – 包含 SDK 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。

从源代码构建 SDK 的早期版本（不推荐）

预建表单中仅提供完整 SDK 的最新版本，作为可下载 jar。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/。

1. 前往 SDK 的 GitHub 页面，网址为:适用于 Java 的 AWS SDK (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 -Dgpg.skip=true
   ```

   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。

使用构建工具

使用构建工具有助于管理 Java 项目的开发。有几种构建工具可用，但我们将演示如何使用 Maven 和 Gradle 这两种流行的构建工具来启动和运行。本主题介绍如何使用这两种构建工具管理项目所需的适用于 Java 的 SDK 依赖项。

主题

- 将 SDK 与 Apache Maven 结合使用
- 将 SDK 与 Gradle 结合使用

将 SDK 与 Apache Maven 结合使用

您可以使用 Apache Maven 配置和构建适用于 Java 的 AWS SDK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；有关随该软件打包的原型的列表，请访问 Maven archetypes 页面。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 Getting Started Guide》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

要在项目中使用适用于 Java 的 AWS SDK，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，可以导入单个组件或整个开发工具包。

指定单独的开发工具包模块

要选择单个开发工具包模块，请使用适用于 Java 的 AWS SDK 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>1.11.1000</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

要查看 Maven Central 中提供的最新版本的适用于 Java 的 AWS SDK 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 <dependencies> 文件的 pom.xml 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 代码示例目录 来了解要用于给定 AWS 服务的依赖项。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 服务的依赖项，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>1.11.1000</version>
  </dependency>
</dependencies>
```

重新构建项目。

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 `0jar` 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 SDK 与 Gradle 结合使用

要管理 Gradle 项目的 SDK 依赖关系，请将的 Maven BOM 导 适用于 Java 的 AWS SDK 入到应用程序的文件中。`build.gradle`

> **Note**
>
> 在以下示例中，将生成文件*1.12.529*中的替换为的有效版本 适用于 Java 的 AWS SDK。在 Maven Central 存储库中查找最新版本。

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 支持功能来导入材料清单 (BOM) 文件。

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 添加到应用程序 `build.gradle` 文件的 dependencies 部分。

   ```
   ...
   dependencies {
       implementation platform('com.amazonaws:aws-java-sdk-bom:1.12.529')

       // 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，以下内容包含 Amazon Simple Storage Service (Amazon S3) 的依赖关系。

```
...
dependencies {
    implementation platform('com.amazonaws:aws-java-sdk-bom:1.12.529')
    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。

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

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

apply plugin: 'java'

sourceCompatibility = 1.8

repositories {
  mavenCentral()
}

dependencies {
  implementation platform('com.amazonaws:aws-java-sdk-bom:1.12.529')
  implementation 'com.amazonaws:aws-java-sdk-s3'
}
```

> **Note**
>
> 在前面的示例中，将的 Amazon S3 依赖关系替换为您将在项目中使用的 AWS 服务的依赖关系。由 适用于 Java 的 AWS SDK BOM 管理的模块（依赖关系）列在 M aven 中央存储库中。

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 适用于 Java 的 AWS SDK 依赖关系，请使用 Spring 的 Gradle 依赖关系管理插件导入 SDK 的 Maven BOM。

1. 向应用程序的 build.gradle 文件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:1.12.529'
    }
}
```

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。

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

```
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:1.12.529'
  }
}

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

> **Note**
>
> 在前面的示例中，将的 Amazon S3 依赖关系替换为您将在项目中使用的 AWS 服务的依赖关系。由 适用于 Java 的 AWS SDK BOM 管理的模块（依赖关系）列在 M aven 中央存储库中。

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。

设置 AWS 临时证书和 AWS 区域 用于开发

要使用连接到任何支持的服务 适用于 Java 的 AWS SDK，您必须提供 AWS 临时证书。 AWS SDKs 和 CLIs 使用提供者链在许多不同的位置查找 AWS 临时证书，包括 system/user 环境变量和本地 AWS 配置文件。

本主题提供有关使用设置用于本地应用程序开发的 AWS 临时证书的基本信息 适用于 Java 的 AWS SDK。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：

- 使用 EC2 实例时，创建一个 IAM 角色，然后授予您的 EC2 实例访问该角色的权限，如上的 “使用 IAM 角色授予 AWS 资源访问权限” 中所示 Amazon EC2。
- 使用在 Eclipse 中设置 AWS 凭据。AWS Toolkit for Eclipse有关更多信息，请参阅《AWS Toolkit for Eclipse 用户指南》中的 “设置 AWS 凭据”。

配置临时凭证

您可以通过多种 适用于 Java 的 AWS SDK 方式为配置临时证书，但以下是推荐的方法：

- 在本地系统的凭 AWS 证配置文件中设置临时证书，该文件位于：
  - `~/.aws/credentials` (在 Linux、macOS 或 Unix) 上
  - Windows 上的 `C:\Users\USERNAME\.aws\credentials`

  有关如何获取临时凭证的说明，请参阅本指南中的the section called “为 SDK 设置临时凭证”。
- 设置 `AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY`和 `AWS_SESSION_TOKEN` 环境变量。

  要在 Linux、macOS 或 Unix 上设置这些变量，请使用 ：

  ```
  export AWS_ACCESS_KEY_ID=your_access_key_id
  export AWS_SECRET_ACCESS_KEY=your_secret_access_key
  export AWS_SESSION_TOKEN=your_session_token
  ```

  要在 Windows 上设置这些变量，请使用 ：

  ```
  set AWS_ACCESS_KEY_ID=your_access_key_id
  set AWS_SECRET_ACCESS_KEY=your_secret_access_key
  set AWS_SESSION_TOKEN=your_session_token
  ```
- 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。有关其工作原理的 Amazon EC2详细讨论，请参阅 Linux 实例 Amazon EC2 用户指南中的 IAM 角色。

使用其中一种方法设置 AWS 临时证书后，将使用默认的凭证提供程序链自动加载这些证书。 适用于 Java 的 AWS SDK 有关在 Java 应用程序中使用 AWS 凭据的更多信息，请参阅使用 AWS 凭据。

刷新 IMDS 凭证

适用于 Java 的 AWS SDK 支持选择加入每隔 1 分钟在后台刷新 IMDS 凭证，无论凭证到期时间如何。这使您可以更频繁地刷新凭证，并减少未到达 IMDS 影响感知 AWS 可用性的可能性。

```
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

您应该设置一个默认值 AWS 区域 ，该默认值将用于访问 AWS 服务 适用于 Java 的 AWS SDK。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。有关每项服务的区域列表，请参阅 Amazon Web Services 一般参考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。

您可以使用与设置凭据类似的方法来设置默认 AWS 区域：

- AWS 区域 在本地系统的 AWS 配置文件中进行设置，该文件位于：
  - Linux、macOS 或 Unix 上的 ~/.aws/config
  - 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
  ```

+

用你想要的 AWS 区域（例如 “us-east-1”）代替你的 _aws_region。

- 设置 AWS_REGION 环境变量。

  在 Linux、macOS 或 Unix 上，请使用 ：

  ```
  export AWS_REGION=your_aws_region
  ```

  在 Windows 上，请使用 ：

  ```
  set AWS_REGION=your_aws_region
  ```

  其中 y our_aws_region 是所需的名称。 AWS 区域

使用 适用于 Java 的 AWS SDK

本节提供有关使用编程的重要一般信息 适用于 Java 的 AWS SDK ，这些信息适用于您可能在 SDK 中使用的所有服务。

有关特定于服务的编程信息和示例（ Amazon EC2针对 Amazon S3、 Amazon SWF、等），请参阅适用于 Java 的 AWS SDK 代码示例。

主题

- 最佳实践适用于 AWS 与... 一起发展 适用于 Java 的 AWS SDK
- 创建服务客户端
- 向提供临时证书 适用于 Java 的 AWS SDK
- AWS 区域 Selection
- 异常处理
- 异步编程
- 日志记录 适用于 Java 的 AWS SDK 呼叫
- 客户端配置
- 访问控制策略
- 为 DNS 名称查找设置 JVM TTL
- 为启用指标 适用于 Java 的 AWS SDK

最佳实践适用于 AWS 与... 一起发展 适用于 Java 的 AWS SDK

以下最佳做法可以帮助您在使用开发 AWS 应用程序时避免出现问题或麻烦 适用于 Java 的 AWS SDK。这些最佳实践已按服务分类整理。

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，可能会遇到网络连接或超时问题。默认情况下， 适用于 Java 的 AWS SDK 尝试重试传输失败的方法是在传输开始之前标记输入流，然后在重试之前对其进行重置。

如果直播不支持标记和重置，则当出现暂时性故障并启用重试ResetException时，SDK 会抛出。

最佳实践

建议您使用支持标记和重置操作的流。

避免 a 的最可靠方法ResetException是使用文件或来提供数据 FileInputStream，它们 适用于 Java 的 AWS SDK 可以在不受标记和重置限制的限制的情况下处理这些数据。

如果直播不是，FileInputStream但支持标记和重置，则可以使用`setReadLimit`方法设置标记限制RequestClientOptions。其默认值为 128KB。将读取限制值设置为比流大小大一个字节将可靠地避免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

要向发出请求 Amazon Web Services，请先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。

每个都 AWS 服务 有一个服务接口，其中包含服务 API 中每个操作的方法。例如，DynamoDB 的服务接口名为。AmazonDynamoDBClient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。的客户端生成器类名 DynamoDB 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();
    }
}
```

该生成器还为ClientConfiguration和提供了流畅的设置器 RequestMetricCollector，以及一个包含 2 的自定义列表。RequestHandler

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

每个服务（除外）适用于 Java 的 AWS SDK 都有异步（或异步 Amazon S3）客户端，每个服务都有相应的异步客户端生成器。

使用默认值创建异步 DynamoDB 客户端 ExecutorService

```
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外，异步客户端还允许您设置自定义ExecutorFactory以更改异步客户端ExecutorService使用的配置。 ExecutorFactory是一个函数式接口，因此它可以与 Java 8 lambda 表达式和方法引用互操作。

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

同步和异步客户端生成器都包含名为 defaultClient 的另一个工厂方法。该方法使用默认配置创建服务客户端，即，使用默认提供程序链加载凭证和 AWS 区域。如果不能根据运行应用程序的环境确定凭证或区域，则对 defaultClient 的调用失败。有关如何确定AWS 凭证和区域的更多信息，请参阅使用凭证和AWS 区域 选择。

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
```

向提供临时证书 适用于 Java 的 AWS SDK

要向发出请求 Amazon Web Services，您必须提供 AWS 临时证书， 适用于 Java 的 AWS SDK 以供其在调用服务时使用。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）。
- 在代码中自行提供临时凭证。

使用默认凭证提供程序链

当您在不提供任何参数的情况下初始化新的服务客户端时，会 适用于 Java 的 AWS SDK 尝试使用该DefaultAWSCredentialsProviderChain类实现的默认凭证提供程序链来查找临时证书。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、AWS_SECRET_KEY 或 AWS_SECRET_ACCESS_KEY 和 AWS_SESSION_TOKEN。 适用于 Java 的 AWS SDK 使用EnvironmentVariableCredentialsProvider类来加载这些证书。
2. Java 系统属性，aws.accessKeyId、aws.secretKey（而不是 aws.secretAccessKey）和 aws.sessionToken。 适用于 Java 的 AWS SDK 使用加载SystemPropertiesCredentialsProvider这些凭证。
3. 来自环境或容器的 Web 身份令牌凭证。
4. 默认凭证配置文件 ——通常位于~/.aws/credentials（位置可能因平台而异），由许多 AWS SDK 和由共享。 AWS CLI 适用于 Java 的 AWS SDK 使用加载ProfileCredentialsProvider这些凭证。

   您可以使用提供的aws configure命令创建凭据文件 AWS CLI，也可以使用文本编辑器编辑该文件来创建凭证文件。有关凭证文件格式的信息，请参阅 AWS 凭证文件格式。
5. Amazon ECS 容器凭证 – 如果设置了环境变量 AWS_CONTAINER_CREDENTIALS_RELATIVE_URI，则从 Amazon ECS 加载该凭证。 适用于 Java 的 AWS SDK 使用加载ContainerCredentialsProvider这些凭证。可以指定此值的 IP 地址。
6. 实例配置文件证书-用于 EC2 实例，并通过 Amazon EC2 元数据服务提供。 适用于 Java 的 AWS SDK 使用加载InstanceProfileCredentialsProvider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。请参阅EC2ContainerCredentialsProviderWrapper了解更多信息。

设置临时凭证

为了能够使用 AWS 临时证书，必须至少在上述位置之一进行设置。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the section called “配置临时凭证”。
- 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。
- 要在您的 EC2 实例上设置和使用实例配置文件证书，请参阅上的 “使用 IAM 角色授予对 AWS 资源的访问权限” Amazon EC2。

设置备用凭证配置文件

默认 适用于 Java 的 AWS SDK 使用默认配置文件，但也有一些方法可以自定义哪个配置文件来自凭据文件。

您可以使用 P AWS rofile 环境变量来更改 SDK 加载的配置文件。

例如，在 Linux、macOS 或 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环境变量会影响所有官方支持的 AWS SDK 和工具（包括和）的 AWS CLI 凭据加载。 AWS Tools for Windows PowerShell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

会自动从默认凭证文件位置 适用于 Java 的 AWS SDK 加载 AWS 临时证书。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证书文件所在的位置（例如，通过使用命令行设置此变量）。 适用于 Java 的 AWS SDK 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 凭据文件的位置。
  - 在 Linux、macOS 或 Unix 上，请使用：

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

  - 在 Windows 上，请使用：

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

##文件格式

根据本指南中的基本设置说明，您的凭证文件应采用以下基本格式。

```
[default]
aws_access_key_id=<value from AWS access portal>
aws_secret_access_key=<value from AWS access portal>
aws_session_token=<value from AWS access portal>

[profile2]
aws_access_key_id=<value from AWS access portal>
aws_secret_access_key=<value from AWS access portal>
aws_session_token=<value from AWS access portal>
```

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 credentials 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 metadata_service_timeout 和 metadata_service_num_attempts。无法使用 CLI 配置这些文件 – 如果您想使用这些文件，则必须手动编辑它们。有关配置文件及其可用字段的更多信息，请参阅 AWS Command Line Interface 用户指南AWS Command Line Interface中的配置。

加载凭证

在设置临时凭证后，SDK 使用默认凭证提供程序链来加载这些凭证。

为此，您需要实例化 AWS 服务 客户端，而无需向生成器明确提供证书，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。

您可以向以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。以下示例演示使用环境 凭证的具体情况。

```
AmazonS3 s3Client = AmazonS3ClientBuilder.standard()
                        .withCredentials(new EnvironmentVariableCredentialsProvider())
                        .build();
```

有关 适用于 Java 的 AWS SDK提供的凭证提供者和提供者链的完整列表，请参阅中的所有已知实现类。AWSCredentialsProvider

> **Note**
>
> 您可以使用此技术来提供凭证提供程序或提供者链，这些证书提供程序或通过使用自己实现AWSCredentialsProvider接口的凭证提供程序或对类进行子类来创建。AWSCredentialsProviderChain

明确指定临时凭证

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。如果您使用检索了临时证书 AWS STS，请使用此方法指定 AWS 访问凭证。

1. 实例化该BasicSessionCredentials类，并为其提供 SDK 用于连接的 AWS 访问 AWS 密钥、密钥和会 AWS 话令牌。
2. AWSStaticCredentialsProvider用AWSCredentials对象创建。
3. 使用 AWSStaticCredentialsProvider 配置客户端生成器并构建客户端。

示例如下：

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

更多信息

- 注册 AWS 并创建 IAM 用户
- 为开发设置 AWS 凭证和区域
- 使用 IAM 角色授予对 AWS 资源的访问权限 Amazon EC2

AWS 区域 Selection

区域使您能够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。

查看区域的服务可用性

要查看某个地区是否有特定 AWS 服务 内容可用，请在要使用的区域上使用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 终端节点前缀是在AmazonDynamo数据库中定义的。

选择区域

从 1.4 版本开始 适用于 Java 的 AWS SDK，您可以指定区域名称，SDK 将自动为您选择合适的终端节点。要自行选择终端节点，请参阅选择特定终端节点。

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("{region_api_default}")
                    .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。如果您要 AWS 区域 为同一服务使用多个客户端，则应创建多个客户端，每个区域一个。

选择特定终端节点

通过在创建 AWS 客户端时调用withEndpointConfiguration方法，可以将每个客户端配置为使用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当前列表，请参阅区域和终端节点。

根据环境自动确定区域

> **Important**
>
> 本节仅在使用客户端生成器访问 AWS 服务时适用。 AWS 使用客户端构造函数创建的客户端不会自动从环境中确定区域，而是使用默认 SDK 区域 (usEast1)。

在 Amazon EC2 或 Lambda 上运行时，您可能需要将客户端配置为使用与代码运行相同的区域。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 credential/region 提供程序链从环境中确定区域，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**
>
> 此环境变量由 Lambda 容器设置。

3. SDK 会检查 AWS 共享配置文件（通常位于`~/.aws/config`）。如果 region 属性存在，则 SDK 会使用该属性。
   - `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 SDK 要加载的配置文件。
4. SDK 尝试使用 Amazon EC2 实例元数据服务来确定当前正在运行的 Amazon EC2 实例的区域。
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 应用程序时，常见的方法是使用共享配置文件（如使用默认凭证提供程序链中所述）来设置本地开发的区域，并在 AWS 基础设施上运行时依靠默认区域提供商链来确定区域。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

了解 适用于 Java 的 AWS SDK 抛出异常的方式和时间对于使用 SDK 构建高质量的应用程序非常重要。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。

为什么使用未选中的异常？

出于以下原因， 适用于 Java 的 AWS SDK 使用运行时（或未选中）异常而不是已检查的异常：

- 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。

有关使用选中和未选中的异常的更多信息，请参阅：

- 未选中的异常 - 争议
- 使用选中的异常时的问题
- Java 中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是您在使用时遇到的最常见的异常 适用于 Java 的 AWS SDK。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 将返回错误响应，并且该错误响应的所有详细信息都将包含在抛出的错误响应中。AmazonServiceException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。

遇到时AmazonServiceException，您就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 错误码
- 来自服务的详细错误消息
- AWS 失败请求的请求 ID

AmazonServiceException还包括有关失败的请求是调用者的错误（具有非法值的请求）还是调用 AWS 服务者的错误（内部服务错误）的信息。

AmazonClientException

AmazonClientException表示 Java 客户端代码内部出现问题，无论是在尝试向发送请求时 AWS 还是尝试解析来自 AWS的响应时。AmazonClientException通常比 a 更严重AmazonServiceException，表示存在导致客户端无法向 AWS 服务发出服务调用的主要问题。例如，当您尝试在其中一个客户端上调用操作时，AmazonClientException如果没有可用的网络连接，则会 适用于 Java 的 AWS SDK 抛出。

异步编程

您可以使用同步或异步方法来调用对 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。 适用于 Java 的 AWS SDK 提供了两种方法：F uture 对象和回调方法。

Java Futures

中的异步方法 适用于 Java 的 AWS SDK 返回一个 F ut ure 对象，该对象包含将来异步操作的结果。

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。

以下是一个异步操作的示例，该异步操作调用 Lambda 函数，接收Future可以容纳InvokeResult对象的函数。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\":\"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 对象监控异步请求的状态外，SDK 还允许您实现使用该AsyncHandler接口的类。 AsyncHandler提供了两种根据请求完成方式调用的方法：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\":\"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、快速执行的代码最适合您的AsyncHandler实现。 Long-running 或者在处理程序方法中阻塞代码可能会导致异步客户端使用的线程池争用，并可能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

中的异步客户端 适用于 Java 的 AWS SDK 提供了一个适用于大多数应用程序的默认线程池。您可以实现自定义ExecutorService并将其传递给 适用于 Java 的 AWS SDK 异步客户端，以便更好地控制线程池的管理方式。

例如，您可以提供一个ExecutorService实现，该实现使用自定义ThreadFactory来控制池中线程的命名方式，或者记录有关线程使用情况的其他信息。

异步访问

SDK 中的TransferManager类为使用提供了异步支持 Amazon S3。 TransferManager管理异步上传和下载，提供详细的传输进度报告，并支持对不同事件的回调。

日志记录 适用于 Java 的 AWS SDK 呼叫

使用 Apache Commons Log ging 进行检测，Apache Commons Logging 是一个抽象层，允许在运行时使用多个日志系统中的任何一个。 适用于 Java 的 AWS SDK

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。 适用于 Java 的 AWS SDK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 SDK 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。

Service-Specific 错误和警告

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 客户端检测到您的应用程序未正确关闭`InputStream`并可能泄漏资源，则 S3 客户端会通过警告消息将其报告到日志。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
```

Request/Response 摘要记录

对的每个请求都会 AWS 服务 生成一个唯一的 AWS 请求 ID，如果您在如何处理请求时遇到问题，AWS 服务 这会很有用。 AWS 对于任何失败的服务调用，都可以通过软件开发工具包中的异常对象以编程方式访问请求 ID，也可以通过 “com.amazonaws.request” 记录器中的 DEBUG 日志级别报告请求 ID。

以下 log4j.properties 文件提供了请求和响应的摘要，包括 AWS 请求 ID。

```
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 适用于 Java 的 AWS SDK 发送和接收的确切请求和响应可能会很有用。您不应在生产系统中启用此日志记录，因为写出大型请求（例如，正在上传到的文件 Amazon S3）或响应可能会大大降低应用程序的速度。如果您确实需要访问这些信息，可以通过 Apache HttpClient 4 记录器临时将其启用。如果在 `org.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 文件在 Apache HttpClient 4 中开启了全线日志记录，并且只能暂时打开，因为它可能会对应用程序的性能产生重大影响。

```
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 SDK 指标包的更多信息，请参阅对 适用于 Java 的 AWS SDK启用指标。

```
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=[{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

适用于 Java 的 AWS SDK 允许您更改默认的客户机配置，这在您想要执行以下操作时很有用：

- 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，您可以传入一个可选ClientConfiguration对象来自定义客户端的配置。

如果您通过代理服务器连接到 Internet，则需要通过`ClientConfiguration`对象配置代理服务器设置（代理主机、端口和 username/password）。

HTTP 传输配置

您可以使用ClientConfiguration对象配置多个 HTTP 传输选项。偶尔会添加新选项；要查看您可以检索或设置的选项的完整列表，请参阅 适用于 Java 的 AWS SDK API 参考。

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

最大连接数

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

> **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 适用于 Java 的 AWS SDK API 参考中的常量字段值。

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。

  要自己设置此值，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。

  要设置 HTTP 连接 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 TCP 参数的高级用户还可以通过ClientConfiguration对象设置 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

AWS 访问控制策略使您能够对资源指定精细的访问控制。 AWS 访问控制策略包含一组语句，其形式如下：

在条件 D 适用的情况下，账户 A 有权对资源 C 执行操作 B。

其中：

- A 是委托人 AWS 账户 ，即请求访问或修改您的某个 AWS 资源。
- B 是操作-访问或修改 AWS 资源的方式，例如向 Amazon SQS 队列发送消息或将对象存储在存储 Amazon S3 桶中。
- C 是资源-委托人想要访问的 AWS 实体，例如 Amazon SQS 队列或存储在中的对象 Amazon S3。
- D 是一组条件 – 指定何时允许或拒绝主体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

以下示例演示了一项策略，该策略允许任何人访问存储桶中的所有对象，但将向该存储桶上传对象的访问权限限制为两个特定 AWS 账户的（存储桶拥有者的账户除外）。

```
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

策略的一个常见用途是授权 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 根据订阅主题请求的协议（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），提供了允许或拒绝订阅 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()));
```

为 DNS 名称查找设置 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 资源使用的 DNS 名称条目偶尔会发生变化，因此我们建议您将 JVM 的 TTL 值配置为 5 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，如果在应用程序仍在运行时 AWS 资源的 IP 地址发生变化，则在您手动重启 JVM 并刷新缓存的 IP 信息之前，它将无法使用该资源。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。

如何设置 JVM TTL

要修改 JVM 的 TTL，请设置 net workaddress.cache.ttl 安全属性值。请注意，`networkaddress.cache.ttl`这是一个安全属性，而不是系统属性，也就是说，它不能使用`-D`命令行标志进行设置。

选项 1：在应用程序中以编程方式进行设置

在应用程序启动的`java.security.Security.setProperty()`早期、创建任何 AWS SDK 客户端之前和发出任何网络请求之前调用：

```
import java.security.Security;
```

```
public class MyApplication {
    public static void main(String[] args) {
        Security.setProperty("networkaddress.cache.ttl", "5");

        // ... create SDK clients and run application
    }
}
```

选项 2：将其设置在 java.security 文件中

在 Java 8 `$JAVA_HOME/jre/lib/security/java.security` 的文件中设置该`networkaddress.cache.ttl`属性，在 Java 11 或更高版本`$JAVA_HOME/conf/security/java.security`的文件中设置该属性。

以下是 `java.security` 文件中的代码段，该代码段显示 TTL 缓存设置为 5 秒。

```
#
# The Java-level namelookup cache policy for successful lookups:
#
# any negative value: caching forever
# any positive value: the number of seconds to cache an address for
# zero: do not cache
#
...
networkaddress.cache.ttl=5
...
```

在由`$JAVA_HOME`环境变量表示的 JVM 上运行的所有应用程序都使用此设置。

选项 3：使用 JDK 系统属性回退（命令行）

如果无法修改安全配置或代码，则可以使用 JDK 系统属性。如果未定义安全属性，则这些属性充当后备选项。

- `sun.net.inetaddr.ttl`— 控制成功查找（TTL 为正）
- `sun.net.inetaddr.negative.ttl`— 控制失败的查找（负的 TTL）

```
java -Dsun.net.inetaddr.ttl=5 -Dsun.net.inetaddr.negative.ttl=1 -jar myapp.jar
```

> **Note**
>
> 这些 JDK-internal 属性记录在 Oracle Java 8 网络属性参考文献中，作为 “未来的版本可能不支持” 的私有属性。尽可能使用选项 1-2。

为启用指标 适用于 Java 的 AWS SDK

适用于 Java 的 AWS SDK 可以生成用于通过 Amazon 进行可视化和监控的指标，这些指标 CloudWatch可以衡量：

- 您的应用程序在访问时的性能 AWS
- 与一起使用时 JVM 的性能 AWS
- 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

如何启用 Java SDK 指标生成

您需要添加以下 Maven 依赖项才能让 SDK 向其发送指标。 CloudWatch

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-java-sdk-bom</artifactId>
      <version>1.12.490*</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-cloudwatchmetrics</artifactId>
    <scope>provided</scope>
  </dependency>
  <!-- Other SDK dependencies. -->
</dependencies>
```

*将版本号替换为 Maven Central 上可用的最新版 SDK。

适用于 Java 的 AWS SDK 默认情况下，指标处于禁用状态。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
```

您需要指定证书文件的路径，以便 SDK 可以将收集到的数据点上传到以 CloudWatch 供日后分析。

> **Note**
>
> 如果您使用 Amazon EC2 实例元数据服务 AWS 从 Amazon EC2 实例进行访问，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
> ```

捕获的所有指标 适用于 Java 的 AWS SDK 都在命名空间下 AWSSDK/Java，并上传到 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 区域设置为 us-ea st-1，请使用：

```
-Dcom.amazonaws.sdk.enableDefaultMetrics=credentialFile=/path/
aws.properties,cloudwatchRegion={region_api_default}
```

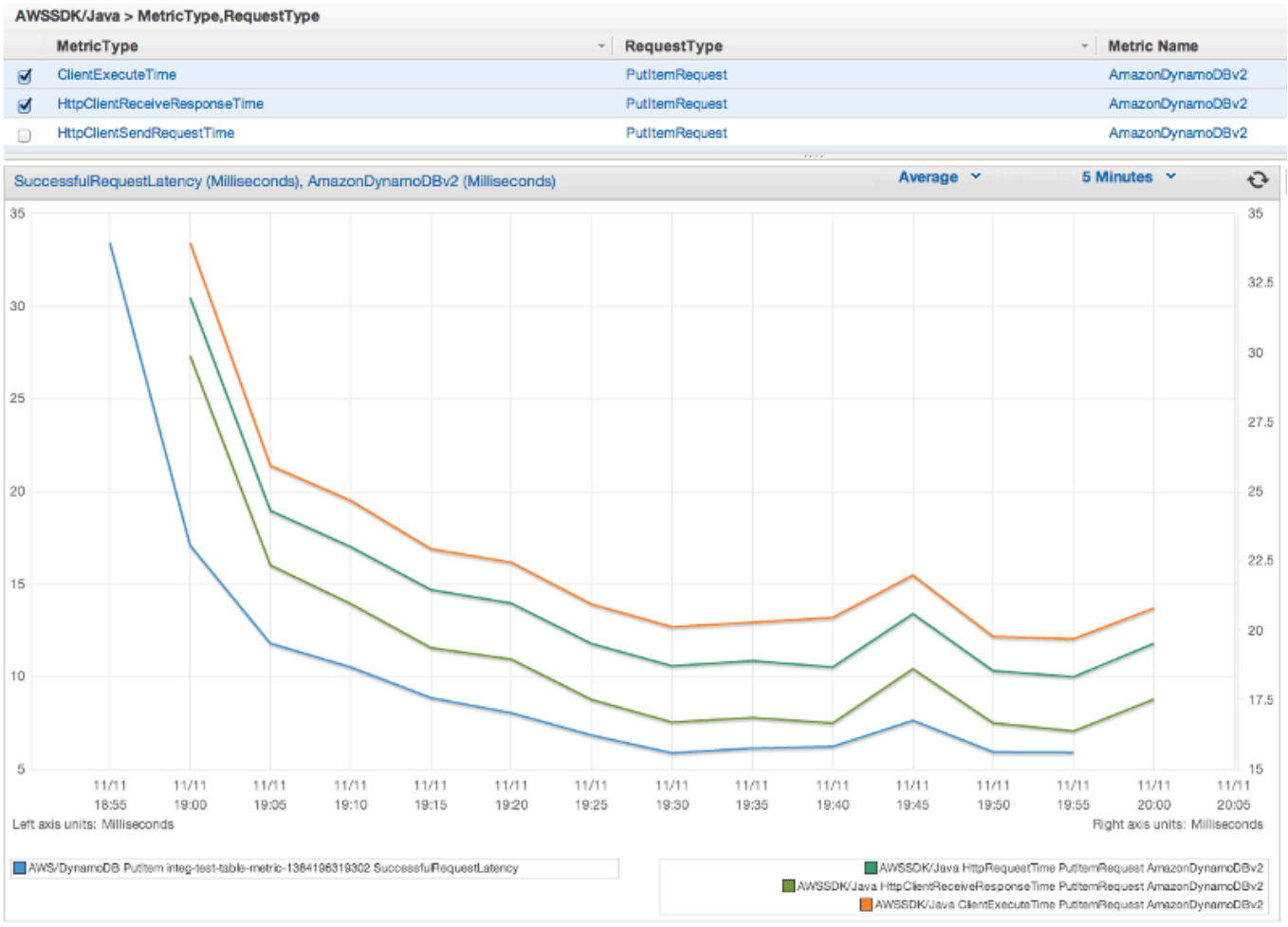
启用该功能后，每次有 AWS 来自的服务请求时，都会生成指标数据点 适用于 Java 的 AWS SDK，排队等候统计摘要，然后异步上传到 CloudWatch 大约每分钟一次。指标一旦上传，您就可以使用 AWS 管理控制台将其可视化，并设置潜在问题的警报，如内存泄露、文件描述符泄露等等。

可用指标类型

默认指标组分为三大类：

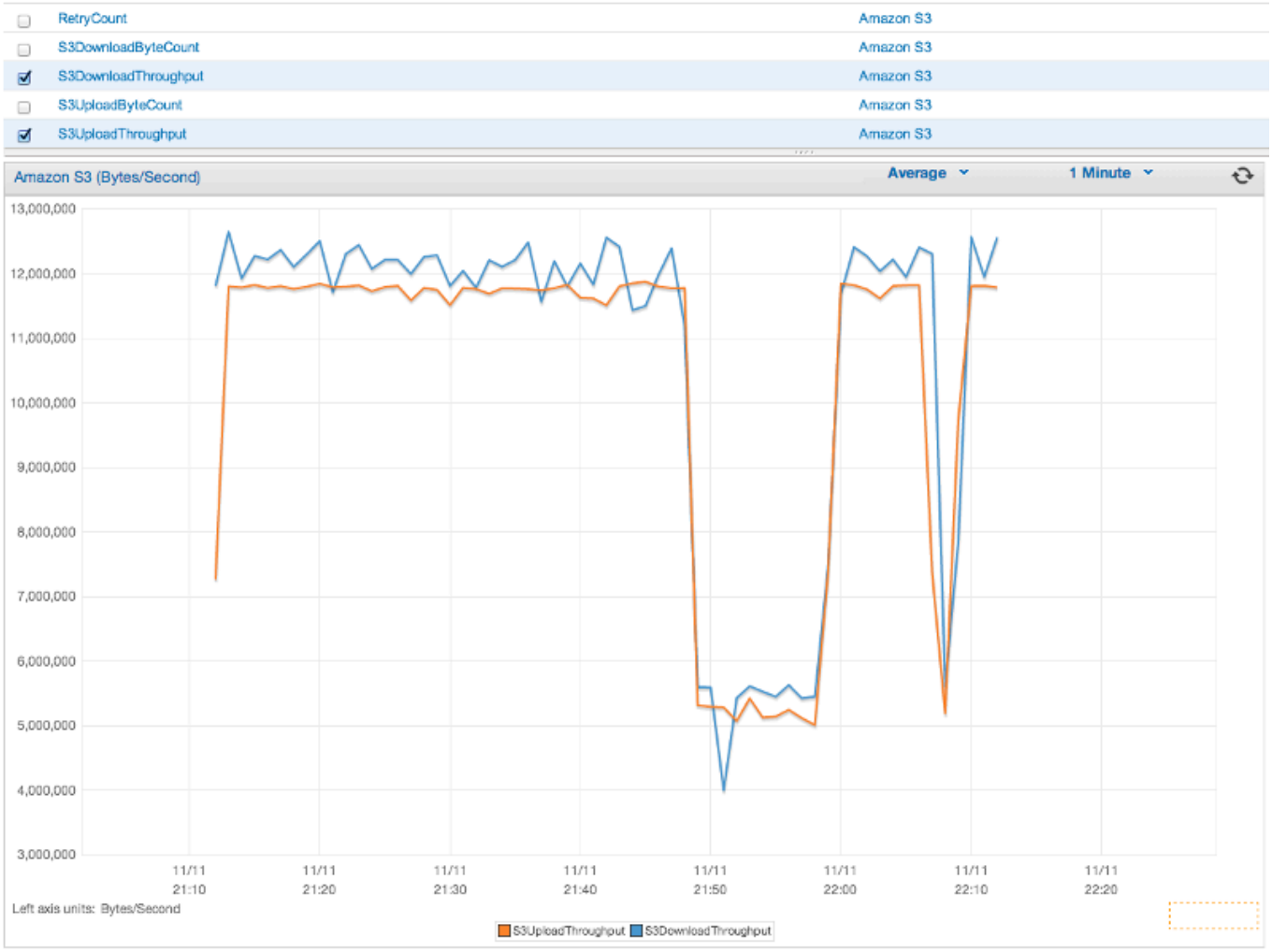
AWS 请求指标

- 涵盖诸如 HTTP 延迟、请求数 request/response、异常和重试次数等方面。



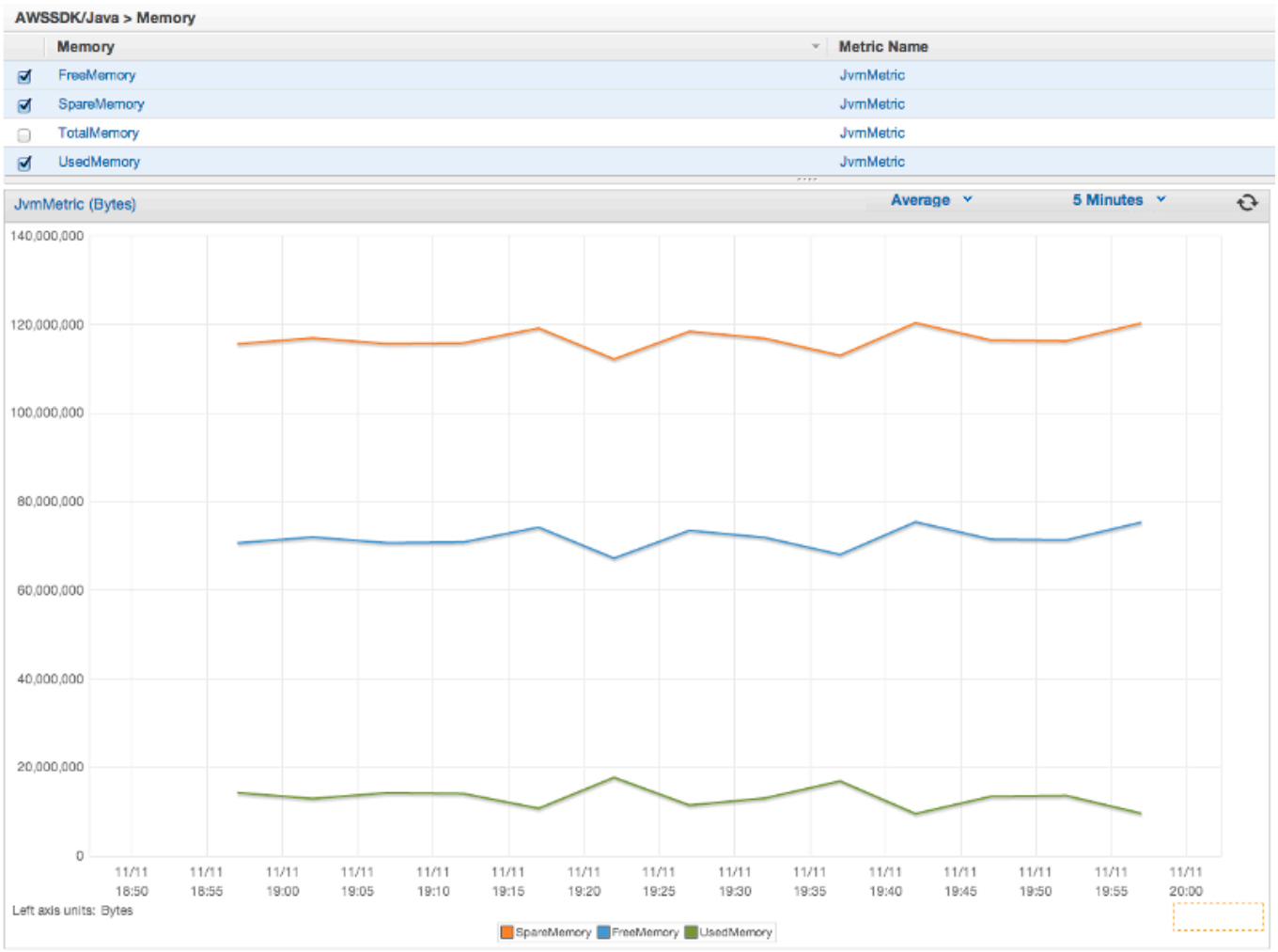
AWS 服务 指标

- 包括 AWS 服务特定数据，例如 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 软件包摘要。
- 适用于 Java 的 AWS SDK 在 “ CloudWatch 使用CloudWatch 示例” 中了解如何使用 适用于 Java 的 AWS SDK。
- 要了解有关性能调整的更多信息，请参阅 “调整 适用于 Java 的 AWS SDK 以提高弹性” 博客文章。

适用于 Java 的 AWS SDK 代码示例

本部分提供使用适用于 Java 的 AWS SDK v1 对 AWS 服务进行编程的教程和示例。

您可以在 GitHub 上的代码示例库 AWS 文档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 上的代码示例存储库中的 Contributing guidelines。

适用于 Java 的 AWS SDK 2.x

2018 年，AWS 发布了AWS SDK for Java 2.x。本指南包含有关使用最新 Java SDK 的说明以及示例代码。

> **Note**
>
> 如需可供 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适用于 Java 的 AWS SDK！

使用适用于 Java 的 AWS SDK 的 CloudWatch 示例

此部分提供使用适用于 Java 的 AWS SDK 对 CloudWatch 进行编程的示例。

Amazon CloudWatch 实时监控您的 Amazon Web Services (AWS) 资源以及在 AWS 上运行的应用程序。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**
>
> 这些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。您可以在那里下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行这些示例。

主题

- 从 CloudWatch 获取指标
- 发布自定义指标数据
- 使用 CloudWatch 警报
- 在 CloudWatch 中使用警报操作
- 将 事件发送到 CloudWatch

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**
>
> AWS 服务发布的指标和维度列表可在《Amazon CloudWatch 用户指南》的 {https---docs-aws-amazon-com-AmazonCloudWatch-latest-monitoring-CW-Support-For-AWS-html} [Amazon CloudWatch 指标和维度参考] 中找到。

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;
```

代码

```
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;
    }
}
```

调用指标的 `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 `setNextToken` 对象的 `ListMetricsResult` 方法的返回值调用 `getNextToken`，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 服务以“AWS”开头的命名空间发布它们自己的指标。您也可以使用自己的命名空间发布自定义指标数据（不以“AWS”开头即可）。

发布自定义指标数据

要发布自己的指标数据，请使用 PutMetricDataRequest 调用 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”开头的命名空间保留供 Amazon Web Services 产品使用。

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;
```

代码

```
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 用户指南》中的使用 Amazon CloudWatch 指标。
- 《Amazon CloudWatch 用户指南》中的AWS 命名空间。
- 《Amazon CloudWatch API Reference》中的 PutMetricData。

使用 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;
```

代码

```
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;
```

代码

```
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 `setNextToken` 对象的 `DescribeAlarmsResult` 方法的返回值调用 `getNextToken`，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;
```

代码

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

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

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

更多信息

- 《Amazon CloudWatch 用户指南》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 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;
```

代码

```
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其操作的警报的名称）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;
```

代码

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

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

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

更多信息

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

将 事件发送到 CloudWatch

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

添加事件

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

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

导入。

```
import 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;
```

代码

```
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，可选参数如事件模式、与规则相关联的 IAM 角色以及描述规则运行频率的计划表达式。

导入。

```
import 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;
```

代码

```
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;
```

代码

```
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 Adding Events with PutEvents
- 《Amazon CloudWatch Events User Guide》中的 Schedule Expressions for Rules
- 《Amazon CloudWatch Events User Guide》中的 Event Types for CloudWatch Events
- 《Amazon CloudWatch Events User Guide》中的 Events and Event Patterns
- 《Amazon CloudWatch Events API Reference》中的 PutEvents
- 《Amazon CloudWatch Events API Reference》中的 PutTargets
- 《Amazon CloudWatch Events API Reference》中的 PutRule

使用适用于 Java 的 AWS SDK 的 DynamoDB 示例

此部分提供使用适用于 Java 的 AWS SDK 对 DynamoDB 进行编程的示例。

> **Note**
>
> 这些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。您可以在那里下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行这些示例。

主题

- 使用基于 AWS 账户的端点
- 处理 DynamoDB 中的表
- 处理 DynamoDB 中的项目

使用基于 AWS 账户的端点

DynamoDB 提供基于 AWS 账户的端点，通过使用您的 AWS 账户 ID 来简化请求路由，从而提升性能。

要使用此功能，您需要使用适用于 Java 的 AWS SDK 版本 1 的 1.12.771 或更高版本。Maven Central 存储库中列出了 SDK 的最新版本。在受支持的 SDK 版本处于活动状态后，它会自动使用新的端点。

如果要选择退出基于账户的路由，您可以选择四个选项：

- 配置 DynamoDB 服务客户端，将 `AccountIdEndpointMode` 设置为 `DISABLED`。
- 设置环境变量。
- 设置 JVM 系统属性。
- 更新共享的 AWS 配置文件设置。

以下代码片段演示了如何通过配置 DynamoDB 服务客户端来禁用基于账户的路由：

```
ClientConfiguration config = new ClientConfiguration()
    .withAccountIdEndpointMode(AccountIdEndpointMode.DISABLED);
AWSCredentialsProvider credentialsProvider = new
 EnvironmentVariableCredentialsProvider();

AmazonDynamoDB dynamodb = AmazonDynamoDBClientBuilder.standard()
    .withClientConfiguration(config)
    .withCredentials(credentialsProvider)
    .withRegion(Regions.US_WEST_2)
    .build();
```

《AWS SDK 和工具参考指南》提供了有关最后三个配置选项的更多信息。

处理 DynamoDB 中的表

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

对于每个表，您必须定义：

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

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

  每个键值均有一个由 ScalarAttributeType 类枚举的关联的数据类型。键值可以是二进制 (B)、数字 (N) 或字符串 (S)。有关更多信息，请参阅《Amazon DynamoDB 开发人员指南》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。

代码

```
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。

代码

```
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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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 开发人员指南》中的表处理准则
- 《Amazon DynamoDB 开发人员指南》中的处理 DynamoDB 中的表

处理 DynamoDB 中的项目

在 DynamoDB 中，项目是属性的集合，每个项目都包括一个名称和一个值。属性值可以为标量、集或文档类型。有关更多信息，请参阅《Amazon DynamoDB 开发人员指南》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;
```

代码

```
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。

向表添加新项目

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;
```

代码

```
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;
```

代码

```
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 SDK 提供了 DynamoDBMapper 类，使您能够将客户端类映射到 Amazon DynamoDB 表。要使用 DynamoDBMapper 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;
```

代码

以下 Java 代码示例演示如何使用 DynamoDBMapper 类向 Music 表添加内容。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 开发人员指南》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 开发人员指南》中的项目处理准则
- 《Amazon DynamoDB 开发人员指南》中的处理 DynamoDB 中的项目

Amazon EC2使用 的 示例适用于 Java 的 AWS SDK

本部分提供使用适用于 Java 的 AWS SDK 对 Amazon EC2 进行编程的示例。

主题

- 教程：启动 EC2 实例

- 使用 IAM 角色授予对 Amazon EC2 上的 AWS 资源的访问权
- 教程：Amazon EC2 竞价型实例
- 教程：高级 Amazon EC2 竞价型实例请求管理
- 管理 Amazon EC2实例
- 在 Amazon EC2 中使用弹性 IP 地址
- 使用区域和可用区
- 使用 Amazon EC2 密钥对
- 在 Amazon EC2 中使用安全组

教程：启动 EC2 实例

本教程演示如何使用适用于 Java 的 AWS SDK 启动 EC2 实例。

主题

- 先决条件
- 创建 Amazon EC2 安全组
- 创建密钥对
- 运行 Amazon EC2 实例

先决条件

在开始之前，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。

创建 Amazon EC2 安全组

EC2-Classic 将停用

> **Warning**
>
> 我们将于 2022 年 8 月 15 日停用 EC2-Classic。我们建议您从 EC2-Classic 迁移到 VPC。有关更多信息，请参阅博客文章 EC2-Classic-Classic Networking is Retiring – Here's How to Prepare。

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 控制台或适用于 Java 的 AWS SDK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 用户指南（适用于 Linux 实例）》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 用户指南（适用于 Linux 实例）》中的 Amazon EC2 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。

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 用户指南（适用于 Linux 实例）》中的 Amazon EC2 密钥对。

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。

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 用户指南（适用于 Linux 实例）》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。如果此数量大于 Amazon EC2 可在目标可用区中启动的实例数，则 Amazon EC2 将启动高于 `MinCount` 的最大可能数量的实例。您可以启动的实例数介于 1 和您允许为该实例类型启动的最大实例数之间。有关更多信息，请参阅 Amazon EC2 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。

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。

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 用户指南（适用于 Linux 实例）》中的连接到您的 Linux 实例。

使用 IAM 角色授予对 Amazon EC2 上的 AWS 资源的访问权

必须通过使用由 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。

本主题介绍如何将 IAM 角色与 Amazon EC2 上运行的 Java SDK 应用程序结合使用。有关 IAM 实例的更多信息，请参阅《Amazon EC2 用户指南（适用于 Linux 实例）》中的适用于 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. Java 系统属性：`aws.accessKeyId` 和 `aws.secretKey`。
2. 系统环境变量：`AWS_ACCESS_KEY_ID` 和 `AWS_SECRET_ACCESS_KEY`。
3. 默认凭证文件 (在不同平台上该文件位于不同位置)。
4. 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. 实例配置文件凭证，包含在与 EC2 实例的 IAM 角色关联的实例元数据中。
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实例配置文件凭证 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，SDK 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。

演练：将 IAM 角色用于 EC2 实例

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。

1. 打开 IAM 管理控制台。
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 Next Step。请记住此名称，因为在启动 Amazon EC2 实例时会用到它。
4. 在选择角色类型页面的 AWS 服务 角色下，选择 Amazon EC2。
5. 在设置权限页面的选择策略模板下，选择 Amazon S3 只读访问权限，然后选择下一步。
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 控制台或适用于 Java 的 AWS SDK，使用 IAM 角色启动 Amazon EC2 实例。

- 要使用控制台启动 Amazon EC2 实例，请按照《Amazon EC2 用户指南（适用于 Linux 实例）》Amazon EC2 Linux 实例入门中的说明操作。

  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。

- 要使用适用于 Java 的 AWS SDK 通过 IAM 角色启动 Amazon EC2 实例，请参阅运行 Amazon EC2 实例。

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，将适用于 Java 的 AWS SDK 库复制到新创建的目录中。如果已将适用于 Java 的 AWS SDK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 final String bucketName = "text-content";
  private static final 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 {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。所以，除非您已经指定 AWS 凭证，否则代码会引发 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 (Amazon EC2)，将程序连同 库传输到适用于 Java 的 AWS SDK 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”。要获取实例的公有 DNS 名称，请打开 EC2 控制台并在描述选项卡中查找公有 DNS 值（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` 目录是适用于 Java 的 AWS SDK中对应的库文件夹。
- `-r` 开关指示 `scp` 应该对`library`版本的 `third-party` 和 适用于 Java 的 AWS SDK 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、macOS 或 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 用户指南（适用于 Linux 实例）》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 EC2 竞价型实例

概览

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适用于 Java 的 AWS SDK 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

先决条件

要使用此指南，您必须已安装适用于 Java 的 AWS SDK 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 SDK。

第 1 步：设置您的证书

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

> **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。

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。

第 2 步：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 客户端数据元之后，我们会创建一个名为“GettingStarted”的`CreateSecurityGroupRequest`数据元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，我们将使用本地电脑子网的 CIDR 表示的 IP 地址范围创建一个 `ipPermission` 数据元，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 Cost 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 管理控制台。
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 来检查各个类型实例的 Spot 价格历史记录。如需更多信息，请转到查看 Spot 价格历史记录。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。要了解更多信息，请访问教程：高级 Amazon EC2 竞价型实例请求管理或《适用于 Java 的 AWS SDK API Reference》中的 RequestSpotInstances 类。

> **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 Spot 请求的状态

下一步是，要一直等到在进行最后一步之前、Spot 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 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！您已经完成了用适用于 Java 的 AWS SDK 开发 Spot 实例软件的入门教程。

后续步骤

继续阅览教程：高级 Amazon EC2 竞价型实例请求管理。

教程：高级 Amazon EC2 竞价型实例请求管理

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。有关竞价型实例的更多信息，请参阅《Amazon EC2 用户指南（适用于 Linux 实例）》中的竞价型实例。

先决条件

要使用此指南，您必须已安装适用于 Java 的 AWS SDK 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 SDK。

设置您的凭证

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

> **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。

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 客户端数据元之后，我们会创建一个名为“GettingStarted”的`CreateSecurityGroupRequest`数据元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，我们将使用本地电脑子网的 CIDR 表示的 IP 地址范围创建一个 `ipPermission` 数据元，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 Managing Security Groups from AWS Cost Explorer。

详细 Spot 实例请求创建选项

正如我们在教程：Amazon EC2 竞价型实例中所介绍的，您需要通过实例类型、亚马逊机器映像 (AMI) 和最高竞标价格来创建您的请求。

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。同样重要的是要认识到，这些参数中的大多数（您的出价是一次性还是永久性出价除外）都可能降低出价实现的可能性。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

管理竞价型实例中断最简单的方法之一是确保定期为您的数据执行到 Amazon Elastic Block Store (Amazon Amazon EBS) 卷的检查点操作。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，并将与其相关联的 Amazon Elastic Block Store (Amazon EBS) 设置到 Amazon EBS 对象，之前我们已对此对象进行配置，如果竞价型实例被终止，此对象 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竞价型实例恢复后连接到该实例的 Amazon 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 Amazon 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 用户指南（适用于 Linux 实例）》中的使用标签。

添加标签请求

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 Amazon 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 DescribeSpotInstanceRequestsRequest 对象调用 Amazon EC2 客户端的 `describeSpotInstanceRequests()` 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 `SpotInstanceRequest` 方法从 `getInstanceId()` 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

要取消竞价型实例请求，请使用 CancelSpotInstanceRequestsRequest 对象调用 Amazon 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竞价型实例的 ID 传递到 Amazon 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，了解如何使用适用于 Java 的 AWS SDK 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，其中包含要使用的亚马逊机器映像 (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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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 `DescribeInstancesResultReservationgetReservations' method, and then call `getInstances` 对象上调用 类的 。

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;
```

代码

```
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信息，请参阅《Amazon EC2 用户指南（适用于 Linux 实例）》中的监控 Amazon EC2。

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

代码

```
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;
```

代码

```
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

EC2-Classic 将停用

> **Warning**
>
> 我们将于 2022 年 8 月 15 日停用 EC2-Classic。我们建议您从 EC2-Classic 迁移到 VPC。有关更多信息，请参阅博客文章 EC2-Classic-Classic Networking is Retiring – Here's How to Prepare。

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，方法是在 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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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，您此后可能不能再使用该地址。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 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 用户指南（适用于 Linux 实例）》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

描述区域

要列出账户可用的区域，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;
```

代码

```
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

要列出账户可用的每个可用区，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;
```

代码

```
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

要描述您的账户，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;
```

代码

```
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 用户指南（适用于 Linux 实例）》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;
```

代码

```
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

要列出密钥对或获取相关信息，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;
```

代码

```
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;
```

代码

```
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 用户指南（适用于 Linux 实例）》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;
```

代码

```
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)。以下示例演示如何将 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;
```

代码

```
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 SecurityGroup 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;
```

代码

```
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。

正在删除安全组

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;
```

代码

```
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 用户指南（适用于 Linux 实例）》中的 Amazon EC2 安全组
- 《Amazon EC2 用户指南（适用于 Linux 实例）》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

使用适用于 Java 的 AWS SDK 的 IAM 示例

本部分提供使用适用于 Java 的 AWS SDK 对 IAM 进行编程的示例。

AWS Identity and Access Management (IAM) 使您能够安全地控制您的用户对 AWS 服务和资源的访问权限。使用 IAM，您可以创建和管理 AWS 用户和组，并使用权限来允许和拒绝他们对 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
- 管理 IAM 用户
- 使用 IAM 账户别名
- 使用 IAM 策略
- 使用 IAM 服务器证书

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;
```

代码

```
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;
```

代码

```
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 setMarker 上调用 ListAccessKeysRequest 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;
```

代码

```
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;
```

代码

```
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 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;
```

代码

```
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;
```

代码

```
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 ListUsersResult 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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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 用户指南》中的 IAM 用户
- 《IAM 用户指南》中的管理 IAM 用户
- 《IAM API Reference》中的 CreateUser
- 《IAM API Reference》中的 ListUsers
- 《IAM API Reference》中的 UpdateUser
- 《IAM API Reference》中的 DeleteUser

使用 IAM 账户别名

如果您希望登录页面的 URL 包含贵公司名称（或其他友好标识符）而不是 AWS 账户 ID，则可以为 AWS 账户创建别名。

> **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;
```

代码

```
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 ListAccountAliasesResult 支持与其他适用于 Java 的 AWS SDK 列出 方法相同的 `getIsTruncated` 和 `getMarker` 方法，但一个 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;
```

代码

```
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

要删除您账户的别名，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;
```

代码

```
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 用户指南》中的您的 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;
```

代码

```
final AmazonIdentityManagement iam =
    AmazonIdentityManagementClientBuilder.defaultClient();

CreatePolicyRequest request = new CreatePolicyRequest()
    .withPolicyName(policy_name)
    .withPolicyDocument(POLICY_DOCUMENT);

CreatePolicyResult response = iam.createPolicy(request);
```

IAM policy 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;
```

代码

```
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

您可以通过调用 AmazonIdentityManagementClient 的 `attachRolePolicy` 方法，在 AttachRolePolicyRequest 中向其提供角色名称和策略 ARN 来将策略附加到 IAM 角色 (http://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html)。

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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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 用户指南》中的 IAM policy 概述。
- 《IAM 用户指南》中的 AWS IAM policy 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 上启用网站或应用程序的 HTTPS 连接，需要 SSL/TLS 服务器证书。您可以使用 AWS Certificate Manager 提供的服务器证书或您从外部提供程序获得的服务器证书。

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

获取服务器证书

您可以通过调用 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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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 用户指南》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使用 的 示例适用于 Java 的 AWS SDK

此部分提供使用Lambda 对 适用于 Java 的 AWS SDK 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、列出和删除 Lambda 函数

调用、列出和删除 Lambda 函数

本部分提供使用适用于 Java 的 AWS SDK 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

- 调用函数
- 列出函数
- 删除函数

调用函数

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-east-1:555556330391:function:HelloFunction。可以通过查看 AWS 管理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;
```

代码

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;
```

代码

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-east-1:555556330391:function:HelloFunction。可以通过查看 AWS 管理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;
```

代码

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 function
        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。

使用适用于 Java 的 AWS SDK 的 Amazon Pinpoint 示例

此部分提供使用适用于 Java 的 AWS SDK 对 Amazon Pinpoint 进行编程的示例。

> **Note**
>
> 这些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。您可以在那里下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行这些示例。

主题

- 在 Amazon Pinpoint 中创建和删除应用程序
- 在 Amazon Pinpoint 中创建端点
- 在 Amazon Pinpoint 中创建分段
- 在 Amazon Pinpoint 中创建市场活动
- 在 Amazon Pinpoint 中更新渠道

在 Amazon Pinpoint 中创建和删除应用程序

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;
```

代码

```
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（其中设置了要删除的应用程序名称）调用 AmazonPinpointClient 的 `deleteApp` 请求。

导入。

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
```

代码

```
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 Apps
- 《Amazon Pinpoint API Reference》中的 App

在 Amazon Pinpoint 中创建端点

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。

创建端点

通过在 Amazon PinpointEndpointRequest 对象中提供终端节点数据，在 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;
```

代码

```
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。

代码

```
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

- 《Amazon Pinpoint Developer Guide》中的 Adding Endpoint
- 《Amazon Pinpoint API Reference》中的 Endpoint

在 Amazon Pinpoint 中创建分段

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 Amazon PinpointSegmentDimensions 对象中创建分段的维度，在 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;
```

代码

```
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);
```

接下来，在 WriteSegmentRequest 中设置 SegmentDimensions 对象，接着使用该对象创建 CreateSegmentRequest 对象。然后，将 CreateSegmentRequest 对象传递到 AmazonPinpointClient 的 `createSegment` 方法。

代码

```
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 Segments
- 《Amazon Pinpoint Developer Guide》中的 Creating Segments

- 《Amazon Pinpoint API Reference》中的 Segments
- 《Amazon Pinpoint API Reference》中的 Segment

在 Amazon Pinpoint 中创建市场活动

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;
```

代码

```
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 Amazon PinpointWriteCampaignRequest 提供给 CreateCampaignRequest 对象，在 中创建新的市场活动。最后，将 CreateCampaignRequest 对象传递到 AmazonPinpointClient 的 `createCampaign` 方法。

代码

```
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 Campaigns
- 《Amazon Pinpoint Developer Guide》中的 Creating Campaigns。
- 《Amazon Pinpoint API Reference》中的 Campaigns
- 《Amazon Pinpoint API Reference》中的 Campaign
- 《Amazon Pinpoint API Reference》中的 Campaign Activities
- 《Amazon Pinpoint API Reference》中的 Campaign Versions
- 《Amazon Pinpoint API Reference》中的 Campaign Version

在 Amazon Pinpoint 中更新渠道

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;
```

代码

```
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 Channels
- 《Amazon Pinpoint API Reference》中的 ADM Channel
- 《Amazon Pinpoint API Reference》中的 APNs Channel
- 《Amazon Pinpoint API Reference》中的 APNs Sandbox Channel
- 《Amazon Pinpoint API Reference》中的 APNs VoIP Channel
- 《Amazon Pinpoint API Reference》中的 APNs VoIP Sandbox Channel
- 《Amazon Pinpoint API Reference》中的 Baidu Channel
- 《Amazon Pinpoint API Reference》中的 Email Channel
- 《Amazon Pinpoint API Reference》中的 GCM Channel
- 《Amazon Pinpoint API Reference》中的 SMS Channel

使用适用于 Java 的 AWS SDK 的 Amazon S3 示例

此部分提供使用适用于 Java 的 AWS SDK 对 Amazon S3 进行编程的示例。

> **Note**
>
> 这些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。您可以在那里下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行这些示例。

主题

- 创建、列出和删除 Amazon S3 桶
- 在 Amazon S3 对象上执行操作
- 管理对桶和对象的 Amazon S3 访问权限
- 使用桶策略管理对 Amazon S3 桶的访问
- 使用 TransferManager 执行 Amazon S3 操作
- 将 Amazon S3 桶配置为网站
- 使用 Amazon S3 客户端加密

创建、列出和删除 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。有关桶及其配置的详细信息，请参阅《Amazon Simple Storage Service 用户指南》中的使用 Amazon S3 桶。

> **Note**
>
> 最佳实践
> 建议您对 存储桶启用 AbortIncompleteMultipartUploadAmazon S3 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 用户指南》中的使用版本控制的桶生命周期配置。

> **Note**
>
> 这些代码示例假定您了解使用适用于 Java 的 AWS SDK 中的内容，并且已使用设置用于开发的 AWS 凭证和区域中的信息配置默认 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;
```

代码

```
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;
```

代码

```
List<Bucket> buckets = s3.listBuckets();
System.out.println("Your {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
- 删除受版本控制的存储桶之前先删除其中的对象
- 删除空存储桶

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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
System.out.println(" OK, bucket ready to delete!");
s3.deleteBucket(bucket_name);
```

请参阅 GitHub 上的完整示例。

在 Amazon S3 对象上执行操作

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中。

> **Note**
>
> 这些代码示例假定您了解使用适用于 Java 的 AWS SDK 中的内容，并且已使用设置用于开发的 AWS 凭证和区域中的信息配置默认 AWS 凭证。

主题

- 上传对象
- 列出对象
- 下载对象
- 复制、移动或重命名对象
- 删除对象
- 一次性删除多个对象

上传对象

使用 AmazonS3 客户端的 `putObject` 方法，并为其提供桶名称、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;
```

代码

```
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

要获取桶中的对象列表，请使用 AmazonS3 客户端的 `listObjects` 方法，并为其提供桶名称。

`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;
```

代码

```
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。

您可以通过对 getObjectContent 调用 S3Object 来获取对象的内容。这将返回一个 S3ObjectInputStream，其行为与标准 Java InputStream 对象的相同。

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;
```

代码

```
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` 方法将对象从一个桶复制到另一个桶。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;
```

代码

```
try {
    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 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。

删除对象

使用 AmazonS3 客户端的 `deleteObject` 方法，并向其传递要删除的桶和对象的名称。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;
```

代码

```
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

使用 AmazonS3 客户端 `deleteObjects` 的方法，您可以将同一个桶中的多个对象的名称传递给 link:sdk-for-java/v1/reference/com/amazonaws/services/s3/model/DeleteObjectsRequest.html 方法，从而删除这些对象。

导入。

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
```

代码

```
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。

管理对桶和对象的 Amazon S3 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 SDK 中的内容，并且已使用设置用于开发的 AWS 凭证和区域中的信息配置默认 AWS 凭证。

获取存储桶的访问控制列表

要获取桶的当前 ACL，请调用 AmazonS3 的 `getBucketAcl` 方法，将桶名称 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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
    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 Object acl
- 《Amazon S3 API Reference》中的 PUT Object acl

使用桶策略管理对 Amazon S3 桶的访问

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;
```

代码

```
    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;
```

代码

```
        new Statement(Statement.Effect.Allow)
                .withPrincipals(Principal.AllUsers)
                .withActions(S3Actions.GetObject)
                .withResources(new Resource(
                        "{region-arn}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 桶的策略，请调用 AmazonS3 客户端的 getBucketPolicy 方法，将桶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;
```

代码

```
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

要删除桶策略，请调用 AmazonS3 客户端的 deleteBucketPolicy，并为其提供桶名称。

导入。

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
```

代码

```
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 Simple Storage Service 用户指南》中的访问策略语言概述

- 《Amazon Simple Storage Service 用户指南》中的桶策略示例

使用 TransferManager 执行 Amazon S3 操作

您可以使用适用于 Java 的 AWS SDK TransferManager 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，以及暂停或恢复上传和下载。

> **Note**
>
> 最佳实践
> 建议您对 存储桶启用 AbortIncompleteMultipartUploadAmazon S3 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 用户指南》中的使用版本控制的桶生命周期配置。

> **Note**
>
> 这些代码示例假定您了解使用适用于 Java 的 AWS SDK 中的内容，并且已使用设置用于开发的 AWS 凭证和区域中的信息配置默认 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的任何 Amazon S3 桶。

主题

- 上传单个文件
- 上传文件列表
- 上传目录

上传单个文件

调用 TransferManager 的 `upload` 方法，提供 Amazon S3 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;
```

代码

```
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。

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 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;
```

代码

```
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。

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和使用 ProgressListener 获取传输进度。

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。

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;
```

代码

```
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。

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和使用 ProgressListener 获取传输进度。

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。

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 桶名称，后跟对象前缀）。

主题

- 下载单个文件
- 下载目录

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 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;
```

代码

```
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。

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 `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;
```

代码

```
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。

请参阅 GitHub 上的完整示例。

复制对象

要将对象从一个 S3 桶复制到另一个 S3 桶，可使用 TransferManager 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;
```

代码

```
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 之前`waitForCompletion` 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 `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。

类	返回方
Copy（复制	`copy`

类	返回方
下载 。	download
MultipleFileDownload	downloadDirectory
上传	upload
MultipleFileUpload	uploadFileList ，uploadDirectory

所有这些类都实施 Transfer 接口。Transfer 提供了用于获取传输进度、暂停或恢复传输以及获取传输的当前状态或最终状态的有用方法。

主题

- 轮询传输的当前进度
- 使用 ProgressListener 获取传输进度
- 获取子传输的进度

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;
```

代码

```
// 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

您可以使用 Transfer 接口的 `addProgressListener` 方法将 ProgressListener 附加到任何传输中。

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;
```

代码

```
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;
```

代码

```
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 Simple Storage Service 用户指南》中的对象键

将 Amazon S3 桶配置为网站

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 SDK 中的内容，并且已使用设置用于开发的 AWS 凭证和区域中的信息配置默认 AWS 凭证。

设置存储桶的网站配置

要设置 Amazon S3 桶网站配置，请使用要设置配置的桶名称，以及包含桶网站配置的 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;
```

代码

```
    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，请参阅使用桶策略管理对 Amazon S3 桶的访问。

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 桶的网站配置，请使用要检索其配置的桶的名称来调用 AmazonS3 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;
```

代码

```
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 桶的网站配置，请使用要从中删除配置的桶的名称来调用 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;
```

代码

```
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 Bucket website
- 《Amazon S3 API Reference》中的 GET Bucket website
- 《Amazon S3 API Reference》中的 DELETE Bucket website

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。

如果您不熟悉加密，请参阅《AWS KMS 开发人员指南》中的加密基础知识，大致了解加密术语和加密算法。要了解有关所有 AWS SDK 的加密支持信息，请参阅 Amazon Web Services 一般参考中的 Amazon S3 客户端加密的 AWS SDK 支持。

> **Note**
>
> 这些代码示例假定您了解使用适用于 Java 的 AWS SDK 中的内容，并且已使用设置用于开发的 AWS 凭证和区域中的信息配置默认 AWS 凭证。

如果您使用的是 1.11.836 或更低版本的适用于 Java 的 AWS SDK，请参阅 Amazon S3 加密客户端迁移，了解有关将应用程序迁移到更高版本的信息。如果您无法迁移，请参阅 GitHub 上的此完整示例。

如果您使用的是 1.11.837 或更高版本的适用于 Java 的 AWS SDK，请浏览下面列出的示例主题以使用 Amazon S3 客户端加密。

主题

- Amazon S3 客户端加密配合客户端主密钥
- Amazon S3 客户端加密配合 AWS KMS 托管密钥

Amazon S3 客户端加密配合客户端主密钥

以下示例使用 AmazonS3EncryptionClientV2Builder 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 托管密钥。

启用客户端 Amazon S3 加密时，您可以从两种加密模式中进行选择：经严格身份验证或经身份验证。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.ClientConfiguration;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3EncryptionClientV2Builder;
import com.amazonaws.services.s3.AmazonS3EncryptionV2;
import com.amazonaws.services.s3.model.CryptoConfigurationV2;
import com.amazonaws.services.s3.model.CryptoMode;
import com.amazonaws.services.s3.model.EncryptionMaterials;
import com.amazonaws.services.s3.model.StaticEncryptionMaterialsProvider;
```

经严格身份验证加密

如果未指定 `CryptoMode`，则默认模式为经严格身份验证加密。

要显式启用此模式，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。

代码

```
AmazonS3EncryptionV2 s3Encryption = AmazonS3EncryptionClientV2Builder.standard()
```

```
        .withRegion(Regions.US_WEST_2)
        .withCryptoConfiguration(new
 CryptoConfigurationV2().withCryptoMode((CryptoMode.StrictAuthenticatedEncryption)))
        .withEncryptionMaterialsProvider(new StaticEncryptionMaterialsProvider(new
 EncryptionMaterials(secretKey)))
        .build();

s3Encryption.putObject(bucket_name, ENCRYPTED_KEY2, "This is the 2nd content to
 encrypt");
```

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，请参阅博客文章 Amazon S3 Client-Side Authenticated Encryption。

> **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 AuthenticatedEncryption 方法中指定 withCryptoConfiguration 值。

代码

```
AmazonS3EncryptionV2 s3EncryptionClientV2 =
 AmazonS3EncryptionClientV2Builder.standard()
        .withRegion(Regions.DEFAULT_REGION)
        .withClientConfiguration(new ClientConfiguration())
        .withCryptoConfiguration(new
 CryptoConfigurationV2().withCryptoMode(CryptoMode.AuthenticatedEncryption))
        .withEncryptionMaterialsProvider(new StaticEncryptionMaterialsProvider(new
 EncryptionMaterials(secretKey)))
        .build();

s3EncryptionClientV2.putObject(bucket_name, ENCRYPTED_KEY1, "This is the 1st content to
 encrypt");
```

Amazon S3 客户端加密配合 AWS KMS 托管密钥

以下示例使用 AmazonS3EncryptionClientV2Builder 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**
>
> 以下示例演示如何将 Amazon S3 客户端加密和 AWS 托管密钥配合使用。要了解如何将加密与您自己的密钥配合使用，请参阅 Amazon S3 客户端加密配合客户端主密钥。

启用客户端 Amazon S3 加密时，您可以从两种加密模式中进行选择：经严格身份验证或经身份验证。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.ClientConfiguration;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.kms.AWSKMS;
import com.amazonaws.services.kms.AWSKMSClientBuilder;
import com.amazonaws.services.kms.model.GenerateDataKeyRequest;
import com.amazonaws.services.kms.model.GenerateDataKeyResult;
import com.amazonaws.services.s3.AmazonS3EncryptionClientV2Builder;
import com.amazonaws.services.s3.AmazonS3EncryptionV2;
import com.amazonaws.services.s3.model.CryptoConfigurationV2;
import com.amazonaws.services.s3.model.CryptoMode;
import com.amazonaws.services.s3.model.EncryptionMaterials;
import com.amazonaws.services.s3.model.KMSEncryptionMaterialsProvider;
```

经严格身份验证加密

如果未指定 `CryptoMode`，则默认模式为经严格身份验证加密。

要显式启用此模式，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。

代码

```
AmazonS3EncryptionV2 s3Encryption = AmazonS3EncryptionClientV2Builder.standard()
         .withRegion(Regions.US_WEST_2)
         .withCryptoConfiguration(new
 CryptoConfigurationV2().withCryptoMode((CryptoMode.StrictAuthenticatedEncryption)))
         .withEncryptionMaterialsProvider(new KMSEncryptionMaterialsProvider(keyId))
         .build();

s3Encryption.putObject(bucket_name, ENCRYPTED_KEY3, "This is the 3rd content to encrypt
 with a key created in the {console}");
System.out.println(s3Encryption.getObjectAsString(bucket_name, ENCRYPTED_KEY3));
```

对 `putObject` 加密客户端调用 Amazon S3 方法以上传对象。

代码

```
s3Encryption.putObject(bucket_name, ENCRYPTED_KEY3, "This is the 3rd content to encrypt
 with a key created in the {console}");
```

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。

代码

```
System.out.println(s3Encryption.getObjectAsString(bucket_name, ENCRYPTED_KEY3));
```

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，请参阅博客文章 Amazon S3 Client-Side Authenticated Encryption。

> **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 AuthenticatedEncryption 方法中指定 withCryptoConfiguration 值。

代码

```
AmazonS3EncryptionV2 s3Encryption = AmazonS3EncryptionClientV2Builder.standard()
         .withRegion(Regions.US_WEST_2)
         .withCryptoConfiguration(new
 CryptoConfigurationV2().withCryptoMode((CryptoMode.AuthenticatedEncryption)))
         .withEncryptionMaterialsProvider(new KMSEncryptionMaterialsProvider(keyId))
         .build();
```

配置 AWS KMS 客户端

除非明确指定了 AWS KMS 客户端，否则默认情况下，Amazon S3 加密客户端会创建一个该客户端。

要为这个自动创建的 AWS KMS 客户端设置区域，请设置 awsKmsRegion。

代码

```
Region kmsRegion = Region.getRegion(Regions.AP_NORTHEAST_1);

AmazonS3EncryptionV2 s3Encryption = AmazonS3EncryptionClientV2Builder.standard()
        .withRegion(Regions.US_WEST_2)
        .withCryptoConfiguration(new
 CryptoConfigurationV2().withAwsKmsRegion(kmsRegion))
        .withEncryptionMaterialsProvider(new KMSEncryptionMaterialsProvider(keyId))
        .build();
```

或者，您可以使用自己的 AWS KMS 客户端来初始化加密客户端。

代码

```
AWSKMS kmsClient = AWSKMSClientBuilder.standard()
        .withRegion(Regions.US_WEST_2);
        .build();
```

```
AmazonS3EncryptionV2 s3Encryption = AmazonS3EncryptionClientV2Builder.standard()
        .withRegion(Regions.US_WEST_2)
        .withKmsClient(kmsClient)
        .withCryptoConfiguration(new
 CryptoConfigurationV2().withCryptoMode((CryptoMode.AuthenticatedEncryption)))
        .withEncryptionMaterialsProvider(new KMSEncryptionMaterialsProvider(keyId))
        .build();
```

使用适用于 Java 的 AWS SDK 的 Amazon SQS 示例

此部分提供使用适用于 Java 的 AWS SDK 对 Amazon SQS 进行编程的示例。

> **Note**
>
> 这些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。您可以在那里下载单个源文件，也可以将存储库复制到本地以获得所有示例，然后构建并运行这些示例。

主题

- 使用 Amazon SQS 消息队列
- 发送、接收和删除 Amazon SQS 消息
- 为 Amazon SQS 消息队列启用长轮询
- 在 Amazon SQS 中设置可见性超时
- 在 Amazon SQS 中使用死信队列

使用 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 Developer Guide》。

本主题介绍如何使用适用于 Java 的 AWS SDK 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，并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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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。

导入。

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
```

代码

```
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 How Amazon SQS Queues Work
- 《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。提供包含队列 URL、消息正文和可选延迟值（以秒为单位）的 SendMessageRequest 对象。

导入。

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.SendMessageRequest;
```

代码

```
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;
```

代码

```
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` 方法并为其传递队列的 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;
```

代码

```
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。

代码

```
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 How Amazon SQS Queues Work
- 《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

为 Amazon SQS 消息队列启用长轮询

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;
```

代码

```
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;
```

代码

```
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;
```

代码

```
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 Long Polling

- 《Amazon SQS API Reference》中的 CreateQueue
- 《Amazon SQS API Reference》中的 ReceiveMessage
- 《Amazon SQS API Reference》中的 SetQueueAttributes

在 Amazon SQS 中设置可见性超时

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;
```

代码

```
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;
```

代码

```
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 Visibility Timeout
- 《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;
```

代码

```
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;
```

代码

```
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 Using Amazon SQS Dead Letter Queues
- 《Amazon SQS API Reference》中的 SetQueueAttributes

Amazon SWF使用 的 示例适用于 Java 的 AWS SDK

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。

通过适用于 Java 的 AWS SDK 使用 Amazon SWF 有两种方法：使用 SWF client 对象或者使用适用于 Java 的 AWS Flow Framework。适用于 Java 的 AWS Flow Framework 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，您可使用适用于 Java 的 AWS Flow Framework 来节省编写代码的时间。有关更多信息，请参阅《AWS Flow Framework for Java Developer Guide》。

此部分提供了直接使用Amazon SWF客户端来为 适用于 Java 的 AWS SDK 编程的示例。

主题

- SWF 基本知识
- 构建简单 Amazon SWF 应用程序
- Lambda 任务
- 适当地关闭活动和工作流工作线程
- 注册域

- 列出域

SWF 基本知识

这些是通过 Amazon SWF 使用适用于 Java 的 AWS SDK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。

依赖项

基本 Amazon SWF 应用程序将需要适用于 Java 的 AWS SDK附带的以下依赖项：

- aws-java-sdk-1.12.*.jar
- commons-logging-1.2.*.jar
- httpclient-4.3.*.jar
- httpcore-4.3.*.jar
- jackson-annotations-2.12.*.jar
- jackson-core-2.12.*.jar
- jackson-databind-2.12.*.jar
- joda-time-2.8.*.jar

> **Note**
>
> 虽然这些程序包的版本号将因您拥有的 SDK 版本而异，但 SDK 附带的版本已经过兼容性测试，并且您应使用这些版本。

适用于 Java 的 AWS Flow Framework 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 Developer Guide》。

导入

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;
```

如果您使用适用于 Java 的 AWS Flow Framework，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**
>
> 除了适用于 Java 的 AWS SDK 的基本要求外，适用于 Java 的 AWS Flow Framework 还有额外要求。有关更多信息，请参阅《AWS Flow Framework for Java Developer Guide》。

使用 SWF 客户端类

您通过 Amazon SWFAmazonSimpleWorkflowClient 或 AmazonSimpleWorkflowAsyncClient 类与 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();
```

构建简单 Amazon SWF 应用程序

此主题探讨如何使用适用于 Java 的 AWS SDK 编写 Amazon SWF 应用程序，并在此过程中介绍了一些重要概念。

关于示例

示例项目将创建带有一个活动的工作流，接受通过 AWS Cloud 传递的工作流数据（在 HelloWorld 的传统中，这应该是要问候的某个人的名字）并在响应中打印问候语。

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 Cloud 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。

先决条件

开发环境

此教程中使用的开发环境包括：

- -适用于 Java 的 AWS SDK 。
- 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**
>
> 如果您使用的构建系统不是 Maven，则仍可以使用适用于您环境的相应步骤创建项目，并在这个过程中使用此处提供的概念。适用于 Java 的 AWS SDK入门中提供了在不同编译系统中配置和使用 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 SDK 实施。

所有必需的外部依赖项包括在适用于 Java 的 AWS SDK 中，因此无需下载其他内容。

AWS 访问

要成功完成本教程，您必须有权访问 AWS 访问门户，如本指南的基本设置部分所述。

这些说明描述了如何访问您复制并粘贴到本地共享 credentials 文件中的临时凭证。您粘贴的临时凭证必须与 AWS IAM Identity Center 中有权访问 Amazon SWF 的 IAM 角色关联。粘贴临时凭证后，您的 credentials 文件将类似于以下内容。

```
[default]
aws_access_key_id=AKIAIOSFODNN7EXAMPLE
aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
aws_session_token=IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZVERYLONGSTRI
```

这些临时凭证与 default 配置文件相关联。

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>1.11.1000</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/aws/example/helloswf/ 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. 为 AmazonSimpleWorkflowClient 类和 com.amazonaws.services.simpleworkflow.model 命名空间中的多个类添加 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

我们将从创建新的可执行类 HelloTypes.java 开始。此文件将包含共享数据，您的工作流中的不同部分需要这些数据，例如活动的名称和版本以及工作流类型，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Amazon SWF向 适用于 Java 的 AWS SDK 方法提供的基本接口。

```
private static final AmazonSimpleWorkflow swf =

 AmazonSimpleWorkflowClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

前面的代码片段假设临时凭证与 default 配置文件相关联。如果您使用其他配置文件，请按如下方式修改上面的代码，然后将 *profile_name* 替换为实际配置文件的名称。

```
private static final AmazonSimpleWorkflow swf =
        AmazonSimpleWorkflowClientBuilder
```

```
                .standard()
                .withCredentials(new ProfileCredentialsProvider("profile_name"))
                .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 至 256 个字符组合）以及保留期，这是在工作流执行完成后，Amazon SWF 保留工作流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**
>
> 此代码演示了使用适用于 Java 的 AWS SDK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 `0with*` 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 Timeout Types。

如果您尝试注册的活动类型已存在，则将引发 TypeAlreadyExistsException。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类型已存在，则将引发 TypeAlreadyExistsException。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，您可以编译并运行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（未生成任何错误），则 worker 通过调用 SWF 客户端的 `respondActivityTaskCompleted` 方法来响应 SWF，该方法使用包含任务令牌和活动结果数据的 RespondActivityTaskCompletedRequest 对象。

另一方面，如果任务失败，则我们通过调用带有 RespondActivityTaskFailedRequest 对象的 `respondActivityTaskFailed` 方法进行响应，向其传递任务令牌和有关错误的信息。

> **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，这指示工作流执行已启动 (通常意味着您应该运行工作流中的第一个活动)，并且这提供了初始输入 (提供到工作流中)。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。

+ 注意：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());
```

1. 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。

2. 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 = "{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。

返回的运行`startWorkflowExecution`对象提供了运行 ID，这是用于在 Amazon SWF 的工作流执行历史记录中标识此特定工作流执行的值。

+ 注意：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、macOS 或 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 SDK 库及其依赖项。您可以将 CLASSPATH 环境变量设置为 AWS SDK 库的位置，以及 SDK 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、工作流和活动类型中实施。

在编译之后，如果您已设置 CLASSPATH，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、macOS 或 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。

有关更多信息

- 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。
- 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 Developer Guide》。
- 您可以使用适用于 Java 的 AWS Flow Framework，使用注释以更讲究的 Java 样式编写更复杂的工作流。如需了解更多信息，请参阅《AWS Flow Framework for Java Developer Guide》。

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适用于 Java 的 AWS SDK 实施 Amazon SWF Lambda 任务。有关 Lambda 任务的更多一般性信息，请参阅《Amazon SWF Developer Guide》中的 AWS Lambda Tasks。

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，授予让它代表您运行 Lambda 函数的 Amazon SWF 权限。有关如何完成该操作的完整信息，请参阅 AWS Lambda Tasks。

在注册将使用 Lambda 任务的工作流时，将需要此 IAM 角色的 Amazon 资源名称 (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代码，所有 Amazon SWF 工作流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 Amazon SWF 应用程序中的活动。

- 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 = "{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 开发人员指南》中的创建部署包 (Java)。您可能还希望查看标题为使用 Java 编写 Lambda 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 `withDefaultLambdaRole`RegisterWorkflowTypeRequest`setDefaultLambdaRole` 的 或 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。您提供一条决策，该决策具有“ScheduleLambdaFunction”DecisionType 和 ScheduleLambdaFunctionDecisionAttributes。

```
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 Amazon SWF 用于在历史记录日志中标识 Lambda 函数的名称。

还可以为 Lambda 函数提供可选的 input 并设置其 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 任务会使用 LambdaEventType 值 (如 、LambdaFunctionScheduled 和 LambdaFunctionStarted) 生成与 LambdaFunctionCompleted 任务生命周期对应的多个事件，在工作流工作线程中轮询决策任务时可以对这些事件执行操作。如果 Lambda 函数失败或运行时间超出其超时值，您会分别收到 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 HistoryEvent 上接收 LambdaFunctionCompleted`EventType, you can retrieve your 0 function's return value by first calling `getLambdaFunctionCompletedEventAttributes 时，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

您可以在 Github 上的 aws-java-developer-guide 存储库中，浏览 complete source :github:`<awsdocs/aws-java-developer-guide/tree/master/doc_source/snippets/helloswf_lambda/>，以查看此示例。

适当地关闭活动和工作流工作线程

构建简单 Amazon SWF 应用程序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 final 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域，以在其中运行。

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. 使用 RegisterDomainRequest 对象调用 AmazonSimpleWorkflowClient.registerDomain 方法。
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。

1. 创建 ListDomainsRequest 对象，然后指定目标域的注册状态（必填项）。
2. 使用 ListDomainRequest 对象调用 AmazonSimpleWorkflowClient.listDomains。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. 在每个 DomainInfo 对象上调用 getName 来获取其名称。

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 中包含的代码示例

适用于 Java 的 AWS SDK 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。您可以学习或修改这些程序，以使用适用于 Java 的 AWS SDK 实现您自己的 AWS 解决方案。

如何获取示例

适用于 Java 的 AWS SDK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 SDK 中的信息下载并安装 SDK，则您的系统中已包含示例。

您也可以在适用于 Java 的 AWS SDK 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。

先决条件

在运行任何适用于 Java 的 AWS SDK 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

1. 对包含示例代码的目录所做的更改。例如，如果您在 AWS SDK 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 {EC2} instance(s) running.

You have 13 Amazon SimpleDB domain(s) containing a total of 62 items.

You have 23 {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，也可以基于适用于 Java 的 AWS SDK在 Eclipse 中启动新项目或将该开发工具包添加到现有 Java 项目。

先决条件

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 中的窗口菜单选择首选项，然后选择 AWS Toolkit 部分。

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 项目。
4. 选择下一步。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 SDK 示例组显示了 SDK 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 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. 选择完成。这将创建项目并将其添加到 Project Explorer。
9. 选择要运行的示例 `.java` 文件。例如，对于 Amazon S3 示例，选择 `S3Sample.java`。
10. 从 Run 菜单中选择 Run。
11. 右键单击 Project Explorer 中的项目，指向 Build Path，然后选择 Add Libraries。
12. 选择 AWS Java SDK，然后选择下一步，并按照其余的屏幕说明执行操作。

安全性 适用于 Java 的 AWS SDK

云安全性一直是 Amazon Web Services（AWS）的重中之重。作为 AWS 客户，您将从专为满足大多数安全敏感型企业的要求而打造的数据中心和网络架构中受益。安全是双方共同承担 AWS 的责任。责任共担模式将其描述为云的安全性和云中的安全性。

云安全 — AWS 负责保护运行 AWS 云中提供的所有服务的基础架构，并为您提供可以安全使用的服务。我们的安全责任是重中之重 AWS，作为AWS 合规计划的一部分，第三方审计师定期测试和验证我们安全的有效性。

云端安全 — 您的责任由您使用的 AWS 服务以及其他因素决定，包括数据的敏感性、组织的要求以及适用的法律和法规。

本 AWS 产品或服务通过其支持的特定 Amazon Web Services (AWS) 服务遵循分担责任模式。有关 AWS 服务安全信息，请参阅AWS 服务安全文档页面和合规计划合 AWS 规工作范围内的AWS 服务。

主题

- 中的数据保护 适用于 Java 的 AWS SDK 1.x
- 适用于 Java 的 AWS SDK 支持 TLS
- 身份和访问管理
- 为此进行合规性验证 AWS 产品或服务
- 为此具有韧性 AWS 产品或服务
- 为此提供基础设施安全 AWS 产品或服务
- Amazon S3 加密客户端迁移

中的数据保护 适用于 Java 的 AWS SDK 1.x

责任共担模式适用于本 AWS 产品或服务中的数据保护。如本模型所述 AWS ，负责保护运行所有 AWS 云的全球基础架构。您负责维护对托管在此基础设施上的内容的控制。此内容包括您所使用的 AWS 服务的安全配置和管理任务。有关数据隐私的更多信息，请参阅数据隐私常见问题。有关欧洲数据保护的信息，请参阅 AWS 安全博客上的责任AWS 共担模型和 GDPR 博客文章。

出于数据保护目的，我们建议您保护 AWS 账户 凭据并使用 AWS Identity and Access Management (IAM) 设置个人用户帐户。这仅向每个用户授予履行其工作职责所需的权限。我们还建议您通过以下方式保护数据：

- 对每个账户使用多重身份验证（MFA）。

- 用于 SSL/TLS 与 AWS 资源通信。
- 使用设置 API 和用户活动日志 AWS CloudTrail。
- 使用 AWS 加密解决方案，在 AWS 服务中使用所有默认安全控制。
- 使用高级托管安全服务，例如 Amazon Macie，该服务有助于发现和保护存储在 Amazon S3中的个人数据。
- 如果您在 AWS 通过命令行界面或 API 进行访问时需要经过 FIPS 140-2 验证的加密模块，请使用 FIPS 端点。有关可用的 FIPS 端点的更多信息，请参阅美国联邦信息处理标准（FIPS）第 140-2 版。

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（例如名称字段）。这包括您使用控制台、API 或 AWS SDK 使用本 AWS 产品或 AWS 服务或其他服务时。 AWS CLI您在本 AWS 产品或服务或其他服务中输入的任何数据都可能会被提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。

适用于 Java 的 AWS SDK 支持 TLS

以下信息仅适用于 Java SSL 实现（中的默认 SSL 实现 适用于 Java 的 AWS SDK）。如果您使用的是其他 SSL 实现，请参阅与该特定 SSL 实现相关的信息，以了解如何强制执行 TLS 版本。

如何查看 TLS 版本

请查阅 Java 虚拟机 (JVM) 提供商的文档，以确定您的平台支持哪些 TLS 版本。对于某些 JVM，以下代码将打印支持哪些 SSL 版本。

```
System.out.println(Arrays.toString(SSLContext.getDefault().getSupportedSSLParameters().getProto
```

要查看 SSL 握手的运行情况以及使用的 TLS 版本，可使用系统属性 javax.net.debug。

```
java app.jar -Djavax.net.debug=ssl
```

> **Note**
>
> TLS 1.3 与适用于 Java 的 SDK 版本 1.9.5 至 1.10.31 不兼容。有关更多信息，请参阅以下博客文章。
> https://aws.amazon.com/blogs/developer/tls-1-3-incompatibility-with-aws-sdk-for-java-versions-1-9-5-to-1-10-31/

强制实施最低 TLS 版本

SDK 始终会将平台和服务支持的最新 TLS 版本作为其首选。如果您希望强制指定特定的最低 TLS 版本，请查阅 JVM 的文档。对于 OpenJDK-based JVM，您可以使用系统属性。`jdk.tls.client.protocols`

```
java app.jar -Djdk.tls.client.protocols=PROTOCOLS
```

有关支持的 PROTOCOLS 值，请参阅您的 JVM 文档。

身份和访问管理

AWS Identity and Access Management (IAM) AWS 服务 可帮助管理员安全地控制对 AWS 资源的访问权限。IAM 管理员控制谁可以进行身份验证（登录）和授权（有权限）使用 AWS 资源。您可以使用 IAM AWS 服务 ，无需支付额外费用。

主题

- 受众
- 使用身份进行身份验证
- 使用策略管理访问
- 操作方法 AWS 服务 使用 IAM
- 故障排除 AWS 身份和访问权限

受众

您的使用方式 AWS Identity and Access Management (IAM) 会有所不同，具体取决于您所做的工作 AWS。

服务用户-如果您 AWS 服务 曾经完成工作，则您的管理员会为您提供所需的凭证和权限。当你使用更多 AWS 功能来完成工作时，你可能需要额外的权限。了解如何管理访问权限有助于您向管理员请求适合的权限。如果您无法访问中的功能 AWS，请参阅故障排除 AWS 身份和访问权限或 AWS 服务 您正在使用的用户指南。

服务管理员-如果您负责公司的 AWS 资源，则可能拥有完全访问权限 AWS。您的工作是确定您的服务用户应访问哪些 AWS 功能和资源。然后，您必须向 IAM 管理员提交请求以更改服务用户的权限。请查看该页面上的信息以了解 IAM 的基本概念。要详细了解您的公司如何使用 IAM AWS，请参阅 AWS 服务 您正在使用的用户指南。

IAM 管理员：如果您是 IAM 管理员，您可能希望了解如何编写策略以管理对 AWS的访问权限的详细信息。要查看您可以在 IAM 中使用的 AWS 基于身份的策略示例，请参阅 AWS 服务 您正在使用的用户指南。

使用身份进行身份验证

身份验证是您 AWS 使用身份凭证登录的方式。您必须以 IAM 用户身份进行身份验证 AWS 账户根用户，或者通过担任 IAM 角色进行身份验证。

您可以使用来自身份源的证书 AWS IAM Identity Center （例如（IAM Identity Center）、单点登录身份验证或 Google/Facebook 证书，以联合身份登录。有关登录的更多信息，请参阅《AWS 登录 用户指南》中的如何登录您的 AWS 账户。

对于编程访问， AWS 提供 SDK 和 CLI 来对请求进行加密签名。有关更多信息，请参阅《IAM 用户指南》中的适用于 API 请求的AWS 签名版本 4。

AWS 账户 根用户

创建时 AWS 账户，首先会有一个名为 AWS 账户 root 用户的登录身份，该身份可以完全访问所有资源 AWS 服务 和资源。我们强烈建议不要使用根用户进行日常任务。有关需要根用户凭证的任务，请参阅《IAM 用户指南》中的需要根用户凭证的任务。

联合身份

作为最佳实践，要求人类用户使用与身份提供商的联合身份验证才能 AWS 服务 使用临时证书进行访问。

联合身份是指来自您的企业目录、Web 身份提供商的用户 Directory Service ，或者 AWS 服务 使用来自身份源的凭据进行访问的用户。联合身份代入可提供临时凭证的角色。

要集中管理访问权限，建议使用。 AWS IAM Identity Center有关更多信息，请参阅《AWS IAM Identity Center 用户指南》中的什么是 IAM Identity Center？。

IAM 用户和群组

IAM 用户是对某个人员或应用程序具有特定权限的一个身份。建议使用临时凭证，而非具有长期凭证的 IAM 用户。有关更多信息，请参阅 IAM 用户指南中的要求人类用户使用身份提供商的联合身份验证才能 AWS 使用临时证书进行访问。

IAM 组指定一组 IAM 用户，便于更轻松地对大量用户进行权限管理。有关更多信息，请参阅《IAM 用户指南》中的 IAM 用户使用案例。

IAM 角色

IAM 角色是具有特定权限的身份，可提供临时凭证。您可以通过从用户切换到 IAM 角色（控制台）或调用 AWS CLI 或 AWS API 操作来代入角色。有关更多信息，请参阅《IAM 用户指南》中的担任角色的方法。

IAM 角色对于联合用户访问、临时 IAM 用户权限、跨账户访问、跨服务访问以及在 Amazon EC2 上运行的应用程序非常有用。有关更多信息，请参阅《IAM 用户指南》中的 IAM 中的跨账户资源访问。

使用策略管理访问

您可以 AWS 通过创建策略并将其附加到 AWS 身份或资源来控制中的访问权限。策略定义了与身份或资源关联时的权限。 AWS 在委托人提出请求时评估这些政策。大多数策略都以 JSON 文档的 AWS 形式存储在中。有关 JSON 策略文档的更多信息，请参阅《IAM 用户指南》中的 JSON 策略概述。

管理员使用策略，通过定义哪个主体可以在什么条件下对哪些资源执行哪些操作来指定谁有权访问什么。

默认情况下，用户和角色没有权限。IAM 管理员创建 IAM 策略并将其添加到角色中，然后用户可以担任这些角色。IAM 策略定义权限，与执行操作所用的方法无关。

Identity-based 政策

Identity-based 策略是您附加到身份（用户、组或角色）的 JSON 权限策略文档。这些策略控制身份可以执行什么操作、对哪些资源执行以及在什么条件下执行。要了解如何创建基于身份的策略，请参阅《IAM 用户指南》中的使用客户管理型策略定义自定义 IAM 权限。

Identity-based 策略可以是内联策略（直接嵌入到单个身份中）或托管策略（附加到多个身份的独立策略）。要了解如何在托管式策略和内联策略之间进行选择，请参阅《IAM 用户指南》中的在托管式策略与内联策略之间进行选择。

Resource-based 政策

Resource-based 策略是您附加到资源的 JSON 策略文档。示例包括 IAM 角色信任策略和 Amazon S3 存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。您必须在基于资源的策略中指定主体。

Resource-based 策略是位于该服务中的内联策略。您不能在基于资源的策略中使用 IAM 中的 AWS 托管策略。

访问控制列表（ACL）

访问控制列表（ACL）控制哪些主体（账户成员、用户或角色）有权访问资源。ACL 与基于资源的策略类似，但它们不使用 JSON 策略文档格式。

Amazon S3 和 Amazon VPC 就是支持 ACL 的服务示例。 AWS WAF要了解有关 ACL 的更多信息，请参阅《Amazon Simple Storage Service 开发人员指南》中的访问控制列表（ACL）概览。

其他策略类型

AWS 支持其他策略类型，这些策略类型可以设置更常见的策略类型授予的最大权限：

- 权限边界 – 设置基于身份的策略可以授予 IAM 实体的最大权限。有关更多信息，请参阅《 IAM 用户指南》中的 IAM 实体的权限边界。
- 服务控制策略（SCP）– 指定 AWS Organizations中组织或组织单元的最大权限。有关更多信息，请参阅《AWS Organizations 用户指南》中的服务控制策略。
- 资源控制策略（RCP）– 设置对账户中资源的最大可用权限。有关更多信息，请参阅《AWS Organizations 用户指南》中的资源控制策略（RCP）。
- 会话策略 – 在为角色或联合用户创建临时会话时，作为参数传递的高级策略。有关更多信息，请参阅《IAM 用户指南》中的会话策略。

多个策略类型

当多个类型的策略应用于一个请求时，生成的权限更加复杂和难以理解。要了解在涉及多种策略类型时如何 AWS 确定是否允许请求，请参阅 IAM 用户指南中的策略评估逻辑。

操作方法 AWS 服务 使用 IAM

要全面了解如何 AWS 服务 使用大多数 IAM 功能，请参阅 IAM 用户指南中的与 IAM 配合使用的AWS 服务。

要了解如何在 IAM 中 AWS 服务 使用特定的，请参阅相关服务的用户指南的安全部分。

故障排除 AWS 身份和访问权限

使用以下信息来帮助您诊断和修复在使用 AWS 和 IAM 时可能遇到的常见问题。

主题

- 我无权在以下位置执行操作 AWS

- 我无权执行 iam：PassRole
- 我想允许我以外的人进入 AWS 账户 访问我的 AWS 资源

我无权在以下位置执行操作 AWS

如果您收到错误提示，指明您无权执行某个操作，则必须更新策略以允许执行该操作。

当 mateojackson IAM 用户尝试使用控制台查看有关虚构 *my-example-widget* 资源的详细信息，但不拥有虚构 awes:*GetWidget* 权限时，会发生以下示例错误。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 awes:GetWidget on resource: my-example-widget
```

在此情况下，必须更新 mateojackson 用户的策略，以允许使用 awes:*GetWidget* 操作访问 *my-example-widget* 资源。

如果您需要帮助，请联系您的 AWS 管理员。您的管理员是提供登录凭证的人。

我无权执行 iam：PassRole

如果您收到一个错误，表明您无权执行 iam:PassRole 操作，则必须更新策略以允许您将角色传递给。 AWS

有些 AWS 服务 允许您将现有角色传递给该服务，而不是创建新的服务角色或服务相关角色。为此，您必须具有将角色传递到服务的权限。

当名为 marymajor 的 IAM 用户尝试使用控制台在 AWS中执行操作时，会发生以下示例错误。但是，服务必须具有服务角色所授予的权限才可执行此操作。Mary 不具有将角色传递到服务的权限。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

在这种情况下，必须更新 Mary 的策略以允许她执行 iam:PassRole 操作。

如果您需要帮助，请联系您的 AWS 管理员。您的管理员是提供登录凭证的人。

我想允许我以外的人进入 AWS 账户 访问我的 AWS 资源

您可以创建一个角色，以便其他账户中的用户或您组织外的人员可以使用该角色来访问您的资源。您可以指定谁值得信赖，可以代入角色。对于支持基于资源的策略或访问控制列表（ACL）的服务，您可以使用这些策略向人员授予对您的资源的访问权。

要了解更多信息，请参阅以下内容：

- 要了解是否 AWS 支持这些功能，请参阅操作方法 AWS 服务 使用 IAM。
- 要了解如何提供对您拥有的资源的访问权限 AWS 账户 ，请参阅 IAM 用户指南中的向您拥有 AWS 账户 的另一个 IAM 用户提供访问权限。
- 要了解如何向第三方提供对您的资源的访问权限 AWS 账户，请参阅 IAM 用户指南中的向第三方提供访问权限。 AWS 账户
- 要了解如何通过身份联合验证提供访问权限，请参阅《IAM 用户指南》中的为经过外部身份验证的用户（身份联合验证）提供访问权限。
- 要了解使用角色和基于资源的策略进行跨账户访问之间的差别，请参阅《IAM 用户指南》中的 IAM 中的跨账户资源访问。

为此进行合规性验证 AWS 产品或服务

我们全新的 AWS 注册体验不是为受监管的工作负载而设计的。如果您正在使用我们的全新 AWS 注册体验，但又想 AWS 用于受监管的工作负载，则可以注册 AWS （高级）或激活适合您 AWS 环境的高级功能。

要了解是否属于特定合规计划的范围，请参阅AWS 服务 “按合规计划划分的范围” ”，然后选择您感兴趣的合规计划。 AWS 服务 有关一般信息，请参阅AWS 合规计划AWS。

您可以使用下载第三方审计报告 AWS Artifact。有关更多信息，请参阅中的 “下载报告” 中的 “ AWS Artifact。

您在使用 AWS 服务 时的合规责任取决于您的数据的敏感性、贵公司的合规目标以及适用的法律和法规。有关您在使用时的合规责任的更多信息 AWS 服务，请参阅AWS 安全文档。

本 AWS 产品或服务通过其支持的特定 Amazon Web Services (AWS) 服务遵循分担责任模式。有关 AWS 服务安全信息，请参阅AWS 服务安全文档页面和合规计划合 AWS 规工作范围内的AWS 服务。

为此具有韧性 AWS 产品或服务

AWS 全球基础设施是围绕 AWS 区域 可用区构建的。

AWS 区域 提供多个物理分隔和隔离的可用区，这些可用区通过低延迟、高吞吐量和高度冗余的网络连接。

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失效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、容错能力和可扩展性。

有关 AWS 区域和可用区的更多信息，请参阅AWS 全球基础设施。

本 AWS 产品或服务通过其支持的特定 Amazon Web Services (AWS) 服务遵循分担责任模式。有关 AWS 服务安全信息，请参阅AWS 服务安全文档页面和合规计划合 AWS 规工作范围内的AWS 服务。

为此提供基础设施安全 AWS 产品或服务

本 AWS 产品或服务使用托管服务，因此受到 AWS 全球网络安全的保护。有关 AWS 安全服务以及如何 AWS 保护基础设施的信息，请参阅AWS 云安全。要使用基础设施安全的最佳实践来设计您的 AWS 环境，请参阅 S AWS ecurity Pillar Well-Architected Fram ework 中的基础设施保护。

您可以使用 AWS 已发布的 API 调用通过网络访问此 AWS 产品或服务。客户端必须支持以下内容：

- 传输层安全性协议（TLS）。我们要求使用 TLS 1.2，并建议使用 TLS 1.3。
- 具有完全向前保密 (PFS) 的密码套件，例如 DHE（短暂的）或 ECDHE（椭圆曲线短暂的 Diffie-Hellman）。 Diffie-Hellman大多数现代系统（如 Java 7 及更高版本）都支持这些模式。

此外，必须使用访问密钥 ID 和与 IAM 主体关联的秘密访问密钥来对请求进行签名。或者，您可以使用 AWS Security Token Service（AWS STS）生成临时安全凭证来对请求进行签名。

本 AWS 产品或服务通过其支持的特定 Amazon Web Services (AWS) 服务遵循分担责任模式。有关 AWS 服务安全信息，请参阅AWS 服务安全文档页面和合规计划合 AWS 规工作范围内的AWS 服务。

Amazon S3 加密客户端迁移

本主题介绍如何将应用程序从 () 加密客户端的版本 1 (V1) 迁移到版本 2 Amazon Simple Storage Service (Amazon S3 V2)，并确保应用程序在整个迁移过程中的可用性。

先决条件

Amazon S3 客户端加密需要满足以下条件：

- 应用程序环境中安装了 Java 8 或更高版本。 适用于 Java 的 AWS SDK 它可与 Oracle Java SE 开发套件以及红帽 OpenJDK 和 JDK Amazon Corretto等开放 Java 开发套件 (OpenJDK) 的发行版配合使用。AdoptOpen

- Bouncy Castle 加密套餐。你可以将 Bouncy Castle .jar 文件放在应用程序环境的类路径上，也可以在 Maven pom.xml 文件中添加 artifactId bcprov-ext-jdk15on（groupId 为 org.bouncycastle）的依赖项。

迁移概述

此迁移分为两个阶段：

1. 更新现有客户端以读取新格式。将您的应用程序更新为使用 1.11.837 或更高版本， 适用于 Java 的 AWS SDK 然后重新部署该应用程序。这使应用程序中的 Amazon S3 客户端加密服务客户端能够解密由 V2 服务客户端创建的对象。如果您的应用程序使用多个 AWS SDK，则必须分别更新每个 SDK。
2. 将加密和解密客户端迁移到 V2。在所有 V1 加密客户端都能读取 V2 加密格式后，请更新应用程序代码中的 Amazon S3 客户端加密和解密客户端，以使用其 V2 等效格式。

更新现有客户端以读取新格式

V2 加密客户端使用旧版本 适用于 Java 的 AWS SDK 不支持的加密算法。

迁移的第一步是更新您的 V1 加密客户端，使其使用版本 1.11.837 或更高版本的 适用于 Java 的 AWS SDK。（建议您更新到最新发行版本，您可以在 Java API Reference version 1.x 中找到该版本。）为此，请更新项目配置中的依赖项。更新项目配置后，重新构建项目并重新部署。

完成这些步骤后，应用程序的 V1 加密客户端将能够读取 V2 加密客户端写入的对象。

更新项目配置中的依赖项

修改项目配置文件（例如 pom.xml 或 build.gradle）以使用版本 1.11.837 或更高版本的 适用于 Java 的 AWS SDK。然后重新构建项目并重新部署。

在部署新的应用程序代码之前完成此步骤，有助于确保整个实例集在迁移过程中的加密和解密操作保持一致。

使用 Maven 的 示例

pom.xml 文件中的代码段：

```
<dependencyManagement>
  <dependencies>
```

```
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-java-sdk-bom</artifactId>
      <version>1.11.837</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

使用 Gradle 的示例

build.gradle 文件中的代码段：

```
dependencies {
  implementation platform('com.amazonaws:aws-java-sdk-bom:1.11.837')
  implementation 'com.amazonaws:aws-java-sdk-s3'
}
```

将加密和解密客户端迁移到 V2

使用最新的 SDK 版本更新项目后，您可以修改应用程序代码以使用 V2 客户端。为此，请先更新您的代码以使用新的服务客户端生成器。然后在该生成器上使用已重命名的方法提供加密材料，并根据需要进一步配置您的服务客户端。

这些代码片段演示了如何使用客户端加密 适用于 Java 的 AWS SDK，并提供了 V1 和 V2 加密客户端之间的比较。

V1

```
// minimal configuration in V1; default CryptoMode.EncryptionOnly.
EncryptionMaterialsProvider encryptionMaterialsProvider = ...
AmazonS3Encryption encryptionClient = AmazonS3EncryptionClient.encryptionBuilder()
             .withEncryptionMaterials(encryptionMaterialsProvider)
             .build();
```

V2

```
// minimal configuration in V2; default CryptoMode.StrictAuthenticatedEncryption.
EncryptionMaterialsProvider encryptionMaterialsProvider = ...
AmazonS3EncryptionV2 encryptionClient = AmazonS3EncryptionClientV2.encryptionBuilder()
```

```
            .withEncryptionMaterialsProvider(encryptionMaterialsProvider)
            .withCryptoConfiguration(new CryptoConfigurationV2()
                        // The following setting allows the client to read V1
encrypted objects
                        .withCryptoMode(CryptoMode.AuthenticatedEncryption)
            )
            .build();
```

上面的示例将 cryptoMode 设置为 AuthenticatedEncryption。此设置允许 V2 加密客户端读取 V1 加密客户端写入的对象。如果您的客户端不需要读取 V1 客户端写入的对象的功能，建议改用默认设置 StrictAuthenticatedEncryption。

构建 V2 加密客户端

可以通过调用 AmazonS3EncryptionClientV2.encryptionBuilder() 来构建 V2 加密客户端。

您可以将所有现有的 V1 加密客户端替换为 V2 加密客户端。只要您通过将 V2 加密客户端配置为使用 `来允许 V2 加密客户端写入的任何对象，V2 加密客户端将始终能够读取 V1 加密客户端写入的任何对象。AuthenticatedEncryption `cryptoMode

创建新的 V2 加密客户端与创建 V1 加密客户端的方式非常相似。但还是有几个区别：

- 您将使用 CryptoConfigurationV2 对象而不是 CryptoConfiguration 对象来配置客户端。此参数为必需参数。
- V2 加密客户端的默认 cryptoMode 设置是 StrictAuthenticatedEncryption。对于 V1 加密客户端，该默认设置是 EncryptionOnly。
- 加密客户端生成器上带有 EncryptionMaterials () 的方法已重命名为 w it EncryptionMaterialsProvider h ()。这只是一个表面更改，可以更准确地反映参数类型。配置服务客户端时必须使用新方法。

> **Note**
>
> 使用解密时 AES-GCM，在开始使用解密后的数据之前，请将整个对象读到最后。这是为了验证自加密以来是否未对对象进行过修改。

使用加密材料提供程序

您可以继续使用与 V1 加密客户端相同的加密材料提供程序和加密材料对象。这些类负责提供加密客户端用来保护数据的密钥。它们可以在 V2 和 V1 加密客户端中互换使用。

配置 V2 加密客户端

V2 加密客户端配置了一个 CryptoConfigurationV2 对象。可以通过调用其默认构造函数，然后根据需要修改其属性的默认值来构造此对象。

CryptoConfigurationV2 默认值如下所示：

- cryptoMode = CryptoMode.StrictAuthenticatedEncryption
- storageMode = CryptoStorageMode.ObjectMetadata
- secureRandom = SecureRandom 的实例
- rangeGetMode = CryptoRangeGetMode.DISABLED
- unsafeUndecryptableObjectPassthrough = false

请注意 EncryptionOnly，V2 加密客户端不支持cryptoMode该功能。V2 加密客户端将始终使用经过身份验证的加密来加密内容，并使用 V2 KeyWrap 对象保护内容加密密钥 (CEK)。

以下示例演示如何在 V1 中指定加密配置，以及如何实例化 CryptoConfigurationV2 对象以传递给 V2 加密客户端生成器。

V1

```
CryptoConfiguration cryptoConfiguration = new CryptoConfiguration()
        .withCryptoMode(CryptoMode.StrictAuthenticatedEncryption);
```

V2

```
CryptoConfigurationV2 cryptoConfiguration = new CryptoConfigurationV2()
        .withCryptoMode(CryptoMode.StrictAuthenticatedEncryption);
```

其他示例

以下示例演示如何解决与从 V1 迁移到 V2 相关的特定用例。

将服务客户端配置为读取 V1 加密客户端创建的对象

要读取以前使用 V1 加密客户端写入的对象，请将设置 cryptoMode 为 AuthenticatedEncryption。以下代码段演示如何使用此设置构造配置对象。

```
CryptoConfigurationV2 cryptoConfiguration = new CryptoConfigurationV2()
```

```
        .withCryptoMode(CryptoMode.AuthenticatedEncryption);
```

配置服务客户端以获取对象的字节范围

要能够从加密的 S3 对象中 get 字节范围，请启用新的配置设置 rangeGetMode。默认情况下，此设置在 V2 加密客户端上处于禁用状态。请注意，即使启用了此设置，具有范围的 get 也只能对使用客户端 cryptoMode 设置支持的算法进行加密的对象起作用。有关更多信息，请参阅 适用于 Java 的 AWS SDK API 参考CryptoRangeGetMode中的。

如果您计划使用 V2 加密客户端 Amazon S3 TransferManager 对加密 Amazon S3 对象执行分段下载，则必须先在 V2 加密客户端上启用该rangeGetMode设置。

以下代码段演示如何配置 V2 客户端以执行具有范围的 get。

```
// Allows range gets using AES/CTR, for V2 encrypted objects only
CryptoConfigurationV2 cryptoConfiguration = new CryptoConfigurationV2()
       .withRangeGetMode(CryptoRangeGetMode.ALL);

// Allows range gets using AES/CTR and AES/CBC, for V1 and V2 objects
CryptoConfigurationV2 cryptoConfiguration = new CryptoConfigurationV2()
       .withCryptoMode(CryptoMode.AuthenticatedEncryption)
       .withRangeGetMode(CryptoRangeGetMode.ALL);
```

适用于 适用于 Java 的 AWS SDK 的 OpenPGP 密钥

适用于 Java 的 AWS SDK 的所有公开可用的 Maven 构件均使用 OpenPGP 标准进行签名。验证构件签名所需的公有密钥可在下一部分中找到。

当前密钥

下表显示了 SDK for Java 1x 和 SDK for Java 2.x 的当前版本的 OpenPGP 密钥信息。

密钥 ID	0xAC107B386692DADD
类型	RSA
大小	4096/4096
创建时间	2016-06-30
过期时间	2026-09-27
用户 ID	AWS SDK 和工具 <aws-dr-tools@amazon.com>
密钥指纹	FEB9 209F 2F2F 3F46 6484 1E55 AC10 7B38 6692 DADD

要将以下适用于 SDK for Java 的 OpenPGP 公有密钥复制到剪贴板，请选择右上角的“复制”图标。

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Comment: Hostname:
Version: Hockeypuck 2.2

xsFNBFd1gAUBEACqbmmFbxdJgz1lD7wrlskQA1LLuSAC4p8ny9u/D2zLR8Ynk3Yz
mzJuQ+Kfjne2t+xTDex6MPJlMYpOviSWsX2psgvdmeyUpW9ap0lrThNYkc+W5fRc
buFehfbi9LSATZGJi8RG0sCCr5FsYVz0gEk85M2+PeM24cXhQIOZtQUjswX/pdk/
KduGtZASqNAYLKROmRODzUuaokLPo24pfm9bnr1RnRtwt5ktPAA5bM9ZZaGKriej
kT2lPffbBjp8F5AZvmGLtNm2Cmg4FKBvI04SQjy2jjrQ3wBzi5Lc9HTxDuHK/rtV
u6PewUe2WPlnxlXenhMZU1UK4YoSB9E9StQ2VxQiySLHSdxR7Ma4WgYdVLn9bOie
nj3QxLuQ1ZUKF79ES6JaM4tOz1gGcQeU1+UklgjFLuKwmzWRdEIFfxMyvH6qgKnd
U+DioH5mcUwhwffAAsuIJyAdMIEUYh7IfzJJXQf+fF+XfOCl6byOJFWrIGQkAzMu
```

```
CEvaCfwtHC2Lpzo33/WRFeMAuzzd0QJ4uz4xFFvaSOSZHMLHWI9YV/+Pea3X99Ms
0Nlek/LolAJh67MynHeVBOHKrq+fluorWepQivctzN6Y1NOkx5naTPGGaKWK7G2q
TbcY5SMnkIWfLFSougj0Fvmjczq8iZRwYxWA+i+LQvsR9WEXEiQffIWRoQARAQAB
zSxBV1MgU0RLcyBhbmQgVG9vbHMgPGF3cy1kci10b29sc0BhbWF6b24uY29tPsLB
lAQTAQoAPgIbAwULCQgHAwUVCgkICwUWAgMBAAIeAQIXgBYhBP65IJ8vLz9GZIQe
VawQezhmktrdBQJo12ZrBQkTQxnmAAoJEKwQezhmktrdi18P/A3De83MBx8bdcWJ
Fot71Vk1TyBQFErgrtcytSU0czEHx3tGbzgQLbMlyzjirOT03usxEkOeqTVK+RU+
5uFXNZYQLwMJlHJ6S8tnfLe/ExM5WQ2KPwIUPfZs1GDDRQB2dIKSc+qYrP1O1vf4
O4iPgfLHMW2bFh3zjjxcaHCJyqc7Cau33eZFBAsRni1jOUo7MeyX0hlXfW8pd48Q
wZllQVZ/6KmDiFWA0CZ+2svJ5cL0tgPoh1OQjoz0nHpNfuDILMrZ+e7tx2VTlkGH
UGeNSydnrK8v9ztFn34KtU/k7NEWoVSyEi+5ICZL18FBwPqTwdVWXwXrqZCKiIpr
8ZdJWDz2sJfgDFNCC6rKgCQ6FrmaD9G76dYWkQ4AbZqABlUzU3q36W1K0r3iOAb5
G4tdOt4yqXHTe1x+ZUNaeW7gaCmtXAxLw0OfeJrcq/44b/SQP+qJ8sSOv76Yg2oF
BsF5DWOVUFghbTyokHAoVROyhBR4dUUisY39AqLSL8+Lp9Pr3wNuGl9GLrMD5701
piUb88B3Gwe1EiKV1gaKrvZ3mECDUiSMVO0Z5iG8E4QDpNmVbJbV1uT821ubvtOv
2Ko10Fa0uwCYGssdRGqEXNy6jz/Er8LAC3+nmGINDJQzrF+loYoSSkI2Nu7lhMuL
7iWwUPF7OhDXoVSAn4X3x6q2rGK0wsGUBBMBCgA+AhsDBQsJCAcDBRUKCQgLBRYC
AwEAAh4BAheAFiEE/rkgny8vP0ZkhB5VrBB7OGaS2t0FAmjXZTsFCRNDGLYACgkQ
rBB7OGaS2t0/Ow//YIv51vHtD+kwMmIvk3zpizDHY0zW2dOezAo+C/DsSyC7wDll
Dixw34EQ1yLXH5xLR8CH1zupl3JmmEp1ucdQggoefbidxDl8Fld7tJOD1y3GGnTD
0jAl2ZC+W65Oh+wS1mD1FlaKjMGGkvJf0dA7RtU2T8dv3vt8dsxg76FMFS3+fqlC
FNOAsNTn9zWR1SqBIfkMJK83aq6s/rcEV9VrAYgDgqex58fygB5EuTf842/IF7WZ
Q9gd6fupB0mMZP5YWd2uj/vsBTYakG+mgQwDxZuKPeEzAqnqqS7biSQOUO6Wozlq
Yy4fSczE9GkBAvg0pGmbko+zHvpnjvX/h1CUpC6odvFyOAhZp6zyhs0QWz9thfqV
lU8WlbgJ2atFDn5GUSxF/fe0Yzovlbbs6sbYXuvMG9RiEOuJ1mBbZR3aIdZ1U6Do
BHc/vjc5mWcV7JQSP7i4W/8W7X3UAuN9LdxB+IvF3Cwrgtlw2BWvA5Alco5Tnz8t
P/CIVmBjk+sLme8W4kfLK3IWEbwClOdNnErI/MHRm65A2Y5EMIhwjrOi07SU1Pxa
nPpg3OYJCdvjzdB8QE3/DBiMfOl4dISfKDVEWnfK8mZaYd/BeRm2gUAa9UrqSFCG
BlA7Lg+eLI3US0FvwWJ4j5bBJqgLu+y7crIkiUOPAQuLk3lO+5uYU/I3DuLCwZQE
EwEKAD4CGwMFCwkIBwMFFQoJCAsFFgIDAQACHgECF4AWIQT+uSCfLy8/RmSEHlWs
EHs4ZpLa3QUCZwAXCAUJEWvKgwAKCRCsEHs4ZpLa3ZdTEACMBLg2q9zk8ZH02nDz
Sg5zc8Wlqq8WdxU0Pj8qx4UOrrMca7wyiUvrgoxPW5lh1RVNUeMkDRfu9pSXcOVI
V9LvmYE/WnwKROubgGbsC4T7M/LqV0/AulXil4d7IXcO6l4toa8LTNWtD5bODgrN
gvay1AzCU8kq1Qw1cKZ2gAfvA3Ba7PWyLeUN4HTlGrXcw73G+0CofY1L8wqWxHCJ
29XqQzeTEc6MDEeIlNlVdUcy8Qr5uwkEsl34H9AxS5F1opJ4TqvXiDZsrSRRv57R
XYmRZDWeYT+9PZaMsHXza5qgej7BfATxhYfICsNaY6MK3x6b+nDSKkoZgO+i09zh
1YjpahhQe6G336v/3mRj0dKGCRQ6znQ9ghUaB5z9zfvgH5AOEkTe3l8MqM+j5A6P
VjSBBJAHKejxr7+wKJKIA6P+DqpsYAunzftwUzrLVqb+BZQ+DcTmVrE7OPcMYJD5
QglX/Le+WmWZHI154NXgpWWUOUgZUbUge4DKrT+zCJ9iecPLKTW7OcULyXO+rjb8
8BGrD5GPlHB3dOUXXTlMKCqg3qy1Bu2KnZTQiaEEdZgSIGQbrW0JTMmmXJkKjokd
JMA4vYeg5en51G9nRQjScPngx77IxvByNyFWTJdG1ENpJpsK9TtmENcpyUJtJZTJ
ZSOIRVPP5RzR5vInuXWq6VV0BMLBlAQTAQoAPgIbAwULCQgHAwUVCgkICwUWAgMB
AAIeAQIXgBYhBP65IJ8vLz9GZIQeVawQezhmktrdBQJlJEoiBQkPj/2dAAoJEKwQ
ezhmktrdx1YP/0vvym3jgX/pwnR7K1rafZMb1iKQBr0ISG8cdbaf4pqX5vuUZnyj
```

```
w9Cl/oONn7jJjnQxOIIzuBoxne2WN28ftM2w0nVXm85mAmz2fwQz/fdKDyonXcOh
pfD2iMqn7gESjhEgRE7wMDYMDuLdqHI7OKWGVfgrh7xEmKapLh45h7cnumo2VjL9
uDYY1aOBHz993T7oE41y43rhk+6kKbGFd2uuo7h5j1ZF8Lj6sYfcEzXOU1OhRlD0
nyBjDy9MYWu0YNouc70WgMceGx6hjvCAM/5fxP7SZFecZ7ePeB0GpvVA24hSNENE
0r3tUekuOf1I0FunMnMnbh7ZO9rPYqWvWDNIpU3S4CjFhY82L+IeKnmLy8N6ASRk
HsPiNCOHSK8C/0ynrd9xLhX8Jsk/TGiQYaleoHhWkNLlZsL86QHL8SKEqkqZCQf5
AEqghDP6NEGS7lnOenA7JjIrA9KLlT7fnNWZOwFi5X+o/CymE2ytEMS0Yf3nmY4U
n9x56Wgn6J2zqB5nqOXf6NxGdAIgOBm098YEnKCIFzk+yhoDlprVpHcnd2b5f6Oq
uh8KYOEbKgpMJ3zZuWSL5kwGF1nNoYiAkonMaz9H3pOQnOMVYCUeUTDRsiO/prrd
UhNlry4TAsBMpeXnFhdLVM3vFQZVpByadGOJNmnaN/Wavw2a00UGBFa4wsF9BBMB
CgAnAhsDBQsJCAcDBRUKCQgLBRYCAwEAAh4BAheABQJhMqGaBQkLnlUVAAoJEKwQ
ezhmktrd2sQP/3YHM+U+BbOy1nSEAfykZ71+uCM2hkHMLdxQYWB/rBWkmg/pbu+d
r4t45RsTASrNjRcZOntlPMQRIq973ymHfpmeS+noFwvTGH7zDv1BRBR9wPrd1XUz
iSuEUHGi/fqxUVXQ5mbonzfThX8tuXeuiQmeToqoB00FYlZm6xsNnEHcjVl66mC4
IPoJLWnZJs4rOCeoRf5XvDTgX6xt5/kLYRZf79qaWGFvaZpsc1CH+rQJUdVa/D4T
7pI7hX6zy0S91z4iuC5HZUiOTF+y5auEZHGTdTWNS1kvOvfcCTi0XK/GkGL82SZu
7X2VGnpCeUnFyViRGlk+KaDG1sVyDY+lcBPg6ilr45M6MQV0iHS5OF04QNXSKt5+
UnzJH7lldgNsR6ibRMyNV3k5v3fyUcSBvIYyLORTTBiVEjQDSbk1QNqbrQlX9CWz
+EJWn16BFTmMFvxBSWPm640GncHP5J3/0MbMw3Cm90x7k8UfNANIemcrJrSxIDwm
g9cVAg3a+D+wxjrVe8jGg0ejvECpm+0yswigj5x6Lqj09A4UgdjEauN+/pn0nhBo
Gv7DzMXtM/LoDtgp6wn93qZVN2TsuHnkEk4UyntB6eWJbBdXHWUr47exiWh0dvQN
tpwCWPT6I7ZTPtA5K/zx+q9m6797BLgAkTYc6gloQL3vs1Z1S3m/hZNawsF9BBMB
CgAnAhsDBQsJCAcDBRUKCQgLBRYCAwEAAh4BAheABQJgmrz2BQkLBnBxAAoJEKwQ
ezhmktrd36oP/2rB2EkwSOCKC4m0heWSfDWi6OBKoEbbDtFtc6/HwqBW8SPsiKlq
zV0e3qBY/LVju04+ktJEK+EGXLnC3iC36MegrQ8zt391kEx/Zv9LIuVOCX90QIAX
dL8MVUkkjRLCFFH8pTgRy1cJYWk1X4dYdXWYc29fCwNVartNdNBhsb2ht3VJeKDE
kUivBHmkjuISDPEnI1coY7Lj0ZtY5cHdRF2eZpB0RkTBpsIt18rCYyHkERZrhmvb
j3rOyPyvOa+1/dQS8/hv5pEmbKx8cy8RdJkmbUHYatPBsjHkJSWr7O7G9VFW4GoN
9CRAI4KkbDSEDjCL5dv2pq0Sew1MkLuWJGULAMgiIUlWcOs5SZZGFSksNQrtSFV9
Z/wGocecMGkGQNXQ06JV/Fry/TvyphBlmylEqL+NLqEcEjnlz90IVu+ZA+M09J96
UlHO7V5GvBgM+QK/q/dJeMHPWrNlo1gA6Nwl/HBdM0DqzdZ2jEPvsQSABvZrPMty
+BAqEar4wqY1AH4X5ccEjO7nJQoBQSDRSki1fkBsc1nx44N/m0kHdIa0Z/Y+Mw4v
WiZhREkOospG1I4lBa3CNTVAhSs9msGsYfkqvFJGHL7sZY8XSv82GBBvA0nUNrsJ
bLBwo2FaQG9eoatRAGkqp4b/OtNtBuGeiQoNwFGbfUZTAaStj5/zZj0swsF9BBMB
CgAnAhsDBQsJCAcDBRUKCQgLBRYCAwEAAh4BAheABQJe+9bwBQkJZ4plAAoJEKwQ
ezhmktrd+ScP/RoaUKriVVAgLHOGs+/mnfKtnfTlClzi5dsdI9/6H0vLpmSWK/Cl
2cT6gary45VMgAeVK+H1lQXafYj+FY++I5kYoe2GrSvIXhpjaFAJyNf/dKleTsqR
Tm371i8b3FDYs5kvy2CnTbmHB8MsOGxck8/YHd1x+g8WpO2IgF89yYCSF3CAdxC3
6bHbs6Z3C3lcM/3SoWF+Yie2P8XeBMPCGp/BcjQzUcHF6G06TwDDYhixucUi6vEY
EH5JtOwVVQ7bubT8OFeOoJwVxlzYz4UoqxjKDWymarTzu03AUIT0PXPece94bJAK
mSh68ItQe3H8tSPMubERWz2tEV3lVkChDGXcC7BYQmxHseolxz/qzCtJ0iX9BvZR
dniZNeNJ/Cu8M2pDp47zdNFXzf/Q/sQ9pQlws22G2g119rWDneBku9n1vTP80/er
SB+VLTBjDiArlCY5y9+BG8wbscExJySoQxkB9j/nlMzPY5rgk0SyxsNj9GbqH+hr
EjS3/uacNwSLxGcOT2E9Teot5pfTEO6fQVq+35QhfAlP8c8jze01W/+u+wXu1Ui9
```

```
azRSzYtCHanGyyet6U1mlBpAkqkZzH6t3CA5czc9i6FbzjvFVZnbRUZIRzfISYew
lF5WqgTn2iYVdxagPRvLF5kjd696brGW9d5HwirCVGaK04VsXWlAblB9wsF9BBMB
CgAnBQJXdYAFAhsDBQkHhh+ABQsJCAcDBRUKCQgLBRYCAwEAAh4BAheAAAoJEKwQ
ezhmktrdWigP/3QWl7aO81BUWyby4HEhN4SdAoWGY/FLqO4mCtuplcnMgRUCSiL9
l2BSCTMCtUcdSWtYwOgSChN2mMsdi1U2FNR5HvNunYR/pFdqjfQurf1ZmKVeG5/4
uuKaOxMw9e8pK5uYAfs+O7gr8gu/f6/Drp7NZk3/yVKpf4WCY9oX9TA1q9O/11nN
cwS45U/d7YP+N1YM9cBXa1DnDcdfm0BlykzouAF0qd1Lwi/tmLENvybD3+2c2WsE
rlFZGSa5ZafO0tTIWXh5k6wh5FdRRycrnSyRK3B9N9+yaXfMQ0XpOypa8dqQEnCi
IsngDCJPxtTrhMWKhBFRUMzK/WZTDboTQSQDK+YVRrE4K8MtoZSKwZLV2r9O3TpX
kpbKsPVYmexerfdMeZfjZMF1bC7BmEs7jciH6JjbqAoAPnHzN0481aeNarINSViX
PQWr2mp9qShei2/RavLtx2ZNrvmGW72ZKpF8E3WWUDpBJqFVeGNRvOm3aZj8o/Hl
ewtNjcT4ouJfqlfKiULv+g7ANEMDLQTFDTg5twRdvmZlB7oTBsavf+LwxPIXhH32
IR7TX7VeicMMxmZnmZK2ANT/QBi3laf+ojVHvB+f6D74eLNq0Zqjfi/3UFNYsYjg
E+YgCqEUBpHbl61nOHwGOSsQwfap2uKKlzukD/KxH5SPBC3DYGBI+KCbzsFNBFd1
gAUBEAC8zNArPWb3dPMThL2xAY+fS60vXdB1SkOtYJpDWpFgvo0d+VQ+hV6XulGA
HAS6xG1WHysPT9KejIRSgLG+e9CaM5yhsxNa1WFGUM4Q9ESo3t+a75Go7xHIxgFj
C046/O6Vh3g9N/PREeuG8zkZ3H2v5fmD+ejyPgk4W9sFL00zjRiZD0FKVYR/j9ue
nEC/2NBcLuFy3q6cDfmCoDEOO62kXMnaGz3knzEK/X1SkcjsxRDq7zaQlQ1Kou+3
dICwy4x5SJQ8jl+eeeEvF2C2/dXmDohb57tqUwioohMUQkmCtvZgEHjypUwgp0MT
o25gWxkvJlSJKUOb6b1786WNySIzF2gxqlkkEmBl4RAssQkeXjrSmGwsMDyHNqyJ
eYFusl8sPaSpo+V2n0z+2B070Uq+wmf1S5A5FpegH0PZzzoNZo8I6QxaZje9YSZU
ijGmZIdEBleRVt3Svhi8MYlnasd4bW2RK1sr7plkBf8QRe6biiQRF3KDOSn5CbmX
pAcHJ1ZHzRRdkXZDNQC6vCJxsy13O0TrhJtAV1Yq347uyUbVi291ISVgroUVtprs
mHoEk5GoOTHbg9SCSt+xi/FiJQC+ubWmIGXoFKMR3UmhDnnzobKcbnbs/Hd981Fd
VghYYvq//gTAkJk0WxfGqO30wtXRndPOA0T+qhP3TE+LtGRJ+wARAQABwsF8BBgB
CgAmAhsMFiEE/rkgny8vP0ZkhB5VrBB7OGaS2t0FAmjXZm4FCRNDGegACgkQrBB7
OGaS2t3y5g/7BFXp/fdanzuQPToJTPen7AVwhLloKaiYhG3GjdXfMPLvu6UtaaGm
qynLolUNNooobptFqc1G9BKoAghQrta7CsDHtsQF2xyc3Mfu0gmpL/7X5a7sFIeJ
j08UjfweHx4DSG4LEZgNaAoWFjZltp4+8cqijkAHXt+r+1ayQG4VVHOWyXXqmSH4
9HqtbPcPyRzxdoVLeshZC9jmhHhhKqw/LwGyipWSOUKQDjWarBwdyhNmWCaLvxH1
ndMp4tq8DPGC3G4T9tYAbANrn7nKfZgHebMSzMw9kSp0L6QvwwTDjJyIWz85WyeH
WHeBysDaBOit3XDlehUew27y7N6a9hQSYjnXuwvre5mjDIOqJon/31R6ui2Z1y9P
a+bC11hbLXXh9tLCXRuoOt6thh9Cq5X1a76PPpEv30o3bpsb6l2hbrut1OKezwvK
l7txito/jfMiWfsZHA9O4SoM+8GnmVingHtZ805n1T4RddJvT/vaqplfI6zf7jmf
a69lALP420riFOQcwntNUM5tVmFUZsnFp2YRd4Ls7MiXVjtABahlSbb94l5WSVc0
jrOLDf94edvzk4R8i2Ob8CfVZNqEsTR6bHz8dT7Q+xQzEdjUujyyZY1UUl157Qeb
OsHjhCtuZYCI04X9hZ37nKnZXSxRlRDCnt5BEiyFu2WD1RscUe6PcVDCwXwEGAEK
ACYCGwwWIQT+uSCfLy8/RmSEHlWsEHs4ZpLa3QUCaNdlPQUJE0MYuAAKCRCsEHs4
ZpLa3XCpD/42DrcveE+q2ulrAIYPDlUlHiwIMejqBDRm6zmr1KSAeb4E6/MFcP4s
rXSSscMlrqG6NVynjNCXjD2YzWii68EwoXLJkgoD3r2ifzkV62EX2MIEeNZAVwuy
KNxorzmy6bhuWltRYNK/hITs2AG5orOk9ADEJ8PixKymrWlhesPaWX6Yhp9/tWaC
RHOSRiLbRVaJ+7sqT88urLmkV9Hqx949Zxv4+cgBVUGL6WXKsfWhHjbDMNJnozWB
SZaIJznLApOM8z+1DNrqUYyfR8SkF4IOVmg6HDzoyuseJJ8JvMAlkvT6F9VBq/iE
yeDYdEEQxwHwozKrEx5Ybxl5mntbqwCXy6kHSx2+/3RZWpZQ8K29YP9QEk0KeGF8
```

```
9Vap3jjNrX4u3cuRNQpeblQc4uFn3Nzaj+cVV4YzcRw94NifecXpujSvk8XU2ytJ
/JgMBxPIBKglN4eEMet9b4FRB5XeBdPAm19/LXyb4IlIipGNXlgNz/HCuBzidzHT
QQdqfA9rZVx1hwFr7AJCVqWaXVsx1oEAhKqpTtsLMyj594DvnRuwKw5Vse+1eydW
MIHYdbxmJccsTGIt/hsOpc8zfm+QYk5752jshhOKEBy+Ey3QZI1WbO547NOb2Hwr
Pgt7fw2NCKMPElSu98zmneFPhqNHf7L5urBe5gADj81E8lm6t/oVxcLBfAQYAQoA
JgIbDBYhBP65IJ8vLz9GZIQeVawQezhmktrdBQJnABcLBQkRa8qFAAoJEKwQezhm
ktrde3MP/13CLWp99XvRROrzD/bWOfWjAenT2PE/tYdOY9YcTQFbnIUhaVUDWAo3
pibR3D4u9LlY4olpGfJ7BTIHFa9myfpaVvmrNjueYI4omli24JQ/CKqNdY8Qzxz+
/QyiNK7Aw5cEBWIu84WGB1SsefWWT3rZe9YBb77gNcWHZ15pXTXrcgUxGY4808MC
I9YFWq8EA0iHawtFnmB3UFfClWt37Hy3PKvr1is3uG60+ULI8RQz3/+ZwSG8U+xt
b+I7H9+gITc1eFCb+tIwp5xWflyxcFXYk6UzOL7y3Fg2tIEuSNtIHUC9NDVobf6c
I0KAzZcMvKiPQiuBnVOjgDLmCZM5H6axj9x+gi4oVh6ea3HLqMzyjm5JkeCGgKWv
H0gD3yGEZDvcbavkQOle5T+4JefndKzCPrluX0iyx+oQiiOL8WieSSkSB6BsZcUN
SeuGJwM79Y7Oqld/YVrQNBZj5Vz+m3nZ+0EWDDMI0hRgMpSEIc+dnTC0u103Z+Rc
c2IJq8INmU653sUcfCZE12ParW4rF7ib6kViYrABT8f4e2TP0aOyP5kp51ied9qL
azaBA6tt/C9X1V2EJZK4srXtmcZO2Im45RAiVXyfpBAmmiF3eZWCbKe7qBC4rDRh
LZG4RQW/S86Da0BID7gQz9IFSkaG504MsDhvnA7iAqaHUHUepCsiwsF8BBgBCgAm
AhsMFiEE/rkgny8vP0ZkhB5VrBB7OGaS2t0FAmUkSiQFCQ+P/Z8ACgkQrBB7OGaS
2t3AwA/9GkXKUgvjKGCxwE4SdDt7c2jw6to2TTP9iFJ3Xbk3+5BURT3gkZCuu9D7
gt+97aVo/B4EM7Xz8DQKyY7Ic9VAwDRra/Hwi1V0hw1zyIWQ/gAnX3baU6qLRWHR
vVR5meV8r35C+rg9DaWFYmvS7PIv9LfxESwBPUjbmx8k4/5EJpHUwf12bzkTnot5
7q5lHxKQa6IvqQak+Hp9ZM2KPdsgKO2HWJJIIvYcI5byW9zBKV0O7YR8gtRAJKp9
IbtsXx0WT6cqHOFVc5SSzdcaMt0gLFl7BTnJyvKK2l9GABGBmzYDjeCyF2J+Ippf
oqxqfTe6EoOsuEMc2PbLTs9SsWjyCC2VGlX8+uUH9SoKwL0VQ6LFsP6fhkVKqi/a
rB6UuPR/iZnrKIuxMNQ4U+t2Q6UdMlmXsAXTNdkwzoK9oJRokIrH0ZV1KtH4sjjA
tCic+tOddq+GQLiKe2WpJfxlA0uESCB0TxjAwQmfn1H+dUhPeLlbNimHlH0/hXPd
ifuNGozzADIRseQDyzjl8xGL1qRZLD3cfmda6RyZ+S3dQRuaRrcFCDccpY/pO+F8
jbx64zyqqNs+KV+SkQGOcKFhWTZGCfQ/zMDtDmQKjb3eTAkv1zdEOMw9zEjjmS0q
8FNl+2wO3VnvXwvBbtDdVCIaIq+jVcsy5XtnnV+bJ19Q9yue/XvCwWUEGAEKAA8C
GwwFAmEyoZoFCQueVRUACgkQrBB7OGaS2t1uHBAAhOYVvrtchRmzCvdNER1DtkIs
bgQPJ9OxbyfvmvoD06qxH7PrycLZKbt7yYpAUU/CMc86GwaEe0I5Nm1CTs6NvDIv
g3e7EPIS859tyQflbM56NlwbsopCuoCJYknuroIf/M6dW6vJKNXLMmnL/AtalUBw
X+5pblmGUUJep49oTOxQEnvnuqyvaGjXgFXix5PVFJD2ed5NnQeFpvfCpc/ioNOj
z7ORO82j1ht5nWqPraXX5AYhQFM/kwR1cK4LV7gVDd/q+dfGYHzpxQ/HtyX/Lasi
N6I52QqA95SM1ZZLPFLaNh6EvnB7uC9pLCYS8nvilX7/cez5PFff1e1gXCOT0jv3
mJ2exLmXV0BbfKgjccFCxhrdRLtukfiDfJkySy1zdscnpfng8wJ3xKRv43cUTz7M
Z24OYNMqK26aJZVXEQUYjCwsBylY/F5wjYAwgwZ8yF5RFix28P/K8JsIHb3QrAJK
sNWQAb03ZWis3N3spR5M9Mw3VuDZ3WUXq7mxB5M3kpVoZ3vETU5cwTbADYNPf4Sw
BDK2uIVtxabezxSBtz0FcyYoF+OW8q7r4WvoyC9/+3GfnozZLJcEIVDk4W2pMW4A
UhG/6drKTm3HkSDWIDu7d1sHWMffLEYfUHtN5DKkDkGoPfHvZvu9teR5yLfUrPTf
ktihPn/JMrmwa9pwi8LCwWUEGAEKAA8CGwwFAmCavPcFCQsGcHIACgkQrBB7OGaS
2t0uaA//UWRaRiHEAKeRqBG/T2ak+XZJNu7QHfNgoUEAub9Zru8oPPXx2AJLcHEN
KWmeFlLxADdWOZs4Bm9oOew3VQnR/dBqjnXfob9Rc+eYUjA3rXazM/QrqcU8Syi3
MjNGUmjdL5aQF+IppAMgOBLG1TEnM7C5/PvrGJuYpGEnkKEwMK/GYhqg2V60pHEV
```

```
Pvs66mefJpCzbZSy56qtknSt6yBNWc14XgDX6VTn2kW4CV/3vVJUuvjvYs9SPyY8
mKEXa6QvUd3PcXv6RiWk4lGYuT1+jh2VkcFQ+JnUwv9TbKFB9b5jq1bvW9+LMDEl
YXux7pBP5RPk+OLpyiExIRFWhi3x7aMWOzQ+I9yuNTeYkTHiEAQRUhs/1Fh4oLgI
v9QZgC0mRSN3zm8plQdivs1ZlAosAqqkA9BQwqsgosQe7P92irYIJqay0si9wGCD
wSMsmeXdIF6wW3/UMJZl66aarPeiZApGX0QdTZwjMh/QK/8gTKyeZulKmNkNfwWq
O17OirWqLKssVHtg3VUM8EIdh+oNqDDXSeWtYUmpPpWp+yWZ0x1MFFZhUQHQZTGu
TIj4A92LQzbrfj/jXRvWm2SrJMivUoiDUn+qxKIpVwFlI5gVb+uyTFhw89PCkphr
JwRi052RLoU9yd6Ek46UH4XfZZWrZuzY+zzB7oqGONphLgi/h3DCwWUEGAEKAA8C
GwwFAl771b8FCQlniTUACgkQrBB7OGaS2t2/MxAAjoEGPdzavhsOlXdPCRd1D5QJ
r8T/NSEV2z1cp8ZvdrkjNF09TBP4qsBnKJiuvY1Iw7OGX9W2okvXxgJizE45v9MH
WEMz4hmIjmAfRwcqENgpOc1IY/T0/+kkCW8dB6d30J1kT0n2PCRzN9L5vPqZXGTG
mLvd9MOjH1256w4uxLb+e1HMDTCqEN1ppq9G+EAR/29q8JZWs1marbZZWxSWcg/E
1YYbNafzklgjq4CLh/j8AEWSvLr39zRy9uvQ/yqAKZ4K4aZfh/SPupGDvsD6ZK54
EPHxErQ7aiXTbUHtvwhxWLOP6WmxFA3Shr6L6YUb6jq+0PVliFC517g3mxFHJtwt
yXGNIKhmzmr0l9OlsHafulJ/9QPfK3Ce32SkPhW/11MYA8HzduMv5Arp7cBczXSP
EUTmNIVKv3gTjSQrzRhwhHmMuqyDZ/rXQQ1jl2sxIDjO4MUMvVjYKF+OCNm42gVs
8ca3/wN9ZNU6hyFWeKQDuCAqPPbT5GO/DKseFEwB+07wwyH1RXbyl0v4fneg605X
S7lqhNtw2p1hDL0HYHDiV+aPZ+LoOmX6+dmnqE6bQJaIlVb922KWmliO7F3DkqP7
0jFlhoE1gfiXWkxP4Gy8wOobNfEMgvz02djkGQy+oQqeNdIcZFZgzPTGKB/nVgpt
9CcRDWjPltFCd2e1FBbCwWUEGAEKAA8FAld1gAUCGwwFCQeGH4AACgkQrBB7OGaS
2t1PIQ//Qc5VYfBCxpaMysaPQ44wXPEZSjxIGZhhMGzb1UzzAEYOw+RgKN5nNTXq
L2KoOkOrGnKqZOKByMdXwIPH/rGwwEsbbIpopnibf5ic5B/+xCTIK+qLIwX2ZLuk
NhbL6Y+E+7DxMMh+KqBWHONKkgwVY+rFWOfoops839ABKvc9/Ry4/qqkcb40AzpD
l1iQJ5vK/DMuaDWxWeKXqJLIl3WMGPcPfheuBZL1u7LEEHYKMgzvpbF81WIn3MBo
8jvxf2/o+kMafSSDqgvOu6yu8GOhmScpCbRJn7jV/HrG+tM+zy48TN6/MkGWSR7q
TD34pqBjyatVfVl6dGD6xj/i/Emt5hZB6qXruCDH7AWMoNx+FkDubs4sc4PKysZU
Itya6KdQFo2UeYsNwZhdn6QwKhd85um4JUHJCY0mARvjsQgWXH/5MR40cow77bbE
vVq0XNd+QRVlyT42CEtnIUOFLeDVuZrum5Tuvvna6ImMDoi/z6QcNeL79XsY2m6I
QVRiHr1BDb/8JLkfnWiwL8GRv169Kf8unx0y5u1YBpcMYkyDD2+pnnk3TY0rR+8X
8goecaS8fbyu/Q48K85ZMD8wKW/bzLQ+tK9y8xed24u2QERftMhIw9b6f45Nrrf/
PhgV8RnuwUusSbdDe8kw3eYTmLdzD4kZc9K7SdO2CqT+hm//9JI=
=uGHC
-----END PGP PUBLIC KEY BLOCK-----
```

以前的密钥

> **Important**
>
> 在之前的密钥过期之前会创建新密钥。因此，在任何给定时刻，可能会有多个密钥有效。从构件创建之日起，密钥就用于对其进行签名，因此当密钥的有效性重叠时，请使用最近发布的密钥。

到期日期：2025 年 10 月 4 日

密钥 ID	0xAC107B386692DADD
类型	RSA
大小	4096/4096
创建时间	2016-06-30
到期日期	2025-10-04
用户 ID	AWS SDK 和工具 <aws-dr-tools@amazon.com>
密钥指纹	FEB9 209F 2F2F 3F46 6484 1E55 AC10 7B38 6692 DADD

要将以下适用于 SDK for Java 的 OpenPGP 公有密钥复制到剪贴板，请选择右上角的“复制”图标。

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Comment: Hostname:
Version: Hockeypuck 2.2

xsFNBFd1gAUBEACqbmmFbxdJgz1lD7wrlskQA1LLuSAC4p8ny9u/D2zLR8Ynk3Yz
mzJuQ+Kfjne2t+xTDex6MPJlMYpOviSWsX2psgvdmeyUpW9ap0lrThNYkc+W5fRc
buFehfbi9LSATZGJi8RG0sCCr5FsYVz0gEk85M2+PeM24cXhQIOZtQUjswX/pdk/
KduGtZASqNAYLKROmRODzUuaokLPo24pfm9bnr1RnRtwt5ktPAA5bM9ZZaGKriej
kT2lPffbBjp8F5AZvmGLtNm2Cmg4FKBvI04SQjy2jjrQ3wBzi5Lc9HTxDuHK/rtV
u6PewUe2WPlnxlXenhMZU1UK4YoSB9E9StQ2VxQiySLHSdxR7Ma4WgYdVLn9bOie
nj3QxLuQ1ZUKF79ES6JaM4tOz1gGcQeU1+UklgjFLuKwmzWRdEIFfxMyvH6qgKnd
U+DioH5mcUwhwffAAsuIJyAdMIEUYh7IfzJJXQf+fF+XfOCl6byOJFWrIGQkAzMu
CEvaCfwtHC2Lpzo33/WRFeMAuzzd0QJ4uz4xFFvaSOSZHMLHWI9YV/+Pea3X99Ms
0Nlek/LolAJh67MynHeVBOHKrq+fluorWepQivctzN6Y1NOkx5naTPGGaKWK7G2q
TbcY5SMnkIWfLFSougj0Fvmjczq8iZRwYxWA+i+LQvsR9WEXEiQffIWRoQARAQAB
zSxBV1MgU0RLcyBhbmQgVG9vbHMgPGF3cy1kci10b29sc0BhbWF6b24uY29tPsLB
lAQTAQoAPgIbAwULCQgHAwUVCgkICwUWAgMBAAIeAQIXgBYhBP65IJ8vLz9GZIQe
VawQezhmktrdBQJnABcIBQkRa8qDAAoJEKwQezhmktrdl1MQAIwEuDar3OTxkfTa
cPNKDnNzxaWqrxZ3FTQ+PyrHhQ6usxxrvDKJS+uCjE9bmWHVFU1R4yQNF+72lJdw
5UhX0u+ZgT9afApE65uAZuwLhPsz8upXT8C6VeKXh3shdw7qXi2hrwtM1a0Pls4O
Cs2C9rLUDMJTySrVDDVwpnaAB+8DcFrs9bIt5Q3gdOUatdzDvcb7QKh9jUvzCpbE
```

```
cInb1epDN5MRzowMR4iU2VV1RzLxCvm7CQSyXfgf0DFLkXWiknhOq9eINmytJFG/
ntFdiZFkNZ5hP709loywdfNrmqB6PsF8BPGFh8gKw1pjowrfHpv6cNIqShmA76LT
3OHViOlqGFB7obffq//eZGPR0oYJFDrOdD2CFRoHnP3N++AfkA4SRN7eXwyoz6Pk
Do9WNIEEkAcp6PGvv7AokogDo/4OqmxgC6fN+3BTOstWpv4FlD4NxOZWsTs49wxg
kPlCCVf8t75aZZkcjXng1eClZZQ5SBlRtSB7gMqtP7MIn2J5w8spNbs5xQvJc76u
NvzwEasPkY+UcHd05RddOUwoKqDerLUG7YqdlNCJoQR1mBIgZButbQlMyaZcmQqO
iR0kwDi9h6Dl6fnUb2dFCNJw+eDHvsjG8HI3IVZMl0bUQ2kmmwr1O2YQ1ynJQm0l
lMllI4hFU8/lHNHm8ie5darpVXQEwsGUBBMBCgA+AhsDBQsJCAcDBRUKCQgLBRYC
AwEAAh4BAheAFiEE/rkgny8vP0ZkhB5VrBB7OGaS2t0FAmUkSiIFCQ+P/Z0ACgkQ
rBB7OGaS2t3HVg//S+/KbeOBf+nCdHsrWtp9kxvWIpAGvQhIbxx1tp/impfm+5Rm
fKPD0KX+g42fuMmOdDE4gjO4GjGd7ZY3bx+0zbDSdVebzmYCbPZ/BDP990oPKidd
w6Gl8PaIyqfuARKOESBETvAwNgwO4t2ocjs4pYZV+CuHvESYpqkuHjmHtye6ajZW
Mv24NhjVo4EfP33dPugTjXLjeuGT7qQpsYV3a66juHmPVkXwuPqxh9wTNc5TU6FG
UPSfIGMPL0xha7Rg2i5zvRaAxx4bHqGO8IAz/l/E/tJkV5xnt494HQam9UDbiFI0
Q0TSve1R6S45/UjQW6cycyduHtk72s9ipa9YM0ilTdLgKMWFjzYv4h4qeYvLw3oB
JGQew+I0I4dIrwL/TKet33EuFfwmyT9MaJBhqV6geFaQ0uVmwvzpAcvxIoSqSpkJ
B/kASqCEM/o0QZLuWc56cDsmMisD0ouVPt+c1Zk7AWLlf6j8LKYTbK0QxLRh/eeZ
jhSf3HnpaCfonbOoHmeo5d/o3EZ0AiA4GbT3xgScoIgXOT7KGgOWmtWkdyd3Zvl/
o6q6Hwpg4RsqCkwnfNm5ZIvmTAYXWc2hiICSicxrP0fek5Cc4xVgJR5RMNGyI7+m
ut1SE2WvLhMCwEyl5ecWF0tUze8VBlWkHJp0Y4k2ado39Zq/DZrTRQYEVrjCwX0E
EwEKACcCGwMFCwkIBwMFFQoJCAsFFgIDAQACHgECF4AFAmEyoZoFCQueVRUACgkQ
rBB7OGaS2t3axA//dgcz5T4Fs7LWdIQB/KRnvX64IzaGQcwt3FBhYH+sFaSaD+lu
752vi3jlGxMBKs2NFxk6e2U8xBEir3vfKYd+mZ5L6egXC9MYfvMO/UFEFH3A+t3V
dTOJK4RQcaL9+rFRVdDmZuifN9OFfy25d66JCZ5OiqgHTQViVmbrGw2cQdyNWXrq
YLgg+gktadkmzis4J6hF/le8NOBfrG3n+QthFl/v2ppYYW9pmmxzUIf6tAlR1Vr8
PhPukjuFfrPLRL3XPiK4LkdlSI5MX7Llq4RkcZN1NY1LWS8699wJOLRcr8aQYvzZ
Jm7tfZUaekJ5ScXJWJEaWT4poMbWxXINj6VwE+DqKWvjkzoxBXSIdLk4XThA1dIq
3n5SfMkfuWV2A2xHqJtEzI1XeTm/d/JRxIG8hjIs5FNMGJUSNANJuTVA2putCVf0
JbP4QlafXoEVOYwW/EFJY+brjQadwc/knf/QxszDcKb3THuTxR80A0h6ZysmtLEg
PCaD1xUCDdr4P7DGOtV7yMaDR6O8QKmb7TKzCKCPnHouqPT0DhSB2MRq437+mfSe
EGga/sPMxe0z8ugO2CnrCf3eplU3ZOy4eeQSThTKe0Hp5YlsF1cdZSvjt7GJaHR2
9A22nAJY9PojtlM+0Dkr/PH6r2brv3sEuACRNhzqCWhAve+zVnVLeb+Fk1rCwX0E
EwEKACcCGwMFCwkIBwMFFQoJCAsFFgIDAQACHgECF4AFAmCavPYFCQsGcHEACgkQ
rBB7OGaS2t3fqg//asHYSTBI4IoLibSF5ZJ8NaLo4EqgRtsO0W1zr8fCoFbxI+yI
qWrNXR7eoFj8tWO7Tj6S0kQr4QZcucLeILfox6CtDzO3f3WQTH9m/0si5U4Jf3RA
gBd0vwxVSSSNEsIUUfylOBHLVwlhaTVfh1h1dZhzb18LA1Vqu0100GGxvaG3dUl4
oMSRSK8EeaSO4hIM8ScjVyhjsuPRm1jlwd1EXZ5mkHRGRMGmwi3XysJjIeQRFmuG
a9uPes7I/K85r7X91BLz+G/mkSZsrHxzLxF0mSZtQdhq08GyMeQlJavs7sb1UVbg
ag30JEAjgqRsNIQOMIvl2/amrRJ7DUyQu5YkZQsAyCIhSVZw6zlJlkYVKSw1Cu1I
VX1n/Aahx5wwaQZA1dDTolX8WvL9O/KmEGWbKUSov40uoRwSOeXP3QhW75kD4zT0
n3pSUc7tXka8GAz5Ar+r90l4wc9as2WjWADo3CX8cF0zQOrN1naMQ++xBIAG9ms8
y3L4ECoRqvjCpjUAfhflxwSM7uclCgFBINFKSLV+QGxzWfHjg3+bSQd0hrRn9j4z
Di9aJmFESQ6iykbUjiUFrcI1NUCFKz2awaxh+Sq8UkYcvuxljxdK/zYYEG8DSdQ2
uwlssHCjYVpAb16hq1EAaSqnhv86020G4Z6JCg3AUZt9RlMBpK2Pn/NmPSzCwX0E
```

```
EwEKACcCGwMFCwkIBwMFFQoJCAsFFgIDAQACHgECF4AFAl771vAFCQlnimUACgkQ
rBB7OGaS2t35Jw/9GhpQquJVUCAsc4az7+ad8q2d9OUKXOLl2x0j3/ofS8umZJYr
8KXZxPqBqvLjlUyAB5Ur4fWVBdp9iP4Vj74jmRih7YatK8heGmNoUAnI1/90qV5O
ypFObfvWLxvcUNizmS/LYKdNuYcHwyw4bFyTz9gd3XH6Dxak7YiAXz3JgJIXcIB3
ELfpsduzpncLeVwz/dKhYX5iJ7Y/xd4Ew8Ian8FyNDNRwcXobTpPAMNiGLG5xSLq
8RgQfkm07BVVDtu5tPw4V46gnBXGXNjPhSirGMoNbKZqtPO7TcBQhPQ9c95x73hs
kAqZKHrwi1B7cfy1I8y5sRFbPa0RXeVWQKEMZdwLsFhCbEex6iXHP+rMK0nSJf0G
9lF2eJk140n8K7wzakOnjvN00VfN/9D+xD2lCXCzbYbaDXX2tYOd4GS72fW9M/zT
96tIH5UtMGMOICuUJjnL34EbzBuxwTEnJKhDGQH2P+eUzM9jmuCTRLLGw2P0Zuof
6GsSNLf+5pw3BIvEZw5PYT1N6i3ml9MQ7p9BWr7flCF8CU/xzyPN7TVb/677Be7V
SL1rNFLNi0IdqcbLJ63pTWaUGkCSqRnMfq3cIDlzNz2LoVvOO8VVmdtFRkhHN8hJ
h7CUXlaqBOfaJhV3FqA9G8sXmSN3r3pusZb13kfCKsJUZorThWxdaUBuUH3CwX0E
EwEKACcFAld1gAUCGwMFCQeGH4AFCwkIBwMFFQoJCAsFFgIDAQACHgECF4AACgkQ
rBB7OGaS2t1aKA//dBaXto7zUFRbJvLgcSE3hJ0ChYZj8Uuo7iYK26mVycyBFQJK
Iv2XYFIJMwK1Rx1Ja1jA6BIKE3aYyx2LVTYU1Hke826dhH+kV2qN9C6t/VmYpV4b
n/i64po7EzD17ykrm5gB+z47uCvyC79/r8Ouns1mTf/JUql/hYJj2hf1MDWr07/X
Wc1zBLjlT93tg/43Vgz1wFdrUOcNx1+bQGXKTOi4AXSp3UvCL+2YsQ2/JsPf7ZzZ
awSuUVkZJrllp87S1MhZeHmTrCHkV1FHJyudLJErcH0337Jpd8xDRek7Klrx2pAS
cKIiyeAMIk/G1OuExYqEEVFQzMr9ZlMNuhNBJAMr5hVGsTgrwy2hlIrBktXav07d
OleSlsqw9ViZ7F6t90x5l+NkwXVsLsGYSzuNyIfomNuoCgA+cfM3TjzVp41qsg1J
WJc9Bavaan2pKF6Lb9Fq8u3HZk2u+YZbvZkqkXwTdZZQOkEmoVV4Y1G86bdpmPyj
8eV7C02NxPii4l+qV8qJQu/6DsA0QwMtBMUNODm3BF2+ZmUHuhMGxq9/4vDE8heE
ffYhHtNftV6JwwzGZmeZkrYA1P9AGLeVp/6iNUe8H5/oPvh4s2rRmqN+L/dQU1ix
iOAT5iAKoRQGkduXrWc4fAY5KxDB9qna4oqXO6QP8rEflI8ELcNgYEj4oJvOwU0E
V3WABQEQALzM0Cs9Zvd08xOEvbEBj59LrS9d0HVKQ61gmkNakWC+jR35VD6FXpe6
UYAcBLrEbVYfKw9P0p6MhFKAsb570JoznKGzE1rVYUZQzhD0RKje35rvkajvEcjG
AWMLTjr87pWHeD0389ER64bzORncfa/l+YP56PI+CThb2wUvTTONGJkPQUpVhH+P
256cQL/Y0Fwu4XLerpwN+YKgMQ47raRcydobPeSfMQr9fVKRyOzFEOrvNpCVDUqi
77d0gLDLjHlIlDyOX5554S8XYLb91eYOiFvnu2pTCKiiExRCSYK29mAQePKlTCCn
QxOjbmBbGS8mVIkpQ5vpvXvzpY3JIjMXaDGqWSQSYGXhECyxCR5eOtKYbCwwPIc2
rIl5gW6yXyw9pKmj5XafTP7YHTvRSr7CZ/VLkDkWl6AfQ9nPOg1mjwjpDFpmN71h
JlSKMaZkh0QGV5FW3dK+GLwxiWdqx3htbZErWyvumWQF/xBF7puKJBEXcoM5KfkJ
uZekBwcnVkfNFF2RdkM1ALq8InGzLXc7ROuEm0BXVirfju7JRtWLb3UhJWCuhRW2
muyYegSTkag5MduD1IJK37GL8WIlAL65taYgZegUoxHdSaEOefOhspxuduz8d33z
UV1WCFhi+r/+BMCQmTRbF8ao7fTC1dGd084DRP6qE/dMT4u0ZEn7ABEBAAHCwXwE
GAEKACYCGwwWIQT+uSCfLy8/RmSEHlWsEHs4ZpLa3QUCZwAXCwUJEWvKhQAKCRCs
EHs4ZpLa3XtzD/9dwi1qffV70UTq8w/21jn1owHp09jxP7WHTmPWHE0BW5yFIWlV
A1gKN6Ym0dw+LvS5WOKJaRnyewUyBxWvZsn6Wlb5qzY7nmCOKJpYtuCUPwiqjXWP
EM8c/v0MojSuwMOXBAViLvOFhgdUrHn1lk962XvWAW++4DXFh2deaV0163IFMRmO
PNPDAiPWBVqvBANIh2sLRZ5gd1BXwpVrd+x8tzyr69YrN7hutPlCyPEUM9//mcEh
vFPsbW/iOx/foCE3NXhQm/rSMKecVn5csXBV2JOlMzi+8txYNrSBLkjbSB1AvTQ1
aG3+nCNCgM2XDLyoj0IrgZ1To4Ay5gmTOR+msY/cfoIuKFYenmtxy6jM8o5uSZHg
hoClrx9IA98hhGQ73G2r5EDpXuU/uCXn53Sswj65bl9IssfqEIoji/FonkkpEgeg
bGXFDUnrhicDO/WOzqpXf2Fa0DQWY+Vc/pt52ftBFgwzCNIUYDKUhCHPnZ0wtLtd
```

```
N2fkXHNiCavCDZlOud7FHHwmRNdj2q1uKxe4m+pFYmKwAU/H+Htkz9Gjsj+ZKedY
nnfai2s2gQOrbfwvV9VdhCWSuLK17ZnGTtiJuOUQIlV8n6QQJpohd3mVgmynu6gQ
uKw0YS2RuEUFv0vOg2tASA+4EM/SBUpGhudODLA4b5wO4gKmh1B1HqQrIsLBfAQY
AQoAJgIbDBYhBP65IJ8vLz9GZIQeVawQezhmktrdBQJlJEokBQkPj/2fAAoJEKwQ
ezhmktrdwMAP/RpFylIL4yhgscBOEnQ7e3No8OraNk0z/YhSd125N/uQVEU94JGQ
rrvQ+4Lfve2laPweBDO18/A0CsmOyHPVQMA0a2vx8ItVdIcNc8iFkP4AJ1922lOq
i0Vh0b1UeZnlfK9+Qvq4PQ2lhWJr0uzyL/S38REsAT1I25sfJOP+RCaR1MH9dm85
E56Lee6uZR8SkGuiL6kGpPh6fWTNij3bICjth1iSSCL2HCOW8lvcwSldDu2EfILU
QCSqfSG7bF8dFk+nKhzhVXOUks3XGjLdICxZewU5ycryitpfRgARgZs2A43gshdi
fiKaX6Ksan03uhKDrLhDHNj2y07PUrFo8ggtlRpV/PrlB/UqCsC9FUOixbD+n4ZF
Sqov2qwelLj0f4mZ6yiLsTDUOFPrdkOlHTJZl7AF0zXZMM6CvaCUaJCKx9GVdSrR
+LI4wLQonPrTnXavhkC4intlqSX8ZQNLhEggdE8YwMEJn59R/nVIT3i5WzYph5R9
P4Vz3Yn7jRqM8wAyEbHkA8s45fMRi9akWSw93H5nWukcmfkt3UEbmka3BQg3HKWP
6TvhfI28euM8qqjbPilfkpEBjnChYVk2Rgn0P8zA7Q5kCo293kwJL9c3RDjMPcxI
45ktKvBTZftsDt1Z718LwW7Q3VQiGiKvo1XLMuV7Z51fmydfUPcrnv17wsFlBBgB
CgAPAhsMBQJhMqGaBQkLnlUVAAoJEKwQezhmktrdbhwQAITmFb67XIUZswr3TREd
Q7ZCLG4EDyfTsW8n75r6A9OqsR+z68nC2Sm7e8mKQFFPwjHPOhsGhHtCOTZtQk7O
jbwyL4N3uxDyEvOfbckH5WzOejZcG7KKQrqAiWJJ7q6CH/zOnVurySjVyzJpy/wL
WpVAcF/uaW5ZhlFCXqePaEzsUBJ757qsr2ho14BV4seT1RSQ9nneTZ0Hhab3wqXP
4qDTo8+zkTvNo9YbeZ1qj62l1+QGIUBTP5MEdXCuC1e4FQ3f6vnXxmB86cUPx7cl
/y2rIjeiOdkKgPeUjNWWSzxS2jYehL5we7gvaSwmEvJ74pV+/3Hs+TxX39XtYFwj
k9I795idnsS5l1dAW3yoI3HBQsYa3US7bpH4g3yZMkstc3bHJ6X54PMCd8Skb+N3
FE8+zGduDmDTKitumiWVVxEFGIwsLAcpWPxecI2AMIMGfMheURYsdvD/yvCbCB29
0KwCSrDVkAG9N2VorNzd7KUeTPTMN1bg2d1lF6u5sQeTN5KVaGd7xE1OXME2wA2D
T3+EsAQytriFbcWm3s8Ugbc9BXMmKBfjlvKu6+Fr6Mgvf/txn56M2SyXBCFQ5OFt
qTFuAFIRv+nayk5tx5Eg1iA7u3dbB1jH3yxGH1B7TeQypA5BqD3x72b7vbXkeci3
1Kz035LYoT5/yTK5sGvacIvCwsFlBBgBCgAPAhsMBQJgmrz3BQkLBnByAAoJEKwQ
ezhmktrdLmgP/1FkWkYhxACnkagRv09mpPl2STbu0B3zYKFBALm/Wa7vKDz18dgC
S3BxDSlpnhZS8QA3VjmbOAZvaDnsN1UJ0f3Qao5136G/UXPnmFIwN612szP0K6nF
PEsotzIzRlJo3S+WkBfiKaQDIDgSxtUxJzOwufz76xibmKRhJ5ChMDCvxmIaoNle
tKRxFT77OupnnyaQs22UsueqrZJ0resgTVnNeF4A1+lU59pFuAlf971SVLr472LP
Uj8mPJihF2ukL1Hdz3F7+kYlpOJRmLk9fo4dlZHBUPiZ1ML/U2yhQfW+Y6tW71vf
izAxJWF7se6QT+UT5Pji6cohMSERVoYt8e2jFjs0PiPcrjU3mJEx4hAEEVIbP9RY
eKC4CL/UGYAtJkUjd85vKZUHYr7NWZQKLAKqpAPQUMKrIKLEHuz/doq2CCamstLI
vcBgg8EjLJnl3SBesFt/1DCWZeummqz3omQKRl9EHU2cIzIf0Cv/IEysnmbpSpjZ
DX8Fqjtezoq1qiyrLFR7YN1VDPBCHYfqDagw10nlrWFJqT6VqfslmdMdTBRWYVEB
0GUxrkyI+APdi0M2634/410b1ptkqyTIr1KIg1J/qsSiKVcBZSOYFW/rskxYcPPT
wpKYaycEYtOdkS6FPcnehJOOlB+F32WVq2bs2Ps8we6KhjjaYS4Iv4dwwsFlBBgB
CgAPAhsMBQJe+9W/BQkJZ4k1AAoJEKwQezhmktrdvzMQAI6BBj3c2r4bDpV3TwkX
dQ+UCa/E/zUhFds9XKfGb3a5IzRdPUwT+KrAZyiYrr2NSMOzhl/VtqJL18YCYsxO
Ob/TB1hDM+IZiI5gH0cHKhDYKTnNSGP09P/pJAlvHQend9CdZE9J9jwkczfS+bz6
mVxkxpi73fTDox9duesOLsS2/ntRzA0wqhDdaaavRvhAEf9vavCWVrNZmq22WVsU
lnIPxNWGGzWn85JYI6uAi4f4/ABFkry69/c0cvbr0P8qgCmeCuGmX4f0j7qRg77A
+mSueBDx8RK0O2ol021B7b8IcVizj+lpsRQN0oa+i+mFG+o6vtD1ZYhQude4N5sR
```

```
RybcLclxjSCoZs5q9JfTpbB2n7pSf/UD3ytwnt9kpD4Vv9dTGAPB83bjL+QK6e3A
XM10jxFE5jSFSr94E40kK80YcIR5jLqsg2f610ENY5drMSA4zuDFDL1Y2ChfjgjZ
uNoFbPHGt/8DfWTVOochVnikA7ggKjz20+RjvwyrHhRMAftO8MMh9UV28pdL+H53
oOtOV0u5aoTbcNqdYQy9B2Bw4lfmj2fi6Dpl+vnZp6hOm0CWiJVW/dtilppYjuxd
w5Kj+9IxZYaBNYH4l1pMT+BsvMDqGzXxDIL89NnY5BkMvqEKnjXSHGRWYMz0xigf
51YKbfQnEQ1oz5bRQndntRQWwsFlBBgBCgAPBQJXdYAFAhsMBQkHhh+AAAoJEKwQ
ezhmktrdTyEP/0HOVWHwQsaWjMrGj0OOMFzxGUo8SBmYYTBs29VM8wBGDsPkYCje
ZzU16i9iqDpDqxpyqmTigcjHV8CDx/6xsMBLG2yKaKZ4m3+YnOQf/sQkyCvqiyMF
9mS7pDYWy+mPhPuw8TDIfiqgVhzjSpIMFWPqxVjn6KKbPN/QASr3Pf0cuP6qpHG+
NAM6Q5dYkCebyvwzLmg1sVnil6iSyJd1jBj3D34XrgWS9buyxBB2CjIM76WxfNVi
J9zAaPI78X9v6PpDGn0kg6oLzrusrvBjoZknKQm0SZ+41fx6xvrTPs8uPEzevzJB
lkke6kw9+KagY8mrVX1ZenRg+sY/4vxJreYWQeql67ggx+wFjKDcfhZA7m7OLHOD
ysrGVCLcmuinUBaNlHmLDcGYXZ+kMCoXfObpuCVByQmNJgEb47EIFlx/+TEeNHKM
O+22xL1atFzXfkEVZck+NghLZyFDhS3g1bma7puU7r752uiJjA6Iv8+kHDXi+/V7
GNpuiEFUYh69QQ2//CS5H51osC/Bkb9evSn/Lp8dMubtWAaXDGJMgw9vqZ55N02N
K0fvF/IKHnGkvH28rv0OPCvOWTA/MClv28y0PrSvcvMXnduLtkBEX7TISMPW+n+O
Ta63/z4YFfEZ7sFLrEm3Q3vJMN3mE5i3cw+JGXPSu0nTtgqk/oZv//SS
=bboB
-----END PGP PUBLIC KEY BLOCK-----
```

到期日期：2024 年 10 月 8 日

密钥 ID	0xAC107B386692DADD
类型	RSA
大小	4096/4096
创建时间	2016-06-30
到期日期	2024-10-08
用户 ID	AWS SDK 和工具 <aws-dr-tools@amazon.com>
密钥指纹	FEB9 209F 2F2F 3F46 6484 1E55 AC10 7B38 6692 DADD

要将以下适用于 SDK for Java 的 OpenPGP 公有密钥复制到剪贴板，请选择右上角的“复制”图标。

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
xsFNBFd1gAUBEACqbmmFbxdJgz1lD7wrlskQA1LLuSAC4p8ny9u/D2zLR8Ynk3Yz
mzJuQ+Kfjne2t+xTDex6MPJlMYpOviSWsX2psgvdmeyUpW9ap0lrThNYkc+W5fRc
buFehfbi9LSATZGJi8RG0sCCr5FsYVz0gEk85M2+PeM24cXhQIOZtQUjswX/pdk/
KduGtZASqNAYLKROmRODzUuaokLPo24pfm9bnr1RnRtwt5ktPAA5bM9ZZaGKriej
kT2lPffbBjp8F5AZvmGLtNm2Cmg4FKBvI04SQjy2jjrQ3wBzi5Lc9HTxDuHK/rtV
u6PewUe2WPlnxlXenhMZU1UK4YoSB9E9StQ2VxQiySLHSdxR7Ma4WgYdVLn9bOie
nj3QxLuQ1ZUKF79ES6JaM4tOz1gGcQeU1+UklgjFLuKwmzWRdEIFfxMyvH6qgKnd
U+DioH5mcUwhwffAAsuIJyAdMIEUYh7IfzJJXQf+fF+XfOCl6byOJFWrIGQkAzMu
CEvaCfwtHC2Lpzo33/WRFeMAuzzd0QJ4uz4xFFvaSOSZHMLHWI9YV/+Pea3X99Ms
0Nlek/LolAJh67MynHeVBOHKrq+fluorWepQivctzN6Y1NOkx5naTPGGaKWK7G2q
TbcY5SMnkIWfLFSougj0Fvmjczq8iZRwYxWA+i+LQvsR9WEXEiQffIWRoQARAQAB
zsFNBFd1gAUBEAC8zNArPWb3dPMThL2xAY+fS60vXdB1SkOtYJpDWpFgvo0d+VQ+
hV6XulGAHAS6xG1WHysPT9KejIRSgLG+e9CaM5yhsxNa1WFGUM4Q9ESo3t+a75Go
7xHIxgFjC046/O6Vh3g9N/PREeuG8zkZ3H2v5fmD+ejyPgk4W9sFL00zjRiZD0FK
VYR/j9uenEC/2NBcLuFy3q6cDfmCoDEOO62kXMnaGz3knzEK/X1SkcjsxRDq7zaQ
lQ1Kou+3dICwy4x5SJQ8jl+eeeEvF2C2/dXmDohb57tqUwioohMUQkmCtvZgEHjy
pUwgp0MTo25gWxkvJlSJKUOb6b1786WNySIzF2gxqlkkEmBl4RAssQkeXjrSmGws
MDyHNqyJeYFusl8sPaSpo+V2n0z+2B070Uq+wmf1S5A5FpegH0PZzzoNZo8I6Qxa
Zje9YSZUijGmZIdEBleRVt3Svhi8MYlnasd4bW2RK1sr7plkBf8QRe6biiQRF3KD
OSn5CbmXpAcHJ1ZHzRRdkXZDNQC6vCJxsy13O0TrhJtAV1Yq347uyUbVi291ISVg
roUVtprsmHoEk5GoOTHbg9SCSt+xi/FiJQC+ubWmIGXoFKMR3UmhDnnzobKcbnbs
/Hd981FdVghYYvq//gTAkJk0WxfGqO30wtXRndPOA0T+qhP3TE+LtGRJ+wARAQAB
wsFlBBgBCgAPBQJXdYAFAhsMBQkHhh+AAAoJEKwQezhmktrdTyEP/0HOVWHwQsaW
jMrGj0OOMFzxGUo8SBmYYTBs29VM8wBGDsPkYCjeZzU16i9iqDpDqxpyqmTigcjH
V8CDx/6xsMBLG2yKaKZ4m3+YnOQf/sQkyCvqiyMF9mS7pDYWy+mPhPuw8TDIfiqg
VhzjSpIMFWPqxVjn6KKbPN/QASr3Pf0cuP6qpHG+NAM6Q5dYkCebyvwzLmg1sVni
l6iSyJd1jBj3D34XrgWS9buyxBB2CjIM76WxfNViJ9zAaPI78X9v6PpDGn0kg6oL
zrusrvBjoZknKQm0SZ+41fx6xvrTPs8uPEzevzJBlkke6kw9+KagY8mrVX1ZenRg
+sY/4vxJreYWQeql67ggx+wFjKDcfhZA7m7OLHODysrGVCLcmuinUBaNlHmLDcGY
XZ+kMCoXfObpuCVByQmNJgEb47EIFlx/+TEeNHKMO+22xL1atFzXfkEVZck+NghL
ZyFDhS3g1bma7puU7r752uiJjA6Iv8+kHDXi+/V7GNpuiEFUYh69QQ2//CS5H51o
sC/Bkb9evSn/Lp8dMubtWAaXDGJMgw9vqZ55N02NK0fvF/IKHnGkvH28rv0OPCvO
WTA/MClv28y0PrSvcvMXnduLtkBEX7TISMPW+n+OTa63/z4YFfEZ7sFLrEm3Q3vJ
MN3mE5i3cw+JGXPSu0nTtgqk/oZv//SS
=Z9u3
-----END PGP PUBLIC KEY BLOCK-----
```

文档历史记录

本页列出了《适用于 Java 的 AWS SDK 开发人员指南》在其发展历程中的重要更改。

本指南发布于：2025 年 10 月 1 日。

2025 年 10 月 1 日

添加将于 2026 年 9 月 27 日到期的新 PGP 密钥。

2024 年 10 月 5 日

更新当前 OpenPGP 密钥信息。

2024 年 9 月 4 日

为 DynamoDB 添加基于 AWS 账户的端点的相关信息。请参阅the section called “使用基于 AWS 账户的端点”。

2024 年 5 月 21 日

移除有关使用 java 命令行系统属性设置 `networkaddress.cache.ttl` 安全属性的说明。请参阅如何设置 JVM TTL。

2024 年 1 月 12 日

添加宣布终止对 适用于 Java 的 AWS SDK v1.x 的支持的横幅。

2023 年 12 月 6 日

- 提供当前 OpenPGP 密钥。

2023 年 3 月 14 日

- 更新了指南，使其符合 IAM 最佳实践。有关更多信息，请参阅 IAM 安全最佳实践。

2022 年 7 月 28 日

- 添加了 EC2-Classic 将于 2022 年 8 月 15 日停用的提醒。

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 Amazon S3 加密客户端的加密示例，包括新主题：使用 Amazon S3 客户端加密、Amazon S3 客户端加密配合 AWS KMS 托管密钥和 Amazon S3 客户端加密配合客户端主密钥。

2017 年 8 月 14 日

- 对使用适用于 Java 的 AWS SDK 的 Amazon S3 示例部分进行了多处更新，包括新主题：管理对桶和对象的 Amazon S3 访问权限和将 Amazon S3 桶配置为网站。

2017 年 4 月 4 日

- 新增主题为适用于 Java 的 AWS SDK 启用指标，描述如何为适用于 Java 的 AWS SDK 生成应用程序和 SDK 性能指标。

2017 年 4 月 3 日

- 使用适用于 Java 的 AWS SDK 的 CloudWatch 示例部分添加了新的 CloudWatch 示例：从 CloudWatch 获取指标、发布自定义指标数据、使用 CloudWatch 警报、在 CloudWatch 中使用警报操作和将事件发送到 CloudWatch

2017 年 3 月 27 日

- 在使用适用于 Java 的 AWS SDK 的 Amazon EC2 示例部分添加了更多 Amazon EC2 示例：管理 Amazon EC2 实例、在 Amazon EC2 中使用弹性 IP 地址、使用区域和可用区、使用 Amazon EC2 密钥对以及在 Amazon EC2 中使用安全组。

2017 年 3 月 21 日

- 使用适用于 Java 的 AWS SDK 的 IAM 示例部分添加了一组新的 IAM 示例：管理 IAM 访问密钥、管理 IAM 用户、使用 IAM 账户别名、使用 IAM policy 和使用 IAM 服务器证书

2017 年 3 月 13 日

- Amazon SQS 部分添加了三个新主题：为 Amazon SQS 消息队列启用长轮询、在 Amazon SQS 中设置可见性超时和在 Amazon SQS 中使用死信队列。

2017 年 1 月 26 日

- 增加了新的 Amazon S3 主题（使用 TransferManager 执行 Amazon S3 操作）以及使用适用于 Java 的 AWS SDK 部分的新主题使用适用于 Java 的 AWS SDK 进行 AWS 开发的最佳实践。

2017 年 1 月 16 日

- 增加了一个新的 Amazon S3 主题（使用桶策略管理对 Amazon S3 桶的访问）和两个新的 Amazon SQS 主题（使用 Amazon SQS 消息队列和发送、接收和删除 Amazon SQS 消息。

2016 年 12 月 16 日

- 增加了面向 DynamoDB 的新示例主题：处理 DynamoDB 中的表和处理 DynamoDB 中的项目。

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 SDK 中，因为这些主题实际上是有关使用 SDK 的中心主题。

2016 年 8 月 25 日

- 已将一个新的创建服务客户端主题添加到使用适用于 Java 的 AWS SDK 中，该主题说明如何使用客户端生成器更轻松地生成 AWS 服务客户端。

  适用于 Java 的 AWS SDK 代码示例部分已用新的 S3 示例进行了更新，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 5 月 2 日

- 已将一个新的异步编程主题添加到使用适用于 Java 的 AWS SDK 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

- Amazon SWF 部分添加了一个新的 Lambda 任务主题，该主题说明如何实施 Amazon SWF 工作流，此工作流将 Lambda 函数作为任务进行调用，以作为使用传统 Amazon SWF 活动的替代方法。

2016 年 3 月 4 日

- 已使用新内容更新使用适用于 Java 的 AWS SDK 的 Amazon SWF 示例部分：
  - Amazon SWF 基础知识 – 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 – 一个新的教程，为初次使用 Amazon SWF 的 Java 开发人员提供分步指南。
  - 适当地关闭活动工作线程和工作流工作线程 – 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

- 《适用于 Java 的 AWS SDK 开发人员指南》的来源已移至 aws-java-developer-guide。

2015 年 12 月 28 日

- the section called “为 DNS 名称查找设置 JVM TTL” 已从高级部分移入使用适用于 Java 的 AWS SDK 部分，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。

2015 年 8 月 4 日

- SSL 证书要求 是入门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 5 月 14 日

- 已对简介和入门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 凭证和区域的指南。

  对代码示例的讨论已移至其在其他文档和资源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的信息已移入简介部分。

2014 年 5 月 9 日

- 简化了适用于 Java 的 AWS SDK文档的总体结构，并更新了入门和其他文档和资源主题。

  添加了新主题：

  - 使用 AWS 凭证 – 讨论可用于指定要与适用于 Java 的 AWS SDK 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 – 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《适用于 Java 的 AWS SDK 开发人员指南》所做的更改。它旨在与发布说明历史记录一起提供。