

SDK version 1.x のデベロッパーガイド

AWS SDK for Java 1.x



AWS SDK for Java 1.x: SDK version 1.x のデベロッパーガイド

Table of Contents

.....	viii
AWS SDK for Java 1.x	1
SDK のバージョン 2 をリリースしました	1
その他のドキュメントとリソース	1
Eclipse IDE サポート	2
Android 向けアプリケーションの開発	2
SDK の改訂履歴の表示	2
旧バージョンの SDK 用の Java リファレンスドキュメントのビルド	2
使用開始	4
基本セットアップ	4
概要	4
AWS アクセスポータルにサインインします。	5
共有設定ファイルを設定します。	5
Java 開発環境をインストールします。	7
を取得する方法 AWS SDK for Java	8
前提条件	8
構築ツールの使用	8
構築済みの jar をダウンロードする	8
ソースからビルドする	9
構築ツールの使用	10
Apache Maven で SDK を使用する	10
Gradle とともに SDK を使用する	13
一時的な認証情報、リージョン	17
一時的な認証情報の設定	17
IMDS 認証情報の更新	18
AWS リージョン の設定	19
の使用 AWS SDK for Java	21
を使用した AWS 開発のベストプラクティス AWS SDK for Java	21
S3	21
サービスクライアントの作成	22
クライアントビルダーの取得	22
非同期クライアントの作成	24
の使用 DefaultClient	24
クライアントのライフサイクル	25

一時的な認証情報の提供	25
デフォルトの認証情報プロバイダチェーンの使用	25
認証情報プロバイダーまたはプロバイダチェーンの指定	29
一時的な認証情報を明示的に指定する	30
詳細情報	30
AWS リージョン 選択	30
リージョンでのサービス可用性の確認	30
リージョンを選択する	31
特定のエンドポイントの選択	32
環境に基づくリージョンの自動的な決定	32
例外処理	34
非チェック例外を使用する理由	34
AmazonServiceException (およびサブクラス)	34
AmazonClientException	35
非同期プログラミング	35
Java Future	35
非同期コールバック	37
ベストプラクティス	39
AWS SDK for Java 通話のログ記録	39
Log4J JAR のダウンロード	40
クラスパスの設定	40
サービス固有のエラーと警告	41
リクエストおよびレスポンスの概要のログ記録	41
詳細なワイヤログ記録	42
レイテンシーメトリクスのログ記録	43
クライアント設定	44
プロキシ設定	44
HTTP トランスポートの設定	44
TCP ソケットバッファのサイズに関するヒント	46
アクセスコントロールポリシー	46
Amazon S3 例	47
Amazon SQS 例	47
Amazon SNS の例	48
DNS 名参照用の JVM TTL を設定する	48
JVM TTL を設定する方法	49
のメトリクスの有効化 AWS SDK for Java	49

Java SDK メトリクス生成を有効にする方法	50
利用可能なメトリクスのタイプ	51
詳細情報	54
コードの例	56
AWS SDK for Java 2.x	56
Amazon CloudWatch の例	56
CloudWatch からのメトリクスの取得	57
カスタムメトリクスデータを発行する	59
CloudWatch アラームの使用	60
CloudWatch でのアラームアクションの使用	63
CloudWatch にイベントを送信する	65
Amazon DynamoDB の例	68
DynamoDB でのテーブルの操作	68
DynamoDB での項目の操作	75
Amazon EC2 の例	82
チュートリアル: EC2 インスタンスの開始	83
Amazon EC2 での IAM ロールを使用した AWS リソースへのアクセスの許可	88
チュートリアル: Amazon EC2 スポットインスタンス	94
チュートリアル: Amazon EC2 スポットリクエストの高度な管理	106
Amazon EC2 インスタンスの管理	123
での Elastic IP アドレスの使用 Amazon EC2	128
リージョンとアベイラビリティゾーンを使用する	132
Amazon EC2 キーペアでの作業	135
Amazon EC2 でセキュリティグループを操作する	137
AWS Identity and Access Management (IAM) の例	140
IAM アクセスキーの管理	141
IAM ユーザーの管理	146
IAM アカウントエイリアスの使用	149
IAM ポリシーの使用	151
IAM サーバー証明書の使用	156
Amazon Lambda の例	160
サービスオペレーション	160
Amazon Pinpoint の例	164
Amazon Pinpoint のアプリの作成および削除	164
Amazon Pinpoint でのエンドポイントの作成	166
Amazon Pinpoint でのセグメントの作成	168

Amazon Pinpoint でのキャンペーンの作成	170
Amazon Pinpoint のチャンネルの更新	171
Amazon S3 の例	173
Amazon S3 バケットの作成、一覧表示、削除	173
Amazon S3 オブジェクトに対する操作の実行	178
バケットおよびオブジェクトの Amazon S3 アクセス許可の管理	184
バケットポリシーを使用した Amazon S3 バケットへのアクセス管理	188
Amazon S3 操作の TransferManager の使用	191
ウェブサイトとしての Amazon S3 バケットの設定	204
Amazon S3 クライアント側の暗号化を使用する	207
Amazon SQS の例	213
Amazon SQS メッセージキューの使用	214
Amazon SQS メッセージの送信、受信、削除	217
Amazon SQS メッセージキューのロングポーリングの有効化	219
Amazon SQS で可視性タイムアウトを設定する	222
Amazon SQS でのデッドレターキューの使用	224
Amazon SWF の例	226
SWF の基本	227
シンプルな Amazon SWF アプリケーションの構築	229
Lambda タスク	248
アクティビティおよびワークフローワーカーの適切なシャットダウン	253
ドメインの登録	256
ドメインの一覧表示	257
SDK に含まれるコードサンプル	257
サンプルの入手方法	257
コマンドラインを使用したサンプルのビルドと実行	258
Eclipse IDE を使用したサンプルのビルドと実行	259
セキュリティ	261
データ保護	261
最小 TLS バージョンの適用	262
TLS のバージョンを確認する方法	263
最小 TLS バージョンの適用	263
Identity and Access Management	263
対象者	264
アイデンティティを使用した認証	264
ポリシーを使用したアクセスの管理	268

IAM の AWS のサービス 仕組み	271
AWS ID とアクセスのトラブルシューティング	271
コンプライアンス検証	273
耐障害性	274
インフラストラクチャセキュリティ	275
S3 暗号化クライアント移行	276
前提条件	276
移行の概要	276
新しいフォーマットを読み取るために既存のクライアントを更新する	276
暗号化および復号クライアントを V2 に移行する	278
その他の例	280
OpenPGP 鍵	282
現在のキー	282
ドキュメント履歴	284

end-of-support の今後の AWS SDK for Java (v1) [を公表しました](#)。 [AWS SDK for Java v2](#) に移行することをお勧めします。日付、その他の詳細、移行方法については、リンク先の発表内容を参照してください。

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。

デベロッパーガイド - AWS SDK for Java 1.x

[AWS SDK for Java](#) は AWS のサービス用の Java API を提供します。この SDK を使用すると、Amazon S3、Amazon EC2、DynamoDB など動作する Java アプリケーションを簡単に構築できます。AWS SDK for Java には、新しいサービスのサポートが定期的に追加されています。サポートされるサービスと、SDK の各リリースに含まれる API バージョンのリストについては、使用しているバージョンの[リリースノート](#)を参照してください。

SDK のバージョン 2 をリリースしました

<https://github.com/aws/aws-sdk-java-v2/> で新しい AWS SDK for Java 2.x をご覧ください。HTTP 実装をプラグインする方法など、待望の機能が含まれています。開始するには、「[AWS SDK for Java 2.x のデベロッパーガイド](#)」を参照してください。

その他のドキュメントとリソース

このガイドに加えて、以下の AWS SDK for Java 開発者のための貴重なオンラインリソースもあります。

- [AWS SDK for Java API リファレンス](#)
- [Java 開発者ブログ](#)
- [Java 開発者フォーラム](#)
- GitHub:
 - [ドキュメントソース](#)
 - [ドキュメントに関する問題](#)
 - [SDK ソース](#)
 - [SDK に関する問題](#)
 - [SDK サンプル](#)
 - [Gitter チャンネル](#)
- [AWS Code Sample Catalog](#)
- [@awsforjava \(Twitter\)](#)
- [リリースノート](#)

Eclipse IDE サポート

Eclipse IDE を使用してコードを開発する場合は、[AWS Toolkit for Eclipse](#) を使用して、既存の Eclipse プロジェクトに AWS SDK for Java を追加したり、新しい AWS SDK for Java プロジェクトを作成したりできます。このツールキットでは、Lambda 関数の作成およびアップロード、Amazon EC2 インスタンスの起動およびモニタリング、IAM ユーザーおよびセキュリティグループの管理、AWS CloudFormation テンプレートエディタなどもサポートされています。

詳細なドキュメントについては、「[AWS Toolkit for Eclipse ユーザーガイド](#)」を参照してください。

Android 向けアプリケーションの開発

Amazon Web Services では、Android 開発者向けに、Android 開発専用の SDK である [Amplify Android \(AWS Mobile SDK for Android\)](#) を提供しています。

SDK の改訂履歴の表示

SDK バージョンごとの変更とサポートされるサービスを含む、AWS SDK for Java のリリース履歴を確認するには、SDK の [リリースノート](#) を参照してください。

旧バージョンの SDK 用の Java リファレンスドキュメントのビルド

[AWS SDK for Java API リファレンス](#) は、SDK のバージョン 1.x の最新ビルドを表しています。1.x バージョンの旧ビルドを使用している場合、使用しているバージョンに合った SDK リファレンスドキュメントにアクセスできます。

ドキュメントをビルドする最も簡単な方法は、Apache の [Maven](#) ビルドツールを使用することです。システムに Maven がインストールされていない場合は、まず Maven をダウンロードしてインストールして、以下の手順に従ってリファレンスドキュメントをビルドします。

1. GitHub の SDK リポジトリの [\[Releases\]](#) ページで、使用している SDK バージョンを見つけて選択します。
2. zip (Windows を含むほとんどのプラットフォーム) または tar.gz (Linux、macOS、または Unix) のいずれかのリンクを選択し、SDK を自分のコンピュータにダウンロードします。
3. そのアーカイブをローカルディレクトリに解凍します。

4. コマンドラインで、アーカイブを解凍したディレクトリに移動して以下のコマンドを入力します。

```
mvn javadoc:javadoc
```

5. ビルドが完了すると、aws-java-sdk/target/site/apidocs/ ディレクトリに HTML ドキュメントが生成されています。

使用開始

このセクションでは、AWS SDK for Java のインストール方法、セットアップ方法、および使用方法に関する情報を示します。

トピック

- [AWS のサービスの使用に必要な基本設定](#)
- [を取得する方法 AWS SDK for Java](#)
- [構築ツールの使用](#)
- [開発用の AWS 認証情報と AWS リージョン のセットアップ](#)

AWS のサービスの使用に必要な基本設定

概要

AWS SDK for Java で AWS のサービス を使用してアクセスするアプリケーションを正常に開発するには、次の条件が必要です。

- AWS IAM Identity Center にある[AWS アクセスポータルにサインイン](#)できる必要があります。
- SDK 用に設定された [IAM ロールのアクセス許可](#)により、アプリケーションが必要とする AWS のサービス へのアクセスが許可されている必要があります。PowerUserAccess AWS 管理ポリシーに関連するアクセス許可は、ほとんどの開発ニーズに十分対応できます。
- 次の要素を備えた開発環境:
 - 以下の方法で設定される[共有設定ファイル](#)。
 - config ファイルには、AWS リージョン を指定するデフォルトプロファイルが含まれています。
 - credentials ファイルには、デフォルトプロファイルの一部として一時的な認証情報が含まれています。
 - 適切な [Java インストール](#)。
 - [Maven](#) や [Gradle](#) などの[構築オートメーションツール](#)。
 - コードを使用するテキストエディター。
 - [\(オプションだが推奨\) IntelliJ IDEA、Eclipse、NetBeans](#) などの IDE (統合開発環境)。

IDE を使用すると、AWS Toolkit を統合してより簡単に AWS のサービスを使用することもできます。[AWS Toolkit for IntelliJ](#) と、[AWS Toolkit for Eclipse](#) は Java 開発に使用できる 2 つのツールキットです。

Important

このセットアップセクションの手順は、ユーザーまたは組織が IAM アイデンティティセンターを使用していることを前提としています。組織が IAM アイデンティティセンターとは独立して機能する外部 ID プロバイダーを使用している場合は、SDK for Java で使用するための一時的な認証情報を取得する方法をご確認ください。[以下の手順](#)に従って、`~/.aws/credentials` ファイルに一時的な認証情報を追加します。

ID プロバイダーが一時的な認証情報を `~/.aws/credentials` ファイルに自動的に追加する場合は、SDK AWS CLI またはプロファイル名を指定する必要がないように、プロファイル名が `[default]` であることを確認してください。

AWS アクセスポータルにサインインします。

AWS アクセスポータルは、IAM アイデンティティセンターに手動でサインインするウェブ上の場所です。URL のフォーマットは `d-xxxxxxxxxx.awsapps.com/start`、または `your_subdomain.awsapps.com/start` です。

AWS アクセスポータルに慣れていない場合は、AWS SDK およびツールリファレンスガイドの [IAM Identity Center authentication](#) にあるアカウントアクセスに関するガイダンスに従ってください。ステップ 2 には従わないでください。AWS SDK for Java 1.x では、ステップ 2 で説明されている SDK の自動トークン更新や SDK の一時的な認証情報の自動取得がサポートされていないためです。

共有設定ファイルを設定します。

共有設定ファイルは開発ワークステーションにあり、すべての AWS SDK と AWS Command Line Interface (CLI) で使用される基本設定が含まれています。共有設定ファイルには [多数の設定](#) を含めることができますが、これらの手順は SDK を使用するために必要な基本要素を設定します。

共有 `config` ファイルをセットアップする

共有 `config` ファイルの内容の例を以下に示します。

```
[default]
```

```
region=us-east-1
output=json
```

開発目的では、コードを実行する予定の AWS リージョン に 最も近い 場所を使用してください。config ファイルで使用する リージョンコードのリスト については、「Amazon Web Services 全般のリファレンス ガイド」を参照してください。出力形式の json 設定は、いくつかの可能な値 のうちの 1 つです。

このセクション のガイダンスに従って config ファイルを作成します。

SDK の一時的な認証情報を設定します。

AWS アクセスポータルから AWS アカウント と IAM ロールにアクセスできるようになったら、SDK がアクセスするための一時的な認証情報を使用して開発環境を設定します。

一時的な認証情報を使用してローカル **credentials** ファイルを設定する手順

1. 共有 credentials ファイルを作成します。
2. credentials ファイルに、作業用の一時認証情報を貼り付けるまで次のプレースホルダーテキストを貼り付けます。

```
[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. ファイルを保存します。これで、ファイル `~/.aws/credentials` はローカルの開発システムに存在しているはずですが、このファイルには、特定の名前付きプロファイルが指定されていない場合に SDK for Java が使用する [default] プロファイル が含まれています。
4. AWS アクセスポータルにサインインします。
5. AWS アクセスポータルから IAM ロール認証情報をコピーするには、「手動での認証情報更新」にある次の手順を行います。
 - a. リンク先の手順のステップ 4 で、開発ニーズに合ったアクセスを許可する IAM ロールの名前を選択します。通常、このロールには PowerUserAccess や Developer などの名前が付いています。
 - b. ステップ 7 で、[AWS 認証情報ファイルにプロファイルを手動で追加] オプションを選択し、内容をコピーします。

を取得する方法 AWS SDK for Java

前提条件

を使用するには AWS SDK for Java、以下が必要です。

- [AWS IAM Identity Centerにある AWS アクセスポータルにサインイン](#)できる必要があります。
- 適切な [Java インストール](#)。
- ローカル共有 credentials ファイルに設定された一時的な認証情報。

SDK for Java を使用するための設定方法については、[the section called “基本セットアップ”](#) トピックを参照してください。

ビルドツールを使用して SDK for Java の依存関係を管理する (推奨)

SDK for Java の必要な依存関係にアクセスするには、プロジェクトで Apache Maven または Gradle を使用することをお勧めします。[このセクション](#)では、それらのツールを使用する方法について説明します。

SDK をダウンロードして解凍する (非推奨)

プロジェクトの SDK にアクセスするには構築ツールを使用することをお勧めしますが、最新バージョンの SDK のビルド済み jar をダウンロードすることもできます。

Note

以前のバージョンの SDK のダウンロードおよび構築方法の詳細については、「[以前のバージョンの SDK のインストール](#)」を参照してください。

1. <https://amazonwebservices.com/latest/aws-java-sdksdk-for-java.zip> から SDK をダウンロードします。
2. SDK をダウンロードしたら、そのコンテンツをローカルディレクトリに抽出します。

SDK には次のディレクトリが含まれます。

- documentation - API ドキュメントが含まれます (ウェブの [AWS SDK for Java API リファレンス](#)にも掲載されています)。

- lib - SDK の .jar ファイルが置かれています。
- samples - SDK の使用方法を示す作業サンプルコードが置かれています。
- third-party/lib - Apache Commons Logging、AspectJ、Spring framework など、SDK で使用されているサードパーティーライブラリが置かれています。

SDK を使用するには、lib と third-party ディレクトリの完全パスをビルドファイルの依存関係に追加し、コードを実行する Java の CLASSPATH にそれらを追加します。

SDK の以前のバージョンをソースから構築 (非推奨)

完全な SDK の最新バージョンのみ、ダウンロード可能な jar として構築済みの形で提供されます。ただし、Apache Maven (オープンソース) を使用して、以前のバージョンの SDK を構築できます。Maven が、必要なすべての依存関係をダウンロードし、1つのステップで SDK を構築およびインストールします。インストールの手順と詳細については、<http://maven.apache.org/> を参照してください。

1. SDK の GitHub ページ ([AWS SDK for Java GitHub](#)) に移動します。
2. 使用する SDK のバージョン番号に該当するタグを選択します。例えば 1.6.10 です。
3. [Download ZIP] ボタンをクリックして、選択したバージョンの SDK をダウンロードします。
4. 開発システムのディレクトリにファイルを解凍します。多くのシステムでは、グラフィカルなファイルマネージャーを使用してこれを行うことができます。または、ターミナルウィンドウで unzip コーティリティを使用できます。
5. ターミナルウィンドウで、SDK ソースを解凍したディレクトリに移動します。
6. 次のコマンドを実行 ([Maven](#) が必要) して SDK をビルドおよびインストールします。

```
mvn clean install -Dgpg.skip=true
```

.jar ファイルが target ディレクトリで構築されます。

7. (オプション) 次のコマンドを使用して API リファレンスのドキュメントを構築します。

```
mvn javadoc:javadoc
```

ドキュメントは target/site/apidocs/ ディレクトリに構築されます。

構築ツールの使用

構築ツールを使用すると、Java プロジェクトの開発を管理しやすくなります。構築ツールはいくつかありますが、ここでは一般的な 2 つの構築ツール (Maven と Gradle) を使って起動して実行する方法を説明します。このトピックでは、これらの構築ツールを使用して、プロジェクトに必要な SDK for Java の依存関係を管理する方法を説明します。

トピック

- [Apache Maven で SDK を使用する](#)
- [Gradle とともに SDK を使用する](#)

Apache Maven で SDK を使用する

[Apache Maven](#) を使用して、AWS SDK for Java プロジェクトの設定およびビルド、SDK 自体のビルドを行うことができます。

Note

このトピックのガイダンスを使用するには、Maven をインストールしている必要があります。まだインストールしていない場合は、<http://maven.apache.org/> にアクセスし、ダウンロードしてインストールしてください。

新しい Maven パッケージを作成する

Maven の基本パッケージを作成するには、ターミナル (コマンドライン) ウィンドウを開いて次のコマンドを実行します。

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

org.example.basicapp はアプリケーションの完全パッケージ名前空間に置き換え、myapp はプロジェクト名 (これがプロジェクトのディレクトリ名になります) に置き換えます。

デフォルトで、[quickstart](#) アーキタイプを使用してプロジェクトテンプレートが作成されます。このテンプレートは、多くのプロジェクトで手始めとして使用できます。使用できるアーキタイプは他に

もあります。[Maven アーキタイプ](#)のページでは、次でパッケージ化されているアーキタイプの一覧を参照できます。`-DarchetypeArtifactId` コマンドに `archetype:generate` 引数を追加すると、特定のアーキタイプを選択して使用できます。例:

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

Note

プロジェクトの作成と設定に関する詳細情報は、[Maven の開始方法ガイド](#)で提供されています。

SDK を Maven 依存関係に設定する

プロジェクトで AWS SDK for Java を使用するには、プロジェクトの `pom.xml` ファイルで SDK を依存関係として宣言する必要があります。バージョン 1.9.0 以降では、[個々のコンポーネント](#)または [SDK 全体](#)をインポートできます。

個別の SDK モジュールの指定

個々 SDK モジュールを選択するには、AWS SDK for Java の Maven 用の部品表 (BOM) を使用します。そうすることによって、指定したモジュールで同じバージョンの SDK が使用され、それらは互いに互換性があることが保証されます。

BOM を使用するには、アプリケーションの `<dependencyManagement>` ファイルに `pom.xml` セクションを追加し、そのセクションで `aws-java-sdk-bom` を依存関係として追加し、使用する SDK のバージョンを指定します。

```
<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>
```

Maven Central で使用可能な最新バージョンの AWS SDK for Java BOM を表示するには、<https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-bom> を参照してください。どのモジュール (依存関係) が、プロジェクトの pom.xml ファイルの <dependencies> セクション内に含めることができる BOM により管理されるのかを確認するには、このページを参照してください。

これで、アプリケーションで使用する個々のモジュールを SDK から選択できるようになりました。BOM で SDK バージョンを既に宣言しているため、各コンポーネントのバージョン番号を指定する必要はありません。

```
<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>
```

AWS Code Sample Catalog を参照して、特定の AWS のサービスで使用する依存関係を確認することもできます。特定のサービス例の下での POM ファイルを参照してください。例えば、AWS S3 サービスの依存関係に関心がある場合は、GitHub の [完全な例](#) を参照してください (/java/example_code/s3 の下の POM を参照してください)。

すべての SDK モジュールのインポート

SDK 全体を依存関係にする場合は、BOM による方法を使用せずに、pom.xml で次のように宣言するだけです。

```
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk</artifactId>
    <version>1.11.1000</version>
  </dependency>
```

```
</dependencies>
```

プロジェクトを構築する

プロジェクトのセットアップが完了したら、Maven の `package` コマンドを使用してアプリケーションをビルドできます。

```
mvn package
```

このコマンドでは、`0jar` ディレクトリに `target` ファイルが作成されます。

Maven を使用して SDK をビルドする

Apache Maven を使用して、SDK をソースからビルドできます。そのためには、[GitHub から SDK コードをダウンロード](#)し、ローカルに解凍して、次の Maven コマンドを実行します。

```
mvn clean install
```

Gradle とともに SDK を使用する

[Gradle](#) プロジェクトの SDK 依存関係を管理するには、AWS SDK for Java の Maven BOM をアプリケーションの `build.gradle` ファイルにインポートします。

Note

次の例では、構築ファイルの `1.12.529` を有効なバージョンの AWS SDK for Java に置き換えます。[Maven Central リポジトリ](#)で最新バージョンを検索してください。

Gradle 4.6 以降のプロジェクト設定

[Gradle 4.6 以降](#)、BOM で依存関係を宣言することにより、Gradle の改善された POM サポート機能を使用して部品表 (BOM) ファイルをインポートできます。

1. Gradle 5.0 以降を使用している場合は、ステップ 2 に進みます。それ以外の場合は、`settings.gradle` ファイルで `IMPROVED_POM_SUPPORT` 機能を有効にします。

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

2. アプリケーションの `build.gradle` ファイルの `dependencies` セクションに BOM を追加します。

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

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

3. `dependencies` セクションで使用する SDK モジュールを指定します。たとえば、次の例には Amazon Simple Storage Service (Amazon S3) の依存関係が含まれています。

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

Gradle は BOM の情報を使用して、正しいバージョンの SDK 依存関係を自動的に解決します。

以下に、Amazon S3 の依存関係を含む完全な `build.gradle` ファイルの例を示します。

```
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

前の例で、Amazon S3 の依存関係を、プロジェクトで使用する AWS のサービスの依存関係に置き換えます。AWS SDK for Java BOM によって管理されるモジュール (依存関係) は [Maven Central リポジトリ](#) に一覧表示されます。

4.6 より前の Gradle バージョンのプロジェクト設定

4.6 より前の Gradle バージョンでは、ネイティブ BOM がサポートされていません。プロジェクトの AWS SDK for Java 依存関係を管理するには、Spring の Gradle 用 [依存関係管理プラグイン](#) を使用して、SDK の Maven BOM をインポートします。

1. ご自身のアプリケーションの `build.gradle` ファイルに依存関係管理プラグインを追加します。

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "io.spring.gradle:dependency-management-plugin:1.0.9.RELEASE"
    }
}

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

2. ファイルの `dependencyManagement` セクションに BOM を追加します。

```
dependencyManagement {
    imports {
        mavenBom 'com.amazonaws:aws-java-sdk-bom:1.12.529'
    }
}
```

3. `dependencies` セクションで使用する SDK モジュールを指定します。たとえば次の例には、Amazon S3 の依存関係が含まれています。

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

Gradle は BOM の情報を使用して、正しいバージョンの SDK 依存関係を自動的に解決します。

以下に、Amazon S3 の依存関係を含む完全な build.gradle ファイルの例を示します。

```
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

前の例で、Amazon S3 の依存関係を、プロジェクトで使用する AWS のサービスの依存関係に置き換えます。AWS SDK for Java BOM によって管理されるモジュール (依存関係) は [Maven Central リポジトリ](#) に一覧表示されます。

BOM を使用した SDK の依存関係の指定の詳細については、[Apache Maven とともに SDK を使用する](#)を参照してください。

開発用の AWS 認証情報と AWS リージョンのセットアップ

AWS SDK for Java でサポートされている任意のサービスに接続するには、AWS の一時的な認証情報を指定する必要があります。AWS SDK と CLI では、プロバイダーチェーンを使用して、システムまたはユーザーの環境変数やローカルの AWS 設定ファイルなど、いくつかの場所にある AWS の一時的な認証情報が検索されます。

このトピックでは、AWS SDK for Java を使用して、ローカルアプリケーション開発用に AWS の一時的な認証情報をセットアップするための基本情報について説明します。EC2 インスタンス内で使用するための認証情報をセットアップする必要がある場合、または開発に Eclipse IDE を使用している場合は、代わりに以下のトピックを参照してください。

- EC2 インスタンスを使用している場合は、[Amazon EC2 での IAM ロールを使用した AWS リソースへの許可の付与](#)での説明に従って、IAM ロールを作成し、そのロールへのアクセス権を EC2 インスタンスに付与します。
- [AWS Toolkit for Eclipse](#) を使用して Eclipse 内で AWS 認証情報を設定します。詳細については、「[AWS Toolkit for Eclipse ユーザーガイド](#)」の「[Set up AWS Credentials](#)」を参照してください。

一時的な認証情報の設定

AWS SDK for Java の一時的な認証情報の設定はさまざまな方法で行うことができますが、推奨される手法を次に示します。

- 次の場所にあるローカルシステム上の AWS の一時的な認証情報プロファイルファイルで認証情報を設定します。
 - ~/.aws/credentialsLinux、macOS、Unix の場合は
 - Windows の C:\Users\USERNAME\.aws\credentials

一時的な認証情報を取得する方法については、本ガイドの「[the section called “SDK の一時的な認証情報を設定します。”](#)」を参照してください。

- AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEY、および AWS_SESSION_TOKEN 環境変数を設定します。

これらの変数を Linux、macOS、または Unix で設定するには、`awscli` を使用します。

```
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
```

Windows でこれらの変数を設定するには、`set` を使用します。

```
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
```

- EC2 インスタンスの場合、IAM ロールを指定し、そのロールへのアクセスを EC2 インスタンスに提供します。これがどのように機能するかについての詳細な説明については、「Linux インスタンス用 Amazon EC2 ユーザーガイド」の「[IAM Roles for Amazon EC2](#)」を参照してください。

これらの方法のいずれかを使用して AWS の一時的な認証情報を設定すると、デフォルトの認証情報プロバイダチェーンを使用して AWS SDK for Java によって自動的にロードされます。Java アプリケーションでの AWS 認証情報の使用の詳細については、[AWS 認証情報の使用](#)を参照してください。

IMDS 認証情報の更新

AWS SDK for Java は、認証情報の有効期限に関係なく、バックグラウンドで 1 分ごとに IMDS 認証情報を更新するオプションをサポートしています。これにより、認証情報をより頻繁に更新できるようになり、IMDS に到達できないことで認識される AWS の可用性に影響が出る可能性が低くなります。

```
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();
```

AWS リージョン の設定

AWS SDK for Java で AWS のサービスにアクセスするために使用するデフォルトの AWS リージョンを設定する必要があります。最適なパフォーマンスを得るために、地理的に近いリージョンを選択します。各サービスのリージョンのリストについては、Amazon Web Services の全般のリファレンスの[リージョンとエンドポイント](#)を参照してください。

Note

リージョンを選択しないと、デフォルトで us-east-1 が使用されます。

認証情報の設定と同様の方法でデフォルトの AWS リージョンを設定できます。

- ローカルシステムの次の場所にある AWS Config ファイルで AWS リージョンを設定します。
 - Linux、macOS、または Unix の `~/.aws/config`
 - Windows の `C:\Users\USERNAME\.aws\config`

このファイルには以下の形式の行が含まれている必要があります。

+

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

+

`your_aws_region` を目的の AWS リージョン (「us-west-1」など) に置き換えます。

- AWS_REGION 環境変数を設定します。

Linux、macOS、または Unix では、`export` を使用してください:

```
export AWS_REGION=your_aws_region
```

Windows では、`set` を使用します。

```
set AWS_REGION=your_aws_region
```

`your_aws_region` は目的の AWS リージョン 名です。

の使用 AWS SDK for Java

このセクションでは、SDK で使用するすべてのサービス AWS SDK for Java に適用される を使用したプログラミングに関する重要な一般情報を提供します。

サービス固有のプログラミング情報と例 (Amazon EC2、 など) については Amazon S3 Amazon SWF、[AWS SDK for Java 「コード例」](#) を参照してください。

トピック

- [を使用した AWS 開発のベストプラクティス AWS SDK for Java](#)
- [サービスクライアントの作成](#)
- [への一時的な認証情報の提供 AWS SDK for Java](#)
- [AWS リージョン 選択](#)
- [例外処理](#)
- [非同期プログラミング](#)
- [AWS SDK for Java 通話のログ記録](#)
- [クライアント設定](#)
- [アクセスコントロールポリシー](#)
- [DNS 名参照用の JVM TTL を設定する](#)
- [のメトリクスの有効化 AWS SDK for Java](#)

を使用した AWS 開発のベストプラクティス AWS SDK for Java

以下のベストプラクティスは、 で AWS アプリケーションを開発する際の問題や問題を回避するのに役立ちます AWS SDK for Java。ベストプラクティスはサービスごとに整理してあります。

S3

回避 ResetExceptions

ストリーム Amazon S3 を使用して (AmazonS3クライアントまたは を介してTransferManager) オブジェクトを にアップロードすると、ネットワーク接続またはタイムアウトの問題が発生する可能性があります。デフォルトでは、 は転送の開始前に入カストリームをマークし、 AWS SDK for Java 再試行する前にリセットすることで、失敗した転送を再試行しようとします。

ストリームがマークとリセットをサポートしていない場合、SDK は一時的な障害が発生し、再試行が有効になってい[ResetException](#)ると をスローします。

ベストプラクティス

mark オペレーションと reset オペレーションをサポートするストリームを使用することをお勧めします。

を回避する最も信頼性の高い方法は、[ファイル](#)または を使用してデータを提供することと[ResetException](#)です。これは[FileInputStream](#)、 がマークとリセットの制限に制約されることなく処理 AWS SDK for Java できます。

ストリーム[FileInputStream](#)が ではなく、マークとリセットをサポートしている場合は、 の `setReadLimit`メソッドを使用してマーク制限を設定できます[RequestClientOptions](#)。デフォルト値は 128 KB です。読み取り制限値をストリームのサイズよりも 1 バイト大きく設定すると、 は確実に回避されます[ResetException](#)。

たとえば、ストリームの最大予想サイズが 100,000 バイトである場合、読み取り制限を 100,001 (100,000 + 1) バイトに設定します。mark と reset は常に 100,000 バイト以下で動作します。これにより、一部のストリームではそのバイト数がメモリにバッファされる場合があることに注意してください。

サービスクライアントの作成

にリクエストを行うには Amazon Web Services、まずサービスクライアントオブジェクトを作成します。推奨される方法は、サービスクライアントビルダーを使用することです。

各 AWS のサービス には、サービス API の各アクションのメソッドを含むサービスインターフェイスがあります。例えば、DynamoDB のサービスインターフェイスの名前は [AmazonDynamoDBClient](#) です。各サービスインターフェイスに対応するクライアントビルダーがあり、サービスインターフェイスの実装を構築する際に使用できます。のクライアントビルダークラス DynamoDB は [AmazonDynamoDB ClientBuilder](#)という名前です。

クライアントビルダーの取得

クライアントビルダーのインスタンスを取得するには、次の例に示す静的ファクトリメソッド `standard` を使用します。

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

ビルダーを取得すると、ビルダー API で多数の fluent setter を使用して、クライアントのプロパティをカスタマイズできます。たとえば、カスタムリージョンおよびカスタム認証情報プロバイダーは以下のように設定できます。

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

Note

fluent withXXX メソッドは builder オブジェクトを返し、メソッドの呼び出しを連鎖させて利便性とコードの可読性を向上させることができます。必要なプロパティを設定後、build メソッドを呼び出してクライアントを作成することができます。作成後のクライアントはイミュータブルで、setRegion または setEndpoint の呼び出しはすべて失敗します。

ビルダーでは、同じ構成で複数のクライアントを作成できます。アプリケーションを記述している場合、ビルダーがミュータブルであり、スレッドセーフでないことに注意してください。

次のコードでは、クライアントインスタンスのファクトリとしてビルダーが使用されます。

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

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

ビルダーは、[ClientConfiguration](#) との流暢なセッターと [RequestMetricCollector](#)、カスタムリスト [RequestHandler2](#) も公開します。

すべての設定可能なプロパティをオーバーライドする完全な例を次に示します。

```
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();
```

非同期クライアントの作成

AWS SDK for Java には、すべてのサービス (を除く) に非同期 (または非同期 Amazon S3) クライアントがあり、すべてのサービスに対応する非同期クライアントビルダーがあります。

デフォルトの を使用して非同期 DynamoDB クライアントを作成するには
ExecutorService

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

同期 (または同期) クライアントビルダーがサポートする設定オプションに加えて、非同期クライアントでは、非同期クライアントExecutorServiceが使用する [ExecutorFactory](#) を変更するカスタムを設定できます。ExecutorFactoryは機能インターフェイスであるため、Java 8 Lambda 式およびメソッドリファレンスと相互運用されます。

非同期クライアントをカスタムエグゼキューターで作成するには

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

の使用 DefaultClient

同期クライアントビルダーと非同期クライアントビルダーの両方に、defaultClient という名前の別のファクトリメソッドがあります。このメソッドは、デフォルトプロバイダチェーンを使用して認証情報と AWS リージョンをロードすることで、デフォルト構成を持つサービスクライアントを作成します。アプリケーションを実行している環境から認証情報またはリージョンが決定できない場

合、defaultClient の呼び出しは失敗します。[AWS 認証情報とリージョンの決定方法の詳細については](#)、「[認証情報の使用とAWS リージョン 選択](#)」を参照してください。

デフォルトサービスクライアントを作成するには

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

クライアントのライフサイクル

SDK のサービスクライアントはスレッドセーフであり、最大のパフォーマンスを得るには、存続期間の長いオブジェクトとしてこれら进行处理する必要があります。各クライアントは独自の接続プールリソースを備えています。リソースリークを回避するために、不要になったときは明示的にクライアントをシャットダウンしてください。

明示的にクライアントをシャットダウンする場合は、shutdown メソッドを呼び出します。shutdown の呼び出し後は、すべてのクライアントリソースが解放され、クライアントが使用できなくなります。

クライアントをシャットダウンするには

```
AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
ddb.shutdown();
// Client is now unusable
```

への一時的な認証情報の提供 AWS SDK for Java

にリクエストを行うには Amazon Web Services、 が サービスを呼び出すときに AWS SDK for Java 使用する AWS 一時的な認証情報を指定する必要があります。これは以下の方法で対応できます。

- デフォルトの認証情報プロバイダーチェーンを使用する (推奨)。
- 特定の認証情報プロバイダーまたはプロバイダーチェーンを使用する (または独自のものを作成する)。
- 一時的な認証情報はご自分でコードに入力してください。

デフォルトの認証情報プロバイダーチェーンの使用

引数を指定せずに新しいサービスクライアントを初期化すると、[AWSCredentialsProviderChain](#)はデフォルトクラスによって実装されたデフォルトの認証情報プロバイダーチェーンを使用して一時的

な認証情報を検索 AWS SDK for Java しようとします。デフォルトの認証情報プロバイダーチェーンは、次の順序で認証情報を検索します。

1. 環境変数 - `AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY`、`AWS_SESSION_TOKEN`。AWS SDK for Java は [EnvironmentVariableCredentialsProvider](#) クラスを使用してこれらの認証情報をロードします。
2. Java のシステムプロパティ - `aws.accessKeyId`、`aws.secretKey`、`aws.sessionToken`。AWS SDK for Java は を使用してこれらの認証情報 [SystemPropertiesCredentialsProvider](#) をロードします。
3. 環境またはコンテナからのウェブアイデンティティトークンの認証情報。
4. デフォルトの認証情報プロファイルファイル - 通常、 にあり `~/.aws/credentials` (場所はプラットフォームによって異なります)、多くの AWS SDKs とによって共有されます AWS CLI。AWS SDK for Java は を使用してこれらの認証情報 [ProfileCredentialsProvider](#) をロードします。

が提供する `aws configure` コマンドを使用して認証情報ファイルを作成することも AWS CLI、テキストエディタでファイルを編集して作成することもできます。認証情報ファイル形式に関する情報については、「[AWS Credentials File Format](#)」を参照してください。

5. Amazon ECS コンテナの認証情報 - 環境変数 `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` が設定されている場合に、Amazon ECS からロードされます。AWS SDK for Java は を使用してこれらの認証情報 [ContainerCredentialsProvider](#) をロードします。この値の IP アドレスを指定できます。
6. インスタンスプロファイルの認証情報 - EC2 インスタンスで使用され、Amazon EC2 メタデータサービスを介して配信されます。AWS SDK for Java は を使用してこれらの認証情報 [InstanceProfileCredentialsProvider](#) をロードします。この値の IP アドレスを指定できます。

Note

インスタンスプロファイル認証情報

は、`AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` が設定されていない場合にのみ使用されます。詳細については、「[EC2ContainerCredentialsProviderWrapper](#)」を参照してください。

一時的な認証情報の設定

AWS 一時的な認証情報を使用するには、前述の場所の少なくとも 1 つに設定する必要があります。認証情報の設定の詳細については、以下のトピックを参照してください。

- 環境またはデフォルトの認証情報プロファイルファイルでの認証情報の指定については、「[the section called “一時的な認証情報の設定”](#)」を参照してください。
- Java システムプロパティの設定については、公式の [Java Tutorials](#) ウェブサイトにある「System Properties」チュートリアルを参照してください。
- EC2 インスタンスでインスタンスプロファイル認証情報をセットアップして使用するには、「[IAM ロールを使用しての AWS リソースへのアクセスを許可する Amazon EC2](#)」を参照してください。

別の認証情報プロファイルの設定

はデフォルトでデフォルトのプロファイル AWS SDK for Java を使用しますが、認証情報ファイルから取得されるプロファイルをカスタマイズする方法があります。

AWS プロファイル環境変数を使用して、SDK によってロードされたプロファイルを変更できます。

例えば、Linux、macOS、または Unix の場合は、次のコマンドを実行してプロファイルを myProfile に変更します。

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

Windows の場合は次のコマンドを使用します。

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

AWS_PROFILE 環境変数を設定すると、公式にサポートされているすべての AWS SDKs とツール (AWS CLI とを含む AWS Tools for Windows PowerShell) の認証情報のロードに影響します。Java アプリケーションのプロファイルだけを変更する場合は、代わりにシステムプロパティ `aws.profile` を使用できます。

Note

環境変数はシステムプロパティより優先されます。

別の認証情報ファイルの場所を設定する

は、デフォルトの認証情報ファイルの場所から AWS 一時的な認証情報を自動的に AWS SDK for Java ロードします。ただし、認証情報ファイルへのフルパスを使用して

AWS_CREDENTIAL_PROFILES_FILE 環境変数を設定することで、場所を指定することもできます。

この機能を使用して、`aws_credentials_file` が認証情報ファイル AWS SDK for Java を検索する場所を一時的に変更できます (例えば、この変数をコマンドラインで設定するなど)。または、ユーザー環境やシステム環境で環境変数を設定して、ユーザーやシステム全体に対して変数を変更できます。

認証情報ファイルのデフォルトの場所を上書きするには

- AWS_CREDENTIAL_PROFILES_FILE 環境変数を AWS 認証情報ファイルの場所に設定します。
 - Linux、macOS、または Unix では、次を使用します。

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

- Windows では、次を使用します。

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

Credentials ファイル形式

このガイドの[基本設定の指示](#)に従うと、認証情報ファイルは次の基本形式になります。

```
[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>
```

プロファイル名では角括弧 (例: `[default]`) に続いて、キーと値のペアとして、そのプロファイルの設定可能なフィールドを指定します。credentials ファイルで複数のプロファイルを持つことができ、それらは `aws configure --profile PROFILE_NAME` を使用して設定するプロファイルを選択することで追加または編集できます。

`metadata_service_timeout`、`metadata_service_num_attempts` などの追加のフィールドを指定できます。これらは、CLI で設定可能ではなく、使用する場合は手動でファイルを編集する必

必要があります。設定ファイルとその使用可能なフィールドの詳細については、「[ユーザーガイド](#)」の「[の設定 AWS Command Line Interface](#) AWS Command Line Interface」を参照してください。

認証情報の読み込み

一時的な認証情報を設定した後は、SDK はデフォルトの認証情報プロバイダーチェーンを使用してその認証情報をロードします。

これを行うには、次のように、ビルダーに認証情報を明示的に提供せずに AWS のサービス クライアントをインスタンス化します。

```
AmazonS3 s3Client = AmazonS3ClientBuilder.standard()
    .withRegion(Regions.US_WEST_2)
    .build();
```

認証情報プロバイダーまたはプロバイダーチェーンの指定

クライアントビルダーを使用して、デフォルトの認証情報プロバイダーチェーンとは異なる認証情報プロバイダーを指定できます。

[AWSCredentialsProvider](#) インターフェイスを入力として受け取るクライアントビルダーに、認証情報プロバイダーまたはプロバイダーチェーンのインスタンスを提供します。以下の例は、環境認証情報を具体的に使用する方法を示しています。

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

AWS SDK for Javaが提供する認証情報プロバイダーとプロバイダーチェーンの完全なリストについては、「」の「[既知のすべての実装クラス](#)」を参照してください[AWSCredentialsProvider](#)。

Note

この手法を使用して、[AWSCredentialsProvider](#) インターフェイスを実装する独自の認証情報プロバイダーを使用するか、[AWSCredentialsProviderChain](#) クラスをサブクラスすることで、作成した認証情報プロバイダーまたはプロバイダーチェーンを指定できます。

一時的な認証情報を明示的に指定する

デフォルトの認証情報チェーンまたは特定あるいはカスタムのプロバイダーやプロバイダーチェーンがコードに対して機能しない場合は、明示的に指定する認証情報を設定できます。を使用して一時的な認証情報を取得した場合は AWS STS、この方法を使用して AWS アクセス用の認証情報を指定します。

1. [BasicSessionCredentials](#) クラスをインスタンス化し、AWS SDK が接続に使用するアクセスキー、AWS シークレットキー、および AWS セッショントークンを提供します。
2. `AWSCredentials` オブジェクト [AWSStaticCredentialsProvider](#) を使用してを作成します。
3. `AWSStaticCredentialsProvider` を使用してクライアントビルダーを設定し、クライアントをビルドします。

次に例を示します。

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

詳細情報

- [にサインアップ AWS して IAM ユーザーを作成する](#)
- [開発用の AWS 認証情報とリージョンを設定する](#)
- [IAM ロールを使用しての AWS リソースへのアクセスを許可する Amazon EC2](#)

AWS リージョン 選択

リージョンを使用すると、特定の地理的地域に物理的に存在する AWS サービスにアクセスできます。これは、冗長性と、ユーザーがアクセスする場所の近くでのデータとアプリケーションの実行を維持するために有効です。

リージョンでのサービス可用性の確認

特定の AWS のサービスがリージョンで利用可能かどうかを確認するには、使用するリージョンで `isServiceSupported` メソッドを使用します。

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

指定できるリージョンに関しては、「[Regions](#)」クラスのドキュメントを参照して、サービスのエンドポイントプレフィックスを使用してクエリを実行します。各サービスのエンドポイントプレフィックスはサービスインターフェイスで定義されています。例えば、DynamoDB エンドポイントプレフィックスは [AmazonDynamoDB](#) で定義されています。

リージョンを選択する

のバージョン 1.4 以降では AWS SDK for Java、リージョン名を指定でき、SDK は自動的に適切なエンドポイントを選択します。自分でエンドポイントを選択する場合は、「[特定のエンドポイントの選択](#)」を参照してください。

リージョンを明示的に設定するには、「[Regions](#)」列挙型を使用することをお勧めします。これは、公開されている利用可能なすべてのリージョンを列挙したものです。その列挙型からリージョンを使用してクライアントを作成するには、以下のコードを使用します。

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

使用するリージョンが `Regions` 列挙型にない場合は、リージョンの名前を示す `string` を使用してリージョンを設定できます。

```
AmazonEC2 ec2 = AmazonEC2ClientBuilder.standard()
    .withRegion("{region_api_default}")
    .build();
```

Note

ビルダーを使用してクライアントをビルドした後、そのクライアントはイミュータブルとなり、リージョンは変更不可能です。同じサービス AWS リージョン で複数の を使用している場合は、リージョンごとに 1 つずつ、複数のクライアントを作成する必要があります。

特定のエンドポイントの選択

各 AWS クライアントは、クライアントの作成時に `withEndpointConfiguration` メソッドを呼び出すことで、リージョン内の特定のエンドポイントを使用するように設定できます。

例えば、欧州 (アイルランド) リージョンを使用するように Amazon S3 クライアントを設定するには、次のコードを使用します。

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

すべてのサービスの現在の [リージョンと対応するエンドポイントのリスト](#) については、「リージョンとエンドポイント」を参照してください AWS。

環境に基づくリージョンの自動的な決定

Important

このセクションは、[クライアントビルダー](#) を使用して、クライアントコンストラクタを使用して作成された AWS services. AWS clients にアクセスする場合にのみ適用され、環境からリージョンを自動的に決定せず、代わりにデフォルトの SDK リージョン (USEast1) を使用します。

Amazon EC2 または Lambda で実行している場合、コードが実行されているのと同じリージョンを使用するようにクライアントを設定することをお勧めします。そうすることによって、コードとそれが実行される環境が分離され、レイテンシーの低減や冗長化のために複数のリージョンにデプロイすることが容易になります。

コードが実行されているリージョンを SDK で自動的に検出するには、クライアントビルダーを使用する必要があります。

認証情報/リージョンのデフォルトプロバイダーチェーンを使用して環境からリージョンを決定するには、クライアントビルダーの `defaultClient` メソッドを使用します。

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


これは、standard に続けて build を使用するのと同じです。

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

withRegion メソッドを使用してリージョンを明示的に設定しない場合は、SDK がリージョンのデフォルトプロバイダーチェーンに問い合わせ、使用するリージョンを決定しようとします。

デフォルトリージョンプロバイダーチェーン

リージョンルックアップ手順は以下のとおりです。

1. ビルダー自体で withRegion または setRegion を使用して設定されている明示的なリージョンは、その他の設定より優先されます。
2. AWS_REGION 環境変数が確認されます。設定されている場合は、そのリージョンを使用してクライアントが設定されます。

Note

この環境変数は Lambda コンテナによって設定されます。

3. SDK は、AWS 共有設定ファイル (通常は `~/.aws/config`) をチェックします。リージョンプロパティがあると、SDK はこのプロパティを使用します。
 - AWS_CONFIG_FILE 環境変数を使用すると、共有設定ファイルの場所をカスタマイズできません。
 - AWS_PROFILE 環境変数または `aws.profile` システムプロパティを使用すると、SDK によってロードされるプロファイルをカスタマイズできます。
4. SDK は、Amazon EC2 インスタンスメタデータサービスを使用して、現在実行中の Amazon EC2 インスタンスのリージョンを決定しようとします。
5. この時点で SDK によってリージョンがまだ見つからない場合は、クライアント作成が失敗し、例外が発生します。

AWS アプリケーションを開発する場合の一般的な方法は、共有設定ファイル ([デフォルト認証情報プロバイダーチェーンの使用で説明](#)) を使用してローカル開発用のリージョンを設定し、AWS インフラストラクチャで実行するときにデフォルトのリージョンプロバイダーチェーンに依存してリージョンを決定することです。これによって、クライアントの作成が大幅に簡略化され、アプリケーションの性が維持されます。

例外処理

が例外を AWS SDK for Java スローする方法とタイミングを理解することは、SDK を使用して高品質のアプリケーションを構築する上で重要です。以下のセクションでは、SDK によってスローされる例外のさまざまなケース、および例外の適切な処理方法について説明します。

非チェック例外を使用する理由

は、次の理由により、チェックされた例外の代わりにランタイム (またはチェックされていない) 例外 AWS SDK for Java を使用します。

- 懸念がない場合には例外ケースの処理を強制的に適用せずに (さらにコードを冗長にすることなく)、開発者がエラーを細かく制御できるようにするため。
- サイズの大きいアプリケーションで、チェック例外に関連する拡張性の問題が発生するのを防ぐため。

一般的に、チェック例外は小規模なアプリケーションでは役立ちますが、アプリケーションのサイズが大きくなり、複雑化すると、チェック例外が問題となる場合があります。

チェック例外と非チェック例外の使用方法の詳細については、以下を参照してください。

- [非チェック例外に関する論争](#)
- [チェック例外に関連する問題点](#)
- [Java のチェック例外の誤り \(およびその対処方法\)](#)

AmazonServiceException (およびサブクラス)

[AmazonServiceException](#) は、の使用時に発生する最も一般的な例外です AWS SDK for Java。この例外は、AWS のサービスからのエラーレスポンスを表します。例えば、存在しない Amazon EC2 インスタンスを終了しようとする、EC2 はエラーレスポンスを返し、そのエラーレスポンスのすべての詳細がスローされる AmazonServiceException に含まれます。場合によっては、AmazonServiceException のサブクラスがスローされ、開発者は catch ブロックを使用して、エラーケースの処理を細かく制御できるようになります。

が発生すると AmazonServiceException、リクエストは正常に に送信されました AWS のサービスが、正常に処理できなかったことがわかります。これは、リクエストのパラメータに含まれるエラーまたはサービス側の問題が原因です。

AmazonServiceException では次のような情報がわかります。

- 返された HTTP ステータスコード
- 返された AWS エラーコード
- サービスからの詳細なエラーメッセージ
- AWS 失敗したリクエストの リクエスト ID

AmazonServiceException には、失敗したリクエストが発信者の障害 (不正な値を持つリクエスト) か、AWS のサービスの障害 (内部サービスエラー) に関する情報も含まれています。

AmazonClientException

[AmazonClientException](#) は、にリクエストを送信しようとしたとき、AWS または からのレスポンスを解析しようとしたときに、Java クライアントコード内で問題が発生したことを示します AWS。AmazonClientException は通常、 よりも深刻でありAmazonServiceException、クライアントが のサービスへのサービス呼び出しを妨げている重大な問題を示します AWS 。例えば、いずれかのクライアントで オペレーションを呼び出そうとすると、 はネットワーク接続が利用AmazonClientExceptionできない場合に を AWS SDK for Java スローします。

非同期プログラミング

同期メソッドまたは非同期メソッドを使用して、AWS サービスで オペレーションを呼び出すことができます。同期メソッドは、クライアントがサービスからのレスポンスを受信するまでスレッドの実行をブロックします。非同期メソッドはすぐに応答を返し、レスポンスを待機せずに呼び出しスレッドに制御を戻します。

非同期メソッドはレスポンスが可能になる前に応答を返すため、準備ができたらレスポンスを得るための手段が必要になります。AWS SDK for Java には、将来のオブジェクトとコールバックメソッドの 2 つの方法があります。

Java Future

の非同期メソッドは、[将来の](#) の非同期オペレーションの結果を含む Future オブジェクト AWS SDK for Java を返します。

Future isDone() メソッドを呼び出し、サービスが既に応答オブジェクトを提供したかどうかを確認します。レスポンスの準備が整うと、Future get() メソッドを呼び出して応答オブジェクトを

取得できます。このメカニズムを使用して、アプリケーションで他の動作を続行しながら、定期的に非同期オペレーションの結果をポーリングすることができます。

オブジェクトを保持Futureできる を受け取る Lambda 関数を呼び出す非同期オペレーションの例を次に示します[InvokeResult](#)。InvokeResult オブジェクトは、isDone() が true になった後に限り取得されます。

```
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);
}
```

非同期コールバック

Java Future オブジェクトを使用して非同期リクエストのステータスをモニタリングすることに加えて、SDK では、[AsyncHandler](#) インターフェイスを使用するクラスを実装することもできます。AsyncHandlerには、リクエストの完了方法に応じて呼び出される 2 つのメソッド、onSuccessとが用意されていますonError。

コールバックインターフェイスアプローチの主な利点は、リクエストがいつ完了したかを調べるために Future オブジェクトをポーリングする必要がなくなることです。コードによってすぐに次のアクティビティを開始でき、また SDK に依存してハンドラを適時に呼び出すことができます。

```
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(".");
    }
}
}
```

ベストプラクティス

コールバックの実行

AsyncHandler の実装は、非同期クライアントが所有するスレッドプール内で実行されます。素早く実行される短いコードが AsyncHandler 実装内で最も適しています。ハンドラメソッド内の長期実行コードまたはブロックコードにより、非同期クライアントが使用するスレッドプールの競合が起これば、クライアントのリクエスト実行が妨げられる場合があります。コールバックで始める必要がある長期実行タスクがある場合は、コールバックを新しいスレッドで、またはアプリケーションが管理するスレッドプールで実行します。

スレッドプールの構成

の非同期クライアントは、ほとんどのアプリケーションで機能するデフォルトのスレッドプール AWS SDK for Java を提供します。カスタムを実装 [ExecutorService](#) して AWS SDK for Java 非同期クライアントに渡すことで、スレッドプールの管理方法をより詳細に制御できます。

例えば、カスタム を使用してプール内のスレッドの名前付け方法 [ThreadFactory](#) を制御したり、スレッドの使用状況に関する追加情報をログに記録する [ExecutorService](#) 実装を指定できます。

非同期アクセス

SDK の [TransferManager](#) クラスは、 を操作するための非同期サポートを提供します Amazon S3。は非同期のアップロードとダウンロード [TransferManager](#) を管理し、転送に関する詳細な進行状況レポートを提供し、さまざまなイベントへのコールバックをサポートします。

AWS SDK for Java 通話のログ記録

AWS SDK for Java は [Apache Commons Logging](#) で計測されます。これは、実行時に複数のログ記録システムのいずれかを使用できるようにする抽象化レイヤーです。

サポートされるログ記録システムには、Java ロギング フレームワークや Apache Log4j などがあります。このトピックでは Log4j の使用方法を示します。SDK のログ記録機能は、アプリケーションコードを変更せずに使用できます。

[Log4j](#) の詳細については、[Apache ウェブサイト](#) を参照してください。

Note

このトピックでは、Log4j 1.x について説明します。Log4j2 は Apache Commons のログ記録を直接サポートしませんが、Apache Commons ログ記録インターフェイスを使用して Log4j2 にログ記録呼び出しを自動的にダイレクトするアダプターを提供します。詳細については、Log4j2 のドキュメントの「[Commons Logging Bridge](#)」を参照してください。

Log4J JAR のダウンロード

SDK で Log4j を使用するには、Apache ウェブサイトで Log4j JAR をダウンロードする必要があります。SDK に JAR は含まれていません。クラスパス上の場所に JAR ファイルをコピーします。

Log4j では、設定ファイル log4j.properties を使用します。設定ファイルの例を次に示します。クラスパス上のディレクトリに、この設定ファイルをコピーします。Log4j JAR と log4j.properties ファイルは、同じディレクトリに配置しないでください。

log4j.properties 設定ファイルは、[ログ記録レベル](#)、ログ記録出力の送信先 (たとえば [ファイルやコンソール](#))、[出力形式](#)などのプロパティを指定します。ログ記録レベルは、ロガーによって生成される出力の詳細度です。Log4j では、複数のログ記録階層の概念をサポートしています。ログ記録レベルは、階層ごとに個別に設定されます。次の 2 つのログ記録階層が AWS SDK for Java で使用できます。

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

クラスパスの設定

Log4j JAR および log4j.properties ファイルは、クラスパス上に配置する必要があります。[Apache Ant](#) を使用している場合は、Ant ファイルの path 要素でクラスパスを設定します。以下の例では、SDK に含まれている Amazon S3 [サンプル](#)用の Ant ファイルにある path 要素を示します。

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


Eclipse IDE を使用している場合は、メニューを開き、[Project (プロジェクト)] | [Properties (プロパティ)] | [Java Build Path] に移動して、クラスパスを設定します。

サービス固有のエラーと警告

クライアントライブラリからの重要なメッセージを取得するために、"com.amazonaws" ロガー階層は必ず "WARN" に設定しておくことをお勧めします。例えば、Amazon S3 クライアントがアプリケーションが適切に を閉じておらず InputStream、リソースがリークしている可能性があることを検出した場合、S3 クライアントは警告メッセージを通じてログに報告します。これにより、リクエストやレスポンスの処理でクライアントに問題が発生した場合、メッセージが必ずログに記録されます。

次の log4j.properties ファイルでは、rootLogger が WARN に設定されています。これにより、"com.amazonaws" 階層のすべてのロガーで生成された警告とエラーメッセージが取り込まれます。また、com.amazonaws ロガーを WARN に明示的に設定することもできます。

```
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
```

リクエストおよびレスポンスの概要のログ記録

へのすべてのリクエストは、AWS のサービスが AWS リクエストをどのように処理しているかという問題が発生した場合に便利な一意のリクエスト ID AWS のサービスを生成します。AWS リクエスト IDs は、失敗したサービス呼び出しに対して SDK の例外オブジェクトを介してプログラムでアクセスでき、「com.amazonaws.request」ロガーの DEBUG ログレベルを通じて報告することもできます。

次の log4j.properties ファイルは、リクエスト IDs を含む AWS リクエストとレスポンスの概要を有効にします。

```
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
```

ログ出力の例を次に示します。

```
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:
q963XH63Lcovl5Rr71AP1zlye99rmWwT9DfuQaNznkD, ) 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-00000000000000, AWSAccessKeyId: ACCESSKEYID,
Version: 2009-10-16, SignatureVersion: 2, Timestamp:
2009-12-17T17:53:04.467Z, Signature:
9WcgfPwTobvLVcphybrdN7P713uH0oviYQ4yZ+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
```

詳細なワイヤログ記録

場合によっては、が AWS SDK for Java 送受信する正確なリクエストとレスポンスを確認すると便利です。大きなリクエスト (にアップロードされるファイルなど Amazon S3) を書き出すと、レスポンスによってアプリケーションが大幅に遅くなる可能性があるため、本番システムでこのログ記録を有効にしないでください。この情報に本当にアクセスする必要がある場合は、Apache HttpClient 4 ログガーを使用して一時的に有効にできます。org.apache.http.wire ログガーの DEBUG レベルを有効にすると、すべてのリクエストデータとレスポンスデータに対するログ記録が有効になります。

次の log4j.properties ファイルは、Apache HttpClient 4 でのフルワイヤログ記録をオンにします。アプリケーションのパフォーマンスに大きな影響を与える可能性があるため、一時的にのみオンにする必要があります。

```
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
```

レイテンシーメトリクスのログ記録

トラブルシューティングを行っていて、処理に最も時間がかかっているプロセスを示すメトリクスや、サーバー側またはクライアント側のどちらに大きなレイテンシーが発生しているかを示すメトリクスなどを確認する場合、レイテンシーロガーが役立ちます。このロガーを有効にするには、`com.amazonaws.latency` ロガーを `DEBUG` に設定します。

Note

このロガーは、SDK メトリクスが有効な場合にのみ使用できます。SDK メトリクスパッケージの詳細については、「[Enabling Metrics for the AWS SDK for Java](#)」を参照してください。

```
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
```

ログ出力の例を次に示します。

```
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]
```

クライアント設定

AWS SDK for Java では、デフォルトのクライアント設定を変更できます。これは、次の場合に便利です。

- プロキシを使用したインターネットへの接続
- HTTP トラnsポートの設定 (接続タイムアウトやリクエスト再試行など) の変更
- TCP ソケットバッファのサイズに関するヒントの指定

プロキシ設定

クライアントオブジェクトを構築するときに、オプションの[ClientConfiguration](#)オブジェクトを渡してクライアントの設定をカスタマイズできます。

プロキシサーバーを使用してインターネットに接続する場合は、[ClientConfiguration](#) オブジェクトを使用して、プロキシサーバーの設定 (プロキシホスト、ポート、ユーザー名やパスワード) を指定する必要があります。

HTTP トラnsポートの設定

[ClientConfiguration](#) オブジェクトを使用して、複数の HTTP トラnsポートオプションを設定できます。新しいオプションが時折追加されることがあります。取得または設定できるオプションの完全なリストを確認するには、AWS SDK for Java API リファレンスを参照してください。

Note

設定可能な各値には、定数によって定義されるデフォルト値があります。の定数値のリストについては[ClientConfiguration](#)、AWS SDK for Java 「API リファレンス」の「[定数フィールド値](#)」を参照してください。

最大接続数

. [ClientConfiguration.setMaxConnections](#)メソッドを使用して、オープン HTTP 接続の最大許容数を設定できます。

⚠ Important

同時トランザクション数に最大接続数を設定します。接続の競合およびパフォーマンスの低下を回避します。デフォルトの最大接続値については、AWS SDK for Java「API リファレンス」の「[定数フィールド値](#)」を参照してください。

タイムアウトとエラー処理

HTTP 接続でのタイムアウトとエラー処理に関連するオプションを設定できます。

• 接続タイムアウト

接続タイムアウトは、接続を確立するまでに HTTP 接続が待機する時間 (ミリ秒単位) です。デフォルトは 10,000 ミリ秒です。

この値を自分で設定するには、[ClientConfiguration.setConnectionTimeout](#) メソッドを使用します。

• 接続有効期限 (TTL)

デフォルトで、SDK は HTTP 接続を可能な限り長い時間再利用しようとします。サービスが停止したサーバーに対して接続が確立される失敗の状況では、有限の TTL がアプリケーション復旧に役立つ場合があります。たとえば、15 分の TTL を設定すれば、問題が発生しているサーバーに対して接続が確立されていても、15 分以内に新しいサーバーに接続を再確立できます。

HTTP 接続 TTL を設定するには、[ClientConfiguration.setConnectionTTL](#) メソッドを使用します。

• エラーの最大再試行回数

デフォルトの再試行可能なエラーの最大再試行回数は 3 です。[ClientConfiguration.setMaxErrorRetry](#) メソッドを使用して、別の値を設定できます。

ローカルアドレス

HTTP クライアントがバインドするローカルアドレスを設定するには、[ClientConfiguration.setLocalAddress](#) を使用します。

TCP ソケットバッファのサイズに関するヒント

低レベルの TCP パラメータをチューニングしたい上級ユーザーは、さらに [ClientConfiguration](#) オブジェクトを通じて TCP バッファサイズのヒントを設定できます。ほとんどのユーザーはこれらの値を微調整する必要はありません。これらの値は上級ユーザー向けの値です。

アプリケーションに最適な TCP バッファサイズは、ネットワークやオペレーティングシステムの設定と機能に大きく依存します。たとえば、最新のオペレーティングシステムのほとんどでは、TCP バッファサイズを自動的にチューニングするロジックが組み込まれています。このロジックは TCP 接続のパフォーマンスに大きな影響を与える可能性があります。自動チューニングロジックでバッファサイズを最適化するために、TCP 接続を長い時間開いたままにしておくためです。

バッファサイズを大きくすると (2 MB など)、リモートサーバーでその情報の受信を確認しなくてもオペレーティングシステムでメモリにバッファできるデータが多くなるため、ネットワークのレイテンシーが高い場合に役立ちます。

これはヒントのみです。そのヒントを採用するかどうかはオペレーティングシステムが決定します。このオプションを使用する場合、ユーザーはオペレーティングシステムで設定されている制限とデフォルト値を必ず確認する必要があります。ほとんどのオペレーティングシステムでは、最大 TCP バッファサイズの制限が設定されており、最大 TCP バッファサイズの制限を明示的に引き上げない限り、その制限を超えることはできません。

TCP バッファサイズおよびオペレーティングシステムに固有の TCP 設定を指定するために、次を含む多数のリソースが利用できます。

- [ホストのチューニング](#)

アクセスコントロールポリシー

AWS アクセスコントロールポリシーを使用すると、AWS リソースにきめ細かなアクセスコントロールを指定できます。アクセスコントロールポリシーは、次のフォームを持つステートメントのコレクションで構成されます。

アカウント A は、アクション B をリソース C に対して実行する権限があります (条件 D に該当する場合)。

コードの説明は以下のとおりです。

- はプリンシパル - AWS リソースの 1 つへのアクセスまたは変更をリクエスト AWS アカウントしているです。

- B はアクション - Amazon SQS キューへのメッセージの送信やバケットへのオブジェクトの保存など、AWS リソースへのアクセスまたは変更の方法 Amazon S3。
- C はリソース - Amazon SQS キューや に保存されているオブジェクトなど、プリンシパルがアクセスする AWS エンティティです Amazon S3。
- D は一連の条件 - プリンシパルがリソースにアクセスすることを許可または拒否するタイミングを指定するオプションの制約。さまざまな表記の条件を使用できます。条件によっては、サービスごとに固有な条件もあります。たとえば、日付条件を使用して、特定の時刻の前または後でのみリソースへのアクセスを許可することができます。

Amazon S3 例

次の例は、バケット内のすべてのオブジェクトを読み取るアクセスをすべてのユーザーに許可するが、そのバケットへのオブジェクトのアップロードへのアクセスを AWS アカウント (バケット所有者のアカウントに加えて) 2 つの特定の に制限するポリシーを示しています。

```
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 例

ポリシーの一般的な用途の 1 つは、Amazon SNS トピックからメッセージを受信することを Amazon SQS キューに許可することです。

```
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 の例

一部のサービスには、ポリシーで使用できる追加条件があります。Amazon SNS は、トピックへのサブスクライブリクエストのプロトコル (E メール、HTTP、HTTPS など Amazon SQS) とエンドポイント (E メールアドレス、URL、Amazon SQS ARN など) に基づいて、SNS トピックへのサブスクリプションを許可または拒否するための条件を提供します。

```
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()));
```

DNS 名参照用の JVM TTL を設定する

Java 仮想マシン (JVM) は DNS 名参照をキャッシュします。JVM がホスト名を IP アドレスに解決すると、time-to-live(TTL) と呼ばれる指定された期間、IP アドレスがキャッシュされます。

AWS リソースは DNS 名エントリを使用するため、JVM の TTL 値を 5 秒に設定することをお勧めします。これにより、リソースの IP アドレスが変更されたときに、アプリケーションは DNS に対して再度クエリを実行することで、リソースの新しい IP アドレスを取得し、使用できるようになります。

一部の Java 設定では JVM のデフォルトの TTL が設定されるため、JVM が再起動されるまで、DNS エントリが更新されることはありません。したがって、アプリケーションの実行中に AWS リソース

の IP アドレスが変更された場合、JVM を手動で再起動し、キャッシュされた IP 情報が更新されるまで、そのリソースを使用することはできません。この場合、キャッシュされた IP 情報が定期的に更新されるように JVM の TTL を設定することが極めて重要です。

JVM TTL を設定する方法

JVM の TTL を変更するには、[networkaddress.cache.ttl](#) セキュリティプロパティ値を設定し、Java 8 の場合は \$JAVA_HOME/jre/lib/security/java.security ファイルに、Java 11 以降の場合は \$JAVA_HOME/conf/security/java.security ファイルに networkaddress.cache.ttl プロパティを設定します。

以下は、TTL キャッシュが 5 秒に設定されている java.security ファイルからのスニペットです。

```
#
# This is the "master security properties file".
#
# An alternate java.security properties file may be specified
...
# 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
...
```

環境変数で表される JVM で実行されるすべてのアプリケーションは、\$JAVA_HOME この設定を使用します。

のメトリクスの有効化 AWS SDK for Java

AWS SDK for Java は、以下を測定する [Amazon CloudWatch](#) による可視化とモニタリングのメトリクスを生成できます。

- アクセス時のアプリケーションのパフォーマンス AWS
- で使用した場合の JVMs のパフォーマンス AWS
- ヒープメモリ、スレッド数、開かれたファイル記述子などのランタイム環境の詳細

Java SDK メトリクス生成を有効にする方法

SDK が にメトリクスを送信できるようにするには、次の Maven 依存関係を追加する必要があります 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>
```

* バージョン番号を [Maven Central](#) で入手可能な SDK の最新バージョンに置き換えてください。

AWS SDK for Java メトリクスはデフォルトでは無効になっています。これをローカル開発環境で有効にするには、JVM の起動時に AWS セキュリティ認証情報ファイルをポイントするシステムプロパティを含めます。例:

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

SDK が収集したデータポイントを にアップロードして CloudWatch 後で分析できるように、認証情報ファイルへのパスを指定する必要があります。

Note

Amazon EC2 インスタンスメタデータサービスを使用して Amazon EC2 インスタンス AWS から にアクセスする場合は、認証情報ファイルを指定する必要はありません。この場合、指定する必要があるのは以下だけです。

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

によってキャプチャされたすべてのメトリクス AWS SDK for Java は、名前空間 AWSSDK/Java の下にあり、CloudWatch デフォルトのリージョン (us-east-1) にアップロードされます。リージョンを変更するには、システムプロパティの `cloudwatchRegion` 属性を使用して指定します。例えば、CloudWatch リージョンを us-east-1 に設定するには、以下を使用します。

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

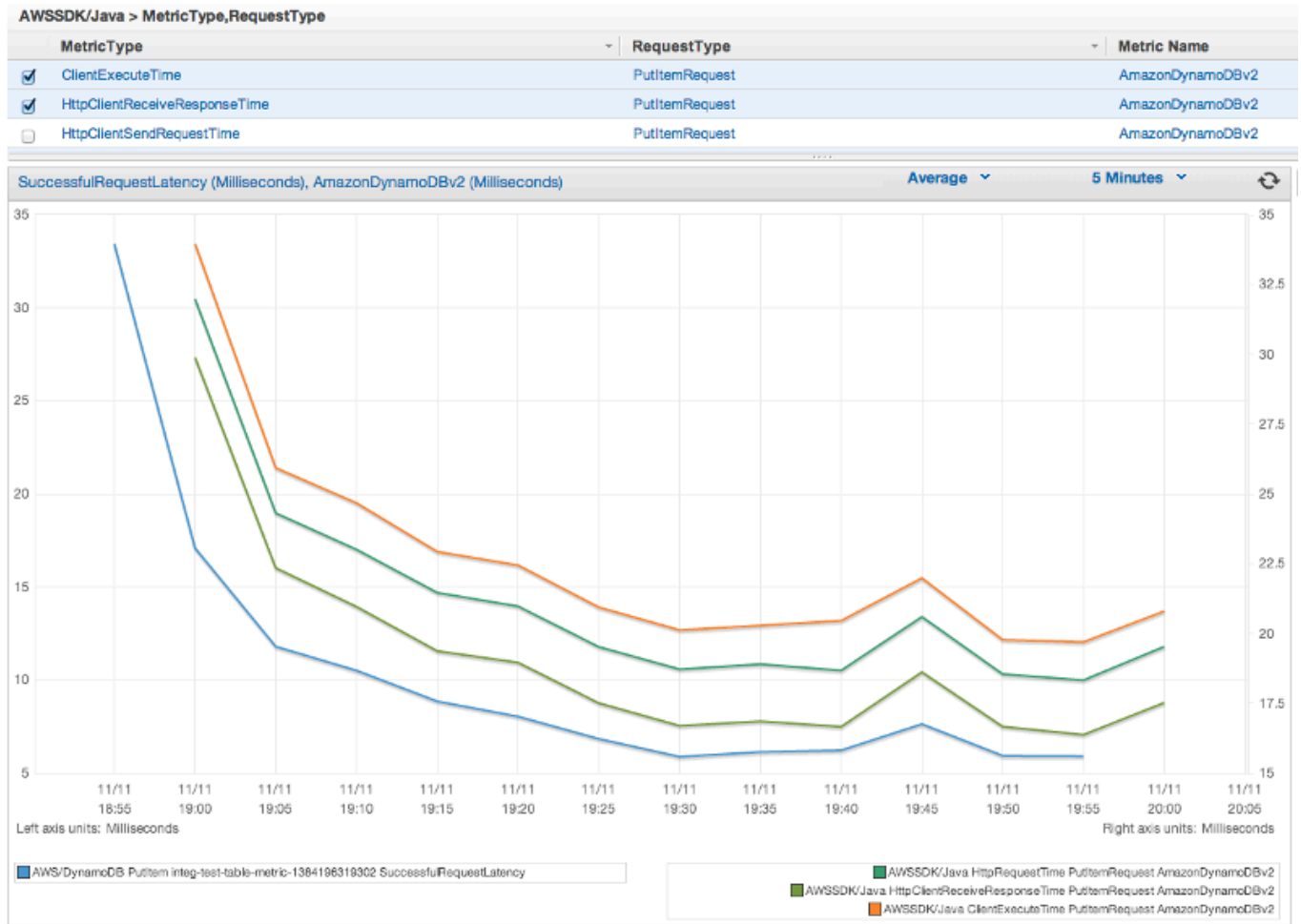
この機能を有効にすると、AWS からへのサービスリクエストがあるたびに AWS SDK for Java、メトリクスデータポイントが生成され、統計サマリー用にキューに入れられ、CloudWatch 約 1 分に 1 回まで非同期的にアップロードされます。メトリクスがアップロードされたら、[AWS Management Console](#) を使用して可視化し、メモリリーク、ファイル記述子リークなどの潜在的な問題にアラームを設定できます。

利用可能なメトリクスのタイプ

デフォルトのメトリクスのセットは 3 つの主要なカテゴリに分かれています。

AWS リクエストメトリクス

- HTTP リクエスト/応答のレイテンシー、リクエスト数、例外および再試行などのエリアを扱います。



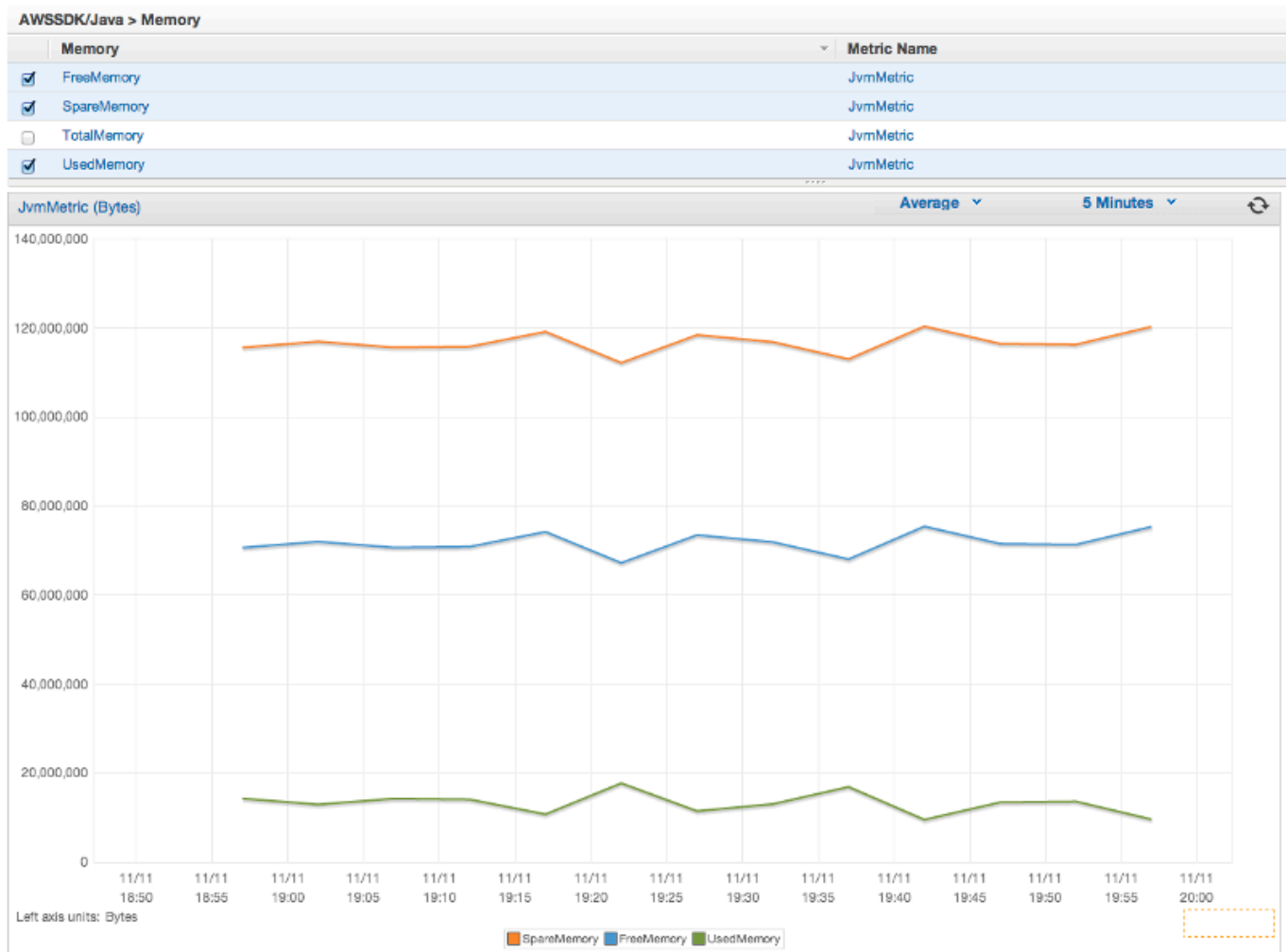
AWS のサービス メトリクス

- S3 のアップロードとダウンロードのスループットやバイト数など、AWS のサービス固有のデータを含めます。



マシンメトリクス

- ヒープメモリ、スレッド数、および開いているファイル記述子を含むランタイム環境を扱います。



マシンメトリクスを除外する場合は、システムプロパティに `excludeMachineMetrics` を追加します。

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

詳細情報

- 事前定義されたコアメトリクスタイプの詳細な一覧については、[amazonaws/metrics package summary](#) を参照してください。
- の使用については、「CloudWatch の使用例 AWS SDK for Java」を参照してください。
[CloudWatch AWS SDK for Java](#)

- パフォーマンスチューニングの詳細については、「[のチューニング AWS SDK for Java による回復性の向上](#)」ブログ記事を参照してください。

AWS SDK for Java CLI コードの例

このセクションでは、AWS SDK for Java v1 を使用して AWS のサービスをプログラミングするためのチュートリアルおよび例を示します。

これらの例などのソースコードについては、AWS ドキュメントの [code examples repository on GitHub](#) を参照してください。

AWS ドキュメントチームに作成を検討してもらいたい新しいコード例を提案するには、新しいリクエストを作成します。チームは、個々の API 呼び出しのみを対象とするシンプルなコードスニペットよりは、より広範なシナリオやユースケースを対象とするコード例を作成しようとしています。手順については、GitHub のコード例リポジトリにある [Contributing guidelines](#) を参照してください。

AWS SDK for Java 2.x

2018 年、AWS は [AWS SDK for Java 2.x](#) をリリースしました。このガイドには、最新の Java SDK の使用方法と、サンプルコードが記載されています。

Note

[開発者が利用できるその他の例と追加のリソースについては、「その他のドキュメントとリソースAWS SDK for Java」](#) を参照してください。

AWS SDK for Java を使用した CloudWatch の例

このセクションでは、[AWS SDK for Java](#) を使用して [CloudWatch](#) をプログラムする例を示します。

Amazon CloudWatch は、Amazon Web Services (AWS) のリソースおよび AWS で実行しているアプリケーションをリアルタイムでモニタリングします。CloudWatch を使用してメトリクスを収集および追跡できます。メトリクスとは、リソースやアプリケーションについて測定できる変数です。CloudWatch アラームは、ユーザーが定義したルールに基づいて、通知を送信したり、モニタリングしているリソースに自動的に変更を加えたりします。

CloudWatch の詳細については、「[Amazon CloudWatch ユーザーガイド](#)」を参照してください。

Note

例には各手法を示すのに必要なコードのみが含まれます。[完全なサンプルコードは GitHub で入手できます](#)。そこから、単一のソースファイルをダウンロードするかリポジトリをローカルにクローン作成して、ビルドし実行するためのすべての例を取得できます。

トピック

- [CloudWatch からのメトリクスの取得](#)
- [カスタムメトリクスデータを発行する](#)
- [CloudWatch アラームの使用](#)
- [CloudWatch でのアラームアクションの使用](#)
- [CloudWatch にイベントを送信する](#)

CloudWatch からのメトリクスの取得

メトリクスの一覧表示

CloudWatch メトリクスを一覧表示するには、[ListMetricsRequest](#) を作成して AmazonCloudWatchClient の `listMetrics` メソッドを呼び出します。ListMetricsRequest を使用して、名前空間、メトリクス名、またはディメンションで返されたメトリクスをフィルタリングできます。

Note

AWS のサービスによって投稿されるメトリクスとディメンションのリストは、Amazon CloudWatch ユーザーガイドの <https---docs-aws-amazon-com-AmazonCloudWatch-latest-monitoring-CW-Support-For-AWS-html> [Amazon CloudWatch のメトリクスおよびディメンションのリファレンス] に記載されています。

インポート

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.ListMetricsRequest;
```

```
import com.amazonaws.services.cloudwatch.model.ListMetricsResult;  
import com.amazonaws.services.cloudwatch.model.Metric;
```

Code

```
final AmazonCloudWatch cw =  
    AmazonCloudWatchClientBuilder.defaultClient();  
  
ListMetricsRequest request = new ListMetricsRequest()  
    .withMetricName(name)  
    .withNamespace(namespace);  
  
boolean done = false;  
  
while(!done) {  
    ListMetricsResult response = cw.listMetrics(request);  
  
    for(Metric metric : response.getMetrics()) {  
        System.out.printf(  
            "Retrieved metric %s", metric.getMetricName());  
    }  
  
    request.setNextToken(response.getNextToken());  
  
    if(response.getNextToken() == null) {  
        done = true;  
    }  
}
```

メトリクスは、`getMetrics` メソッドを呼び出すことによって [ListMetricsResult](#) 内で返されます。結果はページ分割される場合があります。結果の次のバッチを取得するには、`ListMetricsResult` オブジェクトの `getNextToken` メソッドの戻り値を使用して元のリクエストオブジェクトで `setNextToken` を呼び出し、変更したリクエストオブジェクトを `listMetrics` の再呼び出しに渡します。

詳細情報

- Amazon CloudWatch API リファレンスの [ListMetrics](#)

カスタムメトリクスデータを発行する

複数の AWS のサービスが「AWS」で始まる名前空間で[独自のメトリクス](#)を発行します。(「AWS」で始まらない限り) 独自の名前空間を使用してカスタムメトリクスデータを発行することもできます。

カスタムメトリクスデータを発行する

独自のメトリクスデータを発行するには、AmazonCloudWatchClient の putMetricData メソッドを [PutMetricDataRequest](#) で呼び出します。PutMetricDataRequest には、データ用に使用するカスタム名前空間と、[MetricDatum](#) オブジェクト内のデータポイント自体に関する情報が含まれている必要があります。

Note

「AWS」で始まる名前空間を指定することはできません。「AWS」で始まる名前空間は、Amazon Web Services 製品による利用のために予約されています。

インポート

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.cloudwatch.model.Dimension;
import com.amazonaws.services.cloudwatch.model.MetricDatum;
import com.amazonaws.services.cloudwatch.model.PutMetricDataRequest;
import com.amazonaws.services.cloudwatch.model.PutMetricDataResult;
import com.amazonaws.services.cloudwatch.model.StandardUnit;
```

Code

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

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

MetricDatum datum = new MetricDatum()
    .withMetricName("PAGES_VISITED")
    .withUnit(StandardUnit.None)
    .withValue(data_point)
```

```
.withDimensions(dimension);

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

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

詳細情報

- Amazon CloudWatch ユーザーガイドの [Amazon CloudWatch メトリクスの使用](#)。
- Amazon CloudWatch ユーザーガイドの [AWS 名前空間](#)。
- Amazon CloudWatch API リファレンスの [PutMetricData](#)

CloudWatch アラームの使用

アラームの作成

CloudWatch メトリクスに基づいてアラームを作成するには、AmazonCloudWatchClient の putMetricAlarm メソッドをアラーム条件に満たされた [PutMetricAlarmRequest](#) で呼び出します。

インポート

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

Code

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

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

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

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

アラームの一覧表示

作成した CloudWatch アラームを一覧表示するには、AmazonCloudWatchClient の describeAlarms メソッドを、結果のオプションを設定するのに使用できる [DescribeAlarmsRequest](#) で呼び出します。

インポート

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

Code

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

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

while(!done) {
```

```
DescribeAlarmsResult response = cw.describeAlarms(request);

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

request.setNextToken(response.getNextToken());

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

アラームのリストは `getMetricAlarms` を [により返される](#)

`DescribeAlarmsResult` の `describeAlarms` で呼び出すことで取得できます。

結果はページ分割される場合があります。結果の次のバッチを取得するには、`DescribeAlarmsResult` オブジェクトの `getNextToken` メソッドの戻り値を使用して元のリクエストオブジェクトで `setNextToken` を呼び出し、変更したリクエストオブジェクトを `describeAlarms` の再呼び出しに渡します。

Note

また、特定のメトリクスのアラームを取得するには、`AmazonCloudWatchClient` の `describeAlarmsForMetric` メソッドを使用します。使用方法は `describeAlarms` と同様です。

アラームの削除

CloudWatch アラームを削除するには、`AmazonCloudWatchClient` の `deleteAlarms` メソッドを、削除するアラームの名前を 1 つ以上含む [DeleteAlarmsRequest](#) で呼び出します。

インポート

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

Code

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

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

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

詳細情報

- Amazon CloudWatch ユーザーガイドの [Amazon CloudWatch アラームの作成](#)
- Amazon CloudWatch API リファレンスの [PutMetricAlarm](#)
- Amazon CloudWatch API リファレンスの [DescribeAlarms](#)
- Amazon CloudWatch API リファレンスの [DeleteAlarms](#)

CloudWatch でのアラームアクションの使用

CloudWatch アラームアクションを使用して、Amazon EC2 インスタンスを自動的に停止、終了、再起動、または復旧するといったアクションを実行するアラームを作成できます。

Note

[アラームの作成](#)時に `setAlarmActionsPutMetricAlarmRequest` のメソッドを使用することで、アラームアクションをアラームに追加することができます。

アラームアクションの有効化

CloudWatch アラームのアラームアクションを有効化するには、アクションを有効にしたい 1 つ以上のアラームの名前を含む [EnableAlarmActionsRequest](#) で `AmazonCloudWatchClient` の `enableAlarmActions` を呼び出します。

インポート

```
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
```

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

Code

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

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

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

アラームアクションの無効化

CloudWatch アラームのアラームアクションを無効化するには、アクションを無効にしたい 1 つ以上のアラームの名前を含む [DisableAlarmActionsRequest](#) で AmazonCloudWatchClient の `disableAlarmActions` を呼び出します。

インポート

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

Code

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

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

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

詳細情報

- Amazon CloudWatch ユーザーガイドの [インスタンスを停止、終了、再起動、または復旧するアラームを作成する](#)

- Amazon CloudWatch API リファレンスの [PutMetricAlarm](#)
- Amazon CloudWatch API リファレンスの [EnableAlarmActions](#)
- Amazon CloudWatch API リファレンスの [DisableAlarmActions](#)

CloudWatch にイベントを送信する

CloudWatch Events は、AWS リソースの変更を示すシステムイベントのほぼリアルタイムのストリームを、Amazon EC2 インスタンス、Lambda 関数、Kinesis ストリーム、Amazon ECS タスク、Step Functions ステートマシン、Amazon SNS トピック、Amazon SQS キュー、または組み込みターゲットに振り分けます。簡単なルールを使用して、一致したイベントを 1 つ以上のターゲット関数またはストリームに振り分けることができます。

イベントの追加

カスタム CloudWatch イベントを追加するには、各イベントに関する詳細情報を提供している [PutEventsRequestEntry](#) オブジェクトを 1 つ以上含む [PutEventsRequest](#) オブジェクトを使用して `AmazonCloudWatchEventsClient` の `putEvents` メソッドを呼び出します。イベントのソースとタイプ、イベントに関連付けられたリソースなど、エントリの複数のパラメータを指定できます。

Note

`putEvents` への呼び出しごとに最大 10 個のイベントを指定できます。

インポート

```
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEvents;
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEventsClientBuilder;
import com.amazonaws.services.cloudwatchevents.model.PutEventsRequest;
import com.amazonaws.services.cloudwatchevents.model.PutEventsRequestEntry;
import com.amazonaws.services.cloudwatchevents.model.PutEventsResult;
```

Code

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

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

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

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

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

ルールの追加

ルールを作成または更新するには、ルールの名前を含む [PutRuleRequest](#) と、[イベントパターン](#)、ルールと関連付ける IAM ロール、およびルールの実行頻度を説明する [スケジュール式](#)などを含む任意指定のパラメータを `AmazonCloudWatchEventsClient` の `putRule` メソッドを使用して呼び出します。

インポート

```
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEvents;
import com.amazonaws.services.cloudwatchevents.AmazonCloudWatchEventsClientBuilder;
import com.amazonaws.services.cloudwatchevents.model.PutRuleRequest;
import com.amazonaws.services.cloudwatchevents.model.PutRuleResult;
import com.amazonaws.services.cloudwatchevents.model.RuleState;
```

Code

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

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

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

ターゲットの追加

ターゲットは、ルールがトリガーされたときに呼び出されるリソースです。ターゲット例には、Amazon EC2 インスタンス、Lambda 関数、Kinesis ストリーム、Amazon ECS タスク、Step Functions ステートマシン、組み込みターゲットが含まれます。

ルールにターゲットを追加するには、更新するルールを含む [PutTargetsRequest](#) とルールに追加するターゲットのリストを使用して AmazonCloudWatchEventsClient の putTargets メソッドを呼び出します。

インポート

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

Code

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

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

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

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

詳細情報

- Amazon CloudWatch Events ユーザーガイドの [PutEvents を使用したイベントの追加](#)
- Amazon CloudWatch Events ユーザーガイドの [ルールのスケジュール式](#)
- Amazon CloudWatch Events ユーザーガイドの [CloudWatch イベントのイベントタイプ](#)
- Amazon CloudWatch Events ユーザーガイドの [イベントとイベントパターン](#)

- Amazon CloudWatch Events API リファレンスの [PutEvents](#)
- Amazon CloudWatch Events API リファレンスの [PutTargets](#)
- Amazon CloudWatch Events API リファレンスの [PutRule](#)

DynamoDBAWS SDK for Java を使用した例

このセクションでは、[AWS SDK for Java](#) を使用して [DynamoDB](#) をプログラムする例を示します。

Note

例には各手法を示すのに必要なコードのみが含まれます。[完全なサンプルコードは GitHub で入手できます](#)。そこから、単一のソースファイルをダウンロードするリポジトリをローカルにクローン作成して、ビルドし実行するためのすべての例を取得できます。

トピック

- [DynamoDB でのテーブルの操作](#)
- [DynamoDB での項目の操作](#)

DynamoDB でのテーブルの操作

テーブルは、DynamoDB データベースのすべての項目のコンテナです。DynamoDB のデータの追加または削除を行う前に、テーブルを作成する必要があります。

テーブルごとに、以下を定義する必要があります。

- アカウントおよびリージョンに一意であるテーブル名。
- プライマリキー。すべての値は一意でなければならず、テーブル内のどの 2 つの項目も同じプライマリキー値を持つことはできません。

プライマリキーは、単一のパーティション (ハッシュ) キーで構成されるシンプルなプライマリキーにするか、パーティションとソート (範囲) キーで構成される複合プライマリキーにすることができます。

各キーバリューには、[ScalarAttributeType](#) クラスによって列挙されるデータ型が関連付けられています。キー値は、バイナリ (B)、数値 (N)、または文字列 (S) になります。詳細については、Amazon DynamoDB デベロッパーガイドの[命名規則とデータ型](#)を参照してください。

- テーブル用に予約された読み込み/書き込みキャパシティーユニットの数を定義するプロビジョニングされたスループットの値。

Note

[Amazon DynamoDB の料金](#)は、テーブルに設定したプロビジョニングされたスループット値に基づくため、テーブルに必要と予想される分だけの容量を予約します。

テーブルのプロビジョニングされたスループットはいつでも変更できるため、必要に応じてキャパシティーを調整できます。

テーブルを作成する

新しい DynamoDB テーブルを作成するには、[DynamoDB クライアント](#)の `createTable` メソッドを使用します。テーブルのプライマリキーを識別するために使用する、テーブル属性とテーブルスキーマを構築する必要があります。また、最初のプロビジョニングされたスループット値およびテーブル名を指定する必要があります。DynamoDB テーブルの作成時に、キーテーブルの属性のみを定義します。

Note

選択した名前のテーブルが既に存在している場合は、[AmazonServiceException](#) がスローされます。

インポート

```
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;
```

シンプルプライマリキーを使用してテーブルを作成する

このコードでは、シンプルプライマリキー (「Name」) を持つテーブルを作成します。

Code

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

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

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

[GitHub](#) で完全な例をご覧ください。

複合プライマリキーを使用してテーブルを作成する

別の [AttributeDefinition](#) および [KeySchemaElement](#) を [CreateTableRequest](#) に追加します。

Code

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

[GitHub](#) で完全な例をご覧ください。

テーブルの一覧表示

特定のリージョンのテーブルを一覧表示するには、[DynamoDB クライアント](#)の `listTables` メソッドを呼び出します。

Note

指定したテーブルがアカウントやリージョンにない場合は、[ResourceNotFoundException](#) がスローされます。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.ListTablesRequest;
import com.amazonaws.services.dynamodbv2.model.ListTablesResult;
```

Code

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

ListTablesRequest request;

boolean more_tables = true;
String last_name = null;

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

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

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

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

デフォルトでは、1 回の呼び出しで最大 100 個のテーブルが返されます。評価された最後のテーブルを取得するには、返された [ListTablesResult](#) オブジェクトに対して `getLastEvaluatedTableName` を使用します。この値を使用して、前回の一覧表示で返された最後の値以降から、一覧表示を開始できます。

[GitHub](#) で完全な例をご覧ください。

テーブルの説明 (テーブルに関する情報の取得)

[DynamoDB クライアント](#) の `describeTable` メソッドを呼び出します。

Note

指定したテーブルがアカウントやリージョンにない場合は、[ResourceNotFoundException](#) がスローされます。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughputDescription;
import com.amazonaws.services.dynamodbv2.model.TableDescription;
```

Code


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

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

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

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

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

[GitHub](#) で完全な例をご覧ください。

テーブルの変更 (更新)

[DynamoDB クライアント](#)の `updateTable` メソッドを呼び出すことで、テーブルのプロビジョニングされたスループット値を随時変更できます。

Note

指定したテーブルがアカウントやリージョンにない場合は、[ResourceNotFoundException](#) がスローされます。

インポート

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.AmazonServiceException;
```

Code

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

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

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

[GitHub](#) で完全な例をご覧ください。

テーブルの削除

[DynamoDB クライアント](#)の `deleteTable` メソッドを呼び出し、それにテーブルの名前を渡します。

Note

指定したテーブルがアカウントやリージョンにない場合は、[ResourceNotFoundException](#) がスローされます。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
```

Code

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

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

[GitHub](#) で完全な例をご覧ください。

詳細

- Amazon DynamoDB デベロッパーガイドの [テーブルの操作のガイドライン](#)
- Amazon DynamoDB デベロッパーガイドの [DynamoDB のテーブルの操作](#)

DynamoDB での項目の操作

DynamoDB で、項目とは、属性のコレクションで、それぞれに名前と値があります。属性値はスカラー型、セット型、ドキュメント型のいずれかです。詳細については、Amazon DynamoDB デベロッパーガイドの [命名規則とデータ型](#) を参照してください。

テーブルからの項目の取り出し (取得)

AmazonDynamoDB の `getItem` メソッドを呼び出して、指定する項目のテーブル名とプライマリキーバリューを持つ [GetItemRequest](#) オブジェクトを渡します。 [GetItemResult](#) オブジェクトが返されます。

返された `GetItemResult` オブジェクトの `getItem()` メソッドを使用して、項目に関連付けられているキー (String) と値 ([AttributeValue](#)) のペアの [Map](#) を取得できます。

インポート

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

Code

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

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

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

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

try {
    Map<String, AttributeValue> returned_item =
        ddb.getItem(request).getItem();
```

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

[GitHub](#) で完全な例をご覧ください。

テーブルへの新しい項目の追加

項目の属性を表すキーと値のペアの [マップ](#) を作成します。これらには、テーブルのプライマリキーフィールドの値を含める必要があります。プライマリキーで特定される項目がすでにある場合、フィールドはリクエストによって更新されます。

Note

指定したテーブルがアカウントやリージョンにない場合は、[ResourceNotFoundException](#) がスローされます。

インポート

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

Code

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

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

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

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

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

[GitHub](#) で完全な例をご覧ください。

テーブルの既存の項目の更新

テーブルに既に存在する項目の属性を更新するには、AmazonDynamoDB の `updateItem` メソッドを呼び出して、テーブル名、プライマリキーバリュー、更新するフィールドのマップを渡します。

Note

指定したテーブルがアカウントやリージョンにない場合、または渡したプライマリキーで特定される項目がない場合、[ResourceNotFoundException](#) がスローされます。

インポート

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

Code

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

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

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

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

final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();

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

[GitHub](#) で完全な例をご覧ください。

DynamoDBMapper クラスの使用

[AWS SDK for Java](#) には [DynamoDBMapper](#) クラスが用意されているため、クライアント側のクラスを Amazon DynamoDB テーブルにマッピングできます。[DynamoDBMapper](#) クラスを使用するには、注釈を使用して、DynamoDB テーブルの項目とコード内のそれに対応するオブジェクトインスタンス間の関係を定義します (次のコード例を参照)。[DynamoDBMapper](#) クラスでは、テーブルへのアクセス、さまざまな作成、読み取り、更新、削除 (CRUD) オペレーションの実行、およびクエリを行うことができます。

Note

[DynamoDBMapper](#) クラスでは、テーブルを作成、更新、または削除することはできません。

インポート

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

Code

次の Java サンプルコードは、[DynamoDBMapper](#) クラスを使用して Music テーブルにコンテンツを追加する方法を示しています。コンテンツがテーブルに追加されると、Partition キーと Sort キーを使用して項目がロードされることに注意してください。その後、Awards 項目が更新されます。Music テーブルの作成については、Amazon DynamoDB デベロッパーガイドの[テーブルの作成](#)を参照してください。

```
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.printStackTrace();
    }
}

@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;
    }
}
```

[GitHub](#) で完全な例をご覧ください。

詳細

- Amazon DynamoDB デベロッパーガイドの [項目の操作のガイドライン](#)
- Amazon DynamoDB デベロッパーガイドの [DynamoDB の項目の操作](#)

Amazon EC2AWS SDK for Java を使用した例

このセクションでは、AWS SDK for Java で [Amazon EC2](#) をプログラムする例を示します。

トピック

- [チュートリアル: EC2 インスタンスの開始](#)
- [Amazon EC2 での IAM ロールを使用した AWS リソースへのアクセスの許可](#)
- [チュートリアル: Amazon EC2 スポットインスタンス](#)
- [チュートリアル: Amazon EC2 スポットリクエストの高度な管理](#)
- [Amazon EC2 インスタンスの管理](#)
- [での Elastic IP アドレスの使用 Amazon EC2](#)
- [リージョンとアベイラビリティゾーンを使用する](#)
- [Amazon EC2 キーペアでの作業](#)
- [Amazon EC2 でセキュリティグループを操作する](#)

チュートリアル: EC2 インスタンスの開始

このチュートリアルでは、AWS SDK for Java を使用して EC2 インスタンスを開始する方法を示します。

トピック

- [前提条件](#)
- [Amazon EC2 セキュリティグループを作成する](#)
- [キーペアの作成](#)
- [Amazon EC2 インスタンスを実行する](#)

前提条件

開始する前に、AWS アカウント を作成したこと、および AWS 認証情報を設定したことを確認します。詳細については、「[の使用開始](#)」を参照してください。

Amazon EC2 セキュリティグループを作成する

EC2-Classic は廃止されます

Warning

2022 年 8 月 15 日に、EC2-Classic の提供を終了します。EC2-Classic は、VPC への移行をお勧めします。詳細については、[Amazon EC2](#) EC2-Classic から VPC への移行[Amazon EC2](#)」を参照してください。ブログ記事「[EC2-Classic-Classic Networking is Retiring – Here's How to Prepare](#)」も参照してください。

セキュリティグループを作成します。セキュリティグループは、1 つ以上の EC2 インスタンスのネットワークトラフィックを制御する仮想ファイアウォールとして機能します。デフォルトでは、インバウンドトラフィックを許可しないセキュリティグループにインスタンスを Amazon EC2 関連付けます。EC2 インスタンスが特定のトラフィックを受け付けるようにするセキュリティグループを作成できます。たとえば、Linux インスタンスに接続する必要がある場合は、SSH トラフィックを許可するようにセキュリティグループを設定する必要があります。Amazon EC2 コンソールまたは を使用してセキュリティグループを作成できます AWS SDK for Java。

EC2-Classic または EC2-VPC で使用するセキュリティグループを作成します。EC2-Classic および EC2-VPC の詳細については、Linux インスタンス用 Amazon EC2 ユーザーガイドの「[サポートされているプラットフォーム](#)」を参照してください。

Amazon EC2 コンソールを使用したセキュリティグループの作成の詳細については、Linux インスタンス用 Amazon EC2 ユーザーガイドの[Amazon EC2 「セキュリティグループ」](#)を参照してください。

1. [CreateSecurityGroupRequest](#) インスタンスを作成して初期化します。[withGroupName](#) メソッドを使用してセキュリティグループ名を設定し、[withDescription](#) メソッドを使用してセキュリティグループの説明を次のように設定します。

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

セキュリティグループ名は、Amazon EC2 クライアントを初期化する AWS リージョン内で一意である必要があります。セキュリティグループの名前と説明には、US-ASCII 文字を使用する必要があります。

2. リクエストオブジェクトをパラメータとして [createSecurityGroup](#) メソッドに渡します。メソッドは、次のように [CreateSecurityGroupResult](#) オブジェクトを返します。

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

既存のセキュリティグループと同じ名前で作成しようとする場合、`createSecurityGroup` によって例外がスローされます。

デフォルトでは、新しいセキュリティグループは Amazon EC2 インスタンスへのインバウンドトラフィックを許可しません。インバウンドトラフィックを許可するには、セキュリティグループの進入を明示的に承認する必要があります。個々の IP アドレス、IP アドレスの範囲、特定のプロトコル、および TCP/UDP ポートに対して進入を承認することができます。

1. [IpPermission](#) インスタンスを作成して初期化します。[withIpv4Ranges](#) メソッドを使用して、進入を許可する IP アドレスの範囲を設定し、[withIpProtocol](#) メソッドを使用して IP プロトコルを設定します。[withFromPort](#) および [withToPort](#) メソッドを使用して、次のように進入を承認するポートの範囲を指定します。

```
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);
```

進入が許可されるには、IpPermission オブジェクトで指定したすべての条件を満たしている必要があります。

CIDR 表記を使用して IP アドレスを指定します。プロトコルを TCP/UDP として指定した場合は、送信元ポートと送信先ポートを指定する必要があります。ポートを承認できるのは、TCP または UDP を指定した場合のみです。

2. [AuthorizeSecurityGroupIngressRequest](#) インスタンスを作成して初期化します。withGroupName メソッドを使用してセキュリティグループ名を指定し、次のように、前に初期化した IpPermission オブジェクトを [withIpPermissions](#) メソッドに渡します。

```
AuthorizeSecurityGroupIngressRequest authorizeSecurityGroupIngressRequest =
    new AuthorizeSecurityGroupIngressRequest();

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

3. 次のように、リクエストオブジェクトを [authorizeSecurityGroupIngress](#) メソッドに渡します。

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

進入がすでに承認されている IP アドレスを使用して authorizeSecurityGroupIngress を呼び出すと、メソッドによって例外がスローされます。AuthorizeSecurityGroupIngress を呼び出す前に、新しい IpPermission オブジェクトを作成し、初期化して、異なる IP、ポート、プロトコルに対して進入を承認します。

[authorizeSecurityGroupIngress](#) メソッドまたは [authorizeSecurityGroupEgress](#) メソッドを呼び出すたびに、ルールがセキュリティグループに追加されます。

キーペアの作成

EC2 インスタンスを起動するときはキーペアを指定し、インスタンスに接続するときはキーペアのプライベートキーを指定する必要があります。キーペアを作成することも、他のインスタンスの起動時に使用した既存のキーペアを使用することもできます。詳細については、「Linux インスタンス用 Amazon EC2 ユーザーガイド」の「[Amazon EC2 Key Pairs](#)」を参照してください。

1. [CreateKeyPairRequest](#) インスタンスを作成し、初期化します。[withKeyName](#) メソッドを使用して、キーペアの名前を設定します。次に例を示します。

```
CreateKeyPairRequest createKeyPairRequest = new CreateKeyPairRequest();  
createKeyPairRequest.withKeyName(keyName);
```

Important

キーペア名は一意である必要があります。既存のキーペアと同じキー名でキーペアを作成しようとすると、例外が発生します。

2. [createKeyPair](#) メソッドにリクエストオブジェクトを渡します。このメソッドは、[CreateKeyPairResult](#) インスタンスを返します。次に例を示します。

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

3. 結果のオブジェクトの [getKeyPair](#) メソッドを呼び出して、[KeyPair](#) オブジェクトを取得します。KeyPair オブジェクトの [getKeyMaterial](#) メソッドを呼び出して、暗号化されていない PEM エンコード形式のプライベートキーを取得します。次に例を示します。

```
KeyPair keyPair = new KeyPair();  
  
keyPair = createKeyPairResult.getKeyPair();  
  
String privateKey = keyPair.getKeyMaterial();
```

Amazon EC2 インスタンスを実行する

同じ Amazon Machine Image (AMI) から全く同じに設定された 1 つ以上の EC2 インスタンスを起動するには、以下の手順を使用します。EC2 インスタンスを作成した後は、ステータスを確認できます。EC2 インスタンスが実行した後は、それに接続できます。

1. [RunInstancesRequest](#) インスタンスを作成し、初期化します。指定した AMI、キーペア、およびセキュリティグループが、クライアントオブジェクトを作成したときに指定したリージョンに存在することを確認します。

```
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 の ID。Amazon から提供されるパブリック AMI を見つける方法や独自の AMI を作成する方法については、「[Amazon マシンイメージ \(AMI\)](#)」を参照してください。

[withInstanceType](#)

- 指定した AMI と互換性のあるインスタンスタイプ。詳細については、Amazon EC2 Linux インスタンス用ユーザーガイドの[インスタンスタイプ](#)を参照してください。

[withMinCount](#)

- 起動する EC2 インスタンスの最小数。ターゲットアベイラビリティーゾーンで Amazon EC2 が起動できるインスタンスより多い場合、Amazon EC2 はインスタンスを起動しません。

[withMaxCount](#)

- 起動する EC2 インスタンスの最大数。ターゲットアベイラビリティーゾーンで Amazon EC2 が起動できるインスタンスより多い場合、Amazon EC2 は MinCount より多くて可能な最大数のインスタンスを起動します。1 から、インスタンスタイプに対して許可されているインスタンスの最大数の間で起動できます。詳細については、Amazon EC2 の一般的なよくある質問の Amazon EC2 でいくつかインスタンスを実行できますか? を参照してください。

withKeyName

- EC2 キーペアの名前。キーペアを指定せずにインスタンスを起動すると、接続できません。詳細については、「[キーペアの作成](#)」を参照してください。

withSecurityGroups

- 1 つまたは複数のセキュリティグループ。詳細については、[Amazon EC2 セキュリティグループの作成](#)を参照してください。
2. リクエストオブジェクトを [runInstances](#) メソッドに渡してインスタンスを起動します。このメソッドは、[RunInstancesResult](#) オブジェクトを返します。次に例を示します。

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

インスタンスの実行後は、キーペアを使用してインスタンスにリモート接続することができます。詳細については、Amazon EC2 Linux インスタンス用ユーザーガイドの [Linux インスタンスへの接続](#)を参照してください。

Amazon EC2 での IAM ロールを使用した AWS リソースへのアクセスの許可

Amazon Web Services (AWS) へのリクエストはすべて、AWS が発行した認証情報を使用して暗号で署名される必要があります。IAM ロールを使用することで、Amazon EC2 インスタンスから AWS リソースへのセキュアなアクセスを簡単に付与できます。

このトピックでは、Amazon EC2 で実行されている Java SDK アプリケーションで IAM ロールを使用する方法について説明します。IAM インスタンスの詳細については、「Linux インスタンス用 Amazon EC2 ユーザーガイドの [IAM Roles for Amazon EC2](#)」を参照してください。

デフォルトプロバイダチェーンと EC2 インスタンスプロファイル

アプリケーションでデフォルトのコンストラクタを使用して AWS クライアントを作成する場合、そのクライアントはデフォルトの認証情報プロバイダチェーンを使用して次の順序で認証情報を検索します。

1. Java のシステムプロパティ: `aws.accessKeyId` と `aws.secretKey`。
2. システム環境変数: `AWS_ACCESS_KEY_ID` と `AWS_SECRET_ACCESS_KEY`。
3. デフォルトの認証情報ファイル (このファイルの場所はプラットフォームによって異なります)。

4. `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` 環境変数が設定されていて、セキュリティマネージャーが変数にアクセスするアクセス権限を持っている場合、Amazon EC2 コンテナサービスを介して配信される認証情報。
5. インスタンスプロファイル認証情報。EC2 インスタンスの IAM ロールに関連付けられたインスタンスメタデータ内にあります。
6. 環境またはコンテナからのウェブアイデンティティトークンの認証情報。

デフォルトのプロバイダーチェーンのインスタンスプロファイル認証情報ステップは、アプリケーションを Amazon EC2 インスタンスで実行する場合にのみ使用できます。Amazon EC2 インスタンスを使用する場合にもっとも使い方が簡単でセキュリティに優れた方法です。また、[InstanceProfileCredentialsProvider](#) インスタンスを直接クライアントコンストラクタに渡して、デフォルトプロバイダーチェーン全体を経ることなく、インスタンスプロファイル認証情報を取得することもできます。

例:

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

この方法を使用する場合、SDK はインスタンスプロファイル内の Amazon EC2 インスタンスに関連付けられている IAM ロールに関連付けられたのと同じ許可を持つ一時的な AWS 認証情報を取得します。これらの認証情報は一時的なもので、最終的には失効しますが、`InstanceProfileCredentialsProvider` によって定期的に更新されるため、取得済みの認証情報で引き続き AWS にアクセスできます。

Important

認証情報の自動更新は、デフォルトのプロバイダーチェーンの一部として独自の `InstanceProfileCredentialsProvider` を作成するデフォルトのクライアントコンストラクターを使用する場合、または `InstanceProfileCredentialsProvider` インスタンスをクライアントコンストラクターに直接渡す場合にのみ行われます。その他の手段でインスタンスプロファイル認証情報を取得または渡す場合は、お客様自身で期限切れ認証情報を確認し更新する必要があります。

クライアントコンストラクタが認証情報プロバイダチェーンを使用して証明書を見つけられない場合、[AmazonClientException](#) がスローされます。

ウォークスルー: EC2 インスタンスでの IAM ロールの使用

以下のウォークスルーでは、アクセス権を管理するために IAM ロールを使用して Amazon S3 からオブジェクトを取得する方法を示します。

IAM ロールを作成します。

Amazon S3 に読み取り専用アクセスを付与する IAM ロールを作成します。

1. [\[IAM コンソール\]](#) を開きます。
2. ナビゲーションペインで [\[Roles\]](#)、[\[Create New Role\]](#) の順に選択します。
3. ロールの名前を入力し、[\[Next Step\]](#) (次のステップ) を選択します。この名前は Amazon EC2 インスタンスを起動するときに必要なため、覚えておいてください。
4. [\[ロールタイプの選択\]](#) ページの [\[AWS のサービス ロール\]](#) で、[\[Amazon EC2\]](#) を選択します。
5. [\[許可を設定\]](#) ページの [\[ポリシーテンプレートの選択\]](#) で、[\[Amazon S3 読み取り専用アクセス\]](#) を選択して、[\[次のステップ\]](#) を選択します。
6. [\[Review\]](#) ページで、[\[Create Role\]](#) を選択します。

EC2 インスタンスを起動して IAM ロールを指定する

Amazon EC2 コンソールまたは AWS SDK for Java を使用して、IAM ロールで Amazon EC2 インスタンスを起動できます。

- コンソールを使用して Amazon EC2 インスタンスを起動するには、Amazon EC2 Linux インスタンス用ユーザーガイドの [Amazon EC2 Linux インスタンスの開始方法](#) の指示に従います。

[\[Review Instance Launch \(インスタンス作成の確認\)\]](#) ページを開いたら、[\[Edit instance details \(インスタンスの詳細の編集\)\]](#) を選択します。[\[IAM role\] \(IAM ロール\)](#) で、前に作成した IAM ロールを選択します。指示にしたがって手順を完了します。

Note

そのインスタンスに接続するには、セキュリティグループとキーペアを作成するか、または既存のものを使用する必要があります。

- AWS SDK for Java を使用して IAM ロールを使用する Amazon EC2 インスタンスを起動するには、[Amazon EC2 インスタンスの実行](#)を参照してください。

アプリケーションを作成する

EC2 インスタンスで実行するサンプルアプリケーションを作成してみましょう。まず、チュートリアルファイルを保存するために使用できるディレクトリを作成します (例: GetS3ObjectApp)。

次に、新しく作成したディレクトリに AWS SDK for Java ライブラリをコピーします。AWS SDK for Java を ~/Downloads ディレクトリにダウンロードした場合は、次のコマンドを使用してそれらのライブラリをコピーできます。

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

新規ファイルを開き、GetS3Object.java と名付け、次のコードを追加します。

```
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();
}
}
```

新規ファイルを開き、build.xml と名付け、次の行を追加します。

```
<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>
```

変更を加えたプログラムを構築し、実行します。プログラムには認証情報は保存されていません。このため、AWS 認証情報が既に指定されていない場合、コードによって `AmazonServiceException` がスローされます。例:

```
$ ant
Buildfile: /path/to/my/GetS30bjectApp/build.xml

build:
  [javac] Compiling 1 source file to /path/to/my/GetS30bjectApp

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

BUILD SUCCESSFUL
```

EC2 インスタンスへのコンパイルしたプログラムの転送

Secure Copy (Amazon EC2) を使用して、ライブラリとともに AWS SDK for Java インスタンスにプログラムを転送します。一連のコマンドは、次のようになります。

```
scp -p -i {my-key-pair}.pem GetS30bject.class ec2-user@{public_dns}:GetS30bject.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

使用した Linux ディストリビューションに応じて、ユーザー名は「ec2-user」、「root」、「ubuntu」のいずれかになります。インスタンスのパブリック DNS 名を取得するには、[EC2 コンソール](#)を開き、[Description] (説明) タブで [Public DNS] (パブリック DNS) 値を探します (例: ec2-198-51-100-1.compute-1.amazonaws.com)。

上記のコマンドでは:

- `GetS30bject.class` はコンパイルされたプログラム、
- `build.xml` はプログラムを構築して実行するために使用する ant ファイル、
- `lib` ディレクトリと `third-party` ディレクトリは、AWS SDK for Java の対応するライブラリフォルダです。

- `-r` スイッチは、`scp` が AWS SDK for Java ディストリビューションの `library` ディレクトリと `third-party` ディレクトリのすべてのコンテンツについて、再帰的なコピーを実行することを示しています。
- `-p` スイッチは、ソースファイルがコピー先にコピーされるときに、`scp` ではソースファイルのアクセス許可が維持されることを示しています。

Note

この `-p` スイッチは、Linux、macOS、または Unix でのみ機能します。Windows からファイルをコピーする場合、必要に応じて次のコマンドを使用し、インスタンスでのファイルへのアクセス許可を修正します。

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

EC2 インスタンスでサンプルプログラムを実行する

プログラムを実行するには、Amazon EC2 インスタンスに接続します。詳細については、Amazon EC2 Linux インスタンス用ユーザーガイドの [Linux インスタンスへの接続](#) を参照してください。

ant がインスタンスで使用できない場合は、次のコマンドを使用してインストールします。

```
sudo yum install ant
```

次に、`ant` を使用して次のようにプログラムを実行します。

```
ant run
```

プログラムでは、Amazon S3 オブジェクトのコンテンツがコマンドウィンドウに表示されます。

チュートリアル: Amazon EC2 スポットインスタンス

概要

スポットインスタンスとは、Amazon Elastic Compute Cloud (Amazon EC2) の未使用キャパシティに対してお客様から価格を提示していただき、入札価格がその時点のスポット料金を上回っている限り、お客様がそのインスタンスを取得し、実行できるというシステムです。Amazon EC2 のスポット料金は、需要と供給に基づいて定期的に変動しますが、お客様の入札価格がその価格以上ならば、

空いているスポットインスタンスにアクセスできます。オンデマンドインスタンスやリザーブドインスタンスと同様に、スポットインスタンスは計算キャパシティを増やしたいときの選択肢の 1 つとなります。

スポットインスタンスを利用すると、Amazon EC2 によるバッチ処理、科学研究、画像処理、動画エンコーディング、データと Web のクローリング、財務分析、テストなどのコストの大幅削減を期待できます。加えて、スポットインスタンスは、大量の追加計算キャパシティが必要であるけれどもその緊急性が低いという場合にも適しています。

スポットインスタンスを使用するには、スポットインスタンスリクエストを提出し、このときにインスタンス時間あたりいくらまで支払えるかを指定します。これが入札価格です。入札価格がその時点のスポット価格を超えている場合は、リクエストが受理されてインスタンスを実行できるようになります。このインスタンスの実行は、お客様がインスタンスを終了した時点と、スポット価格が入札価格を上回った時点のいずれか早い方までとなります。

次のことに注意することが重要です。

- 時間当たりの支払い金額が入札価格を下回ることもよくあります。Amazon EC2 のスポット料金は、提出されるリクエストや空きインスタンスの変動に応じて、定期的に変更されます。お客様それぞれの入札価格の方が上かどうにかかわらず、どのお客様もその期間の同一のスポット料金をお支払いいただきます。したがって、お客様が支払う金額は入札価格を下回ることもありますが、入札価格を超えることはありません。
- スポットインスタンスを実行しているときに、お客様の入札価格がその時点のスポット料金以上ではなくなった場合は、そのインスタンスは終了となります。つまり、この変動性の高いキャパシティを活用できる、柔軟性の高いワークロードとアプリケーションに限ってスポットインスタンスを利用することをお勧めします。

スポットインスタンスは稼働中、他の Amazon EC2 インスタンスとまったく同じように動作します。そして他の Amazon EC2 インスタンスと同様に、スポットインスタンスは必要がなくなった場合に終了することができます。お客様がインスタンスを終了した場合は、使用時間の端数分についても料金をいただきます (オンデマンドやリザーブドのインスタンスと同様です)。ただし、スポット価格がお客様の入札価格を超えたためにインスタンスが Amazon EC2 によって終了させられた場合は、使用時間の端数分の料金は発生しません。

このチュートリアルでは、AWS SDK for Java を使用して以下を行う方法について説明します。

- スポットリクエストを提出する
- スポットリクエストが受理されたかどうかを判断する

- スポットリクエストをキャンセルする
- 関連するインスタンスを終了させる

前提条件

このチュートリアルを使用するには、AWS SDK for Java がインストールされており、基本インストール前提条件を満たしている必要があります。詳細については、「[Set up the AWS SDK for Java](#)」を参照してください。

ステップ 1: 認証情報のセットアップ

このサンプルコードの使用を開始するには、AWS 認証情報を設定する必要があります。その方法については、[開発用の AWS 認証情報とリージョンのセットアップ](#)を参照してください。

Note

IAM ユーザーの認証情報を使用してこれらの値を指定することをお勧めします。詳細については、[AWS にサインアップし、IAM ユーザーを作成する](#)を参照してください。

これで設定が完了したので、例に示すコードを使用できるようになります。

ステップ 2: セキュリティグループのセットアップ

セキュリティグループとは、ファイアウォールとしての役割を果たすものであり、インスタンスのグループに対してどのトラフィックの送受信を許可するかを制御します。デフォルトでは、インスタンスの起動時にセキュリティグループは何も設定されていません。つまり、着信 IP トラフィックは、どの TCP ポートであってもすべて拒否されます。したがって、ここでは、スポットリクエストを提出する前に、必要なネットワークトラフィックを許可するセキュリティグループをセットアップすることにします。このチュートリアルの目的に合わせて、ここでは新しいセキュリティグループを「GettingStarted」という名前で作成します。このグループでは、自分のアプリケーションを実行する IP アドレスからの Secure Shell (SSH) トラフィックを許可します。新しいセキュリティグループをセットアップするには、次に示すコードサンプルをインクルードするか実行する必要があります。このコードは、セキュリティグループをプログラムからセットアップするためのものです。

AmazonEC2 クライアントオブジェクトを作成した後で、CreateSecurityGroupRequest オブジェクトを作成し、「GettingStarted」という名前と、セキュリティグループの説明を指定します。その後で、ec2.createSecurityGroup API を呼び出してグループを作成します。

このグループにアクセスできるようにするために、`ipPermission` オブジェクトを作成します。IP アドレス範囲は、ローカルコンピュータのサブネット (CIDR 表現) で設定します。IP アドレスの「/10」というサフィックスが、指定した IP アドレスのサブネットを示します。また、`ipPermission` オブジェクトを設定して TCP プロトコルとポート 22 (SSH) を指定します。最後のステップは、`ec2.authorizeSecurityGroupIngress` を呼び出すことです。このときに、作成したセキュリティグループの名前と `ipPermission` オブジェクトを指定します。

```
// 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());
}
```

このアプリケーションを実行して新しいセキュリティグループを作成する必要があるのは 1 回のみです。

また、AWS Toolkit for Eclipse を使用してセキュリティグループを作成することもできます。詳細については、「[Managing Security Groups from AWS Cost Explorer](#)」を参照してください。

ステップ 3: スポットリクエストを提出する

スポットリクエストを提出するには、最初に、使用するインスタンスタイプ、Amazon マシンイメージ (AMI)、最高入札価格を決定する必要があります。前のステップで設定したセキュリティグループも指定する必要があります。これは、必要に応じてインスタンスにログインできるようにするためです。

選択できるインスタンスタイプにはさまざまなものがあります。すべての一覧については、Amazon EC2 インスタンスタイプのページを参照してください。このチュートリアルでは、最も低価格のインスタンスタイプである t1.micro を使用します。次に、使用する AMI のタイプを決定します。ここでは、ami-a9d09ed1 を使用します。これは、このチュートリアルの執筆時点で最新の Amazon Linux AMI です。最新の AMI は時間の経過と共に変化する可能性があります。次のステップを実行することで最新バージョンの AMI であることを常に判断できます。

1. [Amazon EC2 コンソール](#)を開きます。
2. [Launch Instance (インスタンスの起動)] ボタンを選択します。
3. 最初のウィンドウには、利用可能な AMI が表示されます。各 AMI のタイトルの横には、AMI の ID が表示されます。DescribeImages API を使用することもできますが、このコマンドの利用方法は、このチュートリアルでは取り上げません。

スポットインスタンス入札のアプローチは多数あります。さまざまなアプローチの概要については、[スポットインスタンスの入札](#)の動画をご覧ください。ただし、ここでは初めての方のために、3つの一般的な戦略について説明します。その3つとは、「コストがオンデマンド価格より低くなるように入札する」、「計算処理の結果の価値に基づいて入札する」、「できるだけ早くコンピューティング性能を獲得できるように入札する」です。

- コストをオンデマンドよりも低くする 実行完了までに何時間も、あるいは何日間もかかるバッチ処理ジョブがあるとします。ただし、いつ開始していつ完了するかについては、特に決められていないものとします。このジョブを完了するためのコストを、オンデマンドインスタンスを使用する場合よりも低くできるかどうかを考えます。インスタンスタイプのスポット価格の履歴を、AWS Management Console または Amazon EC2 API を使用して調べます。詳細については、「[スポット価格の履歴の表示](#)」を参照してください。使用したいインスタンスタイプの、特定の Availability Zone での価格履歴を分析した後は、入札のアプローチとして次の2つも考えられます。
- スポット料金の範囲の上限（ただしオンデマンド価格よりは下）で入札します。このようにすれば、この1回限りのスポットリクエストが受理される可能性が高くなり、ジョブが完了するまで連続して実行できるからです。
- または、スポットインスタンスに対して支払う金額をオンデマンドインスタンス料金の % で指定し、1つの永続リクエストで次々とインスタンスを起動することを計画できます。指定された料金を超えた場合、スポットインスタンスは終了します。（この作業を自動化する方法については、このチュートリアルで後ほど説明します。）
- 結果の価値以上は支払わない データ処理ジョブを実行するとします。このジョブの結果の価値は判明しており、計算コストに換算してどれくらいになるかもわかっています。使用するインスタンスタイプのスポット料金履歴の分析が完了した後で、入札価格を選択します。コンピューティング時間のコストがこのジョブの結果の価値を上回ることがないように、価格を決定します。永続リクエストを作成し、スポット料金が入札価格以下となったときに断続的に実行するよう設定します。
- 計算キャパシティをすぐに獲得する 追加のキャパシティが突然、短期間だけ必要になることがあり、オンデマンドインスタンスではそのキャパシティを獲得できないとします。使用するインスタンスタイプのスポット料金履歴の分析が完了した後で、履歴の価格の最大値を超える価格で入札します。このようにすれば、リクエストがすぐに受理される可能性が高まり、完了するまで連続して計算できるようになります。

入札価格を選択すると、スポットインスタンスをリクエストできる状態になります。ここでは、このチュートリアルの目的に合わせて、オンデマンド価格 (0.03 USD) で入札します。これは、受理される可能性を最大にするためです。利用できるインスタンスのタイプと、インスタンスのオンデマンド料金を調べるには、Amazon EC2 の料金のページを参照してください。スポッ

トインスタンスの実行中は、インスタンスが実行された期間で有効なスポット料金を支払い続けます。スポットインスタンス料金は Amazon EC2 で設定され、長期の需要と供給に応じて、スポットインスタンス容量に合わせて緩やかに調整されます。また、スポットインスタンスに対して支払う金額をオンデマンドインスタンス料金の % で指定することもできます。スポットインスタンスをリクエストするには、先ほど選択したパラメータを使用してリクエストを構築するだけです。初めに、`RequestSpotInstanceRequest` オブジェクトを作成します。このリクエストオブジェクトには、起動したいインスタンスの数と入札価格が必要です。さらに、リクエストの `LaunchSpecification` を設定する必要があります。この内容は、インスタンスタイプ、AMI ID、および使用するセキュリティグループです。リクエストの内容が入力されたら、`requestSpotInstances` オブジェクトの `AmazonEC2Client` メソッドを呼び出します。次の例で、スポットインスタンスをリクエストする方法を示します。

```
// 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);
```

このコードを実行すると、新しいスポットインスタンスリクエストが発行されます。他にも、スポットリクエストの設定に使用できるオプションがあります。詳細については、[チュートリアル: Amazon EC2 スポットリクエストの高度な管理](#)または AWS SDK for Java API リファレンスの [RequestSpotInstances](#) クラスにアクセスしてください。

Note

スポットインスタンスが実際に起動されるとお客様への課金が発生するので、料金を抑えるために、リクエストを作成した場合はキャンセルし、インスタンスを起動した場合は終了してください。

ステップ 4: スポットリクエストの状態を特定する

次に、最後のステップに進む前にスポットリクエストの状態が「アクティブ」になるのを待つようにするコードを作成する必要があります。スポットリクエストの状態を特定するには、[describeSpotInstanceRequests](#) メソッドをポーリングすることによって、モニタリング対象のスポットリクエスト ID の状態を調べます。

ステップ 2 で作成したリクエスト ID は、`requestSpotInstances` リクエストへのレスポンスに埋め込まれています。次に示すコード例では、リクエスト ID を `requestSpotInstances` レスポンスから取り出して `ArrayList` への入力に使用方法を示します。

```
// 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());
}
```

リクエスト ID をモニタリングするには、`describeSpotInstanceRequests` メソッドを呼び出してリクエストの状態を特定します。その後で、リクエストが「オープン」状態でなくなるまでループを繰り返します。状態が、例えば「アクティブ」ではなく、「オープン」以外かどうかをモニタリングするのは、リクエストが直接「クローズ済み」に遷移することもあるからです (リクエストの引数に問題がある場合)。次に示すコード例では、このことを実現する具体的な方法を示します。

```
// 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);
```

このコードを実行すると、スポットインスタンスリクエストは完了するか、エラーありで失敗し、そのエラーが画面に出力されます。どちらの場合も、次のステップに進んで、アクティブなリクエストがある場合はクリーンアップし、実行中のインスタンスがある場合は終了させてください。

ステップ 5: スポットリクエストとインスタンスをクリーンアップする

最後に、リクエストとインスタンスをクリーンアップする必要があります。未完了リクエストのキャンセルと、インスタンスの削除の両方を行うことが重要です。リクエストをキャンセルするだけではインスタンスは終了しないので、引き続きお客様への課金が発生することになります。インスタンスを削除すると、スポットリクエストがキャンセルされることもありますが、場合によっては (持続的入札を使用した場合など)、インスタンスを終了しただけでは、リクエストが再度受理されるのを停止できないことがあります。したがって、アクティブな入札のキャンセルと実行中インスタンスの削除の両方を行うことをお勧めします。

次のコードでは、リクエストをキャンセルする方法を示します。

```
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());
}
```

稼働中のインスタンスを終了させるには、そのインスタンスを起動したリクエストに関連付けられているインスタンス ID が必要です。次のコード例は、前に示したインスタンスをモニタリングするためのコードに `ArrayList` を追加したものです。この中に、`describeInstance` レスポンスに関連付けられているインスタンス ID を格納します。

```
// 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);
```

この `ArrayList` に格納されているインスタンス ID を使用して、稼働中のインスタンスを終了させます。コードは次のとおりです。

```
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());
}
```

ステップの集約

これまでに説明したステップは、よりオブジェクト指向的なアプローチをとって 1 つに集約することができます。このステップとは、EC2 クライアントの初期化、スポットリクエストの提出、スポットリクエストがオープン状態でなくなったかどうかの特定、および未完了のスポットリクエストや関連するインスタンスのクリーンアップです。これらのすべてを実行する、`Requests` というクラスを作成します。

さらに、`GettingStartedApp` というクラスも作成します。ここにメインメソッドがあり、ここで高レベルの関数呼び出しを実行します。具体的には、既に説明した `Requests` オブジェクトを初期化します。スポットインスタンスリクエストを提出します。その後は、スポットリクエストが「アクティブ」状態になるまで待ちます。最後に、リクエストとインスタンスをクリーンアップします。

この例の完全なソースコードは、[GitHub](#) で確認またはダウンロードできます。

お疲れ様でした。これで、AWS SDK for Java を使用したスポットインスタンスソフトウェア開発の入門チュートリアルは終了です。

次のステップ

[チュートリアル: Amazon EC2 スポットリクエストの高度な管理](#)に進みます。

チュートリアル: Amazon EC2 スポットリクエストの高度な管理

Amazon EC2 スポットインスタンスとは、Amazon EC2 の未使用キャパシティに対してお客様から価格を提示していただき、入札値段がその時点のスポット料金を上回っている限り、お客様がインスタンスを実行できるというシステムです。Amazon EC2 のスポット料金は、需要と供給に応じて定期的に変動します。スポットインスタンスの詳細については、Amazon EC2 Linux インスタンス用ユーザーガイドの[スポットインスタンス](#)を参照してください。

前提条件

このチュートリアルを使用するには、AWS SDK for Java がインストールされており、基本インストール前提条件を満たしている必要があります。詳細については、「[Set up the AWS SDK for Java](#)」を参照してください。

認証情報のセットアップ

このサンプルコードの使用を開始するには、AWS 認証情報を設定する必要があります。その方法については、[開発用の AWS 認証情報とリージョンのセットアップ](#)を参照してください。

Note

IAM ユーザーの認証情報を使用してこれらの値を指定することをお勧めします。詳細については、[AWS にサインアップし、IAM ユーザーを作成する](#)を参照してください。

これで設定が完了したので、例に示すコードを使用できるようになります。

セキュリティグループのセットアップ

セキュリティグループとは、ファイアウォールとしての役割を果たすものであり、インスタンスのグループに対してどのトラフィックの送受信を許可するかを制御します。デフォルトでは、インスタンスの起動時にセキュリティグループは何も設定されていません。つまり、着信 IP トラフィックは、どの TCP ポートであってもすべて拒否されます。したがって、ここでは、スポットリクエストを提出する前に、必要なネットワークトラフィックを許可するセキュリティグループをセットアップすることにします。このチュートリアル目的に合わせて、ここでは新しいセキュリティグループを「GettingStarted」という名前で作成します。このグループでは、自分のアプリケーションを実行する IP アドレスからの Secure Shell (SSH) トラフィックを許可します。新しいセキュリティグループをセットアップするには、次に示すコードサンプルをインクルードするか実行する必要があります。このコードは、セキュリティグループをプログラムからセットアップするためのものです。

AmazonEC2 クライアントオブジェクトを作成した後で、`CreateSecurityGroupRequest` オブジェクトを作成し、「GettingStarted」という名前と、セキュリティグループの説明を指定します。その後で、`ec2.createSecurityGroup` API を呼び出してグループを作成します。

このグループにアクセスできるようにするために、`ipPermission` オブジェクトを作成します。IP アドレス範囲は、ローカルコンピュータのサブネット (CIDR 表現) で設定します。IP アドレスの「/10」というサフィックスが、指定した IP アドレスのサブネットを示します。また、`ipPermission` オブジェクトを設定して TCP プロトコルとポート 22 (SSH) を指定します。最後のステップは、`ec2.authorizeSecurityGroupIngress` を呼び出すことです。このときに、作成したセキュリティグループの名前と `ipPermission` オブジェクトを指定します。

(次に示すコードは、最初のチュートリアルで使用したのと同じものです)

```
// 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());
}
```

このコードサンプル全体を見るには、`advanced.CreateSecurityGroupApp.java` コードサンプルを参照してください。このアプリケーションを実行して新しいセキュリティグループを作成する必要があるのは 1 回のみです。

Note

また、AWS Toolkit for Eclipse を使用してセキュリティグループを作成することもできます。詳細については、「AWS Toolkit for Eclipse ユーザーガイド」の「[Managing Security Groups from AWS Cost Explorer](#)」を参照してください。

スポットインスタンスリクエスト作成の詳細なオプション

[チュートリアル: Amazon EC2 スポットインスタンス](#)で説明したように、リクエストを作成してインスタンスタイプ、Amazon Machine Image (AMI)、および最高入札価格を指定する必要があります。

初めに、`RequestSpotInstanceRequest` オブジェクトを作成します。このリクエストオブジェクトには、必要なインスタンスの数と入札価格が必要です。さらに、リクエストの `LaunchSpecification` も設定する必要があります。この内容は、インスタンスタイプ、AMI ID、および使用するセキュリティグループです。リクエストの内容が入力されたら、`requestSpotInstances` オブジェクトの `AmazonEC2Client` メソッドを呼び出します。スポットインスタンスをリクエストする方法の例を次に示します。

(次に示すコードは、最初のチュートリアルで使用したのと同じものです)

```
// 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);
```

永続リクエストと 1 回限りのリクエスト

スポットリクエストを作成するときは、複数の任意パラメータを指定できます。最初のパラメータは、そのリクエストが 1 回限りか持続的なものを指定するためのものです。デフォルトでは、リクエストは 1 回限りとなります。1 回限りのリクエストが受理されるのは 1 回だけであり、リクエストしたインスタンスが終了すると、そのリクエストはクローズ済みとなります。永続リクエストは、同じリクエストで実行されているスポットインスタンスがない限り、常に受理の対象となります。リクエストのタイプを指定するには、スポットリクエストの `Type` を設定します。このことを行うコードを次に示します。

```
// 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);
```

リクエストの期間の制限

また、リクエストの有効期間もオプションで指定できます。その期間の開始時点と終了時点の両方を指定できます。デフォルトでは、スポットリクエストが受理の対象とみなされるのは、作成された時点から、そのリクエストが受理されるか作成者によってキャンセルされるまでの間となります。ただし、必要であれば、作成時に有効期間を指定できます。この期間を指定する方法の例を次のコードに示します。

```
// 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);
```

Amazon EC2 スポットインスタンスリクエストのグループ化

スポットインスタンスリクエストには、いくつか異なる方法でグループ化するオプションがあります。ここでは、起動グループ、アベイラビリティゾーングループ、およびプレイスメントグループの利点について説明します。

リクエストしたスポットインスタンスがすべて同時に起動され、同時に終了するようにしたい場合は、起動グループを利用します。起動グループとは、1つにまとめる入札のグループに付けられるラ

ベルです。同じ起動グループ内のインスタンスはすべて、同時に起動されて同時に終了します。なお、起動グループ内のインスタンスが受理済みの場合に、その同じ起動グループで起動される新しいインスタンスも受理されるという保証はありません。起動グループを設定する方法の例を次のコードサンプルで示します。

```
// 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);
```

1つのリクエスト内のすべてのインスタンスが同じアベイラビリティーゾーン内で起動されるようにする必要があるが、どのアベイラビリティーゾーンでもかまわない場合は、アベイラビリティーゾーングループを利用します。アベイラビリティーゾーングループとは、同じアベイラビリティーゾーンにまとめるインスタンスのグループに付けられるラベルです。同じアベイラビリティーゾーングループ

プに属し、同時に受理されたインスタンスはすべて、同じアベイラビリティゾーンで起動されます。アベイラビリティゾーングループを設定する方法の例を次に示します。

```
// 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);
```

リクエストするスポットインスタンスをどのアベイラビリティゾーンで起動したいかを指定できます。次のコードサンプルでは、アベイラビリティゾーンの設定方法を示します。

```
// 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);
```

最後のプレイズメントグループは、ハイパフォーマンスコンピューティング (HPC) スポットインスタンス (クラスターコンピュートインスタンスやクラスター GPU インスタンスなど) を使用する場合に指定できます。プレイズメントグループを利用すると、低レイテンシー、高帯域幅でインスタンス間を接続できます。プレイズメントグループを設定する方法の例を次に示します。

```
// 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);
```

このセクションで示したパラメータはいずれも、省略可能です。また、これらのパラメータのほとんど (入札が 1 回限りであるか永続的であるかを除く) により、入札が履行される可能性を低減できることを理解することも重要です。したがって、これらのオプションは、そのオプションが必要な場合に限って使用することが重要です。これまでに示したコード例すべてを 1 つにまとめたものが `com.amazonaws.codesamples.advanced.InlineGettingStartedCodeSampleApp.java` クラスの中にあります。

中断または終了の後もルートパーティションを永続化する方法

スポットインスタンスの中断を管理する最も簡単な方法は、データのチェックポイントを作成して Amazon Elastic Block Store (Amazon Amazon EBS) ボリュームに保存するという処理を定期的に行

うことです。チェックポイントを定期的に作成しておくこと、中断が発生したときでも、データが失われるのは最後のチェックポイント以降に作成された分だけになります (その間に他の非べき等アクションが実行されていないことを前提とします)。このプロセスを容易にするには、スポットリクエストを設定するときに、中断時や終了時にルートパーティションを削除しないことを指定します。このシナリオを実現する方法を示す新しいコードが、次の例に挿入されています。

追加されたコードの中では、BlockDeviceMapping オブジェクトを作成し、対応する Amazon Elastic Block Store (Amazon EBS) を Amazon EBS オブジェクトに設定しています (このオブジェクトは、スポットインスタンスが終了しても削除しない (not) よう設定済みです)。その後で、この BlockDeviceMapping をマッピングの ArrayList に追加し、起動指定の中でこのマッピングを指定します。

```
// 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);
```

このボリュームがインスタンスの起動時に再度接続されるようにしたい場合は、ブロックデバイスマッピング設定を使用することもできます。別の方法としては、ルート以外のパーティションを接続する場合に、どの Amazon Amazon EBS ボリュームをスポットインスタンス再開後にインスタンスに接続するかを指定できます。このようにするには、スナップショット ID を `EbsBlockDevice` オブジェクトで指定し、代替デバイス名を `BlockDeviceMapping` オブジェクトで指定します。ブロックデバイスマッピングを利用すると、インスタンスのブートストラップが容易になります。

ルートパーティションを使用して重要なデータのチェックポイントを作成しておくこと、インスタンスの中断の可能性を管理するうえで大いに役立ちます。中断の可能性を管理するその他の方法については、[中断の管理についての動画](#)をご覧ください。

スポットリクエストとインスタンスにタグを付加する方法

Amazon EC2 リソースにタグを追加すると、クラウドインフラストラクチャの管理を簡略化できます。タグとは、メタデータの形を取るものであり、わかりやすい名前を付けるのに使用できます。また、検索がしやすくなり、複数ユーザー間での共同作業にも役立ちます。タグは、プロセスのスクリプトや各部分の自動化にも使用できます。Amazon EC2 リソースのタグ付けの詳細については、Amazon EC2 Linux インスタンス用ユーザーガイドの[タグの使用](#)を参照してください。

リクエストのタグ付け

使用するスポットリクエストにタグを追加するには、リソースをリクエストした後でタグを付ける必要があります。requestSpotInstances() からの戻り値によって、タグ付けのためのスポットリクエスト ID を取得する際に使用できる [RequestSpotInstancesResult](#) オブジェクトが提供されます。

```
// 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());
}
```

ID を取得したら、[CreateTagsRequest](#) に ID を追加し、Amazon EC2 クライアントの createTags() メソッドを呼び出してリクエストにタグを追加できます。

```
// 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());
}
```

インスタンスにタグを付ける

同様に、スポットリクエスト自体に対し、インスタンスの作成後 1 つのインスタンスのみにタグを追加でき、またそのタグはスポットリクエストに一致する場合のみ追加されます (オープン状態ではなくなります)。

Amazon EC2 クライアントの `describeSpotInstanceRequests()` メソッドを [DescribeSpotInstanceRequestsRequest](#) オブジェクトとともに呼び出し、リクエストのステータスを確認できます。返される [DescribeSpotInstanceRequestsResult](#) オブジェクトには、スポットリクエストのステータスをクエリし、open 状態でなくなったときにインスタンス ID を取得するために使用できる [SpotInstanceRequest](#) オブジェクトのリストが含まれています。

スポットリクエストがオープン状態でなくなると、`getInstanceId()` メソッドを呼び出すことで、そのインスタンス ID を `SpotInstanceRequest` オブジェクトから取得できます。

```
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);
```

ここで、返されるインスタンスにタグを追加できます。

```
// 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());
}
```

スポットリクエストのキャンセルとインスタンスの削除

スポットリクエストのキャンセル

スポットインスタンスリクエストをキャンセルするには、Amazon EC2 クライアントの `cancelSpotInstanceRequests` を [CancelSpotInstanceRequestsRequest](#) オブジェクトとともに呼び出します。

```
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());
}
```

スポットインスタンスの削除

Amazon EC2 クライアントの `terminateInstances()` メソッドに ID を渡すことで、実行中のすべてのスポットインスタンスを終了できます。

```
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());
}
```

ステップの集約

これまでに説明したステップは、よりオブジェクト指向的なアプローチをとって1つのクラスに集約し、利便性を高めることができます。Requests という名前のクラスをインスタンス化すると、これらのアクションを実行できます。さらに、GettingStartedApp というクラスも作成します。ここにメインメソッドがあり、ここで高レベルの関数呼び出しを実行します。

この例の完全なソースコードは、[GitHub](#) で確認またはダウンロードできます。

お疲れ様でした。これで、AWS SDK for Java を使用してスポットインスタンスソフトウェアを開発するための、高度なリクエスト機能のチュートリアルは終了です。

Amazon EC2 インスタンスの管理

インスタンスを作成する

新しい Amazon EC2 インスタンスを作成するには、AmazonEC2Client の runInstances メソッドを呼び出して、使用する [Amazon マシンイメージ \(AMI\)](#)、および [インスタンスタイプ](#) を含む [RunInstancesRequest](#) を指定します。

インポート

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

Code

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

RunInstancesResult run_response = ec2.runInstances(run_request);
```

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

[完全な例](#)をご覧ください。

インスタンスの起動

Amazon EC2 インスタンスを起動するには、AmazonEC2Client の `startInstances` メソッドを呼び出して、開始するインスタンスの ID を含む [StartInstancesRequest](#) 指定します。

インポート

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

Code

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

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

ec2.startInstances(request);
```

[完全な例](#)をご覧ください。

インスタンスの停止

Amazon EC2 インスタンスを停止するには、AmazonEC2Client の `stopInstances` メソッドを呼び出して、停止するインスタンスの ID を含む [StopInstancesRequest](#) 指定します。

インポート

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

Code

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

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

ec2.stopInstances(request);
```

[完全な例](#)をご覧ください。

インスタンスの再起動

Amazon EC2 インスタンスを再起動するには、AmazonEC2Client の `rebootInstances` メソッドを呼び出して、再起動するインスタンスの ID を含む [RebootInstancesRequest](#) を指定します。

インポート

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

Code

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

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

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

[完全な例](#)をご覧ください。

インスタンスの説明

インスタンスをリスト表示するには、[DescribeInstancesRequest](#) を作成し、AmazonEC2Client の `describeInstances` メソッドを呼び出します。お客様のアカウントとリージョンの [インスタンスをリスト表示するのに使用できる](#) `DescribeInstancesResult` Amazon EC2 オブジェクトが返されます。

インスタンスは予約ごとにグループ化されています。それぞれの予約は、インスタンスを起動した `startInstances` の呼び出しに対応しています。インスタンスをリスト表示するには、ま

ず DescribeInstancesResult クラスの getReservations' method, and then call `getInstances` [予約](#)オブジェクトごとにメソッド名 getInstances を呼び出します。

インポート

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

Code

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

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

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

    request.setNextToken(response.getNextToken());

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

```
}
```

結果はページ分割されます。さらに結果を取得するには、結果オブジェクトの `getNextToken` メソッドから返された値を元のリクエストオブジェクトの `setNextToken` メソッドに渡した後、次の `describeInstances` の呼び出しで同じリクエストオブジェクトを使用します。

[完全な例](#)をご覧ください。

インスタンスの監視

CPU やネットワークの使用率、使用可能なメモリ、ディスクの残り容量など、Amazon EC2 インスタンスのさまざまな側面を監視できます。インスタンスのモニタリングの詳細については、「Linux インスタンス用 Amazon EC2 ユーザーガイド」の「[Monitoring Amazon EC2](#)」を参照してください。

インスタンスのモニタリングを開始するには、モニタリングするインスタンスの ID で [MonitorInstancesRequest](#) を作成し、AmazonEC2Client の `monitorInstances` メソッドに渡します。

インポート

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

Code

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

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

ec2.monitorInstances(request);
```

[完全な例](#)をご覧ください。

インスタンス監視の停止

インスタンスのモニタリングを停止するには、モニタリングを停止するインスタンスの ID で [UnmonitorInstancesRequest](#) を作成し、AmazonEC2Client の `unmonitorInstances` メソッドに渡します。

インポート

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

Code

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

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

ec2.unmonitorInstances(request);
```

[完全な例](#)をご覧ください。

詳細情報

- Amazon EC2 API リファレンスの [RunInstances](#)
- Amazon EC2 API リファレンスの [DescribeInstances](#)
- Amazon EC2 API リファレンスの [StartInstances](#)
- Amazon EC2 API リファレンスの [StopInstances](#)
- Amazon EC2 API リファレンスの [RebootInstances](#)
- Amazon EC2 API リファレンスの [MonitorInstances](#)
- Amazon EC2 API リファレンスの [UnmonitorInstances](#)

での Elastic IP アドレスの使用 Amazon EC2

EC2-Classic は廃止されます

Warning

2022 年 8 月 15 日に、EC2-Classic の提供を終了します。EC2-Classic は、VPC への移行をお勧めします。詳細については、[Amazon EC2](#) EC2-Classic から VPC への移行 [Amazon EC2](#)」を参照してください。ブログ記事「[EC2-Classic-Classic Networking is Retiring – Here's How to Prepare](#)」も参照してください。

Elastic IP アドレスの割り当て

Elastic IP アドレスを使用するには、まずアカウントに 1 つ割り当ててから、それをインスタンスまたはネットワークインターフェイスに関連付けます。

Elastic IP アドレスを割り当てるには、ネットワークタイプ (クラシック EC2 または VPC) を含むオブジェクトを使用して AmazonEC2Client の `allocateAddress` メソッドを呼び出します。

[AllocateAddressRequest](#)

返されるには、の割り当て ID とインスタンス ID を AmazonEC2Client の `associateAddress` メソッドに渡すことで、アドレスをインスタンス [AssociateAddressRequest](#) に関連付けるために使用できる割り当て ID [AllocateAddressResult](#) が含まれています。 AmazonEC2Client

インポート

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

Code

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

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

AllocateAddressResult allocate_response =
    ec2.allocateAddress(allocate_request);

String allocation_id = allocate_response.getAllocationId();

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

AssociateAddressResult associate_response =
```

```
ec2.associateAddress(associate_request);
```

[完全な例](#)をご覧ください。

Elastic IP アドレスの説明

アカウントに割り当てられた Elastic IP アドレスを一覧表示するには AmazonEC2Client の `describeAddresses` メソッドを呼び出します。これは、アカウントの Elastic IP [アドレス](#)を表す Address オブジェクトのリストを取得するために [DescribeAddressesResult](#) 使用できる を返します。

インポート

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

Code

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

DescribeAddressesResult response = ec2.describeAddresses();

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

[完全な例](#)をご覧ください。

Elastic IP アドレスを解放する

Elastic IP アドレスを解放するには、AmazonEC2Client の `releaseAddress` メソッドを呼び出し、解放する Elastic IP アドレスの割り当て ID [ReleaseAddressRequest](#) を含む を渡します。

インポート

```
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.ReleaseAddressRequest;
import com.amazonaws.services.ec2.model.ReleaseAddressResult;
```

Code

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

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

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

Elastic IP アドレスを解放すると、そのアドレスは AWS IP アドレスプールに解放され、その後使用できなくなる場合があります。DNS レコード、およびそのアドレスと通信するすべてのサーバーまたはデバイスを更新してください。既にリリースした Elastic IP アドレスを解放しようとする、そのアドレスが既に別の に割り当てられている場合に AuthFailure エラーが発生します AWS アカウント。

EC2-Classic またはデフォルト VPC を使用している場合、Elastic IP アドレスを解放すると関連付けられているすべてのインスタンスからの関連付けが自動的に解除されます。Elastic IP アドレスを開放せずに関連付けを解除するには、AmazonEC2Client の `disassociateAddress` メソッドを使用します。

デフォルト以外の VPC を使用している場合は、開放しようとする前に必ず `disassociateAddress` を使用して Elastic IP アドレスの関連付けを解除する必要があります。それ以外の場合は、エラー (`InvalidIPAddress`) Amazon EC2 を返します。 `InvalidIPAddressInUse`

[完全な例](#)をご覧ください。

詳細情報

- [「Linux インスタンス用ユーザーガイド」の「Elastic IP アドレス Amazon EC2」](#)
- [AllocateAddress](#) Amazon EC2 API リファレンスの
- [DescribeAddresses](#) Amazon EC2 API リファレンスの
- [ReleaseAddress](#) Amazon EC2 API リファレンスの

リージョンとアベイラビリティゾーンを使用する

リージョンの詳細を表示する

アカウントに使用可能なリージョンを一覧表示するには、AmazonEC2Client の `describeRegions` メソッドを呼び出します。[DescribeRegionsResult](#) が返されます。返されたオブジェクトの `getRegions` メソッドを呼び出して、各リージョンを表す [Region](#) オブジェクトの一覧を取得します。

インポート

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

Code

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

for(Region region : regions_response.getRegions()) {
    System.out.printf(
        "Found region %s " +
        "with endpoint %s",
        region.getRegionName(),
        region.getEndpoint());
}
```

[完全な例](#)をご覧ください。

アベイラビリティゾーンの詳細を表示する

アカウントに使用可能な各アベイラビリティゾーンを一覧表示するには、AmazonEC2Client の `describeAvailabilityZones` メソッドを呼び出します。[DescribeAvailabilityZonesResult](#) が返されます。その `getAvailabilityZones` メソッドを呼び出して、各アベイラビリティゾーンを表す [AvailabilityZone](#) オブジェクトの一覧を取得します。

インポート

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

Code

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

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

[完全な例](#)をご覧ください。

アカウントの説明

アカウントの詳細を表示するには、AmazonEC2Client の describeAccountAttributes メソッドを呼び出します。このメソッドは、[DescribeAccountAttributesResult](#) オブジェクトを返します。このオブジェクト getAccountAttributes メソッドを呼び出して、[AccountAttribute](#) オブジェクトのリストを取得します。リストを反復処理して、[AccountAttribute](#) オブジェクトを取得できます。

アカウントの属性値は、[AccountAttribute](#) オブジェクトの getAttributeValues メソッドを呼び出すことで取得できます。このメソッドは、[AccountAttributeValue](#) オブジェクトのリストを返します。この 2 番目のリストを反復処理して、属性の値を表示できます (次のコード例を参照)。

インポート

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

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

Code

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

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

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

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

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

[GitHub](#) で完全な例をご覧ください。

詳細情報

- Amazon EC2 Linux インスタンス用ユーザーガイドの [リージョンとアベイラビリティゾーン](#)
- Amazon EC2 API リファレンスの [DescribeRegions](#)
- Amazon EC2 API リファレンスの [DescribeAvailabilityZones](#)

Amazon EC2 キーペアでの作業

キーペアを作成する

キーペアを作成するには、そのキーの名前を含む [CreateKeyPairRequest](#) を使用して AmazonEC2Client の createKeyPair メソッドを呼び出します。

インポート

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

Code

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

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

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

[完全な例](#)をご覧ください。

キーペアの詳細を表示する

キーペアを一覧表示したりキーペアに関する情報を入手するには、AmazonEC2Client の describeKeyPairs メソッドを呼び出します。 [メソッドを呼び出すことでキーペアの一覧にアクセスするのに使用できる](#) DescribeKeyPairsResult.getKeyPairs が返され、それにより [KeyPairInfo](#) オブジェクトの一覧が返されます。

インポート

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

Code

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

DescribeKeyPairsResult response = ec2.describeKeyPairs();

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

[完全な例](#)をご覧ください。

キーペアを削除する

キーペアを削除するには、AmazonEC2Client の deleteKeyPair メソッドを呼び出し、それに削除するキーペアの名前を含む [DeleteKeyPairRequest](#) を渡します。

インポート

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

Code

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

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

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

[完全な例](#)をご覧ください。

詳細情報

- Linux インスタンス用 Amazon EC2 ユーザーガイドの [Amazon EC2 キーペア](#)
- Amazon EC2 API リファレンスの [CreateKeyPair](#)

- Amazon EC2 API リファレンスの [DescribeKeyPairs](#)
- Amazon EC2 API リファレンスの [DeleteKeyPair](#)

Amazon EC2 でセキュリティグループを操作する

セキュリティグループを作成する

セキュリティグループを作成するには、そのキーの名前を含む [CreateSecurityGroupRequest](#) を使用して AmazonEC2Client の createSecurityGroup メソッドを呼び出します。

インポート

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

Code

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

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

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

[完全な例](#)をご覧ください。

セキュリティグループを設定する

セキュリティグループは、Amazon EC2 インスタンスへのインバウンド (ingress) とアウトバウンド (egress) トラフィックの両方を制御できます。

セキュリティグループに Ingress ルールを追加するには、AmazonEC2Client の authorizeSecurityGroupIngress メソッドを使用して、セキュリティグループの名前と

[AuthorizeSecurityGroupIngressRequest](#) オブジェクト内で割り当てるアクセスルール ([IpPermission](#)) を指定します。以下の例では、セキュリティグループへの IP のアクセス許可の追加方法を説明します。

インポート

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

Code

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

IpPermission ip_perm = new IpPermission()
    .withIpProtocol("tcp")
    .withToPort(80)
    .withFromPort(80)
    .withIpv4Ranges(ip_range);

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

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

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

セキュリティグループに egress ルールを追加するには、同様のデータを AmazonEC2Client の `authorizeSecurityGroupEgress` メソッドに [AuthorizeSecurityGroupEgressRequest](#) で指定します。

[完全な例](#)をご覧ください。

セキュリティグループについて説明する

セキュリティグループについて記述、またはそれらに関する情報を収集するには、AmazonEC2Client の describeSecurityGroups メソッドを呼び出します。getSecurityGroupsメソッドを呼び出すことでセキュリティグループの一覧にアクセスするのに使用できる [DescribeSecurityGroupsResult](#) が返され、それにより [SecurityGroup](#) オブジェクトの一覧が返されます。

インポート

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

Code

```
final String USAGE =
    "To run this example, supply a group id\n" +
    "Ex: DescribeSecurityGroups <group-id>\n";

if (args.length != 1) {
    System.out.println(USAGE);
    System.exit(1);
}

String group_id = args[0];
```

[完全な例](#)をご覧ください。

セキュリティグループの削除

セキュリティグループを削除するには、AmazonEC2Client の deleteSecurityGroup メソッドを呼び出し、それに削除するセキュリティグループの ID を含む [DeleteSecurityGroupRequest](#) を渡します。

インポート

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

```
import com.amazonaws.services.ec2.model.DeleteSecurityGroupRequest;
import com.amazonaws.services.ec2.model.DeleteSecurityGroupResult;
```

Code

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

DeleteSecurityGroupRequest request = new DeleteSecurityGroupRequest()
    .withGroupId(group_id);

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

[完全な例](#)をご覧ください。

詳細情報

- Linux インスタンス用 Amazon EC2 ユーザーガイドの [Amazon EC2 セキュリティグループ](#)
- Amazon EC2 Linux インスタンス用ユーザーガイドでの [Linux インスタンスのインバウンドトラフィックの承認](#)
- Amazon EC2 API リファレンスの [CreateSecurityGroup](#)
- Amazon EC2 API リファレンスの [DescribeSecurityGroups](#)
- Amazon EC2 API リファレンスの [DeleteSecurityGroup](#)
- Amazon EC2 API リファレンスの [AuthorizeSecurityGroupIngress](#)

AWS SDK for Java を使用した IAM の例

このセクションでは、[AWS SDK for Java](#) を使用して [IAM](#) をプログラムする例を示します。

AWS Identity and Access Management (IAM) を使用すると、AWS のサービスおよびリソースに対するお客様のユーザーのアクセスを安全にコントロールすることができます。IAM を使用すると、AWS のユーザーとグループを作成および管理し、許可を使用して AWS リソースへのアクセスを許可および拒否できます。IAM の詳細なガイドについては、「[IAM ユーザーガイド](#)」を参照してください。

Note

例には各手法を示すのに必要なコードのみが含まれます。[完全なサンプルコードは GitHub で入手できます](#)。そこから、単一のソースファイルをダウンロードするかリポジトリをローカルにクローン作成して、ビルドし実行するためのすべての例を取得できます。

トピック

- [IAM アクセスキーの管理](#)
- [IAM ユーザーの管理](#)
- [IAM アカウントエイリアスの使用](#)
- [IAM ポリシーの使用](#)
- [IAM サーバー証明書の使用](#)

IAM アクセスキーの管理

アクセスキーの作成

IAM アクセスキーを作成するには、`AmazonIdentityManagementClient.createAccessKey` メソッドを [CreateAccessKeyRequest](#) オブジェクトを使用して呼び出します。

`CreateAccessKeyRequest` には 2 つのコンストラクタ、すなわち、ユーザー名を取るものとパラメータのないものとがあります。パラメータを取らないバージョンを使用する場合、`createAccessKey` メソッドに渡す前に `withUserName` setter メソッドを使用してユーザー名を設定する必要があります。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreateAccessKeyRequest;
import com.amazonaws.services.identitymanagement.model.CreateAccessKeyResult;
```

Code

```
final AmazonIdentityManagement iam =
```

```
AmazonIdentityManagementClientBuilder.defaultClient();

CreateAccessKeyRequest request = new CreateAccessKeyRequest()
    .withUserName(user);

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

[GitHub](#) で完全な例をご覧ください。

アクセスキーの一覧表示

特定のユーザーのアクセスキーを一覧表示するには、キーの一覧表示の対象となるユーザー名を含む [ListAccessKeysRequest](#) オブジェクトを作成し、それを `AmazonIdentityManagementClient` の `listAccessKeys` メソッドに渡します。

Note

ユーザー名を `listAccessKeys` に渡さない場合は、リクエストに署名した AWS アカウントに関連付けられているアクセスキーの一覧表示を試行します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.AccessKeyMetadata;
import com.amazonaws.services.identitymanagement.model.ListAccessKeysRequest;
import com.amazonaws.services.identitymanagement.model.ListAccessKeysResult;
```

Code

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

boolean done = false;
ListAccessKeysRequest request = new ListAccessKeysRequest()
    .withUserName(username);

while (!done) {
```

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

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

request.setMarker(response.getMarker());

if (!response.getIsTruncated()) {
    done = true;
}
}
```

`listAccessKeys` の結果はページ分割されます (デフォルトで最大 1 回の呼び出しごとに 100 レコード)。返された [ListAccessKeysResult](#) オブジェクトで `getIsTruncated` を呼び出し、より少ない結果を返されたクエリが利用可能かどうか確認することができます。利用可能な場合は、`ListAccessKeysRequest` で `setMarker` を呼び出し、それを `listAccessKeys` の次の呼び出しに返します。

[GitHub](#) で完全な例をご覧ください。

アクセスキーの最終使用時刻の取得

アクセスキーが最後に使用された時刻を取得するには、そのアクセスキーの ID で `AmazonIdentityManagementClient` の `getAccessKeyLastUsed` メソッドを呼び出します。アクセスキーの ID は [GetAccessKeyLastUsedRequest](#) オブジェクトを使用して、またはアクセスキー ID を直接取るオーバーロードへ直接渡すことができます。

その後返された [GetAccessKeyLastUsedResult](#) オブジェクトを使用して、キーの最終使用時刻を取得できます。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.GetAccessKeyLastUsedRequest;
import com.amazonaws.services.identitymanagement.model.GetAccessKeyLastUsedResult;
```

Code

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

GetAccessKeyLastUsedRequest request = new GetAccessKeyLastUsedRequest()
    .withAccessKeyId(access_id);

GetAccessKeyLastUsedResult response = iam.getAccessKeyLastUsed(request);

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

[GitHub](#) で完全な例をご覧ください。

アクセスキーのアクティブ化や非アクティブ化

アクセスキーをアクティブ化または非アクティブ化するには、[UpdateAccessKeyRequest](#) オブジェクトを作成し、アクセスキー ID、オプションでユーザー名、また目的の[ステータス](#)を渡して、次にそのリクエストオブジェクトを `AmazonIdentityManagementClient` の `updateAccessKey` メソッドに渡します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.UpdateAccessKeyRequest;
import com.amazonaws.services.identitymanagement.model.UpdateAccessKeyResult;
```

Code

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

UpdateAccessKeyRequest request = new UpdateAccessKeyRequest()
    .withAccessKeyId(access_id)
    .withUserName(username)
    .withStatus(status);

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

[GitHub](#) で完全な例をご覧ください。

アクセスキーの削除

アクセスキーを完全に削除するには、AmazonIdentityManagementClient の deleteKey メソッドを呼び出し、それにアクセスキーの ID とユーザーネームを含む [DeleteAccessKeyRequest](#) を渡します。

Note

削除してしまうと、キーは取得することも使用することもできなくなります。後で再度アクティブ化できるようキーを一時的に非アクティブ化するには、代わりに [updateAccessKey](#) メソッドを使用します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteAccessKeyRequest;
import com.amazonaws.services.identitymanagement.model.DeleteAccessKeyResult;
```

Code

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

DeleteAccessKeyRequest request = new DeleteAccessKeyRequest()
    .withAccessKeyId(access_key)
    .withUserName(username);

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

[GitHub](#) で完全な例をご覧ください。

詳細情報

- IAM API リファレンスの [CreateAccessKey](#)
- IAM API リファレンスの [ListAccessKeys](#)
- IAM API リファレンスの [GetAccessKeyLastUsed](#)
- IAM API リファレンスの [UpdateAccessKey](#)

- IAM API リファレンスの [DeleteAccessKey](#)

IAM ユーザーの管理

ユーザーの作成

新しい IAM ユーザーを作成するには、直接、またはユーザー名を含む [CreateUserRequest](#) オブジェクトを使用して、AmazonIdentityManagementClient の `createUser` メソッドにユーザー名を渡します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreateUserRequest;
import com.amazonaws.services.identitymanagement.model.CreateUserResult;
```

Code

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

CreateUserRequest request = new CreateUserRequest()
    .withUserName(username);

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

[GitHub](#) で完全な例をご覧ください。

ユーザーの一覧表示

アカウントの IAM ユーザーを一覧表示するには、新しい [ListUsersRequest](#) を作成して、それを AmazonIdentityManagementClient の `listUsers` メソッドに渡します。返された [ListUsersResult](#) オブジェクトで `getUsers` を呼び出すことでユーザーのリストを取得できます。

`listUsers` によって返されたユーザーのリストはページ分割されます。取得できる結果がさらにあることを確認するには、応答オブジェクトの `getIsTruncated` メソッドを呼び出します。true が返ってきた場合、リクエストオブジェクトの `setMarker()` メソッドを呼び出し、それに応答オブジェクトの `getMarker()` メソッドの戻り値を渡します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListUsersRequest;
import com.amazonaws.services.identitymanagement.model.ListUsersResult;
import com.amazonaws.services.identitymanagement.model.User;
```

Code

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

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

while(!done) {
    ListUsersResult response = iam.listUsers(request);

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

    request.setMarker(response.getMarker());

    if(!response.getIsTruncated()) {
        done = true;
    }
}
```

[GitHub](#) で完全な例をご覧ください。

ユーザーの更新

ユーザーを更新するには、AmazonIdentityManagementClient オブジェクトの updateUser メソッドを呼び出し、それが取得する [UpdateUserRequest](#) オブジェクトを使用して、ユーザーの名前またはパスを変更します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
```

```
import com.amazonaws.services.identitymanagement.model.UpdateUserRequest;
import com.amazonaws.services.identitymanagement.model.UpdateUserResult;
```

Code

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

UpdateUserRequest request = new UpdateUserRequest()
    .withUserName(cur_name)
    .withNewUserName(new_name);

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

[GitHub](#) で完全な例をご覧ください。

ユーザーの削除

ユーザーを削除するには、[UpdateUserRequest](#) オブジェクトに削除するユーザー名を設定して、`AmazonIdentityManagementClient` の `deleteUser` リクエストを呼び出します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteConflictException;
import com.amazonaws.services.identitymanagement.model.DeleteUserRequest;
```

Code

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

DeleteUserRequest request = new DeleteUserRequest()
    .withUserName(username);

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

```
}
```

[GitHub](#) で完全な例をご覧ください。

詳細情報

- IAM ユーザーガイドの [IAM ユーザー](#)
- IAM ユーザーガイドの [IAM ユーザーの管理](#)
- IAM API リファレンスの [CreateUser](#)
- IAM API リファレンスの [ListUsers](#)
- IAM API リファレンスの [UpdateUser](#)
- IAM API リファレンスの [DeleteUser](#)

IAM アカウントエイリアスの使用

サインインページの URL に、AWS アカウント ID ではなく企業の名前または他のわかりやすい識別子を含めるには、AWS アカウントのエイリアスを作成します。

Note

AWS ではアカウントごとに 1 つのアカウントのエイリアスのみがサポートされます。

アカウントエイリアスの作成

アカウントエイリアスを作成するには、そのエイリアス名が含まれる [CreateAccountAliasRequest](#) オブジェクトを使用して `AmazonIdentityManagementClient` の `createAccountAlias` メソッドを呼び出します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;  
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;  
import com.amazonaws.services.identitymanagement.model.CreateAccountAliasRequest;  
import com.amazonaws.services.identitymanagement.model.CreateAccountAliasResult;
```

Code

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

CreateAccountAliasRequest request = new CreateAccountAliasRequest()
    .withAccountAlias(alias);

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

[GitHub](#) で完全な例をご覧ください。

アカウントエイリアスを一覧表示する

アカウントエイリアスを一覧表示するには、AmazonIdentityManagementClient の `listAccountAliases` メソッドを呼び出します。

Note

返される [ListAccountAliasesResult](#) は、他の AWS SDK for Java list メソッドと同じ `getIsTruncated` および `getMarker` メソッドがサポートされますが、AWS アカウントで使用できるアカウントエイリアスは 1 つのみです。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListAccountAliasesResult;
```

コード

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

ListAccountAliasesResult response = iam.listAccountAliases();

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

GitHub で [完全な例](#) をご覧ください。

アカウントエイリアスを削除する

アカウントエイリアスを削除するには、AmazonIdentityManagementClient の `deleteAccountAlias` メソッドを呼び出します。アカウントエイリアスを削除する場合は、[DeleteAccountAliasRequest](#) オブジェクトを使用してその名前を指定する必要があります。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteAccountAliasRequest;
import com.amazonaws.services.identitymanagement.model.DeleteAccountAliasResult;
```

Code

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

DeleteAccountAliasRequest request = new DeleteAccountAliasRequest()
    .withAccountAlias(alias);

DeleteAccountAliasResult response = iam.deleteAccountAlias(request);
```

[GitHub](#) で完全な例をご覧ください。

詳細情報

- IAM ユーザーガイドの [AWS アカウント ID とそのエイリアス](#)
- IAM API リファレンスの [CreateAccountAlias](#)
- IAM API リファレンスの [ListAccountAliases](#)
- IAM API リファレンスの [DeleteAccountAlias](#)

IAM ポリシーの使用

ポリシーの作成

新しいポリシーを作成するには、[CreatePolicyRequest](#) 内のポリシーの名前および JSON 形式のポリシードキュメントを AmazonIdentityManagementClient の `createPolicy` メソッドに渡します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.CreatePolicyRequest;
import com.amazonaws.services.identitymanagement.model.CreatePolicyResult;
```

Code

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

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

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

IAM ポリシードキュメントは [文書による十分な裏づけのある構文](#) を持つ JSON 文字列です。DynamoDB に特定のリクエストをするためのアクセスを提供する例を以下に示します。

```
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\" " +
    "    } " +
    "  ] " +
    "}";
```


[GitHub](#) で完全な例をご覧ください。

ポリシーの取得

既存のポリシーを取得するには、AmazonIdentityManagementClient の `getPolicy` メソッドを呼び出して、[GetPolicyRequest](#) オブジェクト内のポリシーの ARN を渡します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.GetPolicyRequest;
import com.amazonaws.services.identitymanagement.model.GetPolicyResult;
```

Code

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

GetPolicyRequest request = new GetPolicyRequest()
    .withPolicyArn(policy_arn);

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

[GitHub](#) で完全な例をご覧ください。

ロールポリシーのアタッチ

ポリシーを IAMhttp://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html[ロール] にアタッチするには、AmazonIdentityManagementClient の `attachRolePolicy` メソッドを呼び出し、[AttachRolePolicyRequest](#) でロール名とポリシー ARN を指定します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.AttachRolePolicyRequest;
import com.amazonaws.services.identitymanagement.model.AttachedPolicy;
```

Code

```
final AmazonIdentityManagement iam =
```

```
AmazonIdentityManagementClientBuilder.defaultClient();

AttachRolePolicyRequest attach_request =
    new AttachRolePolicyRequest()
        .withRoleName(role_name)
        .withPolicyArn(POLICY_ARN);

iam.attachRolePolicy(attach_request);
```

[GitHub](#) で完全な例をご覧ください。

アタッチ済みロールポリシーの一覧表示

ロールのアタッチ済みポリシーを一覧表示するには、AmazonIdentityManagementClient の `listAttachedRolePolicies` メソッドを呼び出します。このメソッドは、ポリシーを一覧表示するロール名を含む [ListAttachedRolePoliciesRequest](#) オブジェクトを受け取ります。

返された [ListAttachedRolePoliciesResult](#) オブジェクトで `getAttachedPolicies` を呼び出してアタッチ済みポリシーのリストを取得します。結果は切り捨てられる場合があります。ListAttachedRolePoliciesResult オブジェクトの `getIsTruncated` メソッドが `true` を返す場合は、ListAttachedRolePoliciesRequest オブジェクトの `setMarker` メソッドを呼び出し、それを使用して `listAttachedRolePolicies` を再び呼び出し、結果の次のバッチを取得します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListAttachedRolePoliciesRequest;
import com.amazonaws.services.identitymanagement.model.ListAttachedRolePoliciesResult;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
```

Code

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

ListAttachedRolePoliciesRequest request =
    new ListAttachedRolePoliciesRequest()
        .withRoleName(role_name);
```

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

boolean done = false;

while(!done) {
    ListAttachedRolePoliciesResult response =
        iam.listAttachedRolePolicies(request);

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

    if(!response.getIsTruncated()) {
        done = true;
    }
    request.setMarker(response.getMarker());
}
```

[GitHub](#) で完全な例をご覧ください。

ロールポリシーのデタッチ

ロールからポリシーをデタッチするには、`AmazonIdentityManagementClient` の `detachRolePolicy` メソッドを呼び出し、それに [DetachRolePolicyRequest](#) 内のロール名およびポリシー ARN を渡します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DetachRolePolicyRequest;
import com.amazonaws.services.identitymanagement.model.DetachRolePolicyResult;
```

Code

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

DetachRolePolicyRequest request = new DetachRolePolicyRequest()
    .withRoleName(role_name)
```

```
.withPolicyArn(policy_arn);

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

[GitHub](#) で完全な例をご覧ください。

詳細情報

- IAM ユーザーガイドの [IAM ポリシーの概要](#)。
- IAM ユーザーガイドの [AWS IAM ポリシーのリファレンス](#)。
- IAM API リファレンスの [CreatePolicy](#)
- IAM API リファレンスの [GetPolicy](#)
- IAM API リファレンスの [AttachRolePolicy](#)
- IAM API リファレンスの [ListAttachedRolePolicies](#)
- IAM API リファレンスの [DetachRolePolicy](#)

IAM サーバー証明書の使用

AWS でウェブサイトまたはアプリケーションへの HTTPS 接続を有効にするには、SSL/TLS サーバー証明書が必要です。AWS Certificate Manager から提供されたサーバー証明書、または外部プロバイダーから入手したサーバー証明書を使用できます。

ACM を使用してサーバー証明書のプロビジョニング、管理、デプロイを行うことをお勧めします。ACM を使用すると、証明書をリクエストし、それを AWS リソースにデプロイして、証明書の更新を ACM で処理できます。ACM で提供される証明書は無料です。ACM の詳細については、[ACM ユーザーガイド](#)を参照してください。

サーバー証明書の取得

サーバー証明書を取得するには、AmazonIdentityManagementClient の `getServerCertificate` メソッドを呼び出し、それに証明書の名前を含む [GetServerCertificateRequest](#) を渡します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.GetServerCertificateRequest;
import com.amazonaws.services.identitymanagement.model.GetServerCertificateResult;
```

Code

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

GetServerCertificateRequest request = new GetServerCertificateRequest()
    .withServerCertificateName(cert_name);

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

[GitHub](#) で完全な例をご覧ください。

サーバー証明書の一覧表示

サーバー証明書を一覧表示するには、AmazonIdentityManagementClient の `listServerCertificates` メソッドを呼び出し、[ListServerCertificatesRequest](#) を渡します。[ListServerCertificatesResult](#) が返されます。

返された `ListServerCertificateResult` オブジェクトの `getServerCertificateMetadataList` メソッドを呼び出して、各証明書についての情報を取得するために使用できる [ServerCertificateMetadata](#) オブジェクトの一覧を取得します。

結果は切り捨てられる場合があります。`ListServerCertificateResult` オブジェクトの `getIsTruncated` メソッドが `true` を返す場合は、`ListServerCertificatesRequest` オブジェクトの `setMarker` メソッドを呼び出し、それを使用して `listServerCertificates` を再び呼び出し、結果の次のバッチを取得します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.ListServerCertificatesRequest;
import com.amazonaws.services.identitymanagement.model.ListServerCertificatesResult;
import com.amazonaws.services.identitymanagement.model.ServerCertificateMetadata;
```

Code

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

boolean done = false;
ListServerCertificatesRequest request =
```

```
        new ListServerCertificatesRequest());

while(!done) {

    ListServerCertificatesResult response =
        iam.listServerCertificates(request);

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

    request.setMarker(response.getMarker());

    if(!response.getIsTruncated()) {
        done = true;
    }
}
```

[GitHub](#) で完全な例をご覧ください。

サーバー証明書の更新

サーバー証明書の名前やパスを更新するには、`AmazonIdentityManagementClient` の `updateServerCertificate` メソッドを呼び出します。サーバー証明書の現在の名前および使用する新しい名前か新しいパスのいずれかを使って設定した [UpdateServerCertificateRequest](#) オブジェクトが使用されます。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.UpdateServerCertificateRequest;
import com.amazonaws.services.identitymanagement.model.UpdateServerCertificateResult;
```

Code

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

UpdateServerCertificateRequest request =
```

```
new UpdateServerCertificateRequest()
    .withServerCertificateName(cur_name)
    .withNewServerCertificateName(new_name);

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

[GitHub](#) で完全な例をご覧ください。

サーバー証明書の削除

サーバー証明書を削除するには、証明書の名前を含む [DeleteServerCertificateRequest](#) を使用して、AmazonIdentityManagementClient の `deleteServerCertificate` メソッドを呼び出します。

インポート

```
import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;
import com.amazonaws.services.identitymanagement.model.DeleteServerCertificateRequest;
import com.amazonaws.services.identitymanagement.model.DeleteServerCertificateResult;
```

Code

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

DeleteServerCertificateRequest request =
    new DeleteServerCertificateRequest()
        .withServerCertificateName(cert_name);

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

[GitHub](#) で完全な例をご覧ください。

詳細情報

- IAM ユーザーガイドの [サーバー証明書の使用](#)
- IAM API リファレンスの [GetServerCertificate](#)
- IAM API リファレンスの [ListServerCertificates](#)
- IAM API リファレンスの [UpdateServerCertificate](#)

- IAM API リファレンスの [DeleteServerCertificate](#)
- [ACM ユーザーガイド](#)

LambdaAWS SDK for Java を使用した例

このセクションでは、AWS SDK for Java を使用して Lambda をプログラムする例を示します。

Note

例には各手法を示すのに必要なコードのみが含まれます。[完全なサンプルコードは GitHub で入手できます](#)。そこから、単一のソースファイルをダウンロードするリポジトリをローカルにクローン作成して、ビルドし実行するためのすべての例を取得できます。

トピック

- [Lambda 関数の呼び出し、一覧表示、および削除](#)

Lambda 関数の呼び出し、一覧表示、および削除

このセクションでは、AWS SDK for Java を使用した Lambda のサービスのクライアントでのプログラミングの例を示します。Lambda 関数の作成方法については、[AWS Lambda 関数の作成方法](#)を参照してください。

トピック

- [関数を呼び出す](#)
- [関数の一覧表示](#)
- [関数を削除する](#)

関数を呼び出す

[AWSLambda](#) オブジェクトを作成し、その `invoke` メソッドを呼び出すことによって、Lambda 関数を呼び出すことができます。[InvokeRequest](#) オブジェクトを作成して、Lambda 関数に渡す関数名やペイロードなどの追加情報を指定します。関数名は、`arn:aws:lambda:us-east-1:555556330391:function:HelloFunction` と表示されます。AWS Management Console で関数を確認することで、値を取得できます。

ペイロードデータを関数に渡すには、次のコード例に示すように、[InvokeRequest](#) オブジェクトの `withPayload` メソッドを呼び出し、JSON 形式の文字列を指定します。

インポート

```
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.lambda.AWSLambda;
import com.amazonaws.services.lambda.AWSLambdaClientBuilder;
import com.amazonaws.services.lambda.model.InvokeRequest;
import com.amazonaws.services.lambda.model.InvokeResult;
import com.amazonaws.services.lambda.model.ServiceException;

import java.nio.charset.StandardCharsets;
```

Code

次のコード例は、Lambda 関数を呼び出す方法を示しています。

```
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());
```

[GitHub](#) で完全な例をご覧ください。

関数の一覧表示

[AWSLambda](#) オブジェクトを構築し、その `listFunctions` メソッドを呼び出します。このメソッドは、[ListFunctionsResult](#) オブジェクトを返します。このオブジェクトの `getFunctions` メソッドを呼び出して、[FunctionConfiguration](#) オブジェクトのリストを返すことができます。リストを反復処理して、関数に関する情報を取得できます。たとえば、次の Java コード例は、各関数名を取得する方法を示しています。

インポート

```
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.lambda.AWSLambda;
import com.amazonaws.services.lambda.AWSLambdaClientBuilder;
import com.amazonaws.services.lambda.model.FunctionConfiguration;
import com.amazonaws.services.lambda.model.ListFunctionsResult;
import com.amazonaws.services.lambda.model.ServiceException;
import java.util.Iterator;
import java.util.List;
```

Code

次の Java コード例は、Lambda 関数名のリストを取得する方法を示しています。

```
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);
    }
}
```

[GitHub](#) で完全な例をご覧ください。

関数を削除する

[AWSLambda](#) オブジェクトを構築し、その `deleteFunction` メソッドを呼び出します。[DeleteFunctionRequest](#) オブジェクトを作成し、`deleteFunction` メソッドに渡します。このオブジェクトには、削除する関数の名前などの情報が含まれています。関数名は、`arn:aws:lambda:us-east-1:555556330391:function:HelloFunction` と表示されます。AWS Management Console で関数を確認することで、値を取得できます。

インポート

```
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.lambda.AWSLambda;
import com.amazonaws.services.lambda.AWSLambdaClientBuilder;
import com.amazonaws.services.lambda.model.ServiceException;
import com.amazonaws.services.lambda.model.DeleteFunctionRequest;
```

Code

次の Java コードは、Lambda 関数を削除する方法を示しています。

```
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);
}
```

[GitHub](#) で完全な例をご覧ください。

Amazon PinpointAWS SDK for Java を使用した例

このセクションでは、[AWS SDK for Java](#) を使用して [Amazon Pinpoint](#) をプログラムする例を示します。

Note

例には各手法を示すのに必要なコードのみが含まれます。[完全なサンプルコードは GitHub で入手できます](#)。そこから、単一のソースファイルをダウンロードするかリポジトリをローカルにクローン作成して、ビルドし実行するためのすべての例を取得できます。

トピック

- [Amazon Pinpoint のアプリの作成および削除](#)
- [Amazon Pinpoint でのエンドポイントの作成](#)
- [Amazon Pinpoint でのセグメントの作成](#)
- [Amazon Pinpoint でのキャンペーンの作成](#)
- [Amazon Pinpoint のチャンネルの更新](#)

Amazon Pinpoint のアプリの作成および削除

アプリとは、個別のアプリケーションのオーディエンスを定義する Amazon Pinpoint のプロジェクトです。このオーディエンスにカスタマイズされたメッセージを使用して働きかけます。このページの例では、新しいアプリの作成方法や既存のアプリの削除方法を説明します。

アプリの作成

アプリ名を [CreateAppRequest](#) オブジェクトに指定し、そのオブジェクトを `AmazonPinpointClient` の `createApp` メソッドに渡して、Amazon Pinpoint に新しいアプリケーションを作成します。

インポート

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
import com.amazonaws.services.pinpoint.model.CreateAppRequest;
import com.amazonaws.services.pinpoint.model.CreateAppResult;
import com.amazonaws.services.pinpoint.model.CreateApplicationRequest;
```

Code

```
CreateApplicationRequest appRequest = new CreateApplicationRequest()
    .withName(appName);

CreateAppRequest request = new CreateAppRequest();
request.withCreateApplicationRequest(appRequest);
CreateAppResult result = pinpoint.createApp(request);
```

[GitHub](#) で完全な例をご覧ください。

アプリの削除

アプリケーションを削除するには、削除するアプリケーション名を設定した [DeleteAppRequest](#) オブジェクトを指定して `AmazonPinpointClient` の `deleteApp` リクエストを呼び出します。

インポート

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

Code

```
DeleteAppRequest deleteRequest = new DeleteAppRequest()
    .withApplicationId(appID);

pinpoint.deleteApp(deleteRequest);
```

[GitHub](#) で完全な例をご覧ください。

詳細情報

- Amazon Pinpoint API リファレンスの [アプリケーション](#)

- Amazon Pinpoint API リファレンスの[アプリケーション](#)

Amazon Pinpoint でのエンドポイントの作成

エンドポイントでユーザーデバイスを一意に識別し、Amazon Pinpoint でプッシュ通知を送信することができます。アプリで Amazon Pinpoint サポートが有効になっている場合、アプリは、新しいユーザーがアプリを開いたときに Amazon Pinpoint で自動的にエンドポイントを登録します。次の例では、プログラムで新しいエンドポイントを追加する方法について説明します。

エンドポイントの作成

Amazon Pinpoint で新しいエンドポイントを作成するには、[EndpointRequest](#) オブジェクトにエンドポイントデータを指定します。

インポート

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
import com.amazonaws.services.pinpoint.model.UpdateEndpointRequest;
import com.amazonaws.services.pinpoint.model.UpdateEndpointResult;
import com.amazonaws.services.pinpoint.model.EndpointDemographic;
import com.amazonaws.services.pinpoint.model.EndpointLocation;
import com.amazonaws.services.pinpoint.model.EndpointRequest;
import com.amazonaws.services.pinpoint.model.EndpointResponse;
import com.amazonaws.services.pinpoint.model.EndpointUser;
import com.amazonaws.services.pinpoint.model.GetEndpointRequest;
import com.amazonaws.services.pinpoint.model.GetEndpointResult;
```

Code

```
HashMap<String, List<String>> customAttributes = new HashMap<>();
List<String> favoriteTeams = new ArrayList<>();
favoriteTeams.add("Lakers");
favoriteTeams.add("Warriors");
customAttributes.put("team", favoriteTeams);

EndpointDemographic demographic = new EndpointDemographic()
    .withAppVersion("1.0")
    .withMake("apple")
    .withModel("iPhone")
    .withModelVersion("7")
```

```
.withPlatform("ios")
.withPlatformVersion("10.1.1")
.withTimezone("America/Los_Angeles");

EndpointLocation location = new EndpointLocation()
    .withCity("Los Angeles")
    .withCountry("US")
    .withLatitude(34.0)
    .withLongitude(-118.2)
    .withPostalCode("90068")
    .withRegion("CA");

Map<String,Double> metrics = new HashMap<>();
metrics.put("health", 100.00);
metrics.put("luck", 75.00);

EndpointUser user = new EndpointUser()
    .withUserId(UUID.randomUUID().toString());

DateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm'Z'"); // Quoted "Z" to
    indicate UTC, no timezone offset
String nowAsISO = df.format(new Date());

EndpointRequest endpointRequest = new EndpointRequest()
    .withAddress(UUID.randomUUID().toString())
    .withAttributes(customAttributes)
    .withChannelType("APNS")
    .withDemographic(demographic)
    .withEffectiveDate(nowAsISO)
    .withLocation(location)
    .withMetrics(metrics)
    .withOptOut("NONE")
    .withRequestId(UUID.randomUUID().toString())
    .withUser(user);
```

次に、その `EndpointRequest` オブジェクトを使用して [UpdateEndpointRequest](#) オブジェクトを作成します。最後に、`UpdateEndpointRequest` オブジェクトを `AmazonPinpointClient` の `updateEndpoint` メソッドに渡します。

Code

```
UpdateEndpointRequest updateEndpointRequest = new UpdateEndpointRequest()
    .withApplicationId(appId)
```

```
        .withEndpointId(endpointId)
        .withEndpointRequest(endpointRequest);

UpdateEndpointResult updateEndpointResponse =
    client.updateEndpoint(updateEndpointRequest);
System.out.println("Update Endpoint Response: " +
    updateEndpointResponse.getMessageBody());
```

[GitHub](#) で完全な例をご覧ください。

詳細情報

- Amazon Pinpoint デベロッパーガイドの[エンドポイントの追加](#)
- Amazon Pinpoint API リファレンスの[エンドポイント](#)

Amazon Pinpoint でのセグメントの作成

ユーザーセグメントは、ユーザーが最近いつ頃アプリを開いたか、またはどのデバイスを使用しているか、などの共有特性に基づくユーザーのサブセットを表します。次の例では、ユーザーのセグメントを定義する方法を示しています。

セグメントの作成

Amazon PinpointSegmentDimensions [オブジェクトでセグメントのディメンションを定義することで](#)、に新しいセグメントを作成します。

インポート

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
import com.amazonaws.services.pinpoint.model.CreateSegmentRequest;
import com.amazonaws.services.pinpoint.model.CreateSegmentResult;
import com.amazonaws.services.pinpoint.model.AttributeDimension;
import com.amazonaws.services.pinpoint.model.AttributeType;
import com.amazonaws.services.pinpoint.model.RecencyDimension;
import com.amazonaws.services.pinpoint.model.SegmentBehaviors;
import com.amazonaws.services.pinpoint.model.SegmentDemographics;
import com.amazonaws.services.pinpoint.model.SegmentDimensions;
import com.amazonaws.services.pinpoint.model.SegmentLocation;
import com.amazonaws.services.pinpoint.model.SegmentResponse;
import com.amazonaws.services.pinpoint.model.WriteSegmentRequest;
```


Code

```
Pinpoint pinpoint =
    AmazonPinpointClientBuilder.standard().withRegion(Regions.US_EAST_1).build();
Map<String, AttributeDimension> segmentAttributes = new HashMap<>();
segmentAttributes.put("Team", new
    AttributeDimension().withAttributeType(AttributeType.INCLUSIVE).withValues("Lakers"));

SegmentBehaviors segmentBehaviors = new SegmentBehaviors();
SegmentDemographics segmentDemographics = new SegmentDemographics();
SegmentLocation segmentLocation = new SegmentLocation();

RecencyDimension recencyDimension = new RecencyDimension();
recencyDimension.withDuration("DAY_30").withRecencyType("ACTIVE");
segmentBehaviors.setRecency(recencyDimension);

SegmentDimensions dimensions = new SegmentDimensions()
    .withAttributes(segmentAttributes)
    .withBehavior(segmentBehaviors)
    .withDemographic(segmentDemographics)
    .withLocation(segmentLocation);
```

次に [SegmentDimensions](#) オブジェクトを [WriteSegmentRequest](#) に設定します。これは [CreateSegmentRequest](#) オブジェクトの作成に使用されます。その後、CreateSegmentRequest オブジェクトを AmazonPinpointClient の createSegment メソッドに渡します。

Code

```
WriteSegmentRequest writeSegmentRequest = new WriteSegmentRequest()
    .withName("MySegment").withDimensions(dimensions);

CreateSegmentRequest createSegmentRequest = new CreateSegmentRequest()
    .withApplicationId(appId).withWriteSegmentRequest(writeSegmentRequest);

CreateSegmentResult createSegmentResult = client.createSegment(createSegmentRequest);
```

[GitHub](#) で完全な例をご覧ください。

詳細情報

- Amazon Pinpoint ユーザーガイドの [Amazon Pinpoint セグメント](#)
- Amazon Pinpoint デベロッパーガイドの [セグメントの作成](#)

- Amazon Pinpoint API リファレンスの[セグメント](#)
- Amazon Pinpoint API リファレンスの[セグメント](#)

Amazon Pinpoint でのキャンペーンの作成

キャンペーンを使用して、ユーザーとアプリケーション間の関与を向上させることができます。キャンペーンを作成すると、ユーザーのセグメントに合わせてカスタマイズされたメッセージまたは特別なプロモーションを使用してユーザーに連絡できます。この例では、指定されたセグメントにカスタマイズされたプッシュ通知を送信する標準的なキャンペーンを新しく作成する方法を説明します。

キャンペーンの作成

新しいキャンペーンを作成する前に、[スケジュール](#)と[メッセージ](#)を定義し、[WriteCampaignRequest](#) オブジェクトでこれらの値を設定する必要があります。

インポート

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
import com.amazonaws.services.pinpoint.model.CreateCampaignRequest;
import com.amazonaws.services.pinpoint.model.CreateCampaignResult;
import com.amazonaws.services.pinpoint.model.Action;
import com.amazonaws.services.pinpoint.model.CampaignResponse;
import com.amazonaws.services.pinpoint.model.Message;
import com.amazonaws.services.pinpoint.model.MessageConfiguration;
import com.amazonaws.services.pinpoint.model.Schedule;
import com.amazonaws.services.pinpoint.model.WriteCampaignRequest;
```

Code

```
Schedule schedule = new Schedule()
    .withStartTime("IMMEDIATE");

Message defaultMessage = new Message()
    .withAction(Action.OPEN_APP)
    .withBody("My message body.")
    .withTitle("My message title.");

MessageConfiguration messageConfiguration = new MessageConfiguration()
    .withDefaultMessage(defaultMessage);
```

```
WriteCampaignRequest request = new WriteCampaignRequest()
    .withDescription("My description.")
    .withSchedule(schedule)
    .withSegmentId(segmentId)
    .withName("MyCampaign")
    .withMessageConfiguration(messageConfiguration);
```

次に、Amazon Pinpoint `CreateCampaignRequest` オブジェクトに `WriteCampaignRequest` をキャンペーン設定とともに指定し、で新規キャンペーンを作成します。最後に、`CreateCampaignRequest` オブジェクトを `AmazonPinpointClient` の `createCampaign` メソッドに渡します。

Code

```
CreateCampaignRequest createCampaignRequest = new CreateCampaignRequest()
    .withApplicationId(appId).withWriteCampaignRequest(request);

CreateCampaignResult result = client.createCampaign(createCampaignRequest);
```

[GitHub](#) で完全な例をご覧ください。

詳細情報

- Amazon Pinpoint ユーザーガイドの [Amazon Pinpoint キャンペーン](#)
- Amazon Pinpoint デベロッパーガイドの [キャンペーンの作成](#)
- Amazon Pinpoint API リファレンスの [キャンペーン](#)
- Amazon Pinpoint API リファレンスの [キャンペーン](#)
- Amazon Pinpoint API リファレンスの [キャンペーンアクティビティ](#)
- Amazon Pinpoint API リファレンスの [キャンペーンバージョン](#)
- Amazon Pinpoint API リファレンスの [キャンペーンバージョン](#)

Amazon Pinpoint のチャネルの更新

チャネルはメッセージを配信するプラットフォームのタイプを定義します。この例は、APN チャネルを使用してメッセージを送信する方法を示しています。

チャネルの更新

アプリ ID と更新するチャネルタイプのリクエストオブジェクトを指定して、Amazon Pinpoint のチャネルを有効にします。この例では、[APNSChannelRequest](#) オブジェクトを要求する APN

チャンネルが更新されます。これらを [UpdateApnsChannelRequest](#) に設定し、そのオブジェクトを AmazonPinpointClient の updateApnsChannel メソッドに渡します。

インポート

```
import com.amazonaws.services.pinpoint.AmazonPinpoint;
import com.amazonaws.services.pinpoint.AmazonPinpointClientBuilder;
import com.amazonaws.services.pinpoint.model.APNSChannelRequest;
import com.amazonaws.services.pinpoint.model.APNSChannelResponse;
import com.amazonaws.services.pinpoint.model.GetApnsChannelRequest;
import com.amazonaws.services.pinpoint.model.GetApnsChannelResult;
import com.amazonaws.services.pinpoint.model.UpdateApnsChannelRequest;
import com.amazonaws.services.pinpoint.model.UpdateApnsChannelResult;
```

Code

```
APNSChannelRequest request = new APNSChannelRequest()
    .withEnabled(enabled);

UpdateApnsChannelRequest updateRequest = new UpdateApnsChannelRequest()
    .withAPNSChannelRequest(request)
    .withApplicationId(appId);
UpdateApnsChannelResult result = client.updateApnsChannel(updateRequest);
```

[GitHub](#) で完全な例をご覧ください。

詳細情報

- Amazon Pinpoint ユーザーガイドの [Amazon Pinpoint チャンネル](#)
- Amazon Pinpoint API リファレンスの [ADM チャンネル](#)
- Amazon Pinpoint API リファレンスの [APN チャンネル](#)
- Amazon Pinpoint API リファレンスの [APN サンドボックスチャンネル](#)
- Amazon Pinpoint API リファレンスの [APN VoIP チャンネル](#)
- Amazon Pinpoint API リファレンスの [APNs VoIP サンドボックスチャンネル](#)
- Amazon Pinpoint API リファレンスの [Baidu チャンネル](#)
- Amazon Pinpoint API リファレンスの [E メールチャンネル](#)
- Amazon Pinpoint API リファレンスの [GCM チャンネル](#)
- Amazon Pinpoint API リファレンスの [SMS チャンネル](#)

Amazon S3AWS SDK for Java を使用した例

このセクションでは、[AWS SDK for Java](#) を使用して [Amazon S3](#) をプログラムする例を示します。

Note

例には各手法を示すのに必要なコードのみが含まれます。[完全なサンプルコードは GitHub で入手できます](#)。そこから、単一のソースファイルをダウンロードするリポジトリをローカルにクローン作成して、ビルドし実行するためのすべての例を取得できます。

トピック

- [Amazon S3 バケットの作成、一覧表示、削除](#)
- [Amazon S3 オブジェクトに対する操作の実行](#)
- [バケットおよびオブジェクトの Amazon S3 アクセス許可の管理](#)
- [バケットポリシーを使用した Amazon S3 バケットへのアクセス管理](#)
- [Amazon S3 操作の TransferManager の使用](#)
- [ウェブサイトとしての Amazon S3 バケットの設定](#)
- [Amazon S3 クライアント側の暗号化を使用する](#)

Amazon S3 バケットの作成、一覧表示、削除

Amazon S3 の各オブジェクト (ファイル) は、オブジェクトのコレクション (コンテナ) を表すバケット内に存在している必要があります。各バケットはキー (名前) で識別され、それは一意である必要があります。バケットおよびその設定についての詳細は、Amazon Simple Storage Service ユーザーガイドの [Amazon S3 バケットの使用](#) を参照してください。

Note

ベストプラクティス

[バケットで AbortIncompleteMultipartUpload](#) Amazon S3 ライフサイクルルールを有効にすることを勧めます。

このルールは、開始後、指定された日数内に完了しないマルチパートアップロードを中止するよう Amazon S3 に指示できます。設定した時間制限を超えると、Amazon S3 はアップロードを中止して、不完全なアップロードデータを削除します。

詳細については、Amazon S3 ユーザーガイドの[バージョニングを使用したバケットのライフサイクル設定](#)を参照してください。

Note

これらのコード例では、ユーザーが [AWS SDK for Java の使用](#) の内容を理解し、[開発用の AWS 認証情報とリージョンのセットアップ](#) の情報を使用してデフォルトの AWS 認証情報を設定していることを前提としています。

バケットの作成

AmazonS3 クライアントの `createBucket` メソッドを使用します。新しい[バケット](#)が返されます。`createBucket` メソッドでは、バケットが既に存在する場合、例外が発生します。

Note

同じ名前のバケットを作成する前にバケットが既に存在するかどうかを確認するには、`doesBucketExist` メソッドを呼び出してください。バケットが存在する場合は `true` を返し、それ以外の場合は `false` を返します。

インポート

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AmazonS3Exception;
import com.amazonaws.services.s3.model.Bucket;

import java.util.List;
```

Code

```
if (s3.doesBucketExistV2(bucket_name)) {
    System.out.format("Bucket %s already exists.\n", bucket_name);
    b = getBucket(bucket_name);
}
```

```
} else {
    try {
        b = s3.createBucket(bucket_name);
    } catch (AmazonS3Exception e) {
        System.err.println(e.getMessage());
    }
}
return b;
```

[GitHub](#) で完全な例をご覧ください。

バケットの一覧表示

AmazonS3 クライアントの `listBucket` メソッドを使用します。成功すると、[バケット](#) のリストが返されます。

インポート

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.Bucket;

import java.util.List;
```

Code

```
List<Bucket> buckets = s3.listBuckets();
System.out.println("Your {S3} buckets are:");
for (Bucket b : buckets) {
    System.out.println("* " + b.getName());
}
```

[GitHub](#) で完全な例をご覧ください。

バケットの削除

Amazon S3 バケットを削除する前に、バケットが空であることを必ず確認してください。空になっていないとエラーが発生します。[バージョンングされたバケット](#) がある場合は、このバケットに関連付けられているすべてのバージョンングされたオブジェクトも削除する必要があります。

Note

完全な例には、これらの各ステップが順に含まれており、Amazon S3 バケットとそのコンテンツを削除するための完全なソリューションを提供しています。

トピック

- バケットを削除する前にバージョニングされていないバケットからオブジェクトを削除する
- バケットを削除する前にバージョニングされているバケットからオブジェクトを削除する
- 空のバケットを削除する

バケットを削除する前にバージョニングされていないバケットからオブジェクトを削除する

AmazonS3 クライアントの `listObjects` メソッドを使用してオブジェクトのリストおよび `deleteObject` を取得し、それぞれを削除します。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.*;

import java.util.Iterator;
```

Code

```
System.out.println(" - removing objects from bucket");
ObjectListing object_listing = s3.listObjects(bucket_name);
while (true) {
    for (Iterator<?> iterator =
        object_listing.getObjectSummaries().iterator();
        iterator.hasNext(); ) {
        S3ObjectSummary summary = (S3ObjectSummary) iterator.next();
        s3.deleteObject(bucket_name, summary.getKey());
    }

    // more object_listing to retrieve?
    if (object_listing.isTruncated()) {
```



```
        object_listing = s3.listNextBatchOfObjects(object_listing);
    } else {
        break;
    }
}
```

[GitHub](#) で完全な例をご覧ください。

バケットを削除する前にバージョニングされているバケットからオブジェクトを削除する

[バージョニングされたバケット](#)を使用している場合は、バケットを削除する前に、バケットに保存されているすべてのバージョンのオブジェクトも削除する必要があります。

バケット内のオブジェクトを削除する際に使用したのと同じような方法で、バージョニングされたオブジェクトを削除します。まず、AmazonS3 クライアントの `listVersions` メソッドを使用してすべてのバージョニングされたオブジェクトを一覧表示し、次に `deleteVersion` を使用して各オブジェクトを削除します。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.*;

import java.util.Iterator;
```

Code

```
System.out.println(" - removing versions from bucket");
VersionListing version_listing = s3.listVersions(
    new ListVersionsRequest().withBucketName(bucket_name));
while (true) {
    for (Iterator<?> iterator =
        version_listing.getVersionSummaries().iterator();
        iterator.hasNext(); ) {
        S3VersionSummary vs = (S3VersionSummary) iterator.next();
        s3.deleteVersion(
            bucket_name, vs.getKey(), vs.getVersionId());
    }

    if (version_listing.isTruncated()) {
```

```
        version_listing = s3.listNextBatchOfVersions(
            version_listing);
    } else {
        break;
    }
}
```

[GitHub](#) で完全な例をご覧ください。

空のバケットを削除する

バケットからオブジェクト (すべてのバージョニングされたオブジェクトを含む) を削除したら、AmazonS3 クライアントの `deleteBucket` メソッドを使用してバケット自体を削除できます。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.*;

import java.util.Iterator;
```

Code

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

[GitHub](#) で完全な例をご覧ください。

Amazon S3 オブジェクトに対する操作の実行

Amazon S3 オブジェクトは、ファイルまたはデータの集合を表します。すべてのオブジェクトが [バケット](#) 内にある必要があります。

Note

これらのコード例では、ユーザーが [AWS SDK for Java の使用](#) の内容を理解し、[開発用の AWS 認証情報とリージョンのセットアップ](#) の情報を使用してデフォルトの AWS 認証情報を設定していることを前提としています。

トピック

- [オブジェクトのアップロード](#)
- [オブジェクトのリスト化](#)
- [オブジェクトのダウンロード](#)
- [オブジェクトのコピー、移動、または名前の変更](#)
- [オブジェクトの削除](#)
- [複数オブジェクトの一括削除](#)

オブジェクトのアップロード

AmazonS3 クライアントの `putObject` メソッドを使用して、バケット名、キー名、アップロードするファイルを指定します。バケットが存在している必要があり、存在しない場合はエラーが発生します。

インポート

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

Code

```
System.out.format("Uploading %s to S3 bucket %s...\n", file_path, bucket_name);
final AmazonS3 s3 =
    AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    s3.putObject(bucket_name, key_name, new File(file_path));
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
```

[GitHub](#) で完全な例をご覧ください。

オブジェクトのリスト化

バケット内のオブジェクトのリストを取得するには、AmazonS3 クライアントの `listObjects` メソッドを使用して、バケット名を指定します。

`listObjects` メソッドは、バケットのオブジェクトに関する情報を提供する [ObjectListing](#) オブジェクトを返します。オブジェクト名 (キー) を一覧表示するには、`getObjectSummaries` メソッドを使用して、それぞれがバケット内の単一のオブジェクトを表す [S3ObjectSummary](#) オブジェクトのリストを取得し、メソッドを呼び出してオブジェクト名を取得します。それから、`getKey` メソッドを呼び出してオブジェクト名を取得します。

インポート

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.ListObjectsV2Result;
import com.amazonaws.services.s3.model.S3ObjectSummary;
```

Code

```
System.out.format("Objects in S3 bucket %s:\n", bucket_name);
final AmazonS3 s3 =
    AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
ListObjectsV2Result result = s3.listObjectsV2(bucket_name);
List<S3ObjectSummary> objects = result.getObjectSummaries();
for (S3ObjectSummary os : objects) {
    System.out.println("* " + os.getKey());
}
```

[GitHub](#) で完全な例をご覧ください。

オブジェクトのダウンロード

AmazonS3 クライアントの `getObject` メソッドを使用して、ダウンロードするバケットの名前とオブジェクトを渡します。成功すると、このメソッドによって [S3Object](#) が返されます。指定されたバケットとオブジェクトキーが存在している必要があり、存在しない場合エラーが発生します。

オブジェクトのコンテンツは、`getObjectContent` の `S3Object` を呼び出して取得できます。これにより、標準の Java `InputStream` オブジェクトとして動作する [S3ObjectInputStream](#) が返されます。

次の例では、S3 からオブジェクトをダウンロードし、そのコンテンツをファイルに保存します (オブジェクトキーと同じ名前を使用)。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.S3Object;
import com.amazonaws.services.s3.model.S3ObjectInputStream;

import java.io.File;
```

Code

```
System.out.format("Downloading %s from S3 bucket %s...\n", key_name, bucket_name);
final AmazonS3 s3 =
    AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    S3Object o = s3.getObject(bucket_name, key_name);
    S3ObjectInputStream s3is = o.getObjectContent();
    FileOutputStream fos = new FileOutputStream(new File(key_name));
    byte[] read_buf = new byte[1024];
    int read_len = 0;
    while ((read_len = s3is.read(read_buf)) > 0) {
        fos.write(read_buf, 0, read_len);
    }
    s3is.close();
    fos.close();
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
} catch (FileNotFoundException e) {
    System.err.println(e.getMessage());
    System.exit(1);
} catch (IOException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
```

[GitHub](#) で完全な例をご覧ください。

オブジェクトのコピー、移動、または名前の変更

AmazonS3 クライアントの `copyObject` メソッドを使用して、1つのバケットから別のバケットへオブジェクトをコピーできます。コピー元のバケットの名前、コピーするオブジェクト、およびコピー先バケットの名前が継承されます。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
```

Code

```
try {
    s3.copyObject(from_bucket, object_key, to_bucket, object_key);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
System.out.println("Done!");
```

[GitHub](#) で完全な例をご覧ください。

Note

`copyObject` を [deleteObject](#) とともに使用して、最初にオブジェクトを新しい名前でコピーし (コピー元とコピー先の両方に同じバケットの使用が可能)、元の場所からそのオブジェクトを削除することで、オブジェクトの移動または名前変更ができます。

オブジェクトの削除

AmazonS3 クライアントの `deleteObject` メソッドを使用して、削除するバケットの名前とオブジェクトを渡します。指定されたバケットとオブジェクトキーが存在している必要があり、存在しない場合エラーが発生します。

インポート

```
import com.amazonaws.AmazonServiceException;
```

```
import com.amazonaws.regions.Regions;
```

Code

```
final AmazonS3 s3 =
    AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    s3.deleteObject(bucket_name, object_key);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
```

[GitHub](#) で完全な例をご覧ください。

複数オブジェクトの一括削除

AmazonS3 クライアントの `deleteObjects` メソッドを使用すると、<link:sdk-for-java/v1/reference/com/amazonaws/services/s3/model/DeleteObjectsRequest.html> メソッドに名前を渡すことで、同じバケットから複数のオブジェクトを削除できます。

インポート

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

Code

```
final AmazonS3 s3 =
    AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    DeleteObjectsRequest dor = new DeleteObjectsRequest(bucket_name)
        .withKeys(object_keys);
    s3.deleteObjects(dor);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
```

[GitHub](#) で完全な例をご覧ください。

バケットおよびオブジェクトの Amazon S3 アクセス許可の管理

Amazon S3 バケットとオブジェクトのアクセスコントロールリスト (ACL) を使用して、Amazon S3 リソースをきめ細かく制御することができます。

Note

これらのコード例では、ユーザーが [AWS SDK for Java の使用](#) の内容を理解し、[開発用の AWS 認証情報とリージョンのセットアップ](#) の情報を使用してデフォルトの AWS 認証情報を設定していることを前提としています。

バケットのアクセスコントロールリストの取得

バケットの現在の ACL を取得するには、AmazonS3 の `getBucketAcl` メソッドを呼び出して、それにクエリを実行するバケット名を渡します。このメソッドは、[AccessControlList](#) オブジェクトを返します。リスト内の各アクセス権を取得するには、その `getGrantsAsList` メソッドを呼び出します。これにより、[Grant](#) オブジェクトの標準 Java リストが返されます。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.Grant;
```

Code

```
final AmazonS3 s3 =
    AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    AccessControlList acl = s3.getBucketAcl(bucket_name);
    List<Grant> grants = acl.getGrantsAsList();
    for (Grant grant : grants) {
        System.out.format("  %s: %s\n", grant.getGrantee().getIdentifier(),
            grant.getPermission().toString());
    }
} catch (AmazonServiceException e) {
```



```
System.err.println(e.getMessage());
System.exit(1);
}
```

[GitHub](#) で完全な例をご覧ください。

バケットのアクセスコントロールリストの設定

バケットの ACL に許可の追加や変更をするには、AmazonS3 の `setBucketAcl` メソッドを呼び出します。設定する被付与者やアクセスレベルのリストを含む [AccessControlList](#) オブジェクトが使用されます。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.EmailAddressGrantee;
```

Code

```
final AmazonS3 s3 =
    AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    // get the current ACL
    AccessControlList acl = s3.getBucketAcl(bucket_name);
    // set access for the grantee
    EmailAddressGrantee grantee = new EmailAddressGrantee(email);
    Permission permission = Permission.valueOf(access);
    acl.grantPermission(grantee, permission);
    s3.setBucketAcl(bucket_name, acl);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
```

Note

[Grantee](#) クラスを使用して直接被付与者の一意の識別子を提供するか、ここで行ったように、[EmailAddressGrantee](#) クラスを使用して被付与者を E メールで設定することができます。

[GitHub](#) で完全な例をご覧ください。

オブジェクトのアクセスコントロールリストの取得

オブジェクトの現在の ACL を取得するには、AmazonS3 の `getObjectAcl` メソッドを呼び出して、それにクエリを実行するバケット名およびオブジェクト名を渡します。 `getBucketAcl` と同様に、このメソッドは各[権限](#)を調べるのに使用できる [AccessControlList](#) オブジェクトを返します。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.Grant;
```

Code

```
try {
    AccessControlList acl = s3.getObjectAcl(bucket_name, object_key);
    List<Grant> grants = acl.getGrantsAsList();
    for (Grant grant : grants) {
        System.out.format("  %s: %s\n", grant.getGrantee().getIdentifer(),
            grant.getPermission().toString());
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
```

[GitHub](#) で完全な例をご覧ください。

オブジェクトのアクセスコントロールリストの設定

オブジェクトの ACL に許可の追加や変更をするには、AmazonS3 の `setObjectAcl` メソッドを呼び出します。設定する被付与者やアクセスレベルのリストを含む [AccessControlList](#) オブジェクトが使用されます。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.EmailAddressGrantee;
```

Code

```
try {
    // get the current ACL
    AccessControlList acl = s3.getObjectAcl(bucket_name, object_key);
    // set access for the grantee
    EmailAddressGrantee grantee = new EmailAddressGrantee(email);
    Permission permission = Permission.valueOf(access);
    acl.grantPermission(grantee, permission);
    s3.setObjectAcl(bucket_name, object_key, acl);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
}
```

Note

[Grantee](#) クラスを使用して直接被付与者の一意の識別子を提供するか、ここで行ったように、[EmailAddressGrantee](#) クラスを使用して被付与者を E メールで設定することができます。

[GitHub](#) で完全な例をご覧ください。

詳細情報

- Amazon S3 API リファレンスの [GET Bucket acl](#)
- Amazon S3 API リファレンスの [PUT Bucket acl](#)
- Amazon S3 API リファレンスの [GET Object acl](#)
- Amazon S3 API リファレンスの [PUT Object acl](#)

バケットポリシーを使用した Amazon S3 バケットへのアクセス管理

バケットポリシーを設定、取得、または削除して、Amazon S3 バケットへのアクセスを管理できます。

バケットポリシーの設定

特定の S3 バケットにバケットポリシーを設定するには。

- AmazonS3 クライアントの `setBucketPolicy` を呼び出し、[SetBucketPolicyRequest](#) を渡します。
- バケット名とポリシーテキスト (JSON 形式) を受け取る `setBucketPolicy` オーバーロードを使用して、直接ポリシーを設定します。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.policy.Policy;
import com.amazonaws.auth.policy.Principal;
```

Code

```
s3.setBucketPolicy(bucket_name, policy_text);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
```

Policy クラスを使用してポリシーを生成または検証する

バケットポリシーを `setBucketPolicy` に渡す場合、以下のことができます。

- ポリシーを JSON 形式のテキスト文字列として直接指定できます。
- [Policy](#) クラスを使用してポリシーを構築できます。

Policy クラスを使用することで、テキスト文字列を正しくフォーマットすることについて心配する必要はありません。Policy クラスから JSON ポリシーテキストを取得するには、その `toJson` メソッドを使用します。

インポート

```
import com.amazonaws.auth.policy.Resource;
import com.amazonaws.auth.policy.Statement;
import com.amazonaws.auth.policy.actions.S3Actions;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
```

Code

```
new Statement(Statement.Effect.Allow)
    .withPrincipals(Principal.AllUsers)
    .withActions(S3Actions.GetObject)
    .withResources(new Resource(
        "{region-arn}s3:::" + bucket_name + "/*"));
return bucket_policy.toJson();
```

Policy クラスは、渡された JSON 文字列を使用してポリシーの構築を試行できる `fromJson` メソッドも提供します。このメソッドは、確実にテキストが有効なポリシー構造へと変換できることを検証し、ポリシーテキストが無効な場合には `IllegalArgumentException` エラーとなります。

```
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());
}
```

この方法を使用して、ファイルやその他の手段から読み取るポリシーを事前に検証できます。

[GitHub](#) で完全な例をご覧ください。

バケットポリシーの取得

Amazon S3 バケットのポリシーを取得するには、AmazonS3 クライアントの `getBucketPolicy` メソッドを呼び出し、それにポリシーの取得元であるバケットの名前を渡します。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
```

Code

```
try {
    BucketPolicy bucket_policy = s3.getBucketPolicy(bucket_name);
    policy_text = bucket_policy.getPolicyText();
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
```

その名前のバケットが存在しない場合、バケットへのアクセス権がない場合、またはバケットポリシーがない場合は、`AmazonServiceException` がスローされます。

[GitHub](#) で完全な例をご覧ください。

バケットポリシーの削除

バケットポリシーを削除するには、AmazonS3 クライアントの `deleteBucketPolicy` を呼び出して、それにバケット名を渡します。

インポート

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

Code

```
try {
    s3.deleteBucketPolicy(bucket_name);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
```

このメソッドは、バケットにまだポリシーがない場合でも成功します。指定した名前のバケットが存在していないか、バケットへのアクセス権がない場合は、AmazonServiceException がスローされます。

[GitHub](#) で完全な例をご覧ください。

詳細

- Amazon Simple Storage Service ユーザーガイドの[アクセスポリシー言語の概要](#)
- Amazon Simple Storage Service ユーザーガイドの[バケットポリシーの例](#)

Amazon S3 操作の TransferManager の使用

AWS SDK for Java TransferManager クラスを使用して、ローカル環境から Amazon S3 へファイルを確実に転送し、S3 の複数の場所間でオブジェクトをコピーします。TransferManager は、転送の進行状況を取得し、アップロードとダウンロードの一時停止/再開を行うことができます。

Note

ベストプラクティス

[バケットで AbortIncompleteMultipartUpload](#) Amazon S3 ライフサイクルルールを有効にすることを推奨します。

このルールは、開始後、指定された日数内に完了しないマルチパートアップロードを中止するよう Amazon S3 に指示できます。設定した時間制限を超えると、Amazon S3 はアップロードを中止して、不完全なアップロードデータを削除します。

詳細については、Amazon S3 ユーザーガイドの[バージョニングを使用したバケットのライフサイクル設定](#)を参照してください。

Note

これらのコード例では、ユーザーが [AWS SDK for Java の使用](#) の内容を理解し、[開発用の AWS 認証情報とリージョンのセットアップ](#) の情報を使用してデフォルトの AWS 認証情報を設定していることを前提としています。

ファイルとディレクトリのアップロード

TransferManager は、[以前に作成した](#) Amazon S3 バケットのいずれに対してもファイル、ファイルリスト、ディレクトリをアップロードできます。

トピック

- [1つのファイルのアップロード](#)
- [ファイルのリストのアップロード](#)
- [ディレクトリのアップロード](#)

1つのファイルのアップロード

TransferManager の upload メソッドを呼び出し、Amazon S3 バケット名、キー (オブジェクト) 名、アップロードするファイルを表す標準 Java [ファイル](#) オブジェクトを指定します。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.MultipleFileUpload;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;
import com.amazonaws.services.s3.transfer.Upload;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
```

Code

```
File f = new File(file_path);
TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
```



```
try {
    Upload xfer = xfer_mgr.upload(bucket_name, key_name, f);
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    // or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

upload メソッドは、即座に返され、転送の状態を確認する、または、転送が完了するまで待機するための Upload オブジェクトが提供されます。

waitForCompletion を使用して、転送が正常に完了した後に TransferManager の shutdownNow メソッドを呼び出すことの詳細については、[転送の完了の待機](#)を参照してください。転送の完了を待ちながら、転送の状態や進行状況に関する更新情報をポーリングまたはリスンできます。詳細については、「[転送の状態と進行状況の取得](#)」を参照してください。

[GitHub](#) で完全な例をご覧ください。

ファイルのリストのアップロード

複数のファイルを一括してアップロードするには、TransferManageruploadFileList メソッドを呼び出して、以下を指定します。

- Amazon S3 バケット名
- 作成したオブジェクトの名前の前に付加されるキープレフィックス (オブジェクトを置くバケット内のパス)
- ファイルパスの作成元の相対ディレクトリを表す[ファイル](#)オブジェクト
- [リスト](#)オブジェクト (アップロードする一連の[ファイル](#)オブジェクトを含む)

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.MultipleFileUpload;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;
```

```
import com.amazonaws.services.s3.transfer.Upload;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
```

Code

```
ArrayList<File> files = new ArrayList<File>();
for (String path : file_paths) {
    files.add(new File(path));
}

TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
try {
    MultipleFileUpload xfer = xfer_mgr.uploadFileList(bucket_name,
        key_prefix, new File("."), files);
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    // or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

`waitForCompletion` を使用して、転送が正常に完了した後に `TransferManager` の `shutdownNow` メソッドを呼び出すことの詳細については、[転送の完了の待機](#) を参照してください。転送の完了を待ちながら、転送の状態や進行状況に関する更新情報をポーリングまたはリスンできます。詳細については、「[転送の状態と進行状況の取得](#)」を参照してください。

`uploadFileList` から返される [MultipleFileUpload](#) オブジェクトを使用して、転送の状態や進行状況をクエリできます。詳細については、「[転送の現在の進行状況のポーリング](#)」と「[ProgressListener による転送の進行状況の取得](#)」を参照してください。

`MultipleFileUpload` の `getSubTransfers` メソッドを使用して、転送中の各ファイルについて個別の `Upload` オブジェクトを取得することもできます。詳細については、「[サブ転送の進行状況の取得](#)」を参照してください。

[GitHub](#) で完全な例をご覧ください。

ディレクトリのアップロード

TransferManager の `uploadDirectory` メソッドを使用して、ファイルのディレクトリ全体をアップロードし、オプションとしてファイルをサブディレクトリに再帰的にコピーできます。このメソッドに、Amazon S3 バケット名、S3 キープレフィックス、コピーするローカルディレクトリを表す [File](#) オブジェクト、およびサブディレクトリに再帰的にコピーするかどうか (`boolean true` または `false`) を示す 値を渡します。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.MultipleFileUpload;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;
import com.amazonaws.services.s3.transfer.Upload;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
```

Code

```
TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
try {
    MultipleFileUpload xfer = xfer_mgr.uploadDirectory(bucket_name,
        key_prefix, new File(dir_path), recursive);
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    // or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

`waitForCompletion` を使用して、転送が正常に完了した後に TransferManager の `shutdownNow` メソッドを呼び出すことの詳細については、[転送の完了の待機](#)を参照してください。転送の完了を待ちながら、転送の状態や進行状況に関する更新情報をポーリングまたはリスンできます。詳細については、「[転送の状態と進行状況の取得](#)」を参照してください。

uploadFileList から返される [MultipleFileUpload](#) オブジェクトを使用して、転送の状態や進行状況をクエリできます。詳細については、「[転送の現在の進行状況のポーリング](#)」と「[ProgressListener による転送の進行状況の取得](#)」を参照してください。

MultipleFileUpload の getSubTransfers メソッドを使用して、転送中の各ファイルについて個別の Upload オブジェクトを取得することもできます。詳細については、「[サブ転送の進行状況の取得](#)」を参照してください。

[GitHub](#) で完全な例をご覧ください。

ファイルまたはディレクトリのダウンロード

TransferManager クラスを使用して、1つのファイル (Amazon S3 オブジェクト) またはディレクトリ (Amazon S3 バケット名とオブジェクトプレフィックス) を Amazon S3 からダウンロードできます。

トピック

- [1つのファイルのダウンロード](#)
- [ディレクトリのダウンロード](#)

1つのファイルのダウンロード

TransferManager の download メソッドを使用して、ダウンロードするオブジェクトが含まれている Amazon S3 バケット名、キー (オブジェクト) 名、およびローカルシステムで作成するファイルを表す [ファイル](#) オブジェクトを渡します。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.Download;
import com.amazonaws.services.s3.transfer.MultipleFileDownload;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;

import java.io.File;
```

Code

```
File f = new File(file_path);
TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
```

```
try {
    Download xfer = xfer_mgr.download(bucket_name, key_name, f);
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    // or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

`waitForCompletion` を使用して、転送が正常に完了した後に `TransferManager` の `shutdownNow` メソッドを呼び出すことの詳細については、[転送の完了の待機](#) を参照してください。転送の完了を待ちながら、転送の状態や進行状況に関する更新情報をポーリングまたはリスンできます。詳細については、「[転送の状態と進行状況の取得](#)」を参照してください。

[GitHub](#) で完全な例をご覧ください。

ディレクトリのダウンロード

同じキープレフィックス (ファイルシステムのディレクトリに相当) を共有するファイルのセットを Amazon S3 からダウンロードするには、`TransferManagerdownloadDirectory` メソッドを呼び出します。このメソッドに、ダウンロードするオブジェクトが含まれている Amazon S3 バケットの名前、すべてのオブジェクトに共有されているオブジェクトプレフィックス、およびローカルシステムにファイルをダウンロードする先のディレクトリを表す [ファイル](#) オブジェクトを渡します。指定したディレクトリがまだない場合は、自動的に作成されます。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.Download;
import com.amazonaws.services.s3.transfer.MultipleFileDownload;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;

import java.io.File;
```

Code

```
TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
```

```
try {
    MultipleFileDownload xfer = xfer_mgr.downloadDirectory(
        bucket_name, key_prefix, new File(dir_path));
    // loop with Transfer.isDone()
    XferMgrProgress.showTransferProgress(xfer);
    // or block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

`waitForCompletion` を使用して、転送が正常に完了した後に `TransferManager` の `shutdownNow` メソッドを呼び出すことの詳細については、[転送の完了の待機](#) を参照してください。転送の完了を待ちながら、転送の状態や進行状況に関する更新情報をポーリングまたはリスンできます。詳細については、「[転送の状態と進行状況の取得](#)」を参照してください。

[GitHub](#) で完全な例をご覧ください。

オブジェクトのコピー

S3 バケット間でオブジェクトをコピーするには、`TransferManager.copy` メソッドを使用します。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.s3.transfer.Copy;
import com.amazonaws.services.s3.transfer.TransferManager;
import com.amazonaws.services.s3.transfer.TransferManagerBuilder;
```

Code

```
System.out.println("Copying s3 object: " + from_key);
System.out.println("    from bucket: " + from_bucket);
System.out.println("    to s3 object: " + to_key);
System.out.println("    in bucket: " + to_bucket);

TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
try {
    Copy xfer = xfer_mgr.copy(from_bucket, from_key, to_bucket, to_key);
```

```
// loop with Transfer.isDone()
XferMgrProgress.showTransferProgress(xfer);
// or block with Transfer.waitForCompletion()
XferMgrProgress.waitForCompletion(xfer);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

[GitHub](#) で完全な例をご覧ください。

転送が完了するまで待つ

転送が完了するまでアプリケーション (またはスレッド) がブロックできる場合、[Transfer](#) インターフェイスの `waitForCompletion` メソッドを使用して転送が完了するまでブロックします。ブロックしないと、例外が発生します。

```
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);
}
```

を呼び出す前に `waitForCompletion` イベントをポーリングする場合、別個のスレッドにポーリング機構を実装する場合、または [ProgressListener](#) を使用して非同期的に進行状況の更新を受け取る場合は、転送の進行状況が取得されます。

[GitHub](#) で完全な例をご覧ください。

転送の状態および進行状況の取得

`TransferManager.upload*`、`download*`、`copy` メソッドから返される各クラスは、1つのファイルまたは複数のファイルの操作であるかどうかに応じて、以下のクラスのいずれかのインスタンスを返します。

Class	返すクラス
[Copy] (コピー)	copy
ダウンロード	download
MultipleFileDownload	downloadDirectory
アップロード	upload
MultipleFileUpload	uploadFileList , uploadDirectory

これらすべてのクラスは、[Transfer](#) インターフェイスを実装します。Transfer は、転送の進行状況の取得や転送の一時停止/再開、および、転送の現在や最終ステータスを取得するのに役立つメソッドを提供します。

トピック

- [転送の現在の進行状況のポーリング](#)
- [ProgressListener による転送の進行状況の取得](#)
- [サブ転送の進行状況の取得](#)

転送の現在の進行状況のポーリング

このループでは、転送の進行状況を出力し、実行時は現在の進行状況を確認し、完了時は最終の状態を出力します。

インポート

```
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.event.ProgressEvent;
import com.amazonaws.event.ProgressListener;
import com.amazonaws.services.s3.transfer.*;
import com.amazonaws.services.s3.transfer.Transfer.TransferState;

import java.io.File;
import java.util.ArrayList;
import java.util.Collection;
```


Code

```
// print the transfer's human-readable description
System.out.println(xfer.getDescription());
// print an empty progress bar...
printProgressBar(0.0);
// update the progress bar while the xfer is ongoing.
do {
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        return;
    }
    // Note: so_far and total aren't used, they're just for
    // documentation purposes.
    TransferProgress progress = xfer.getProgress();
    long so_far = progress.getBytesTransferred();
    long total = progress.getTotalBytesToTransfer();
    double pct = progress.getPercentTransferred();
    eraseProgressBar();
    printProgressBar(pct);
} while (xfer.isDone() == false);
// print the final state of the transfer.
TransferState xfer_state = xfer.getState();
System.out.println(": " + xfer_state);
```

[GitHub](#) で完全な例をご覧ください。

ProgressListener による転送の進行状況の取得

[Transfer](#) インターフェイスの `addProgressListener` メソッドを使用して、[ProgressListener](#) を任意の転送にアタッチできます。

[ProgressListener](#) は、メソッドとして `progressChanged` だけを必要とし、このメソッドに [ProgressEvent](#) オブジェクトを渡します。このオブジェクトでは、その `getBytes` メソッドを呼び出してオペレーションの総バイト数を取得できます。また、`getBytesTransferred` を呼び出してそれまでに転送されたバイト数を取得できます。

インポート

```
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
```

```
import com.amazonaws.event.ProgressEvent;
import com.amazonaws.event.ProgressListener;
import com.amazonaws.services.s3.transfer.*;
import com.amazonaws.services.s3.transfer.Transfer.TransferState;

import java.io.File;
import java.util.ArrayList;
import java.util.Collection;
```

Code

```
File f = new File(file_path);
TransferManager xfer_mgr = TransferManagerBuilder.standard().build();
try {
    Upload u = xfer_mgr.upload(bucket_name, key_name, f);
    // print an empty progress bar...
    printProgressBar(0.0);
    u.addProgressListener(new ProgressListener() {
        public void progressChanged(ProgressEvent e) {
            double pct = e.getBytesTransferred() * 100.0 / e.getBytes();
            eraseProgressBar();
            printProgressBar(pct);
        }
    });
    // block with Transfer.waitForCompletion()
    XferMgrProgress.waitForCompletion(u);
    // print the final state of the transfer.
    TransferState xfer_state = u.getState();
    System.out.println(": " + xfer_state);
} catch (AmazonServiceException e) {
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
xfer_mgr.shutdownNow();
```

[GitHub](#) で完全な例をご覧ください。

サブ転送の進行状況の取得

[MultipleFileUpload](#) クラスは、その `getSubTransfers` メソッドを呼び出してサブ転送に関する情報を返すことができます。[アップロードオブジェクトの変更不能なコレクション](#)を返して、サブ転送ごとの転送の状態と進行状況を提供します。

インポート

```
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.event.ProgressEvent;
import com.amazonaws.event.ProgressListener;
import com.amazonaws.services.s3.transfer.*;
import com.amazonaws.services.s3.transfer.Transfer.TransferState;

import java.io.File;
import java.util.ArrayList;
import java.util.Collection;
```

Code

```
Collection<? extends Upload> sub_xfers = new ArrayList<Upload>();
sub_xfers = multi_upload.getSubTransfers();

do {
    System.out.println("\nSubtransfer progress:\n");
    for (Upload u : sub_xfers) {
        System.out.println("  " + u.getDescription());
        if (u.isDone()) {
            TransferState xfer_state = u.getState();
            System.out.println("  " + xfer_state);
        } else {
            TransferProgress progress = u.getProgress();
            double pct = progress.getPercentTransferred();
            printProgressBar(pct);
            System.out.println();
        }
    }

    // wait a bit before the next update.
    try {
        Thread.sleep(200);
    } catch (InterruptedException e) {
        return;
    }
} while (multi_upload.isDone() == false);
// print the final state of the transfer.
TransferState xfer_state = multi_upload.getState();
System.out.println("\nMultipleFileUpload " + xfer_state);
```

[GitHub](#) で完全な例をご覧ください。

詳細

- Amazon Simple Storage Service ユーザーガイドの [オブジェクトキー](#)

ウェブサイトとしての Amazon S3 バケットの設定

Amazon S3 バケットを、ウェブサイトのように機能させるよう設定できます。これを行うには、ウェブサイト設定をセットする必要があります。

Note

これらのコード例では、ユーザーが [AWS SDK for Java の使用](#) の内容を理解し、[開発用の AWS 認証情報とリージョンのセットアップ](#) の情報を使用してデフォルトの AWS 認証情報を設定していることを前提としています。

バケットのウェブサイト設定をセットする

Amazon S3 バケットのウェブサイト設定をセットするには、AmazonS3 の `setWebsiteConfiguration` メソッドを設定するバケット名で呼び出し、バケットのウェブサイト設定が含まれる [BucketWebsiteConfiguration](#) オブジェクトを呼び出します。

インデックسدキュメントの設定は必要ですが、他のすべてのパラメータはオプションです。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.BucketWebsiteConfiguration;
```

Code

```
String bucket_name, String index_doc, String error_doc) {
    BucketWebsiteConfiguration website_config = null;
```

```
if (index_doc == null) {
    website_config = new BucketWebsiteConfiguration();
} else if (error_doc == null) {
    website_config = new BucketWebsiteConfiguration(index_doc);
} else {
    website_config = new BucketWebsiteConfiguration(index_doc, error_doc);
}

final AmazonS3 s3 =
    AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    s3.setBucketWebsiteConfiguration(bucket_name, website_config);
} catch (AmazonServiceException e) {
    System.out.format(
        "Failed to set website configuration for bucket '%s'!\n",
        bucket_name);
    System.err.println(e.getMessage());
    System.exit(1);
}
```

Note

ウェブサイト設定をセットしても、バケットのアクセス権限は変更されません。ウェブ上でファイルが表示されるようにするには、バケットのファイルにパブリック読み取りアクセスを許可するバケットポリシーも設定する必要があります。詳細については、[バケットポリシーを使用した Amazon S3 バケットへのアクセス管理](#)を参照してください。

[GitHub](#) で完全な例をご覧ください。

バケットのウェブサイト設定を取得する

Amazon S3 バケットのウェブサイト設定を取得するには、AmazonS3 の `getWebsiteConfiguration` メソッドを、設定を取得するバケットの名前で呼び出します。

この設定は [BucketWebsiteConfiguration](#) オブジェクトとして返されます。バケットのウェブサイト設定がない場合は、`null` が返されます。

インポート

```
import com.amazonaws.AmazonServiceException;
```

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.BucketWebsiteConfiguration;
```

Code

```
final AmazonS3 s3 =
    AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    BucketWebsiteConfiguration config =
        s3.getBucketWebsiteConfiguration(bucket_name);
    if (config == null) {
        System.out.println("No website configuration found!");
    } else {
        System.out.format("Index document: %s\n",
            config.getIndexDocumentSuffix());
        System.out.format("Error document: %s\n",
            config.getErrorDocument());
    }
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.out.println("Failed to get website configuration!");
    System.exit(1);
}
```

[GitHub](#) で完全な例をご覧ください。

バケットのウェブサイト設定を削除する

Amazon S3 バケットのウェブサイト設定を削除するには、AmazonS3 の `deleteWebsiteConfiguration` メソッドを、設定を削除するバケットの名前で呼び出します。

インポート

```
import com.amazonaws.AmazonServiceException;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
```

Code

```
final AmazonS3 s3 =
    AmazonS3ClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
try {
    s3.deleteBucketWebsiteConfiguration(bucket_name);
} catch (AmazonServiceException e) {
    System.err.println(e.getMessage());
    System.out.println("Failed to delete website configuration!");
    System.exit(1);
}
```

[GitHub](#) で完全な例をご覧ください。

詳細情報

- Amazon S3 API リファレンスの [PUT Bucket ウェブサイト](#)
- Amazon S3 API リファレンスの [GET Bucket ウェブサイト](#)
- Amazon S3 API リファレンスの [DELETE Bucket ウェブサイト](#)

Amazon S3 クライアント側の暗号化を使用する

Amazon S3 暗号化クライアントを使用したデータの暗号化は、Amazon S3 に保存された機密情報に保護レイヤーを追加するための 1 つの方法です。このセクションの例では、アプリケーションの Amazon S3 暗号化クライアントを作成および設定する方法を示しています。

暗号化を初めて使用する場合は、AWS KMS デベロッパーガイドの[暗号化の基礎](#)で暗号化の用語やアルゴリズムの基本的な概要を参照してください。AWS SDK 全体の暗号化のサポートの詳細については、「Amazon Web Services 全般のリファレンス」の「[AWS SDK Support for Amazon S3 Client-Side Encryption](#)」を参照してください。

Note

これらのコード例では、ユーザーが [AWS SDK for Java の使用](#) の内容を理解し、[開発用の AWS 認証情報とリージョンのセットアップ](#) の情報を使用してデフォルトの AWS 認証情報を設定していることを前提としています。

AWS SDK for Java のバージョン 1.11.836 以前を使用している場合は、アプリケーションをそれ以降のバージョンに移行する方法について、「[Amazon S3 Encryption Client Migration](#)」を参照してください。移行できない場合は、GitHub で [この完全な例](#) を参照してください。

それ以外の場合であって、AWS SDK for Java のバージョン 1.11.837 以降を使用しているときは、以下にリストされているトピックの例を調べて、Amazon S3 クライアント側の暗号化を使用してください。

トピック

- [クライアントマスターキーを使用した Amazon S3 クライアント側の暗号化](#)
- [AWS KMS マネージドキーを使用した Amazon S3 クライアント側の暗号化](#)

クライアントマスターキーを使用した Amazon S3 クライアント側の暗号化

次の例では、[AmazonS3EncryptionClientV2Builder](#) クラスを使用して、クライアント側の暗号化が有効になった Amazon S3 クライアントを作成します。有効にすると、このクライアントを使用して Amazon S3 にアップロードするすべてのオブジェクトが暗号化されます。このクライアントを使用して Amazon S3 から取得したオブジェクトは、自動的に復号化されます。

Note

次の例では、カスタマー管理のクライアントマスターキーを使用した Amazon S3 クライアント側の暗号化の使用方法を説明します。AWS KMS マネージドの暗号化キーを使用する方法については、「[Amazon S3 client-side encryption with AWS KMS managed keys](#)」を参照してください。

クライアント側の Amazon S3 暗号化を有効にする際に、厳格な認証済みまたは認証済みの 2 つの暗号化モードから選択できます。以下のセクションで、各タイプを有効にする方法を説明します。各モードで使用されるアルゴリズムについては、[CryptoMode](#) の定義を参照してください。

必須のインポート

これらの例では、次のクラスをインポートします。

インポート

```
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;
```

厳格な認証済み暗号化

厳密に認証された暗号化は、CryptoMode が指定されていない場合のデフォルトのモードです。

このモードを明示的に有効にするには、StrictAuthenticatedEncryption 値を withCryptoConfiguration メソッドに指定します。

Note

クライアント側で認証済み暗号化を使用するには、最新の [Bouncy Castle jar](#) ファイルをアプリケーションのクラスパスに含める必要があります。

Code

```
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");
```

認証済み暗号化モード

AuthenticatedEncryption モードを使用すると、暗号化中に適用されるキーのラップアルゴリズムが強化されます。このモードで復号化するときは、アルゴリズムによって復号化されたオブジェクトの整合性が検証され、チェックが失敗した場合は例外がスローされます。認証済み暗号化の動作の詳細については、「[Amazon S3 Client-Side Authenticated Encryption](#)」というブログ記事を参照してください。

Note

クライアント側で認証済み暗号化を使用するには、最新の [Bouncy Castle jar](#) ファイルをアプリケーションのクラスパスに含める必要があります。

このモードを有効にするには、AuthenticatedEncryption 値を withCryptoConfiguration メソッドに指定します。

Code

```
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");
```

AWS KMS マネージドキーを使用した Amazon S3 クライアント側の暗号化

次の例では、[AmazonS3EncryptionClientV2Builder](#) クラスを使用して、クライアント側の暗号化が有効になった Amazon S3 クライアントを作成します。この設定を行うと、このクライアントを使用して Amazon S3 にアップロードするすべてのオブジェクトが暗号化されます。このクライアントを使用して Amazon S3 から取得したオブジェクトは、自動的に復号化されます。

Note

次の例では、AWS KMS マネージドキーを使用した Amazon S3 クライアント側の暗号化の使用方法を説明します。独自の暗号化キーを使用する方法については、「[Amazon S3 client-side encryption with client master keys](#)」を参照してください。

クライアント側の Amazon S3 暗号化を有効にする際に、厳格な認証済みまたは認証済みの 2 つの暗号化モードから選択できます。以下のセクションで、各タイプを有効にする方法を説明します。各モードで使用されるアルゴリズムについては、[CryptoMode](#) の定義を参照してください。

必須のインポート

これらの例では、次のクラスをインポートします。

インポート

```
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;
```

厳格な認証済み暗号化

厳密に認証された暗号化は、`CryptoMode` が指定されていない場合のデフォルトのモードです。

このモードを明示的に有効にするには、`StrictAuthenticatedEncryption` 値を `withCryptoConfiguration` メソッドに指定します。

Note

クライアント側で認証済み暗号化を使用するには、最新の [Bouncy Castle jar](#) ファイルをアプリケーションのクラスパスに含める必要があります。

Code

```
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));
```

Amazon S3 暗号化クライアントで `putObject` メソッドを呼び出して、オブジェクトをアップロードします。

Code

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

同じクライアントを使用してオブジェクトを取得できます。この例では、`getObjectAsString` メソッドを使用して保存された文字列を取得しています。

Code

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

認証済み暗号化モード

AuthenticatedEncryption モードを使用すると、暗号化中に適用されるキーのラップアルゴリズムが強化されます。このモードで復号化するときは、アルゴリズムによって復号化されたオブジェクトの整合性が検証され、チェックが失敗した場合は例外がスローされます。認証済み暗号化の動作の詳細については、「[Amazon S3 Client-Side Authenticated Encryption](#)」というブログ記事を参照してください。

Note

クライアント側で認証済み暗号化を使用するには、最新の [Bouncy Castle jar](#) ファイルをアプリケーションのクラスパスに含める必要があります。

このモードを有効にするには、AuthenticatedEncryption 値を `withCryptoConfiguration` メソッドに指定します。

Code

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

AWS KMS クライアントの設定

Amazon S3 暗号化クライアントは、明示的に指定されていない限り、デフォルトで AWS KMS クライアントを作成します。

この自動作成された AWS KMS クライアントのリージョンを設定するには、`awsKmsRegion` を設定します。

Code

```
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();
```

または、独自の AWS KMS クライアントを使用して暗号化クライアントを初期化することもできます。

Code

```
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();
```

Amazon SQSAWS SDK for Java を使用した例

このセクションでは、[AWS SDK for Java](#) を使用して [Amazon SQS](#) をプログラムする例を示します。

Note

例には各手法を示すのに必要なコードのみが含まれます。[完全なサンプルコードは GitHub で入手できます](#)。そこから、単一のソースファイルをダウンロードするかリポジトリをローカルにクローン作成して、ビルドし実行するためのすべての例を取得できます。

トピック

- [Amazon SQS メッセージキューの使用](#)
- [Amazon SQS メッセージの送信、受信、削除](#)
- [Amazon SQS メッセージキューのロングポーリングの有効化](#)
- [Amazon SQS で可視性タイムアウトを設定する](#)
- [Amazon SQS でのデッドレターキューの使用](#)

Amazon SQS メッセージキューの使用

メッセージキューは、Amazon SQS でメッセージを確実に送信するために使用する論理コンテナです。キューには、標準と先入れ先出し (FIFO) の 2 種類があります。キューおよびキュータイプ間の相違点の詳細については、「[Amazon SQS デベロッパーガイド](#)」を参照してください。

このトピックでは、AWS SDK for Java を使用して Amazon SQS キューの URL の作成、一覧表示、削除、および取得を行う方法について説明します。

キューの作成

AmazonSQS クライアントの `createQueue` メソッドを使用し、キューのパラメータを記述する [CreateQueueRequest](#) オブジェクトを渡します。

インポート

```
import com.amazonaws.services.sqs.AmazonSQS;  
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;  
import com.amazonaws.services.sqs.model.AmazonSQSException;  
import com.amazonaws.services.sqs.model.CreateQueueRequest;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
```

```
CreateQueueRequest create_request = new CreateQueueRequest(QueueName)
    .addAttributesEntry("DelaySeconds", "60")
    .addAttributesEntry("MessageRetentionPeriod", "86400");

try {
    sqs.createQueue(create_request);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}
```

キュー名だけを必要とする簡略化された形式の `createQueue` を使用して、標準キューを作成できます。

```
sqs.createQueue("MyQueue" + new Date().getTime());
```

[GitHub](#) で完全な例をご覧ください。

キューの一覧表示

アカウントの Amazon SQS キューを一覧表示するには、AmazonSQS クライアントの `listQueues` メソッドを呼び出します。

インポート

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

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
ListQueuesResult lq_result = sqs.listQueues();
System.out.println("Your SQS Queue URLs:");
for (String url : lq_result.getQueueUrls()) {
    System.out.println(url);
}
```

パラメータなしで `listQueues` オーバーロードを使用すると、すべてのキューが返されます。返された結果は、`ListQueuesResult` オブジェクトに渡すことでフィルタできます。

インポート

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

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
String name_prefix = "Queue";
lq_result = sqs.listQueues(new ListQueuesRequest(name_prefix));
System.out.println("Queue URLs with prefix: " + name_prefix);
for (String url : lq_result.getQueueUrls()) {
    System.out.println(url);
}
```

[GitHub](#) で完全な例をご覧ください。

キューの URL の取得

AmazonSQS クライアントの `getQueueUrl` メソッドを呼び出します。

インポート

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

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
String queue_url = sqs.getQueueUrl(QueueName).getQueueUrl();
```

[GitHub](#) で完全な例をご覧ください。

キューの削除

キューの [URL](#) を AmazonSQS クライアントの `deleteQueue` メソッドに渡します。

インポート

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


Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
sqs.deleteQueue(queue_url);
```

[GitHub](#) で完全な例をご覧ください。

詳細

- Amazon SQS デベロッパーガイドの [Amazon SQS キューの仕組み](#)
- Amazon SQS API リファレンスの [CreateQueue](#)
- Amazon SQS API リファレンスの [GetQueueUrl](#)
- Amazon SQS API リファレンスの [ListQueues](#)
- Amazon SQS API リファレンスの [DeleteQueues](#)

Amazon SQS メッセージの送信、受信、削除

このトピックでは、Amazon SQS メッセージを送信、受信、削除する方法について説明します。メッセージは、常に [SQS キュー](#) を使用して提供されます。

メッセージの送信

Amazon SQS キューに 1 つのメッセージ追加するには、AmazonSQS クライアントの `sendMessage` メソッドを呼び出します。キューの [URL](#)、メッセージ本文、およびオプションの遅延値 (秒単位) が含まれる [SendMessageRequest](#) オブジェクトを指定します。

インポート

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

Code

```
SendMessageRequest send_msg_request = new SendMessageRequest()
    .withQueueUrl(queueUrl)
    .withMessageBody("hello world")
    .withDelaySeconds(5);
sqs.sendMessage(send_msg_request);
```

[GitHub](#) で完全な例をご覧ください。

一度に複数のメッセージを送信する

複数のメッセージを 1 回のリクエストで送信できます。複数のメッセージを送信するには、AmazonSQS クライアントの `sendMessageBatch` メソッドを呼び出して、キュー URL と送信するメッセージのリスト (各メッセージが [SendMessageBatchRequestEntry](#)) を含む [SendMessageBatchRequest](#) を渡します。メッセージごとにオプションの遅延値を設定することもできます。

インポート

```
import com.amazonaws.services.sqs.model.SendMessageBatchRequest;
import com.amazonaws.services.sqs.model.SendMessageBatchRequestEntry;
```

Code

```
SendMessageBatchRequest send_batch_request = new SendMessageBatchRequest()
    .withQueueUrl(queueUrl)
    .withEntries(
        new SendMessageBatchRequestEntry(
            "msg_1", "Hello from message 1"),
        new SendMessageBatchRequestEntry(
            "msg_2", "Hello from message 2")
            .withDelaySeconds(10));
sqs.sendMessageBatch(send_batch_request);
```

[GitHub](#) で完全な例をご覧ください。

メッセージを受信する

キューに現在含まれているメッセージを取得するには、AmazonSQS クライアントの `receiveMessage` メソッドを呼び出して、キューの URL を渡します。メッセージは、[Message](#) オブジェクトのリストとして返されます。

インポート

```
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.AmazonSQSException;
import com.amazonaws.services.sqs.model.SendMessageBatchRequest;
```

Code

```
List<Message> messages = sqs.receiveMessage(queueUrl).getMessages();
```

受信後にメッセージを削除する

メッセージを受信し、その内容を処理した後で、メッセージをキューから削除するには、メッセージの受信ハンドルとキュー URL を AmazonSQS クライアントの `deleteMessage` メソッドに送信します。

Code

```
for (Message m : messages) {
    sqs.deleteMessage(queueUrl, m.getReceiptHandle());
}
```

[GitHub](#) で完全な例をご覧ください。

詳細

- Amazon SQS デベロッパーガイドの [Amazon SQS キューの仕組み](#)
- Amazon SQS API リファレンスの [SendMessage](#)
- Amazon SQS API リファレンスの [SendMessageBatch](#)
- Amazon SQS API リファレンスの [ReceiveMessage](#)
- Amazon SQS API リファレンスの [DeleteMessage](#)

Amazon SQS メッセージキューのロングポーリングの有効化

Amazon SQS はデフォルトでショートポーリングを使用して、サーバーのサブセットだけに対して (重み付けされたランダムディストリビューションに基づいて) クエリを実行し、レスポンスに含めることができるメッセージがあるかどうかを調べます。

ロングポーリングは、Amazon SQS キューに送信された `ReceiveMessage` リクエストに応答して返信するメッセージがない場合に、偽の空のレスポンスを排除して空のレスポンスの数を減らすことで、Amazon SQS の使用コストを削減します。

Note

1 ~ 20 秒でロングポーリング頻度を設定できます。

キューの作成時にロングポーリングを有効化する

Amazon SQS キューを作成するときにロングポーリングを有効にするには、AmazonSQS クラスの `createQueue` メソッドを呼び出す前に [CreateQueueRequest](#) オブジェクトの `ReceiveMessageWaitTimeSeconds` 属性を設定します。

インポート

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.AmazonSQSException;
import com.amazonaws.services.sqs.model.CreateQueueRequest;
```

Code

```
final AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();

// Enable long polling when creating a queue
CreateQueueRequest create_request = new CreateQueueRequest()
    .withQueueName(queue_name)
    .addAttributesEntry("ReceiveMessageWaitTimeSeconds", "20");

try {
    sqs.createQueue(create_request);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}
```

[GitHub](#) で完全な例をご覧ください。

既存のキューでロングポーリングを有効にする

キューを作成するときにロングポーリングを有効にすることに加えて、AmazonSQS クラスの `setQueueAttributes` メソッドを呼び出す前に、[SetQueueAttributesRequest](#) で `ReceiveMessageWaitTimeSeconds` を設定することで既存のキューでも有効にできます。

インポート

```
import com.amazonaws.services.sqs.model.SetQueueAttributesRequest;
```

Code

```
SetQueueAttributesRequest set_attrs_request = new SetQueueAttributesRequest()
    .withQueueUrl(queue_url)
    .addAttributesEntry("ReceiveMessageWaitTimeSeconds", "20");
sqs.setQueueAttributes(set_attrs_request);
```

[GitHub](#) で完全な例をご覧ください。

メッセージ受信時のロングポーリングを有効にする

メッセージを受信した時にロングポーリングを有効にするには、AmazonSQS クラスの `receiveMessage` メソッドに提供する [ReceiveMessageRequest](#) の待機時間を秒単位で設定します。

Note

次のポーリングイベントの待機中に `receiveMessage` リクエストがタイムアウトしないように、AWS クライアントのリクエストのタイムアウトがロングポーリングの最大値 (20 秒) より長いことを確認します。

インポート

```
import com.amazonaws.services.sqs.model.ReceiveMessageRequest;
```

Code

```
ReceiveMessageRequest receive_request = new ReceiveMessageRequest()
    .withQueueUrl(queue_url)
    .withWaitTimeSeconds(20);
sqs.receiveMessage(receive_request);
```

[GitHub](#) で完全な例をご覧ください。

詳細

- Amazon SQS デベロッパーガイドの [Amazon SQS ロングポーリング](#)
- Amazon SQS API リファレンスの [CreateQueue](#)

- Amazon SQS API リファレンスの [ReceiveMessage](#)
- Amazon SQS API リファレンスの [SetQueueAttributes](#)

Amazon SQS で可視性タイムアウトを設定する

Amazon SQS でメッセージを受信すると、受信確認のために、削除されるまでキューに残ります。削除されなかった受信メッセージは、指定された可視性タイムアウトの後に以降のリクエストで使用でき、メッセージが処理および削除される前に複数回受信することを防ぎます。

Note

[標準キュー](#)を使用している場合、可視性タイムアウトはメッセージを 2 回受信しない保証にはなりません。標準キューを使用している場合は、同じメッセージが複数回配信されるケースをコードが処理できることを確認してください。

単一のメッセージのメッセージ可視性タイムアウトを設定する

メッセージを受信したとき、渡したい [ChangeMessageVisibilityRequest](#) の受信ハンドルを AmazonSQS クラスの `changeMessageVisibility` メソッドに渡すことで、可視性タイムアウトを変更することができます。

インポート

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

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();  
  
// Get the receipt handle for the first message in the queue.  
String receipt = sqs.receiveMessage(queue_url)  
    .getMessages()  
    .get(0)  
    .getReceiptHandle();  
  
sqs.changeMessageVisibility(queue_url, receipt, timeout);
```

[GitHub](#) で完全な例をご覧ください。

複数のメッセージのメッセージ可視性タイムアウトを同時に設定する

複数のメッセージのメッセージ可視性タイムアウトを設定するには、それぞれに一意の ID 文字列と受信ハンドルを含む [ChangeMessageVisibilityBatchRequestEntry](#) オブジェクトのリストを作成します。次に、リストを Amazon SQS クライアントクラスの `changeMessageVisibilityBatch` メソッドに渡します。

インポート

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.ChangeMessageVisibilityBatchRequestEntry;
import java.util.ArrayList;
import java.util.List;
```

Code

```
AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();

List<ChangeMessageVisibilityBatchRequestEntry> entries =
    new ArrayList<ChangeMessageVisibilityBatchRequestEntry>();

entries.add(new ChangeMessageVisibilityBatchRequestEntry(
    "unique_id_msg1",
    sqs.receiveMessage(queue_url)
        .getMessages()
        .get(0)
        .getReceiptHandle())
    .withVisibilityTimeout(timeout));

entries.add(new ChangeMessageVisibilityBatchRequestEntry(
    "unique_id_msg2",
    sqs.receiveMessage(queue_url)
        .getMessages()
        .get(0)
        .getReceiptHandle())
    .withVisibilityTimeout(timeout + 200));

sqs.changeMessageVisibilityBatch(queue_url, entries);
```

[GitHub](#) で完全な例をご覧ください。

詳細

- Amazon SQS デベロッパーガイドの [可視性タイムアウト](#)
- Amazon SQS API リファレンスの [SetQueueAttributes](#)
- Amazon SQS API リファレンスの [GetQueueAttributes](#)
- Amazon SQS API リファレンスの [ReceiveMessage](#)
- Amazon SQS API リファレンスの [ChangeMessageVisibility](#)
- Amazon SQS API リファレンスの [ChangeMessageVisibilityBatch](#)

Amazon SQS でのデッドレターキューの使用

Amazon SQS では、デッドレターキューがサポートされます。デッドレターキューは、正常に処理できないメッセージの送信先として他の (送信元) キューが使用できるキューです。これらのメッセージは、処理が成功しなかった理由を判断するためにデッドレターキューに分離できます。

デッドレターキューの作成

デッドレターキューは、通常のキューと同じ方法で作成されますが、次の制限があります。

- デッドレターキューは、ソースキューと同じタイプのキュー (FIFO または標準) である必要があります。
- デッドレターキューは、ソースキューと同じ AWS アカウント およびリージョンを使用して作成する必要があります。

ここで 2 つの同一の Amazon SQS キューを作成し、そのうちの 1 つがデッドレターキューとして機能します。

インポート

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

Code

```
final AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
```



```
// Create source queue
try {
    sqs.createQueue(src_queue_name);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}

// Create dead-letter queue
try {
    sqs.createQueue(dl_queue_name);
} catch (AmazonSQSException e) {
    if (!e.getErrorCode().equals("QueueAlreadyExists")) {
        throw e;
    }
}
```

[GitHub](#) で完全な例をご覧ください。

ソースキューに対するデッドレターキューの指定

デッドレターキューを指定するには、まず再処理ポリシーを作成し、次にキューの属性でそのポリシーを設定します。再処理ポリシーは JSON で指定され、デッドレターキューの ARN、およびメッセージがデッドレターキューに送信される前に受信できて処理できない最大回数を指定します。

ソースキューに再処理ポリシーを設定するには、JSON 再処理ポリシーで `RedrivePolicy` 属性を設定した [SetQueueAttributesRequest](#) オブジェクトを使用して、AmazonSQS クラスの `setQueueAttributes` メソッドを呼び出します。

インポート

```
import com.amazonaws.services.sqs.model.GetQueueAttributesRequest;
import com.amazonaws.services.sqs.model.GetQueueAttributesResult;
import com.amazonaws.services.sqs.model.SetQueueAttributesRequest;
```

Code

```
String dl_queue_url = sqs.getQueueUrl(dl_queue_name)
    .getQueueUrl();
```

```
GetQueueAttributesResult queue_attrs = sqs.getQueueAttributes(  
    new GetQueueAttributesRequest(dl_queue_url)  
        .withAttributeNames("QueueArn"));  
  
String dl_queue_arn = queue_attrs.getAttributes().get("QueueArn");  
  
// Set dead letter queue with redrive policy on source queue.  
String src_queue_url = sqs.getQueueUrl(src_queue_name)  
    .getQueueUrl();  
  
SetQueueAttributesRequest request = new SetQueueAttributesRequest()  
    .withQueueUrl(src_queue_url)  
    .addAttributesEntry("RedrivePolicy",  
        "{\n\"maxReceiveCount\":\n\"5\", \n\"deadLetterTargetArn\":\n\"\"  
+ dl_queue_arn + \"\"}");  
  
sqs.setQueueAttributes(request);
```

[GitHub](#) で完全な例をご覧ください。

詳細

- Amazon SQS デベロッパーガイドでの [Amazon SQS デッドレターキューの使用](#)
- Amazon SQS API リファレンスの [SetQueueAttributes](#)

Amazon SWFAWS SDK for Java を使用した例

[Amazon SWF](#) は、アクティビティ、子ワークフロー、または [Lambda](#) タスクで構成される並列またはシーケンシャルステップを持つことができる分散ワークフローをデベロッパーが構築およびスケールできるようにする、ワークフロー管理サービスです。

AWS SDK for Java を使用して Amazon SWF を操作する方法は 2 つあります。SWF クライアントオブジェクトを使用する方法と、AWS Flow Framework for Java を使用する方法です。AWS Flow Framework for Java は注釈を多用し、AspectJ や Spring Framework などの追加のライブラリに依存するため、初期設定がより難しくなります。ただし、大量または複雑なプロジェクトでは、AWS Flow Framework for Java を使用すると時間を節約できます。詳細については、「[AWS Flow Framework for Java デベロッパーガイド](#)」を参照してください。

このセクションでは、AWS SDK for Java クライアントを直接使用することにより、Amazon SWF をプログラミングする例を示します。

トピック

- [SWF の基本](#)
- [シンプルな Amazon SWF アプリケーションの構築](#)
- [Lambda タスク](#)
- [アクティビティおよびワークフローワーカーの適切なシャットダウン](#)
- [ドメインの登録](#)
- [ドメインの一覧表示](#)

SWF の基本

以下は、AWS SDK for Java を使用して Amazon SWF を操作する一般的なパターンです。主に参照用です。より詳細な入門チュートリアルについては、[シンプルな Amazon SWF アプリケーションの構築](#)を参照してください。

依存関係

基本的な Amazon SWF アプリケーションでは、次の依存関係が必要です (AWS SDK for Java に含まれています)。

- 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

これらのパッケージのバージョン番号はお手持ちの SDK のバージョンによって異なりますが、SDK で提供するバージョンは互換性についてテスト済みで、使用するバージョンです。

AWS Flow Framework for Java アプリケーションでは、追加のセットアップおよび追加の依存関係が必要です。フレームワークの使用の詳細については、「[AWS Flow Framework for Java デベロッパーガイド](#)」を参照してください。

インポート

一般的に、コード開発には次のインポートを使用できます。

```
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.*;
```

ただし、必要なクラスのみをインポートすることをお勧めします。com.amazonaws.services.simpleworkflow.model ワークスペースで、特定のクラスを指定することになる可能性があります。

```
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;
```

AWS Flow Framework for Java を使用している場合

は、com.amazonaws.services.simpleworkflow.flow ワークスペースからクラスをインポートします。例:

```
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.flow.ActivityWorker;
```

Note

AWS Flow Framework for Java には、AWS SDK for Java の基本的な要件に加えて追加の要件があります。詳細については、「[AWS Flow Framework for Java デベロッパーガイド](#)」を参照してください。

SWF クライアントクラスの使用

Amazon SWF の基本的なインターフェイスは、[AmazonSimpleWorkflowClient](#) または [AmazonSimpleWorkflowAsyncClient](#) クラスを通じて提供されます。これらの主な違い

は、`*AsyncClient` クラスは同時 (非同期) プログラミング用に [Future](#) オブジェクトを返すことで

```
AmazonSimpleWorkflowClient swf = AmazonSimpleWorkflowClientBuilder.defaultClient();
```

シンプルな Amazon SWF アプリケーションの構築

このトピックでは、AWS SDK for Java を使用した [Amazon SWF](#) アプリケーションのプログラミングについて、重要な概念を示しながら説明します。

例について

例のプロジェクトでは、AWS クラウドを通じて渡されるワークフローデータを受け入れる 1 つのアクティビティでワークフローを作成し (HelloWorld では、いつものように、あいさつする相手の名前になります)、応答であいさつを出力します。

これは表面的には非常に単純に見えますが、Amazon SWF アプリケーションは連携して動作する数多くの部分で構成されます。

- ワークフロー実行データの論理コンテナとして使用されるドメイン。
- ワークフローのアクティビティと子ワークフローの実行の論理的順序を定義するコードコンポーネントを表す 1 つ以上のワークフロー。
- 決定タスクをポーリングし、それに応じてアクティビティまたは子ワークフローをスケジュールするワークフローワーカー (ディサイダーとも呼ばれる)。
- それぞれがワークフローのワークの単位を表す、1 つ以上のアクティビティ。
- アクティビティタスクをポーリングし、それに応じてアクティビティメソッドを実行するアクティビティワーカー。
- ワークフローとアクティビティワーカーにリクエストを発行するために使用される によって管理されるキューである、1 つ以上のタスクリスト Amazon SWF。ワークフローワーカーを想定したタスクリスト内のタスクは、決定タスクと呼ばれます。アクティビティワーカーを想定したものは、アクティビティタスクと呼ばれます。
- ワークフローの実行を開始するワークフロースターター。

バックグラウンドでは、Amazon SWF がこれらのコンポーネントの操作を調整して、AWS クラウドからのフローの連携、それらの間のデータの受け渡し、タイムアウトとハートビート通知の処理、およびワークフロー実行履歴のログ記録を行います。

これらの一時的な認証情報は、default プロファイルに関連付けられます。

SWF プロジェクトの作成

1. Maven での新しいプロジェクトの開始

```
mvn archetype:generate -DartifactId=helloswf \
-DgroupId=aws.example.helloswf -DinteractiveMode=false
```

これにより、標準の Maven プロジェクト構造を持つ新しいプロジェクトが作成されます。

```
helloswf
### pom.xml
### src
### main
#   ### java
#       ### aws
#           ### example
#               ### helloswf
#                   ### App.java
### test
### ...
```

test ディレクトリとそれに含まれるすべては、このチュートリアルでは使用しないため、無視または削除できます。また、新しいクラスに置き換えるため、App.java も削除できます。

2. <dependencies> ブロック内で依存関係を追加することで、プロジェクトの pom.xml ファイルを編集して aws-java-sdk-simpleworkflow モジュールを追加します。

```
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-simpleworkflow</artifactId>
    <version>1.11.1000</version>
  </dependency>
</dependencies>
```

3. Maven によって JDK 1.7 以降のサポートがあるプロジェクトが構築されることを確認します。プロジェクトの pom.xml に以下を追加します (<dependencies> ブロックの前または後)。

```
<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>
```

プロジェクトのコーディング

サンプルプロジェクトは 4 つの個別のアプリケーションで構成されます。それらについて 1 つずつ説明します。

- HelloTypes.java--プロジェクトのドメイン、アクティビティ、およびワークフロータイプデータが含まれ、他のコンポーネントと共有されます。また、SWF でのこれらのタイプの登録も処理されます。
- ActivityWorker.java--アクティビティタスクをポーリングし、それに応じてアクティビティを実行するアクティビティワーカーを含みます。
- WorkflowWorker.java--決定タスクをポーリングし、新しいアクティビティをスケジュールするワークフローワーカー (ディサイダー) を含みます。
- WorkflowStarter.java--新しいワークフローの実行を開始するワークフロースターターを含みます。ワークフロースターターにより、SWF はワーカーが使用する決定とワークフロータスクを生成します。

すべてのソースファイルに共通のステップ

Java クラスを格納するために作成するすべてのファイルには、いくつか共通の事柄があります。時間を節約するため、これらのステップはプロジェクトに新しいファイルを追加するたびに暗黙的に示されます。

1. プロジェクトの `src/main/java/aws/example/helloswf/` ディレクトリでファイルを作成します。
2. 各ファイルの先頭に `package` 宣言を追加して名前空間を宣言します。サンプルプロジェクトでは以下を使用します。


```
package aws.example.helloswf;
```

3. [AmazonSimpleWorkflowClient](#) クラスおよび

`com.amazonaws.services.simpleworkflow.model` 名前空間内の複数のクラスの `import` 宣言を追加します。作業を簡素化するため、以下を使用します。

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.*;
```

ドメイン、ワークフロー、およびアクティビティタイプの登録

最初に、新しい実行可能クラス `HelloTypes.java` を作成します。このファイルには、ワークフローの各部分必要となる共有データ (アクティビティとワークフロータイプの名前とバージョン、ドメイン名、タスクリスト名など) が含まれます。

1. テキストエディターを開き、ファイル `HelloTypes.java` を作成して、[共通のステップ](#)に従ってパッケージ宣言とインポートを追加します。
2. `HelloTypes` クラスを宣言し、登録されたアクティビティとワークフロータイプで使用する値を指定します。

```
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";
```

これらの値は、コード全体で使用されます。

3. 文字列を宣言したら、[AmazonSimpleWorkflowClient](#) クラスのインスタンスを作成します。これは、AWS SDK for Java で提供される Amazon SWF メソッドの基本インタフェースです。

```
private static final AmazonSimpleWorkflow swf =

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

前述のスニペットでは、一時的な認証情報が default プロファイルに関連付けられていることを前提としています。別のプロファイルを使用する場合は、上記のコードを次のように変更し、`profile_name` を実際のプロファイル名の名前に置き換えてください。

```
private static final AmazonSimpleWorkflow swf =
    AmazonSimpleWorkflowClientBuilder
        .standard()
        .withCredentials(new ProfileCredentialsProvider("profile_name"))
        .withRegion(Regions.DEFAULT_REGION)
        .build();
```

4. SWF ドメインを登録するための新しい関数を追加します。ドメインは、数多くの関連 SWF アクティビティおよびワークフロータイプ用の論理コンテナです。SWF コンポーネントは、同じドメイン内に存在する場合にのみ相互に通信できます。

```
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!");
}
```

ドメインを登録する場合、名前 (:、/、| を除く 1~256 文字、制御文字、またはリテラル文字列「arn」の任意のセット) と保持期間を指定します。保持期間は、ワークフローの実行が完了してから Amazon SWF がワークフローの実行履歴データを保持する日数です。ワークフロー実行の最大保持期間は 90 日です。詳細については、「[RegisterDomainRequest](#)」を参照してください。

その名前のドメインがすでに存在する場合、[DomainAlreadyExistsException](#) が発生します。ドメインがすでに作成されているかどうかは問題ではないため、この例外は無視できます。

Note

このコードは、AWS SDK for Java のメソッドを使用する場合の一般的なパターン、`simpleworkflow.model` 名前空間のクラスで提供されるメソッドのデータを示します。これは、チェーン可能な `with*` メソッドを使用してインスタンス化および入力します。

5. 新しいアクティビティタイプを登録する関数を追加します。アクティビティはワークフローのワークの単位を表します。

```
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!");
}
```

アクティビティタイプは名前およびバージョンによって識別されます。これらは、登録されているドメインでこのアクティビティを他のアクティビティから一意に識別するために使用されます。アクティビティには、SWF からタスクとデータを受け取るために使用されるデフォルトのタスクリスト、アクティビティ実行の各部分にかかる時間に対する制約適用に使用できるさまざまなタイムアウトなど、数多くのオプションパラメーターも含まれます。詳細については、「[RegisterActivityTypeRequest](#)」を参照してください。

Note

すべてタイムアウト値は秒単位で指定されます。タイムアウトのワークフローの実行への影響の詳細については、「[Amazon SWF Timeout Types](#)」を参照してください。

登録しようとしているアクティビティタイプが既に存在する場合は、[TypeAlreadyExistsException](#)が発生します。新しいワークフロータイプを登録する関数を追加します。ワークフローはディサイダーとも呼ばれ、ワークフロー実行のロジックを表します。

+

```
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!");
}
```

+

アクティビティタイプと同様に、ワークフロータイプは名前とバージョンによって識別され、設定可能なタイムアウトがあります。詳細については、「[RegisterWorkflowTypeRequest](#)」を参照してください。

+

登録しようとしているワークフロータイプが既に存在する場合は、[TypeAlreadyExistsException](#) が発生します。最後に、main メソッドを提供してクラスを実行可能にします。これはドメイン、アクティビティタイプ、およびワークフロータイプを登録します。

+

```
registerDomain();
registerWorkflowType();
registerActivityType();
```

ここで、アプリケーションを**構築**および**実行**して登録スクリプトを実行するか、アクティビティとワークフローワーカーのコーディングを続行できます。ドメイン、ワークフロー、およびアクティビティが登録されたら、これを再度実行する必要はありません。これらのタイプは、ユーザーが自ら非推奨とするまで保持されます。

アクティビティワーカーの実装

アクティビティは、ワークフローのワークの基本単位です。ワークフローは、ディシジョンタスクに応じて、ロジック、実行するスケジュールアクティビティ (または実行するその他のアクション) を提供します。通常、一般的なワークフローは、同期、非同期、またはそれらの組み合わせで実行できる数多くのアクティビティで構成されます。

アクティビティワーカーは、ワークフローの決定に応じて Amazon SWF によって生成されるアクティビティタスクをポーリングするコードです。アクティビティタスクを受け取ると、対応するアクティビティを実行し、ワークフローに成功/失敗の応答を返します。

単一のアクティビティを駆動するシンプルなアクティビティワーカーを実装します。

1. テキストエディターを開き、ファイル `ActivityWorker.java` を作成して、[共通のステップ](#)に従ってパッケージ宣言とインポートを追加します。

```
import com.amazonaws.regions.Regions;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.model.*;
```

2. `ActivityWorker` クラスをファイルに追加し、Amazon SWF を操作するために使用する SWF クライアントを保持するためのデータメンバーを提供します。

```
private static final AmazonSimpleWorkflow swf =
    AmazonSimpleWorkflowClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

3. アクティビティとして使用するメソッドを追加します。

```
private static String sayHello(String input) throws Throwable {
    return "Hello, " + input + "!";
}
```

アクティビティは単純に文字列を受け取り、それをあいさつに組み合わせて、結果を返します。このアクティビティで例外が発生する可能性はほとんどありませんが、何か問題が発生した場合にエラーを発生させるアクティビティを設計することをお勧めします。

4. アクティビティタスクのポーリングメソッドとして使用する `main` メソッドを追加します。最初に、アクティビティタスクについてタスクリストをポーリングするコードを追加します。

```
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();
```

アクティビティは SWF クライアントの `pollForActivityTask` メソッドを呼び出して Amazon SWF からタスクを受け取り、渡された [PollForActivityTaskRequest](#) で使用するドメインとタスクリストを指定します。

タスクを受け取ったら、タスクの `getTaskToken` メソッドを呼び出して、その固有の識別子を取得します。

- 次に、入ってきたタスクを処理するコードを記述します。main メソッドで、タスクをポーリングし、タスクトークンを取得するコードの直後に以下を追加します。

```
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()));  
    }  
}
```

```
}
```

タスクトークンが null でない場合、アクティビティメソッド (sayHello) の実行を開始し、タスクとともに送信された入力データを指定することができます。

タスクが成功した場合 (エラーが生成されなかった場合)、ワーカーは、タスクトークンとアクティビティの結果データを含む [RespondActivityTaskCompletedRequest](#) オブジェクトを使用して SWF クライアントの respondActivityTaskCompleted メソッドを呼び出すことにより、SWF に応答します。

一方、タスクが失敗した場

合、respondActivityTaskFailedRespondActivityTaskFailedRequest [オブジェクトとともに](#) メソッドを呼び出し、タスクトークンとエラーに関する情報を渡して応答します。

Note

このアクティビティは、強制終了した場合は適切にシャットダウンしません。このチュートリアルの対象外ですが、このアクティビティワーカーの代替の実装が、付随するトピック「[アクティビティおよびワークフローワーカーの適切なシャットダウン](#)」で示されています。

ワークフローワーカーの実装

ワークフローは、ワークフローワーカーと呼ばれるコードに置かれます。ワークフローワーカーは、ドメイン内の Amazon SWF によって送信され、ワークフロータイプが登録されたデフォルトのタスクリスト上にある決定タスクをポーリングします。

ワークフローワーカーがタスクを受信すると、何らかの決定 (通常は新しいアクティビティタスクをスケジュールするかどうか) を行い、適切なアクション (アクティビティのスケジュールリングなど) を実行します。

1. テキストエディターを開き、ファイル WorkflowWorker.java を作成して、[共通のステップ](#)に従ってパッケージ宣言とインポートを追加します。
2. いくつかのインポートをファイルに追加します。

```
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. `WorkflowWorker` クラスを宣言し、SWF メソッドへのアクセスに使用する [AmazonSimpleWorkflowClient](#) クラスのインスタンスを作成します。

```
private static final AmazonSimpleWorkflow swf =
    AmazonSimpleWorkflowClientBuilder.standard().withRegion(Regions.DEFAULT_REGION).build();
```

4. `main` メソッドを追加します。このメソッドは連続してループし、SWF クライアントの `pollForDecisionTask` メソッドを使用して決定タスクをポーリングします。 [PollForDecisionTaskRequest](#) によって詳細が提供されます。

```
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();
        }
    }
}
```

タスクを受け取ったら、その `getTaskToken` メソッドを呼び出します。これにより、タスクの識別に使用できる文字列が返されます。返されたトークンが `null` でない場

合、executeDecisionTask メソッドでさらに処理し、タスクトークンおよびそのタスクで送信された [HistoryEvent](#) オブジェクトのリストを渡します。

- executeDecisionTask メソッドを追加し、タスクトークン (String) および HistoryEvent リストを受け取ります。

```
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;
```

また、以下のような項目を追跡するためにいくつかのデータメンバーを設定します。

- タスクの処理結果を報告するために使用される [Decision](#) オブジェクトのリスト。
 - "WorkflowExecutionStarted" イベントによって提供されるワークフロー入力を保持する文字列。
 - すでにスケジュールされているか、現在実行中の場合に、同じアクティビティのスケジュールを回避するためにスケジュールされ、開いている (実行中) のアクティビティの数。
 - アクティビティが完了したことを示すブール。
 - ワークフローの結果として返すためにアクティビティの結果を保持する文字列。
- 次に、executeDecisionTask メソッドによって報告されたイベントタイプに基づいて、タスクとともに送信された HistoryEvent オブジェクトを処理するコードを getEventType に追加します。

```
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("]");
```

ワークフローでは、以下が最も重要です。

- ワークフローの実行が開始したことを示し (通常はワークフローの最初のアクティビティを実行する必要があることを意味します)、ワークフローに渡される最初の入力を提供する、"WorkflowExecutionStarted" イベント。この場合、これはあいさつの名前部分であるため、アクティビティの実行をスケジュールするとき使用する文字列に保存されます。
- スケジュールされたアクティビティが完了すると送信される、"ActivityTaskCompleted" イベント。イベントデータには、完了したアクティビティの戻り値も含まれます。1つのアクティビティのみがあるため、この値をワークフロー全体の結果として使用します。

他のイベントタイプは、ワークフローで必要な場合に使用できます。各イベントタイプの詳細については、[HistoryEvent](#) クラスの説明を参照してください。

+ 注: switch ステートメントの文字列は Java 7 で導入されました。以前のバージョンの Java を使用している場合は、[EventType](#) クラスを使用して、`history_event.getType()` で返される String を列挙値に変換し、必要に応じて String に戻すことができます。

```
EventType et = EventType.fromValue(event.getEventType());
```

1. switch ステートメントの後で、受け取ったタスクに応じて、適切な決定を使用して応答するコードを追加します。

```
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);
```

- まだアクティビティがスケジュールされていない場合、`ScheduleActivityTask` 決定で応答します。これにより、Amazon SWF が次にスケジュールするアクティビティに関する情報が、Amazon SWF がアクティビティに送信するデータを含めて、[ScheduleActivityTaskDecisionAttributes](#) 構造で提供されます。
- アクティビティが完了すると、全体のワークフローが完了したと見なし、`CompletedWorkflowExecution` 決定で応答します。[CompleteWorkflowExecutionDecisionAttributes](#) 構造に入力し、完了したワークフローに関する詳細を提供します。この場合、アクティビティの結果を返します。

いずれの場合も、メソッドの先頭で `Decision` リストに決定情報が追加されます。

2. タスクの処理中に収集された Decision オブジェクトのリストを返して、決定タスクを完了します。このコードを、これまで記述した `executeDecisionTask` メソッドの最後に追加します。

```
swf.respondDecisionTaskCompleted(  
    new RespondDecisionTaskCompletedRequest()  
        .withTaskToken(taskToken)  
        .withDecisions(decisions));
```

SWF クライアントの `respondDecisionTaskCompleted` メソッドは、タスクと、Decision オブジェクトのリストを識別するタスクトークンを受け取ります。

ワークフロースターターの実装

最後に、ワークフロー実行を開始するコードを書きます。

1. テキストエディターを開き、ファイル `WorkflowStarter.java` を作成して、[共通のステップ](#)に従ってパッケージ宣言とインポートを追加します。
2. `WorkflowStarter` クラスを追加します。

```
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 クラスは単一のメソッド main で構成されます。このメソッドはコマンドラインで渡されたオプションの引数を、ワークフローの入力データとして受け取ります。

SWF のクライアントメソッド startWorkflowExecution は、[StartWorkflowExecutionRequest](#) オブジェクトを入力として受け取ります。ここで、ドメインと実行するワークフロータイプの指定に加えて、以下を指定します。

- 人間が読み取れるワークフロー実行の名前
- ワークフローの入力データ (この例のコマンドラインで提供)
- ワークフロー全体の実行時間 (秒単位) を表すタイムアウト値

[が返す](#) RunstartWorkflowExecution オブジェクトは、実行 ID を提供します。これは、ワークフロー実行の Amazon SWF の履歴でこの特定のワークフロー実行を識別するために使用できる値です。

+ 注: 実行 ID は Amazon SWF によって生成され、ワークフローの実行を開始するときに渡すワークフロー実行名と同じではありません。

例の作成

Maven でサンプルプロジェクトを構築するには、helloworld ディレクトリに移動し、次のように入力します。

```
mvn package
```

結果的に生じる `helloswf-1.0.jar` が `target` ディレクトリに生成されます。

例の実行

この例は、4 つの異なる実行可能クラスで構成されます。これらは互いに独立して実行されます。

Note

Linux、macOS、または Unix システムを使用している場合、それらのすべてを 1 つずつ、1 つのターミナルウィンドウで実行できます。Windows を実行している場合は、2 つの追加インスタンスのコマンドラインを開き、それぞれの `helloswf` ディレクトリに移動します。

Java クラスパスの設定

Maven によって依存関係が処理されましたが、この例を実行するには、Java のクラスパスで AWS SDK ライブラリとその依存関係を指定する必要があります。CLASSPATH 環境変数を AWS SDK ライブラリの場所に設定し、必要な依存関係を含む SDK の `third-party/lib` ディレクトリに設定できます。

```
export CLASSPATH='target/helloswf-1.0.jar:/path/to/sdk/lib/*:/path/to/sdk/third-party/lib/*'  
java example.swf.hello.HelloTypes
```

または、`java` コマンドの `-cp` オプションを使用して、各アプリケーションの実行中にクラスパスを設定できます。

```
java -cp target/helloswf-1.0.jar:/path/to/sdk/lib/*:/path/to/sdk/third-party/lib/* \  
example.swf.hello.HelloTypes
```

使用するスタイルはユーザーが選択できます。コードが問題なく作成されても、例を実行しようとするると一連の "NoClassDefFound" エラーが表示される場合、クラスパスが正しく設定されていない可能性があります。

ドメイン、ワークフロー、およびアクティビティタイプの登録

ワーカーおよびワークフロースターターを実行する前に、ドメイン、ワークフロータイプ、およびアクティビティタイプを登録する必要があります。これを行うコードは、[ドメイン、ワークフロー、およびアクティビティタイプの登録](#)で実装しました。

構築後に [CLASSPATH を設定](#)した場合、次のコマンドを実行して登録コードを実行できます。

```
echo 'Supply the name of one of the example classes as an argument.'
```

アクティビティおよびワークフローワーカーの開始

これでタイプが登録されたため、アクティビティとワークフローワーカーを開始できます。これらは継続して実行され、強制終了されるまでタスクをポーリングするため、別のターミナルウィンドウで実行するか、Linux、macOS、または Unix で実行している場合は & 演算子を使用して、実行時にそれぞれが別のプロセスとして生成されるようにできます。

```
echo 'If there are arguments to the class, put them in quotes after the class
name.'
exit 1
```

別のウィンドウでこれらのコマンドを実行している場合は、各行から最終的な & 演算子を省略します。

ワークフロー実行の開始

これでアクティビティとワークフローワーカーがポーリングを実行しているため、ワークフロー実行を開始できます。このプロセスは、ワークフローが完了したステータスを返すまで実行されます。(& オペレーターを使用して新しく生成されたプロセスとしてワーカーを実行していない限り) このプロセスは新しいターミナルウィンドウで実行する必要があります。

```
fi
```

Note

独自の入力データを提供する場合 (最初にワークフロー、次にアクティビティに渡されます)、コマンドラインに追加します。例:

```
echo "## Running $className..."
```

ワークフロー実行を開始すると、両方のワーカーおよびワークフロー実行そのものによって提供された出力が表示され始めます。ワークフローが最終的に完了すると、その出力が画面に表示されます。

この例の完全なソース。

この例の[完全なソース](#)は、Github の `aws-java-developer-guide` レポジトリで参照できます。

詳細情報

- ここに示すワーカーにより、ワークフローポーリングがまだ行われている間にシャットダウンされた場合、タスクが失われる可能性があります。ワーカーを適切にシャットダウンする方法を確認するには、「[アクティビティおよびワークフローワーカーの適切なシャットダウン](#)」を参照してください。
- Amazon SWF の詳細については、[Amazon SWF](#) のホームページにアクセスするか、「[Amazon SWF デベロッパーガイド](#)」を参照してください。
- AWS Flow Framework for Java を使用すると、注釈を使ってスマートな Java スタイルでより複雑なワークフローを記述できます。詳細については、「[AWS Flow Framework for Java デベロッパーガイド](#)」を参照してください。

Lambda タスク

Amazon SWF アクティビティの代わりに、またはこれに併せて [Lambda](#) 関数を使用してワークフローの作業単位を表し、それらをアクティビティに合わせて同様にスケジュールできます。

このトピックでは、AWS SDK for Java を使用して Amazon SWF Lambda タスクを実装する方法について説明します。一般的な Lambda タスクの詳細については、「[Amazon SWF デベロッパーガイド](#)」の「[AWS Lambda Tasks](#)」を参照してください。

Lambda 関数を実行するサービス間 IAM ロールの設定

Amazon SWF が Lambda 関数を実行するには、事前に、ユーザーに代わって Lambda 関数を実行するための Amazon SWF 許可を付与するよう IAM ロールを設定する必要があります。これを行う方法に関する詳細については、「[AWS Lambda Tasks](#)」を参照してください。

Lambda タスクを使用するワークフローを登録するときに、この IAM ロールの Amazon リソースネーム (ARN) が必要になります。

Lambda 関数の作成

Java を含め、多数の異なる言語で Lambda 関数を記述できます。Lambda 関数の作成、デプロイ、および使用の詳細については、「[AWS Lambda デベロッパーガイド](#)」を参照してください。

Note

Lambda 関数の記述に使用する言語は何であってもかまいません。ワークフローコードが記述されている言語にかかわらず、任意の Amazon SWF ワークフローによってスケジュールおよび実行できます。Amazon SWF は関数の実行の詳細を処理し、データをやり取りします。

[シンプル Amazon SWF アプリケーションの構築](#)のアクティビティの代わりに使用できるシンプルな Lambda 関数を次に示します。

- このバージョンは JavaScript で書かれており、[AWS Management Console](#)を使用して直接入力できます。

```
exports.handler = function(event, context) {
    context.succeed("Hello, " + event.who + "!");
};
```

- 次に示すのは、Java で書かれた同じ関数です。これも Lambda でデプロイして実行できます。

```
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

Java 関数の Lambda へのデプロイの詳細については、AWS Lambda デベロッパーガイドの [デプロイパッケージの作成 \(Java\)](#) を参照してください。また、[Java で Lambda 関数を作成するためのプログラミングモデル](#) というタイトルのセクションも参照することをお勧めします。

Lambda 関数は、event または input オブジェクトを最初のパラメータとして受け取り、context オブジェクトを 2 番目のパラメータとして受け取ります。このオブジェクトは、Lambda 関数を実行するリクエストに関する情報を提供します。この特定の関数は、入力が JSON で、who フィールドがあいさつの作成に使用される名前に設定されていることを想定しています。

Lambda で使用するワークフローを登録する

ワークフローで Lambda 関数をスケジュールするには、Lambda 関数を呼び出すアクセス権限を Amazon SWF に提供する IAM ロールの名前を指定する必要があります。これは、withDefaultLambdaRoleRegisterWorkflowTypeRequest.setDefaultLambdaRole の [または](#) メソッドを使用して、ワークフロー登録中に設定できます。

```
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) {
```

Lambda タスクのスケジュール

Lambda タスクのスケジュールは、アクティビティのスケジュールに似ています。`ScheduleLambdaFunction` [DecisionType](#) と [ScheduleLambdaFunctionDecisionAttributes](#) を使用して [Decision](#) を提供します。

```
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(
```

`ScheduleLambdaFunctionDecisionAttributes` で、呼び出す Lambda 関数の ARN である `name` と、履歴ログで Lambda 関数を識別するために Amazon SWF が使用する名前である `id` を指定する必要があります。

また、Lambda 関数のオプションの `input` を指定し、その `start to close timeout` 値を設定できます。これは、`LambdaFunctionTimedOut` イベントを生成する前に Lambda 関数に実行が許可される秒数です。

Note

このコードは、[AWSLambdaClient](#) を使用して、関数名が指定されると Lambda 関数の ARN を取得します。この手法を使用すれば、コードで完全な ARN のハードコーディング (AWS アカウント ID を含む) を行わなくても済みます。

デサイダーでの Lambda 関数イベントの処理

Lambda タスクでは、ワークフローワーカーの決定タスクでポーリングを行うときにアクションを実行できる数多くのイベントを生成します。これらは、Lambda タスクのライフサイクルに対応し、`__`、`LambdaFunctionScheduled` などの `LambdaFunctionStartedEventType` `LambdaFunctionCompleted` 値を取ります。Lambda 関数が失敗するか、設定されたタイムアウト値よりも長い時間がかかる場合、それぞれ `LambdaFunctionFailed` または `LambdaFunctionTimedOut` イベントタイプを受け取ります。

```
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;
```

Lambda 関数からの出力の受け取り

[LambdaFunctionCompletedEventAttributes](#) オブジェクトを取得するために [HistoryEvent](#) で `LambdaFunctionCompleted`EventType`, you can retrieve your 0 function's return value by first calling `getLambdaFunctionCompletedEventAttributes` を受け取るときは、`getResult` メソッドを呼び出して Lambda 関数の出力を取得します。

```
LambdaFunctionCompleted:  
running_functions--;
```

この例の完全なソース。

この例の完全なソース `:github:<awsdocs/aws-java-developer-guide/tree/master/doc_source/snippets/helloswf_lambda/>` は、Github の `aws-java-developer-guide` レポジトリで参照できます。

アクティビティおよびワークフローワーカーの適切なシャットダウン

[シンプル Amazon SWF アプリケーションの構築](#) のトピックでは、登録アプリケーション、アクティビティとワークフローワーカー、およびワークフロースターターで構成されるシンプルなワークフローアプリケーションの完全な実装について説明しました。

ワーカークラスは、アクティビティを実行したり、決定を返したりするため継続して実行し、Amazon SWF によって送信されたタスクをポーリングするよう設計されています。ポーリングリクエストが行われると、Amazon SWF はポーリング元を記録し、それにタスクを割り当てるよう試みます。

長いポーリング中にワークフローワーカーが終了すると、Amazon SWF は終了したワーカーへタスクの送信を引き続き試み、その結果 (タスクのタイムアウトまで) タスクが失われる場合があります。

この状況に対応する 1 つの方法は、ワーカーが終了する前に、すべての長いポーリングリクエストが戻るのを待つことです。

このトピックでは、Java のシャットダウンフックを使用してアクティビティワーカーの適切なシャットダウンを試み、`helloswf` からのアクティビティワーカーを再記述します。

完全なコードは次のとおりです。

```
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()));
            }
        }
    }
}
```

```
        }  
    }  
}  
}
```

このバージョンでは、元のバージョンの main 関数にあったポーリングコードが、独自のメソッドに移動されました。pollAndExecute

main 関数は [CountDownLatch](#) を [シャットダウンフック](#) とともに使用して、終了がリクエストされた後で最大 60 秒待ってから、スレッドをシャットダウンさせます。

ドメインの登録

[Amazon SWF](#) の各ワークフローとアクティビティでは、実行するドメインが必要です。

1. 新しい [RegisterDomainRequest](#) オブジェクトを作成し、これに少なくともドメイン名とワークフロー実行保持期間を指定します (これらのパラメーターは両方とも必須です)。
2. [AmazonSimpleWorkflowClient.registerDomain](#) メソッドを、RegisterDomainRequest オブジェクトで呼び出します。
3. リクエストしているドメインがすでに存在している場合 (その場合、アクションは通常必要ありません)、[DomainAlreadyExistsException](#) をキャッチします。

次のコードは、この手順を示しています。

```
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!");  
    }  
}
```


ドメインの一覧表示

登録タイプ別に、アカウントと AWS リージョンに関連付けられた [Amazon SWF](#) ドメインを一覧表示できます。

1. [ListDomainsRequest](#) オブジェクトを作成し、関心のあるドメインの登録ステータスを指定します。これは必須です。
2. [AmazonSimpleWorkflowClient.listDomains](#) を ListDomainRequest オブジェクトで呼び出します。結果は [DomainInfos](#) オブジェクトで示されます。
3. 返されたオブジェクトで [getDomainInfos](#) を呼び出して、[DomainInfo](#) オブジェクトの一覧を取得します。
4. 各 DomainInfo オブジェクトで [getName](#) を呼び出して、その名前を取得します。

次のコードは、この手順を示しています。

```
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 に含まれるコードサンプル

AWS SDK for Java は、SDK のビルドおよび実行可能なプログラムの多くの機能を示す多数のコードサンプルでパッケージ化されています。AWS SDK for Java を使用して独自の AWS ソリューションを実装する際に、これらを検討または変更できます。

サンプルの入手方法

AWS SDK for Java コードサンプルは、SDK のサンプルディレクトリで提供されています。[AWS SDK for Javaのセットアップ](#)での説明に従って SDK をダウンロードおよびインストールすれば、サンプルは既にシステムに存在しています。

また、AWS SDK for Java の GitHub リポジトリの [src/samples](#) ディレクトリで、最新のサンプルを確認することもできます。

コマンドラインを使用したサンプルのビルドと実行

サンプルには [Ant](#) ビルドスクリプトが含まれているため、コマンドラインからこれらを簡単にビルドおよび実行できます。また、各サンプルには、各サンプルに固有の情報を含む README ファイルが HTML 形式で含まれています。

Note

GitHub でサンプルコードを参照している場合は、サンプルの README.html ファイルを表示するときに、ソースコード表示の [Raw] ボタンをクリックします。raw モードでは、HTML はお使いのブラウザで目的どおりにレンダリングされます。

前提条件

AWS SDK for Java のサンプルを実行する前に、[開発用の AWS 認証情報とリージョンのセットアップ](#)での説明に従って、環境または AWS CLI で AWS 認証情報を設定する必要があります。サンプルは、可能な限りデフォルトの認証情報プロバイダチェーンを使用します。そのため、この方法で認証情報を設定することにより、ソースコードディレクトリ内のファイルに AWS 認証情報を挿入するリスクの高い方法を回避できます (この方法では、意図せずにファイルにチェックインし、ファイルを公開する可能性があります)。

サンプルの実行

1. サンプルのコードを含むディレクトリに変更します。例えば、AWS SDK ダウンロードのルートディレクトリで `AwsConsoleApp` サンプルを実行する場合は、次のように入力します。

```
cd samples/AwsConsoleApp
```

2. `Ant` を使用してサンプルをビルドおよび実行します。デフォルトのビルドターゲットでは両方のアクションが実行されるため、次のように入力できます。

```
ant
```

サンプルは、情報を標準出力に出力します。以下に例を示します。

```
=====

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.
```

Eclipse IDE を使用したサンプルのビルドと実行

AWS Toolkit for Eclipse を使用する場合は、AWS SDK for Java に基づいて Eclipse で新しいプロジェクトを開始したり、既存の Java プロジェクトに SDK を追加したりすることができます。

前提条件

AWS Toolkit for Eclipse をインストールした後で、セキュリティ認証情報を使用して Toolkit を設定することをお勧めします。これは、Eclipse の [Window] メニューから [詳細設定] を選択し、[AWS ツールキット] セクションを選択することで、いつでも実行できます。

サンプルの実行

1. Eclipse を開きます。
2. 新しい AWS Java プロジェクトを作成します。Eclipse の [File] メニューで [New] を選択し、[Project] をクリックします。[New Project] ウィザードが起動します。
3. [AWS] カテゴリを展開し、[AWS Java プロジェクト] を選択します。
4. [Next] (次へ) をクリックします。プロジェクトの設定ページが表示されます。
5. [Pattern Name] ボックスに名前を入力します。AWS SDK for Java サンプルグループに、既に説明した SDK で使用できるサンプルが表示されます。
6. 各チェックボックスをオンにして、プロジェクトに含めるサンプルを選択します。
7. AWS 認証情報を入力します。認証情報を使用してすでに AWS Toolkit for Eclipse を設定している場合、この情報は自動的に入力されます。
8. [終了] を選択します。プロジェクトが作成され、[Project Explorer] に追加されます。

9. 実行するサンプルの .java ファイルを選択します。たとえば、Amazon S3 サンプルの場合は S3Sample.java を選択します。
- 10 [Run] メニューで、[Run] を選択します。
- 11 [Project Explorer] でプロジェクトを右クリックし、[Build Path] をポイントして、[Add Libraries] を選択します。
- 12 [AWS Java SDK] を選択し、[Next] (次へ) を選択して、画面のその他の手順を実行します。

のセキュリティ AWS SDK for Java

クラウドセキュリティは Amazon Web Services (AWS) の最優先事項です。AWS のお客様は、セキュリティを非常に重視する組織の要件を満たせるように構築されたデータセンターとネットワークアーキテクチャーから利点を得ます。セキュリティは、AWS とユーザーの間で共有される責任です。[責任共有モデル](#)では、これをクラウドのセキュリティおよびクラウド内のセキュリティとして説明しています。

クラウドのセキュリティ — AWS クラウドで提供されるすべてのサービスを実行するインフラストラクチャ AWS を保護し、安全に使用できるサービスを提供します。当社のセキュリティ責任は、最優先事項であり AWS、当社のセキュリティの有効性は、[AWS コンプライアンスプログラムの一環として、サードパーティーの監査者によって定期的にテストおよび検証されています](#)。

クラウド内のセキュリティ — お客様の責任は、使用している AWS サービス、およびデータの機密性、組織の要件、適用可能な法律や規制などのその他の要因によって決まります。

この AWS 製品またはサービスは、サポートする特定の Amazon Web Services (AWS) サービスを通じて[責任共有モデル](#)に従います。AWS サービスセキュリティ情報については、[AWS 「サービスセキュリティドキュメント」ページ](#)と[AWS、AWS コンプライアンスプログラムによるコンプライアンスの取り組みの対象となるサービスを参照してください](#)。

トピック

- [AWS SDK for Java 1.x でのデータ保護](#)
- [AWS SDK for Java TLS のサポート](#)
- [Identity and Access Management](#)
- [この AWS 製品またはサービスのコンプライアンス検証](#)
- [この AWS 製品またはサービスの耐障害性](#)
- [この AWS 製品またはサービスのインフラストラクチャセキュリティ](#)
- [Amazon S3 暗号化クライアントの移行](#)

AWS SDK for Java 1.x でのデータ保護

[責任共有モデル](#)は、この AWS 製品またはサービスのデータ保護に適用されます。このモデルで説明されているように、AWS はすべての AWS クラウドを実行するグローバルインフラストラクチャを

保護する責任があります。ユーザーには、このインフラストラクチャでホストされているコンテンツに対する制御を維持する責任があります。このコンテンツには、使用する AWS のサービスに対するセキュリティの設定と管理タスクが含まれます。データプライバシーの詳細については、「[データプライバシーのよくある質問](#)」を参照してください。欧州におけるデータ保護の詳細については、AWS セキュリティブログの[AWS 「責任共有モデルと GDPR」](#) ブログ記事を参照してください。

データ保護の目的で、AWS アカウント 認証情報を保護し、AWS Identity and Access Management (IAM) を使用して個々のユーザーアカウントを設定することをお勧めします。この方法により、それぞれのジョブを遂行するために必要な許可のみを各ユーザーに付与できます。また、次の方法でデータを保護することをお勧めします。

- 各アカウントで多要素認証 (MFA) を使用します。
- SSL/TLS を使用して AWS リソースと通信します。
- API とユーザーアクティビティのログ記録を設定します AWS CloudTrail。
- サービス内のすべてのデフォルトのセキュリティコントロールで AWS 暗号化ソリューション AWS を使用します。
- Amazon Macie などのアドバンスドマネージドセキュリティサービスを使用します。これは、Amazon S3 に保存されている個人データの検出と保護を支援します。
- コマンドラインインターフェイスまたは API AWS を介して にアクセスするときに FIPS 140-2 検証済みの暗号化モジュールが必要な場合は、FIPS エンドポイントを使用します。利用可能な FIPS エンドポイントの詳細については、「[連邦情報処理規格 \(FIPS\) 140-2](#)」を参照してください。

顧客のアカウント番号などの機密の識別情報は、[Name] (名前) フィールドなどの自由形式のフィールドに配置しないことを強くお勧めします。これは、コンソール、API、または SDK を使用して、この AWS 製品またはサービス AWS CLI または他の AWS のサービスを使用する場合も同様です。AWS SDKs この AWS 製品またはサービスまたは他のサービスに入力したデータは、診断ログに取り込まれる可能性があります。外部サーバーへの URL を指定するときは、そのサーバーへのリクエストを検証するための認証情報を URL に含めないでください。

AWS SDK for Java TLS のサポート

次の情報は、Java SSL 実装 (のデフォルトの SSL 実装) にのみ適用されます AWS SDK for Java。別の SSL 実装を使用している場合は、その SSL 実装を参照して、TLS バージョンを適用する方法を確認してください。

TLS のバージョンを確認する方法

Java virtual machine (JVM) プロバイダーのマニュアルを参照して、ご使用のプラットフォームでサポートされている TLS バージョンを確認してください。一部の JVM では、次のコードによってどの SSL バージョンがサポートされているかが出力されます。

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

動作中の SSL ハンドシェイクと使用されている TLS のバージョンを確認するには、システムプロパティ `javax.net.debug` を使用します。

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

Note

TLS 1.3 は Java バージョン 1.9.5 ~ 1.10.31 の SDK と互換性がありません。詳細については、以下のブログ投稿を参照してください。

<https://aws.amazon.com/blogs/developer/tls-1-3-incompatibility-with-aws-sdk-for-java-versions-1-9-5-to-1-10-31/>

最小 TLS バージョンの適用

SDK は常に、プラットフォームとサービスがサポートする最新の TLS バージョンを優先します。特定の最小 TLS バージョンを強制する場合は、「JVM のドキュメント」を参照してください。OpenJDK ベースの JVM では、システムプロパティ `jdk.tls.client.protocols` を使用できます。

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

サポートされている PROTOCOLS の値については、「JVM のドキュメント」を参照してください。

Identity and Access Management

AWS Identity and Access Management (IAM) は、管理者が AWS リソースへのアクセスを安全に制御 AWS のサービス するのに役立つです。IAM 管理者は、誰を認証 (サインイン) し、誰に AWS リ

ソースの使用を承認する (アクセス許可を付与する) を制御します。IAM は、追加料金なしで AWS のサービス 使用できる です。

トピック

- [対象者](#)
- [アイデンティティを使用した認証](#)
- [ポリシーを使用したアクセスの管理](#)
- [IAM の AWS のサービス 仕組み](#)
- [AWS ID とアクセスのトラブルシューティング](#)

対象者

AWS Identity and Access Management (IAM) の使用方法は、 で行う作業によって異なります AWS。

サービスユーザー – AWS のサービス を使用してジョブを実行する場合、管理者から必要な認証情報とアクセス許可が与えられます。さらに多くの AWS 機能を使用して作業を行う場合は、追加のアクセス許可が必要になることがあります。アクセスの管理方法を理解しておく、管理者に適切な許可をリクエストするうえで役立ちます。の機能にアクセスできない場合は AWS、[AWS ID とアクセスのトラブルシューティング](#)「」または AWS のサービス 使用している のユーザーガイドを参照してください。

サービス管理者 – 社内の AWS リソースを担当している場合は、通常、へのフルアクセスがあります AWS。サービスユーザーがどの AWS 機能やリソースにアクセスするかを決めるのは管理者の仕事です。その後、IAM 管理者にリクエストを送信して、サービスユーザーの権限を変更する必要があります。このページの情報を点検して、IAM の基本概念を理解してください。会社で IAM を で使用する方法の詳細については AWS、使用している の AWS のサービス ユーザーガイドを参照してください。

IAM 管理者 - 管理者は、AWSへのアクセスを管理するポリシーの書き込み方法の詳細について確認する場合があります。IAM で使用できる AWS アイデンティティベースのポリシーの例を表示するには、AWS のサービス 使用している のユーザーガイドを参照してください。

アイデンティティを使用した認証

認証とは、ID 認証情報 AWS を使用して にサインインする方法です。として、IAM ユーザーとして AWS アカウントのルートユーザー、または IAM ロールを引き受けて認証 (にサインイン AWS) される必要があります。

ID ソースを介して提供された認証情報を使用して、フェデレーテッド ID AWS としてサインインできます。AWS IAM Identity Center (IAM Identity Center) ユーザー、会社のシングルサインオン認証、Google または Facebook の認証情報は、フェデレーテッド ID の例です。フェデレーテッドアイデンティティとしてサインインする場合、IAM ロールを使用して、前もって管理者により ID フェデレーションが設定されています。フェデレーション AWS を使用してにアクセスすると、間接的にロールを引き受けることとなります。

ユーザーのタイプに応じて、AWS Management Console または AWS アクセスポータルにサインインできます。へのサインインの詳細については AWS、「ユーザーガイド」の「[にサインインする方法 AWS アカウント](#) AWS サインイン」を参照してください。

AWS プログラムでにアクセスする場合、は Software Development Kit (SDK) とコマンドラインインターフェイス (CLI) AWS を提供し、認証情報を使用してリクエストに暗号で署名します。AWS ツールを使用しない場合は、リクエストに自分で署名する必要があります。推奨される方法を使用してリクエストを自分で署名する方法の詳細については、IAM [ユーザーガイドの API AWS リクエスト](#) の署名を参照してください。

使用する認証方法を問わず、追加セキュリティ情報の提供をリクエストされる場合もあります。例えば、AWS では、多要素認証 (MFA) を使用してアカウントのセキュリティを向上させることをお勧めします。詳細については、『AWS IAM Identity Center ユーザーガイド』の「[Multi-factor authentication](#)」(多要素認証) および『IAM ユーザーガイド』の「[AWSにおける多要素認証 \(MFA\) の使用](#)」を参照してください。

AWS アカウント ルートユーザー

を作成するときは AWS アカウント、アカウント内のすべての およびリソースへの AWS のサービス 完全なアクセス権を持つ 1 つのサインインアイデンティティから始めます。この ID は AWS アカウント ルートユーザーと呼ばれ、アカウントの作成に使用した E メールアドレスとパスワードでサインインすることでアクセスできます。日常的なタスクには、ルートユーザーを使用しないことを強くお勧めします。ルートユーザーの認証情報は保護し、ルートユーザーでしか実行できないタスクを実行するときに使用します。ルートユーザーとしてサインインする必要があるタスクの完全なリストについては、『IAM ユーザーガイド』の「[ルートユーザー認証情報が必要なタスク](#)」を参照してください。

フェデレーテッドアイデンティティ

ベストプラクティスとして、管理者アクセスを必要とするユーザーを含む人間のユーザーに、一時的な認証情報を使用してにアクセスするための ID プロバイダーとのフェデレーションの使用を要求 AWS のサービス します。

フェデレーテッド ID は、エンタープライズユーザーディレクトリ、ウェブ ID プロバイダー、AWS Directory Service、アイデンティティセンターディレクトリのユーザー、または ID ソースを通じて提供された認証情報 AWS のサービスを使用してにアクセスするユーザーです。フェデレーテッド ID がにアクセスすると AWS アカウント、ロールが引き受けられ、ロールは一時的な認証情報を提供します。

アクセスを一元管理する場合は、AWS IAM Identity Centerを使用することをお勧めします。IAM Identity Center でユーザーとグループを作成することも、独自の ID ソース内のユーザーとグループのセットに接続して同期して、すべての AWS アカウント とアプリケーションで使用することもできます。IAM Identity Center の詳細については、『AWS IAM Identity Center ユーザーガイド』の「[What is IAM Identity Center?](#)」(IAM Identity Center とは) を参照してください。

IAM ユーザーとグループ

[IAM ユーザー](#)は、単一のユーザーまたはアプリケーションに対して特定のアクセス許可 AWS アカウント を持つ 内のアイデンティティです。可能であれば、パスワードやアクセスキーなどの長期的な認証情報を保有する IAM ユーザーを作成する代わりに、一時認証情報を使用することをお勧めします。ただし、IAM ユーザーでの長期的な認証情報が必要な特定のユースケースがある場合は、アクセスキーをローテーションすることをお勧めします。詳細については、IAM ユーザーガイドの「[長期的な認証情報を必要とするユースケースのためにアクセスキーを定期的にローテーションする](#)」を参照してください。

[IAM グループ](#)は、IAM ユーザーの集団を指定するアイデンティティです。グループとしてサインインすることはできません。グループを使用して、複数のユーザーに対して一度に権限を指定できます。多数のユーザーグループがある場合、グループを使用することで権限の管理が容易になります。例えば、IAMAdmins という名前のグループを設定して、そのグループに IAM リソースを管理する権限を与えることができます。

ユーザーは、ロールとは異なります。ユーザーは 1 人の人または 1 つのアプリケーションに一意に関連付けられますが、ロールはそれを必要とする任意の人が引き受けるようになっています。ユーザーには永続的な長期の認証情報がありますが、ロールでは一時的な認証情報が提供されます。詳細については、『IAM ユーザーガイド』の「[IAM ユーザー \(ロールではなく\) の作成が適している場合](#)」を参照してください。

IAM ロール

[IAM ロール](#)は、特定のアクセス許可 AWS アカウント を持つ 内のアイデンティティです。これは IAM ユーザーに似ていますが、特定のユーザーには関連付けられていません。ロール を切り替える AWS Management Console ことで、[IAM ロール](#)を一時的に引き受けることができます。ロール

を引き受けるには、または AWS API AWS CLI オペレーションを呼び出すか、カスタム URL を使用します。ロールを使用する方法の詳細については、「IAM ユーザーガイド」の「[IAM ロールの使用](#)」を参照してください。

IAM ロールと一時的な認証情報は、次の状況で役立ちます:

- フェデレーションユーザーアクセス – フェデレーティッドアイデンティティに権限を割り当てるには、ロールを作成してそのロールの権限を定義します。フェデレーティッドアイデンティティが認証されると、そのアイデンティティはロールに関連付けられ、ロールで定義されている権限が付与されます。フェデレーションの詳細については、『IAM ユーザーガイド』の「[サードパーティーアイデンティティプロバイダー向けロールの作成](#)」を参照してください。IAM アイデンティティセンターを使用する場合、権限セットを設定します。アイデンティティが認証後にアクセスできるものを制御するため、IAM Identity Center は、権限セットを IAM のロールに関連付けます。権限セットの詳細については、『AWS IAM Identity Center ユーザーガイド』の「[権限セット](#)」を参照してください。
- 一時的な IAM ユーザー権限 - IAM ユーザーまたはロールは、特定のタスクに対して複数の異なる権限を一時的に IAM ロールで引き受けることができます。
- クロスアカウントアクセス - IAM ロールを使用して、自分のアカウントのリソースにアクセスすることを、別のアカウントの人物 (信頼済みプリンシパル) に許可できます。クロスアカウントアクセス権を付与する主な方法は、ロールを使用することです。ただし、一部の AWS サービス、(ロールをプロキシとして使用する代わりに) ポリシーをリソースに直接アタッチできます。クロスアカウントアクセスにおけるロールとリソースベースのポリシーの違いについては、『IAM ユーザーガイド』の「[IAM ロールとリソースベースのポリシーとの相違点](#)」を参照してください。
- クロスサービスアクセス — 一部の は、他の の機能 AWS のサービス を使用します AWS のサービス。例えば、あるサービスで呼び出しを行うと、通常そのサービスによって Amazon EC2 でアプリケーションが実行されたり、Amazon S3 にオブジェクトが保存されたりします。サービスでは、呼び出し元プリンシパルの権限、サービスロール、またはサービスにリンクされたロールを使用してこれを行う場合があります。
- 転送アクセスセッション (FAS) – IAM ユーザーまたはロールを使用して でアクションを実行する場合 AWS、ユーザーはプリンシパルと見なされます。一部のサービスを使用する際に、アクションを実行することで、別のサービスの別のアクションがトリガーされることがあります。FAS は、 を呼び出すプリンシパルのアクセス許可を AWS のサービス、ダウンストリームサービス AWS のサービス へのリクエストのリクエストと組み合わせて使用します。FAS リクエストは、サービスが他の AWS のサービス またはリソースとのやり取りを完了する必要があるリクエストを受け取った場合にのみ行われます。この場合、両方のアクションを実行するため

のアクセス許可が必要です。FAS リクエストを行う際のポリシーの詳細については、「[転送アクセスセッション](#)」を参照してください。

- サービスロール - サービスがユーザーに代わってアクションを実行するために引き受ける [IAM ロール](#)です。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除できます。詳細については、IAM ユーザーガイドの「[AWS のサービスにアクセス許可を委任するロールの作成](#)」を参照してください。
- サービスにリンクされたロール - サービスにリンクされたロールは、にリンクされたサービスロールの一種です AWS のサービス。サービスは、ユーザーに代わってアクションを実行するロールを引き受けることができます。サービスにリンクされたロールは に表示され AWS アカウント、サービスによって所有されます。IAM 管理者は、サービスにリンクされたロールの権限を表示できますが、編集することはできません。
- Amazon EC2 で実行されているアプリケーション - IAM ロールを使用して、EC2 インスタンスで実行され、AWS CLI または AWS API リクエストを行うアプリケーションの一時的な認証情報を管理できます。これは、EC2 インスタンス内でのアクセスキーの保存に推奨されます。AWS ロールを EC2 インスタンスに割り当て、そのすべてのアプリケーションで使用できるようにするには、インスタンスにアタッチされたインスタンスプロファイルを作成します。インスタンスプロファイルにはロールが含まれ、EC2 インスタンスで実行されるプログラムは一時的な認証情報を取得できます。詳細については、『IAM ユーザーガイド』の「[Amazon EC2 インスタンスで実行されるアプリケーションに IAM ロールを使用して権限を付与する](#)」を参照してください。

IAM ロールと IAM ユーザーのどちらを使用するかについては、『IAM ユーザーガイド』の「[\(IAM ユーザーではなく\) IAM ロールをいつ作成したら良いのか?](#)」を参照してください。

ポリシーを使用したアクセスの管理

でアクセスを制御する AWS には、ポリシーを作成し、AWS ID またはリソースにアタッチします。ポリシーは、アイデンティティまたはリソースに関連付けられているときにアクセス許可を定義するオブジェクトです。は、プリンシパル(ユーザー、ルートユーザー、またはロールセッション) AWS がリクエストを行うときに、これらのポリシー AWS を評価します。ポリシーでの権限により、リクエストが許可されるか拒否されるかが決まります。ほとんどのポリシーは JSON ドキュメント AWS として に保存されます。JSON ポリシードキュメントの構造と内容の詳細については、「IAM ユーザーガイド」の「[JSON ポリシー概要](#)」を参照してください。

管理者は AWS JSON ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどんなリソースにどんな条件でアクションを実行できるかということです。

デフォルトでは、ユーザーやロールに権限はありません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。その後、管理者はロールに IAM ポリシーを追加し、ユーザーはロールを引き継ぐことができます。

IAM ポリシーは、オペレーションの実行方法を問わず、アクションの権限を定義します。例えば、iam:GetRole アクションを許可するポリシーがあるとします。そのポリシーを持つユーザーは、AWS Management Console、AWS CLI または AWS API からロール情報を取得できます。

アイデンティティベースのポリシー

アイデンティティベースポリシーは、IAM ユーザー、ユーザーのグループ、ロールなど、アイデンティティにアタッチできる JSON 権限ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件をコントロールします。アイデンティティベースのポリシーを作成する方法については、IAM ユーザーガイドの「[IAM ポリシーの作成](#)」を参照してください。

アイデンティティベースポリシーは、さらにインラインポリシーまたはマネージドポリシーに分類できます。インラインポリシーは、単一のユーザー、グループ、またはロールに直接埋め込まれています。管理ポリシーは、内の複数のユーザー、グループ、ロールにアタッチできるスタンドアロンポリシーです AWS アカウント。管理ポリシーには、AWS 管理ポリシーとカスタマー管理ポリシーが含まれます。マネージドポリシーまたはインラインポリシーのいずれかを選択する方法については、『IAM ユーザーガイド』の「[マネージドポリシーとインラインポリシーの比較](#)」を参照してください。

リソースベースのポリシー

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、[プリンシパルを指定する](#)必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーティッドユーザー、またはを含めることができます AWS のサービス。

リソースベースのポリシーは、そのサービス内にあるインラインポリシーです。リソースベースのポリシーでは、IAM の AWS マネージドポリシーを使用できません。

アクセスコントロールリスト (ACL)

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための権限を持つかをコントロールします。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

Amazon S3、AWS WAF、および Amazon VPC は、ACLs。ACL の詳細については、『Amazon Simple Storage Service デベロッパーガイド』の「[アクセスコントロールリスト \(ACL\) の概要](#)」を参照してください。

その他のポリシータイプ

AWS は、一般的ではない追加のポリシータイプをサポートします。これらのポリシータイプでは、より一般的なポリシータイプで付与された最大の権限を設定できます。

- **アクセス許可の境界** - アクセス許可の境界は、アイデンティティベースのポリシーによって IAM エンティティ (IAM ユーザーまたはロール) に付与できる権限の上限を設定する高度な機能です。エンティティにアクセス許可の境界を設定できます。結果として得られる権限は、エンティティのアイデンティティベースポリシーとそのアクセス許可の境界の共通部分になります。Principal フィールドでユーザーまたはロールを指定するリソースベースのポリシーでは、アクセス許可の境界は制限されません。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。許可の境界の詳細については、「IAM ユーザーガイド」の「[IAM エンティティの許可の境界](#)」を参照してください。
- **サービスコントロールポリシー (SCPs)** - SCPs は、の組織または組織単位 (OU) に対する最大アクセス許可を指定する JSON ポリシーです AWS Organizations。AWS Organizations は、AWS アカウント ビジネスが所有する複数の をグループ化して一元管理するためのサービスです。組織内のすべての機能を有効にすると、サービスコントロールポリシー (SCP) を一部またはすべてのアカウントに適用できます。SCP は、各 を含むメンバーアカウントのエンティティのアクセス許可を制限します AWS アカウントのルートユーザー。Organizations と SCP の詳細については、『AWS Organizations ユーザーガイド』の「[SCP の仕組み](#)」を参照してください。
- **セッションポリシー** - セッションポリシーは、ロールまたはフェデレーションユーザーの一時的なセッションをプログラムで作成する際にパラメータとして渡す高度なポリシーです。結果としてセッションの権限は、ユーザーまたはロールのアイデンティティベースポリシーとセッションポリシーの共通部分になります。また、リソースベースのポリシーから権限が派生する場合があります。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。詳細については、「IAM ユーザーガイド」の「[セッションポリシー](#)」を参照してください。

複数のポリシータイプ

1つのリクエストに複数のタイプのポリシーが適用されると、結果として作成される権限を理解するのがさらに難しくなります。複数のポリシータイプが関与する場合にリクエストを許可するかどうかが AWS を決定する方法については、IAM ユーザーガイドの「[ポリシー評価ロジック](#)」を参照してください。

IAM の AWS のサービス 仕組み

ほとんどの IAM 機能との AWS のサービス 連携方法の概要については、IAM ユーザーガイドの[AWS 「IAM と連携する のサービス」](#)を参照してください。

IAM AWS のサービス で特定の を使用する方法については、関連するサービスのユーザーガイドのセキュリティセクションを参照してください。

AWS ID とアクセスのトラブルシューティング

次の情報は、 と IAM の使用時に発生する可能性がある一般的な問題の診断 AWS と修正に役立ちます。

トピック

- [でアクションを実行する権限がない AWS](#)
- [iam を実行する権限がありません。PassRole](#)
- [自分の 以外のユーザーに自分の AWS リソース AWS アカウント へのアクセスを許可したい](#)

でアクションを実行する権限がない AWS

「I am not authorized to perform an action in Amazon Bedrock」というエラーが表示された場合、そのアクションを実行できるようにポリシーを更新する必要があります。

次のエラー例は、mateojackson IAM ユーザーがコンソールを使用して、ある *my-example-widget* リソースに関する詳細情報を表示しようとしたことを想定して、その際に必要な `aws:GetWidget` アクセス許可を持っていない場合に発生するものです。

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

この場合、`aws:GetWidget` アクションを使用して *my-example-widget* リソースへのアクセスを許可するように、mateojackson ユーザーのポリシーを更新する必要があります。

サポートが必要な場合は、AWS 管理者にお問い合わせください。サインイン資格情報を提供した担当者が管理者です。

iam を実行する権限がありません。PassRole

iam:PassRole アクションを実行する権限がないというエラーが表示された場合は、ポリシーを更新して AWS にロールを渡すことができるようにする必要があります。

一部の AWS のサービスでは、新しいサービスロールまたはサービスにリンクされたロールを作成する代わりに、そのサービスに既存のロールを渡すことができます。そのためには、サービスにロールを渡す権限が必要です。

以下の例のエラーは、marymajor という IAM ユーザーがコンソールを使用して AWS でアクションを実行しようする場合に発生します。ただし、このアクションをサービスが実行するには、サービスロールから付与された権限が必要です。Mary には、ロールをサービスに渡す権限がありません。

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

この場合、Mary のポリシーを更新してメアリーに iam:PassRole アクションの実行を許可する必要があります。

サポートが必要な場合は、AWS 管理者にお問い合わせください。サインイン資格情報を提供した担当者が管理者です。

自分の 以外のユーザーに自分の AWS リソース AWS アカウント へのアクセスを許可したい

他のアカウントのユーザーや組織外の人が、リソースにアクセスするために使用できるロールを作成できます。ロールの引き受けを委託するユーザーを指定できます。リソースベースのポリシーまたはアクセスコントロールリスト (ACL) をサポートするサービスの場合、それらのポリシーを使用して、リソースへのアクセスを付与できます。

詳細については、以下を参照してください:

- がこれらの機能 AWS をサポートしているかどうかを確認するには、「」を参照してください [IAM の AWS のサービス 仕組み](#)。
- 所有 AWS アカウント している のリソースへのアクセスを提供する方法については、[IAM ユーザーガイドの「所有 AWS アカウント している別の の IAM ユーザーへのアクセスを提供する」](#)を参照してください。

- リソースへのアクセスをサードパーティーに提供する方法については AWS アカウント、「IAM ユーザーガイド」の「[サードパーティー AWS アカウント が所有する へのアクセスを提供する](#)」を参照してください。
- ID フェデレーションを介してアクセスを提供する方法については、『IAM ユーザーガイド』の「[外部で認証されたユーザー \(ID フェデレーション\) へのアクセス権限](#)」を参照してください。
- クロスアカウントアクセスでのロールとリソースベースのポリシーの使用の違いの詳細については、「IAM ユーザーガイド」の「[IAM ロールとリソースベースのポリシーとの相違点](#)」を参照してください。

この AWS 製品またはサービスのコンプライアンス検証

AWS のサービスが特定のコンプライアンスプログラムの範囲内にあるかどうかを確認するには、コンプライアンスプログラム [AWS のサービスによる対象範囲内のコンプライアンスプログラム](#) を参照し、関心のあるコンプライアンスプログラムを選択します。一般的な情報については、[AWS 「コンプライアンスプログラム」](#) を参照してください。

を使用して、サードパーティーの監査レポートをダウンロードできます AWS Artifact。詳細については、「[でのレポートのダウンロード AWS Artifact](#)」の」を参照してください。

を使用する際のお客様のコンプライアンス責任 AWS のサービスは、お客様のデータの機密性、貴社のコンプライアンス目的、適用される法律および規制によって決まります。では、コンプライアンスに役立つ以下のリソース AWS を提供しています。

- [セキュリティとコンプライアンスのクイックスタートガイド](#) – これらのデプロイガイドでは、アーキテクチャ上の考慮事項について説明し、セキュリティとコンプライアンスに重点を置いたベースライン環境 AWS を にデプロイする手順について説明します。
- [アマゾン ウェブ サービスにおける HIPAA セキュリティとコンプライアンスのためのアーキテクチャ](#) – このホワイトペーパーでは、企業が AWS を使用して HIPAA 対象アプリケーションを作成する方法について説明します。

Note

すべて AWS のサービス HIPAA の対象となるわけではありません。詳細については、「[HIPAA 対応サービスのリファレンス](#)」を参照してください。

- [AWS コンプライアンスリソース](#) – このワークブックとガイドのコレクションは、お客様の業界や地域に適用される場合があります。

- [AWS カスタマーコンプライアンスガイド](#) — コンプライアンスの観点から責任共有モデルを理解します。このガイドでは、ガイダンスを保護し AWS のサービス、複数のフレームワーク (米国立標準技術研究所 (NIST)、Payment Card Industry Security Standards Council (PCI)、国際標準化機構 (ISO) を含む) のセキュリティコントロールにマッピングするためのベストプラクティスをまとめています。
- 「[デベロッパーガイド](#)」の「[ルールによるリソースの評価](#)」 – この AWS Config サービスは、リソース設定が社内プラクティス、業界ガイドライン、および規制にどの程度準拠しているかを評価します。AWS Config
- [AWS Security Hub](#) – これにより AWS のサービス、内のセキュリティ状態を包括的に確認できます AWS。Security Hub では、セキュリティコントロールを使用して AWS リソースを評価し、セキュリティ業界標準とベストプラクティスに対するコンプライアンスをチェックします。サポートされているサービスとコントロールのリストについては、「[Security Hub のコントロールリファレンス](#)」を参照してください。
- [Amazon GuardDuty](#) — これにより AWS アカウント、疑わしいアクティビティや悪意のあるアクティビティがないか環境を監視することで、、、ワークロード、コンテナ、データに対する潜在的な脅威 AWS のサービス を検出します。GuardDuty は、特定のコンプライアンスフレームワークで義務付けられている侵入検知要件を満たすことで、PCI DSS などのさまざまなコンプライアンス要件への対応に役立ちます。
- [AWS Audit Manager](#) – これにより AWS のサービス、AWS 使用状況を継続的に監査し、リスクの管理方法と規制や業界標準への準拠を簡素化できます。

この AWS 製品またはサービスは、サポートする特定の Amazon Web Services (AWS) サービスを通じて [責任共有モデル](#) に従います。AWS サービスセキュリティ情報については、[AWS 「サービスセキュリティドキュメント」ページ](#) と [AWS、AWS コンプライアンスプログラムによるコンプライアンスの取り組みの対象となるサービス](#) を参照してください。

この AWS 製品またはサービスの耐障害性

AWS グローバルインフラストラクチャは、AWS リージョン およびアベイラビリティゾーンを中心に構築されています。

AWS リージョン は、低レイテンシー、高スループット、高冗長ネットワークで接続された、物理的に分離および隔離された複数のアベイラビリティゾーンを提供します。

アベイラビリティゾーンでは、ゾーン間で中断することなく自動的にフェイルオーバーするアプリケーションとデータベースを設計および運用することができます。アベイラビリティゾーンは、従

来の単一または複数のデータセンターインフラストラクチャよりも可用性が高く、フォールトトレラントで、スケーラブルです。

AWS リージョンとアベイラビリティゾーンの詳細については、[AWS 「グローバルインフラストラクチャ」](#) を参照してください。

この AWS 製品またはサービスは、サポートする特定の Amazon Web Services (AWS) サービスを通じて [責任共有モデル](#) に従います。AWS サービスセキュリティ情報については、[AWS 「サービスセキュリティドキュメント」](#) ページと [AWS、AWS コンプライアンスプログラムによるコンプライアンスの取り組みの対象となるサービス](#) を参照してください。

この AWS 製品またはサービスのインフラストラクチャセキュリティ

この AWS 製品またはサービスはマネージドサービスを使用するため、グローバルネットワークセキュリティによって保護されています。AWS セキュリティサービスと [ガインフラストラクチャ AWS を保護する方法](#) については、[AWS 「クラウドセキュリティ」](#) を参照してください。インフラストラクチャセキュリティのベストプラクティスを使用して AWS 環境を設計するには、「Security Pillar AWS Well-Architected Framework」の [「Infrastructure Protection」](#) を参照してください。

が AWS 公開した API コールを使用して、ネットワーク経由でこの AWS 製品またはサービスにアクセスします。クライアントは以下をサポートする必要があります：

- Transport Layer Security (TLS)。TLS 1.2 は必須で TLS 1.3 がお勧めです。
- DHE (楕円ディフィー・ヘルマン鍵共有) や ECDHE (楕円曲線ディフィー・ヘルマン鍵共有) などの完全前方秘匿性 (PFS) による暗号スイート。これらのモードは、Java 7 以降など、ほとんどの最新システムでサポートされています。

また、リクエストには、アクセスキー ID と、IAM プリンシパルに関連付けられているシークレットアクセスキーを使用して署名する必要があります。または、[AWS Security Token Service \(AWS STS\)](#) を使用して、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

この AWS 製品またはサービスは、サポートする特定の Amazon Web Services (AWS) サービスを通じて [責任共有モデル](#) に従います。AWS サービスセキュリティ情報については、[AWS 「サービスセキュリティドキュメント」](#) ページと [AWS、AWS コンプライアンスプログラムによるコンプライアンスの取り組みの対象となるサービス](#) を参照してください。

Amazon S3 暗号化クライアントの移行

このトピックでは、Amazon Simple Storage Service () 暗号化クライアントのバージョン 1 (V1 Amazon S3) からバージョン 2 (V2) にアプリケーションを移行し、移行プロセス全体でアプリケーションの可用性を確保する方法について説明します。

前提条件

Amazon S3 クライアント側の暗号化には、以下が必要です。

- Java 8 以降がアプリケーション環境にインストールされていること。は、[Oracle Java SE Development Kit](#) と、[Amazon Corretto](#) Red Hat OpenJDK、[AdoptOpenJDK](#) などの Open Java Development Kit (OpenJDK) のディストリビューションで AWS SDK for Java 動作します。
[OpenJDK](#)
- [Bouncy Castle Crypto パッケージ](#)。Bouncy Castle .jar ファイルをアプリケーション環境のクラスパスに配置するか、artifactId bcprov-ext-jdk15on (org.bouncycastle の groupId を使用) における依存関係を Maven pom.xml ファイルに追加できます。

移行の概要

この移行は 2 つのフェーズから構成されます。

1. 新しいフォーマットを読み取るために既存のクライアントを更新します。バージョン 1.11.837 以降の を使用するようにアプリケーションを更新 AWS SDK for Java し、アプリケーションを再デプロイします。これにより、アプリケーションの Amazon S3 クライアント側の暗号化サービスクライアントは、V2 サービスクライアントによって作成されたオブジェクトを復号化できます。アプリケーションで複数の AWS SDKs 個別に更新する必要があります。
2. 暗号化および復号クライアントを V2 に移行します。すべての V1 暗号化クライアントが V2 暗号化形式を読み取れるようになったら、アプリケーションコードの Amazon S3 クライアント側の暗号化クライアントと復号クライアントを更新して V2 に相当するものを使用します。

新しいフォーマットを読み取るために既存のクライアントを更新する

V2 暗号化クライアントは、 の古いバージョンではサポート AWS SDK for Java されていない暗号化アルゴリズムを使用します。

移行の最初のステップは、AWS SDK for Javaのバージョン 1.11.837 以降を使用するように V1 暗号化クライアントを更新することです。([Java API リファレンスバージョン 1.x](#) にある最新リリースバージョンに更新することをお勧めします。) これを実行するには、プロジェクト設定の依存関係を更新します。プロジェクト設定が更新されたら、プロジェクトを再構築して再デプロイします。

これらの手順を完了すると、アプリケーションの V1 暗号化クライアントは、V2 暗号化クライアントによって書き込まれたオブジェクトを読み取ることができるようになります。

プロジェクト設定の依存関係を更新する

プロジェクト設定ファイル (pom.xml や build.gradle など) を変更して、AWS SDK for Javaのバージョン 1.11.837 以降を使用します。その後、プロジェクトを再構築して再デプロイします。

新しいアプリケーションコードをデプロイする前にこのステップを完了すると、移行プロセス中にフリート全体で暗号化と復号の操作の一貫性を保つことができます。

Maven を使用した例

pom.xml ファイルからのスニペット:

```
<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>
```

Gradle を使用した例

build.gradle ファイルからのスニペット:

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

暗号化および復号クライアントを V2 に移行する

プロジェクトが最新の SDK バージョンで更新されたら、V2 クライアントを使用するようにアプリケーションコードを変更できます。これを実行するには、新しいサービスクライアントビルダーを使用するように、最初にコードを更新します。その後、名前が変更されたビルダーのメソッドを使用して暗号化マテリアルを提供し、必要に応じてサービスクライアントをさらに設定します。

これらのコードスニペットは、でクライアント側の暗号化を使用する方法を示し AWS SDK for Java、V1 暗号化クライアントと V2 暗号化クライアントの比較を提供します。

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();
```

上記の例では、`cryptoMode` を `AuthenticatedEncryption` に設定しています。これは、V1 暗号化クライアントによって書き込まれたオブジェクトを V2 暗号化クライアントが読み取れることを許可する設定です。クライアントが V1 クライアントによって書き込まれたオブジェクトを読み取る機能を必要としない場合は、代わりにデフォルト設定の `StrictAuthenticatedEncryption` を使用することをお勧めします。

V2 暗号化クライアントを構築する

V2 暗号化クライアントは、`AmazonS3EncryptionClientV2.encryptionBuilder ()` を呼び出すことで構築できます。

既存のすべての V1 暗号化クライアントを V2 暗号化クライアントに置き換えることができます。V2 暗号化クライアントは、「」を使用するように V2 暗号化クライアントを設定して許可する限り、V1 暗号化クライアントによって書き込まれたオブジェクトを常に読み取ることができます `AuthenticatedEncryption`cryptoMode`。

新しい V2 暗号化クライアントの作成は、V1 暗号化クライアントの作成方法と非常によく似ています。ただし、いくつかの違いがあります。

- `CryptoConfiguration` オブジェクトの代わりに、`CryptoConfigurationV2` オブジェクトを使用してクライアントを設定します。このパラメータは必須です。
- V2 暗号化クライアントのデフォルトの `cryptoMode` 設定は `StrictAuthenticatedEncryption` です。V1 暗号化クライアントの場合は `EncryptionOnly` です。
- 暗号化クライアントビルダーのメソッド `withEncryptionMaterials()` の名前が `withEncryptionMaterialsProvider()` に変更されました。これは、単に引数の型をより正確に反映するための外観上の変更です。サービスクライアントを設定する際には、新しいメソッドを使用する必要があります。

Note

AES-GCM で復号する場合は、復号されたデータの使用を開始する前に、オブジェクト全体を最後まで読み取ります。これは、オブジェクトが暗号化されてから変更されていないことを確認するためものステップです。

暗号化マテリアルプロバイダーを使用する

V1 暗号化クライアントで既に使用しているものと同じ暗号化マテリアルプロバイダーおよび暗号化マテリアルオブジェクトを引き続き使用できます。これらのクラスは、暗号化クライアントがデータを保護するために使用するキーを提供する役割を果たします。これらは、V2 および V1 暗号化クライアントの両方と互換的に使用できます。

V2 暗号化クライアントを設定する

V2 暗号化クライアントは `CryptoConfigurationV2` オブジェクトで設定されます。このオブジェクトは、デフォルトのコンストラクターを呼び出し、必要に応じてデフォルトからプロパティを変更することで構築できます。

CryptoConfigurationV2 のデフォルト値は次のとおりです。

- `cryptoMode = CryptoMode.StrictAuthenticatedEncryption`
- `storageMode = CryptoStorageMode.ObjectMetadata`
- `secureRandom = SecureRandom` のインスタンス
- `rangeGetMode = CryptoRangeGetMode.DISABLED`
- `unsafeUndecryptableObjectPassthrough = false`

EncryptionOnly は V2 暗号化クライアント `cryptoMode` ではサポートされていないことに注意してください。V2 暗号化クライアントは、常に認証された暗号化を使用してコンテンツを暗号化し、V2 KeyWrap オブジェクトを使用してコンテンツ暗号化キー (CEK) を保護します。

次の例は、V1 で暗号化設定を指定する方法と、CryptoConfigurationV2 暗号化クライアントビルダーに渡す V2 オブジェクトをインスタンス化する方法を示しています。

V1

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

V2

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

その他の例

次の例は、V1 から V2 への移行に関連する特定のユースケースに対処する方法を示しています。

V1 暗号化クライアントによって作成されたオブジェクトを読み取るようにサービスクライアントを設定する

以前に V1 暗号化クライアントを使用して作成されたオブジェクトを読み取るには、`cryptoMode` を `AuthenticatedEncryption` に設定します。次のコードスニペットは、この設定で設定オブジェクトを構築する方法を示しています。

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



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

オブジェクトのバイト範囲を取得するようにサービスクライアントを設定する

暗号化された S3 オブジェクトからバイト範囲を get できるようにするには、新しい設定 `rangeGetMode` を有効にします。この設定は、V2 暗号化クライアントではデフォルトで無効になっています。有効になっている場合でも、範囲設定された get は、クライアントの `cryptoMode` 設定でサポートされているアルゴリズムを使用して暗号化されたオブジェクトでのみ機能することに注意してください。詳細については、API リファレンス [CryptoRangeGetMode](#) の AWS SDK for Java 「」を参照してください。

を使用して V2 暗号化クライアントを使用して暗号化された Amazon S3 オブジェクトのマルチパートダウンロード Amazon S3 TransferManager を実行する場合は、まず V2 暗号化クライアントで `rangeGetMode` 設定を有効にする必要があります。

次のコードスニペットは、範囲設定された get を実行するために V2 クライアントを設定する方法を示しています。

```
// 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);
```

AWS SDK for Java 用の OpenPGP 鍵

AWS SDK for Java 用に一般公開されている Maven アーティファクトはすべて OpenPGP 標準を使用して署名されています。アーティファクトの署名を検証するのに必要な公開鍵については、以下のセクションを参照してください。

現在のキー

次の表は、SDK for Java 1x と SDK for Java 2.x の最新リリースの OpenPGP 鍵情報を示しています。

キー ID	0xAC107B386692DADD
タイプ	RSA
サイズ	4096/4096
作成	2016-06-30
有効期限	2024-10-08
ユーザー ID	AWS SDK とツール <aws-dr-tools@amazon.com>
キーフィンガープリント	FEB9 209F 2F2F 3F46 6484 1E55 AC10 7B38 6692 DADD

次の SDK for Java 用の OpenPGP 公開鍵をクリップボードにコピーするには、右上隅にある [コピー] アイコンを選択します。

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

```
xsFNBFd1gAUBEACqbmFbxdJgz1lD7wrlskQA1LLuSAC4p8ny9u/D2zLR8Ynk3Yz
mzJuQ+Kfjne2t+xTDex6MPJlMYp0viSWsX2psgvdmeyUpW9ap01rThNYkc+W5fRc
buFehfbi9LSATZGJi8RG0sCCr5FsYVz0gEk85M2+PeM24cXhQIOZtQUjswX/pdk/
KduGtZASqNAYLKR0mRODzUuaokLPo24pfm9bnr1RnRtw5ktPAA5bM9ZZaGKriej
kT2lPffBbjp8F5AZvmGLtNm2Cmg4FKBvI04SQjy2jjrQ3wBzi5Lc9HTxDuHK/rtV
u6PewUe2WP1nxlXenhMZU1UK4YoSB9E9StQ2VxQiySLHSdxR7Ma4WgYdVLn9b0ie
```

```
nj3QxLuQ1ZUKF79ES6JaM4t0z1gGcQeU1+Uk1gjFLuKwmzWRdEIFfxMyvH6qgKnd
U+DioH5mcUwhwffAAAsuIJyAdMIEUYh7IfzJJXQf+ff+Xf0Cl6by0JFWrIGQkAzMu
CEvaCfwtHC2Lpzo33/WRFEMAuzzd0QJ4uz4xFFvaS0SZHMLHWI9YV/+Pea3X99Ms
0Nlek/LolAJh67MynHeVB0HKrq+fluorWepQivctzN6Y1N0kx5naTPGGaKWK7G2q
TbcY5SMnkIwFLFSougj0Fvmjczq8iZRwYxWA+i+LQvsR9WEXEiQffIWRoQARAQAB
zsFNBFd1gAUBEAC8zNARpWb3dPMThL2xAY+fS60vXdB1Sk0tYJpDWPfGvo0d+VQ+
hV6Xu1GAHAS6xG1WHysPT9KejIRSgLG+e9CaM5yhsxNa1WFGUM4Q9ESo3t+a75Go
7xHIxgFjC046/06Vh3g9N/PREeuG8zkZ3H2v5fmD+ejyPgk4W9sFL00zjRiZD0FK
VYR/j9uenEC/2NBcLuFy3q6cDfmCoDE0062kXMnaGz3knzEK/X1SkcjsxRDq7zaQ
lQ1Kou+3dICwy4x5SJQ8j1+eeeEvF2C2/dXmDohb57tqUwioohMUQkmCtvZgEHjy
pUwgp0MTo25gWxkvJlSJKU0b6b1786WnySIzF2gxq1kkEmB14RAssQkeXjrSmGws
MDyHNqyJeYFus18sPaSpo+V2n0z+2B070Uq+wmf1S5A5FpegH0PZzzoNZo8I6Qxa
Zje9YSZUijGmZIdEBleRVt3Svhi8MY1nasd4bW2RK1sr7plkBf8QRe6biiQRf3KD
0Sn5CbmXpAcHJ1ZHzRRdkXZDNQC6vCJxsy1300TrhJtAV1Yq347uyUbVi291ISVg
roUVtprismHoEk5Go0THbg9SCSt+xi/FiJQC+ubWmIGXoFKMR3UmhDnnzobKcbtnbs
/Hd981FdVghYYvq//gTakJk0WxfGq030wtXRndPOA0T+qhP3TE+LtGRJ+wARAQAB
wsF1BBgBCgAPBQJXdYAFaHsMBQkHhh+AAAoJEKwQezhmktrdTyEP/0H0VWHwQsaW
jMrGj000MFzXGUo8SBmYYTBs29VM8wBGDsPkYCjeZzU16i9iqDpDqxyqmTigcjH
V8CDx/6xsMBLG2yKaKZ4m3+Yn0Qf/sQkyCvqiyMF9mS7pDYWy+mPhPuw8TDIfiqg
VhzjSpIMFWPqxVjn6KKbPN/QASr3Pf0cuP6qpHG+NAM6Q5dYkCebyvwzLmg1sVni
l6iSyJd1jBj3D34XrgWS9buyxBB2CjIM76WxfNViJ9zAaPI78X9v6PpDGn0kg6oL
zrusrvBjoZknKQm0SZ+41fx6xvrTPs8uPEzevzJB1kke6kw9+KagY8mrVX1ZenRg
+sY/4vxJreYWQeq167ggx+wFjKDcfhZA7m70LH0DysrGVCLcmuinUBaN1HmLDcGY
XZ+kMCoXf0bpuCVByQmNJgEb47EIFlx/+TEeNHKM0+22xL1atFzXfkEVZck+NghL
ZyFDhS3g1bma7puU7r752uiJjA6Iv8+kHDXi+/V7GNpuiEFUYh69QQ2//CS5H51o
sC/Bkb9evSn/Lp8dMubtWAaXDGJMgw9vqZ55N02NK0fvF/IKHnGkvH28rv00PCv0
WTA/MClv28y0PrSvvcMXnduLtkBEX7TISMPW+n+0Ta63/z4YFFEZ7sFLrEm3Q3vJ
MN3mE5i3cw+JGXPSu0nTtgqk/oZv//SS
=Z9u3
-----END PGP PUBLIC KEY BLOCK-----
```

ドキュメント履歴

このトピックでは、AWS SDK for Java デベロッパーガイドの履歴における重要な変更点について説明します。

このドキュメントは、2024年5月21日に作成されました。

2024年5月21日

Java コマンドラインシステムプロパティを使用して `networkaddress.cache.ttl` セキュリティプロパティを設定する手順を削除します。「[JVM TTL を設定する方法](#)」を参照してください。

2024-01-12

AWS SDK for Java v1.x のサポート終了を知らせるバナーを追加します。

2023年12月6日

- [現在の OpenPGP 鍵](#) を指定します。

2023年3月14日

- IAM ベストプラクティスに沿ってガイドを更新しました。詳細については、「[IAM のセキュリティのベストプラクティス](#)」を参照してください。

2022年7月28日

- 2022年8月15日付けで EC2-Classic が廃止される旨のアラートを追加。

2018年3月22日

- そのツールがサポートされなくなったため、DynamoDB 例の Tomcat セッションの管理を削除しました。

2017年11月2日

- 新しいトピックを含む暗号化 Amazon S3 クライアントの暗号化例を追加しました。KMS [Amazon S3 マネージドキーによるクライアント側の暗号化](#) とクライアント側の暗号化、および [Amazon S3 クライアントマスターキーによるクライアント側の暗号化](#) を使用します。
[Amazon S3AWS](#)

2017年4月14日

- 新しいトピック「[バケットとオブジェクトのアクセス許可の管理](#)」やウェブサイトとしてのバケットの設定など、[Amazon S3 「を使用した例 AWS SDK for Java」](#) セクションにいくつかの更新を加えました。[Amazon S3 Amazon S3](#)

2017 年 4 月 4 日

- 新しいトピックである[AWS SDK for Javaのメトリクスを有効化](#)では、AWS SDK for Java用にアプリケーションおよび SDK パフォーマンスメトリクスを生成する方法を説明しています。

2017 年 4 月 3 日

- [CloudWatch 「を使用した例 AWS SDK for Java」](#) セクションに新しい CloudWatch 例を追加しました。[「からのメトリクス CloudWatchの取得」](#)、[「カスタムメトリクスデータの発行」](#)、[CloudWatch 「アラームの操作」](#)、[「でのアラームアクションの使用 CloudWatch」](#)、および [「へのイベントの送信 CloudWatch」](#)

2017 年 3 月 27 日

- [Amazon EC2 「インスタンス AWS SDK for Javaの管理」](#)、[「での Elastic IP アドレスの使用」](#)、[Amazon EC2 「リージョンとアベイラビリティゾーンの使用」](#)、[Amazon EC2 「キーペアの使用」](#)、[「セキュリティグループの使用」](#) セクションの例にさらに Amazon EC2 例を追加しました。[Amazon EC2](#) [Amazon EC2](#)

2017 年 3 月 21 日

- 新しい IAM の例のセットが [AWS SDK for Javaを使用した IAM の例](#)のセクションに追加されました。[IAM アクセスキーの管理](#)、[IAM ユーザーの管理](#)、[IAM アカウントエイリアスの使用](#)、[IAM ポリシーの使用](#)、および [IAM サーバー証明書の使用](#)です。

2017 年 3 月 13 日

- Amazon SQS セクションに、[Amazon SQS メッセージキューのロングポーリングの有効化、での可視性タイムアウトの設定 Amazon SQS](#)、[でのデッドレターキューの使用](#)という 3 つの新しいトピックを追加しました [Amazon SQS](#)。

2017 年 1 月 26 日

- [「の使用」](#) セクションに新しい Amazon S3 トピック [「オペレーションでの AWS SDK for Javaの使用」](#)と、[「トピックを使用した AWS 開発の新しいベストプラクティス AWS SDK for Java」](#)を追加しました。[TransferManager Amazon S3](#)

2017 年 1 月 16 日

- 新しい Amazon S3 トピック [「バケットポリシーを使用したバケットへのアクセス Amazon S3 の管理」](#)と、[「メッセージキューの使用 Amazon SQS」](#)と [「メッセージの受信と削除」](#)の 2 つの新しい Amazon SQS トピックを追加しました。[Amazon SQS](#)

2016 年 12 月 16 日

- の新しいトピック例を追加しました DynamoDB: [のテーブルの操作 DynamoDB](#)と [のアイテムの操作 DynamoDB](#)。

2016 年 9 月 26 日

- 詳細セクションのトピックは、SDK [の使用の中心となるため AWS SDK for Java](#)、「[」の使用](#)」に移動しました。

2016 年 8 月 25 日

- 新しいトピック「[サービスクライアントの作成](#)」が「[AWS SDK for Javaの使用](#)」に追加されました。このトピックでは、クライアントビルダーを使用して AWS のサービス クライアントの作成を簡素化する方法を示します。

[AWS SDK for Java 「コード例」セクションが更新され、完全なサンプルコードを含むのリポジトリによってバックアップされる S3 の新しい例が追加されました。](#) [GitHub](#)

2016 年 5 月 02 日

- 新しいトピックである[非同期プログラミング](#)が [AWS SDK for Javaの使用](#)セクションに追加され、Future オブジェクトを返す、または AsyncHandler を取る非同期クライアントメソッドで作業する方法について説明しています。

2016 年 4 月 26 日

- 「SSL 証明書の要件」トピックは、関連性がなくなったため削除されました。SHA-1 署名証明書のサポートは 2015 年に廃止され、テストスクリプトを保存したサイトが削除されました。

2016 年 3 月 14 日

- Amazon SWF セクションに新しいトピック「[Lambda タスク](#)」を追加しました。このトピックでは、従来の Amazon SWF アクティビティを使用する代わりに、Lambda をタスクとして呼び出す Amazon SWF ワークフローを実装する方法について説明します。

2016 年 3 月 4 日

- [AWS SDK for Javaを使用したAmazon SWF の例](#)のセクションが新しい内容に更新されました。
 - [Amazon SWF 基本](#) - プロジェクトに SWF を含める方法に関する基本情報を提供します。
 - [シンプルな Amazon SWF アプリケーションの構築](#) - を初めて使用する Java 開発者向けの step-by-step ガイダンスを提供する新しいチュートリアル Amazon SWF。
 - [アクティビティおよびワークフローワーカーの適切なシャットダウン](#) - Java の同時実行クラスを使用して、Amazon SWF ワーカークラスを適切にシャットダウンする方法を説明します。

2016 年 2 月 23 日

- AWS SDK for Java デベロッパーガイドのソースが [移動されましたaws-java-developer-guide](#)。


2015 年 12 月 28 日

- [the section called “DNS 名参照用の JVM TTL を設定する”](#) は Advanced から [Using the AWS SDK for Java](#) に移動され、わかりやすくするために書き直されました。

[Apache Maven での SDK の使用](#) が更新され、プロジェクトに SDK の部品表 (BOM) を含める方法に関する情報が反映されました。

2015 年 8 月 4 日

- SSL 証明書の要件は、[開始方法](#) のセクションの新規トピックです。AWS の SSL 接続用の SHA256 署名証明書への移行について、および 2015 年 9 月 30 日以降に AWS へのアクセスで必要となる、以前の 1.6 と以前の Java 環境を修正してこれらの証明書を使用する方法について説明しています。

 Note

Java 1.7+ はすでに SHA256 署名証明書を使用できます。

2014 年 5 月 14 日

- 新しいガイド構造をサポートするために、[入門資料](#) と [入門資料](#) が大幅に改訂され、[開発用の AWS 認証情報とリージョンを設定する](#) 方法に関するガイダンスが追加されました。

[コードサンプル](#) の説明が「[その他のドキュメントとリソース](#)」セクションの独自のトピックに移動されました。

[SDK の改訂履歴の表示](#) 方法に関する情報が、「はじめに」に移動されました。

2014 年 5 月 9 日

- AWS SDK for Java ドキュメントの全体的な構造が簡素化され、「開始方法」および [追加のドキュメントとリソース](#) のトピックが更新されました。

新しいトピックが追加されました。

- [AWS 認証情報の使用](#) - AWS SDK for Java とともに使用する認証情報を指定するさまざまな方法について説明します。
- [IAM ロールを使用しての AWS リソースへのアクセスを許可する - EC2 Amazon EC2](#) インスタンスで実行されているアプリケーションの認証情報を安全に指定する方法に関する情報を提供します。EC2

2013 年 9 月 9 日

- このドキュメント履歴のトピックに、AWS SDK for Java デベロッパーガイドの変更が記載されるようになりました。リリースノートの履歴とあわせて利用してください。