デベロッパーガイド

AWS Flow Framework for Java

API バージョン 2021-04-28

AWS Flow Framework for Java: デベロッパーガイド

Table of Contents

AWS Flow Framework for Java とは

を使用すると AWS Flow Framework、ワークフローロジックの実装に集中できます。バックグラウンドでは、フレームワークは Amazon SWF のスケジューリング、ルーティング、状態管理機能を使用してワークフローの実行を管理し、スケーラブルで信頼性が高く、監査可能にします。 AWS Flow Frameworkベースのワークフローは非常に同時実行性が高いです。ワークフローを複数のコンポーネントに分散させ、それぞれを別々のコンピュータ上で別々のプロセスとして実行し、独立して拡張することができます。アプリケーションは、どのコンポーネントが動作していても進行し続けることができるため、高い耐障害性が保たれます。

このガイドの内容

このガイドでは、 をインストール、セットアップ、使用 AWS Flow Framework して Amazon SWF アプリケーションを構築する方法について説明します。

AWS Flow Framework for Java の開始方法

AWS Flow Framework for Java の使用を開始したばかりの場合は、AWS Flow Framework for Java の開始方法「」セクションをお読みください。for AWS Flow Framework Java のダウンロードとインストール、開発環境の設定方法、ワークフロー作成の簡単な例について説明します。

Java AWS Flow Framework の理解

基本的な Amazon SWF と AWS Flow Framework 概念を紹介し、アプリケーションの基本構造 AWS Flow Framework と、分散ワークフローの部分間でのデータ交換方法について説明します。

AWS Flow Framework for Java プログラミングガイド

この章では、 AWS Flow Framework for Java を使用してワークフローアプリケーションを開発するための基本的なプログラミングのガイダンスを提供します。アクティビティタイプとワークフロータイプの登録方法、ワークフロークライアントの実装方法、子ワークフローの作成方法、エラーの処理方法などについて説明します。

AWS Flow Framework for Java のタスクについて

この章では、 AWS Flow Framework for Java の仕組みについて詳しく説明し、非同期ワークフローの実行順序と標準ワークフロー実行の論理的なステップスルーに関する追加情報を提供します。

AWS Flow Framework for Java のトラブルシューティングとデバッグのヒント

この章では、一般的なエラーについて説明します。この情報を使用してワークフローのトラブルシューティングを行い、一般的なエラーの回避方法を確認できます。

AWS Flow Framework for Java リファレンス

この章は、 AWS Flow Framework for Java が SDK for Java に追加する注釈、例外、パッケージへの参照です。

AWS Flow Framework for Java の開始方法

このセクションでは、基本的なプログラミングモデルと API を導入する一連の簡単なサンプルアプリケーションを順を追 AWS Flow Framework って説明します。サンプルアプリケーションは、C および付随するプログラミング言語を導入するために使用される標準の Hello World アプリケーションに基づいています。Hello World の一般的な Java 実装は次のとおりです。

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello World!");
   }
}
```

サンプルアプリケーションの簡単な説明を以下に示します。完全なソースコードが含まれているため、自分でアプリケーションを実装して実行することができます。開始する前に、まず開発環境を設定し、 のように AWS Flow Framework for Java プロジェクトを作成する必要がありますAWS Flow Framework for Java のセットアップ。

- HelloWorld アプリケーション では、標準の Java アプリケーションとして Hello World を実装しますが、ワークフローアプリケーションのように構成して、ワークフローアプリケーションを導入します。
- HelloWorldWorkflow アプリケーション は AWS Flow Framework for Java を使用して、HelloWorld を Amazon SWF ワークフローに変換します。
- HelloWorldWorkflowAsync アプリケーション では、`HelloWorldWorkflow` を変更して、非同期ワークフローメソッドを使用します。
- HelloWorldWorkflowDistributed アプリケーション では、ワークフローおよびアクティビティワーカーを別々のシステムで実行できるように、`HelloWorldWorkflowAsync` を変更します。
- HelloWorldWorkflowParallel アプリケーション では、`HelloWorldWorkflow` を変更して、2 つのアクティビティを並列に実行します。

AWS Flow Framework for Java のセットアップ

AWS Flow Framework for Java は に含まれていますAWS SDK for Java。をまだ設定していない場合は AWS SDK for Java、「 AWS SDK for Java デベロッパーガイド」の「開始方法」で SDK 自体のインストールと設定に関する情報を参照してください。

Maven でフローフレームワークを追加する

Amazon SWF ビルドツールはオープンソースです。コードを表示またはダウンロードするか、ツールを自分で作成するには、https://github.com/aws/aws-swf-build-tools のリポジトリを参照してください。

Amazon は、Maven Central Repository に Amazon SWF ビルドツールを提供しています。

Maven でフローフレームワークを設定するには、次の依存関係をプロジェクトの pom.xml ファイルを追加します。

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-swf-build-tools</artifactId>
    <version>2.0.0</version>
</dependency>
```

HelloWorld アプリケーション

Amazon SWF アプリケーションの開発方法を導入するために、ワークフローのように動作するが、1 つのプロセスでローカルに実行する Java アプリケーションを作成します。アマゾン ウェブ サービスへの接続は必要ありません。

> **Note**
>
> HelloWorldWorkflow の例では、これを基に構成されており、Amazon SWF に接続してワークフローを管理します。

ワークフローアプリケーションは、3 つの基本コンポーネントで構成されています。

- アクティビティワーカーは、一連のアクティビティをサポートしており、各アクティビティは、特定のタスクを実行するために個別に実行するメソッドです。
- ワークフローワーカーは、アクティビティの実行を調整し、データフローを管理します。ワークフロートポロジーのプログラムによる結論です。これは、基本的に、順番または同時に実行されるかどうかに関係なく、さまざまなアクティビティを実行する際に定義するフローチャートです。
- ワークフロースターターは、実行と呼ばれるワークフローインスタンスを開始し、実行時にそのインスタンスと通信できます。

HelloWorld は 3 つのクラスと 2 つの関連インターフェイスとして実装されます。詳細については、次のセクションで説明しています。開始する前に、「」の説明に従って開発環境を設定し、新しい AWS Java プロジェクトを作成する必要がありますAWS Flow Framework for Java のセットアップ。次のウォークスルーで使用されているパッケージの名前はすべて、helloWorld.*XYZ* です。このような名前を使用するには、次のように aop.xml で within 属性を設定します。

```
...
<weaver options="-verbose">
   <include within="helloWorld..*"/>
</weaver>
```

HelloWorld を実装するには、 という名前の AWS SDK プロジェクトに新しい Java パッケージを作成しhelloWorld.HelloWorld、次のファイルを追加します。

- インターフェイス (GreeterActivities.java)
- クラスファイル (GreeterActivitiesImpl.java)。アクティビティワーカーを実装します。
- インターフェイス (GreeterWorkflow.java)。
- クラスファイル (GreeterWorkflowImpl.java)。ワークフローワーカーを実装します。
- クラスファイル (GreeterMain.java)。ワークフロースターターを実装します。

各コンポーネントの完全なコードなどの詳細は、以下のセクションに記載されています。このコードは適切なファイルに追加できます。

HelloWorld アクティビティの実装

HelloWorld は、あいさつ ("Hello World!") をコンソールに出力する全体的なタスクを 3 つのタスクに分割します。各タスクは、 アクティビティメソッド で実行されます。このアクティビティメソッドは、次のように GreeterActivities インターフェイスで定義されています。

```
public interface GreeterActivities {
   public String getName();
   public String getGreeting(String name);
   public void say(String what);
}
```

HelloWorld には 1 つのアクティビティ (GreeterActivitiesImpl) が実装されています。このアクティビティでは、次のように GreeterActivities メソッドが使用されます。

```
public class GreeterActivitiesImpl implements GreeterActivities {
   @Override
   public String getName() {
      return "World";
   }

   @Override
   public String getGreeting(String name) {
      return "Hello " + name + "!";
   }

   @Override
   public void say(String what) {
      System.out.println(what);
   }
}
```

アクティビティは互いに独立しており、多くの場合、異なるワークフローで使用できます。たとえば、どのワークフローでも、say アクティビティを使用して文字列をコンソールに出力できます。ワークフローでは、複数のアクティビティを実装して、それぞれ異なるタスクセットを実行できます。

HelloWorld ワークフローワーカー

「Hello World!」を コンソールに出力するには、正しいデータのアクティビティタスクを適切な順序で実行します。HelloWorld ワークフローワーカーは、以下の図に示すように、シンプルなリニアワークフロートポロジーに基づき、アクティビティの実行を調整します。

3 つのアクティビティは、順番に実行され、そのデータはアクティビティからアクティビティに流れます。

HelloWorld ワークフローワーカーには、ワークフローのエントリポイントとなる 1 つのメソッドがあります。このメソッドは、次のように GreeterWorkflow インターフェイスで定義されています。

```
public interface GreeterWorkflow {
   public void greet();
```

```
}
```

GreeterWorkflowImpl クラスは、次のようにこのインターフェイスを実装します。

```
public class GreeterWorkflowImpl implements GreeterWorkflow{
   private GreeterActivities operations = new GreeterActivitiesImpl();

   public void greet() {
      String name = operations.getName();
      String greeting = operations.getGreeting(name);
      operations.say(greeting);
   }
}
```

greet メソッドは、GreeterActivitiesImpl のインスタンスを作成して、適切な順番で各アクティビティメソッドを呼び出し、各メソッドに適切なデータを渡して、HelloWorld トポロジーを実装します。

HelloWorld ワークフロースターター

ワークフロースターターは、ワークフロー実行を開始するアプリケーションです。実行中にワークフローと通信することもあります。GreeterMain クラスは、次のように、HelloWorld ワークフロースターターを実装します。

```
public class GreeterMain {
   public static void main(String[] args) {
      GreeterWorkflow greeter = new GreeterWorkflowImpl();
      greeter.greet();
   }
}
```

GreeterMain は GreeterWorkflowImpl のインスタンスを作成し、greet を呼び出してワークフローワーカーを実行します。Java アプリケーションとして GreeterMain を実行すると、「Hello World!」が コンソール出力に表示されます。

HelloWorldWorkflow アプリケーション

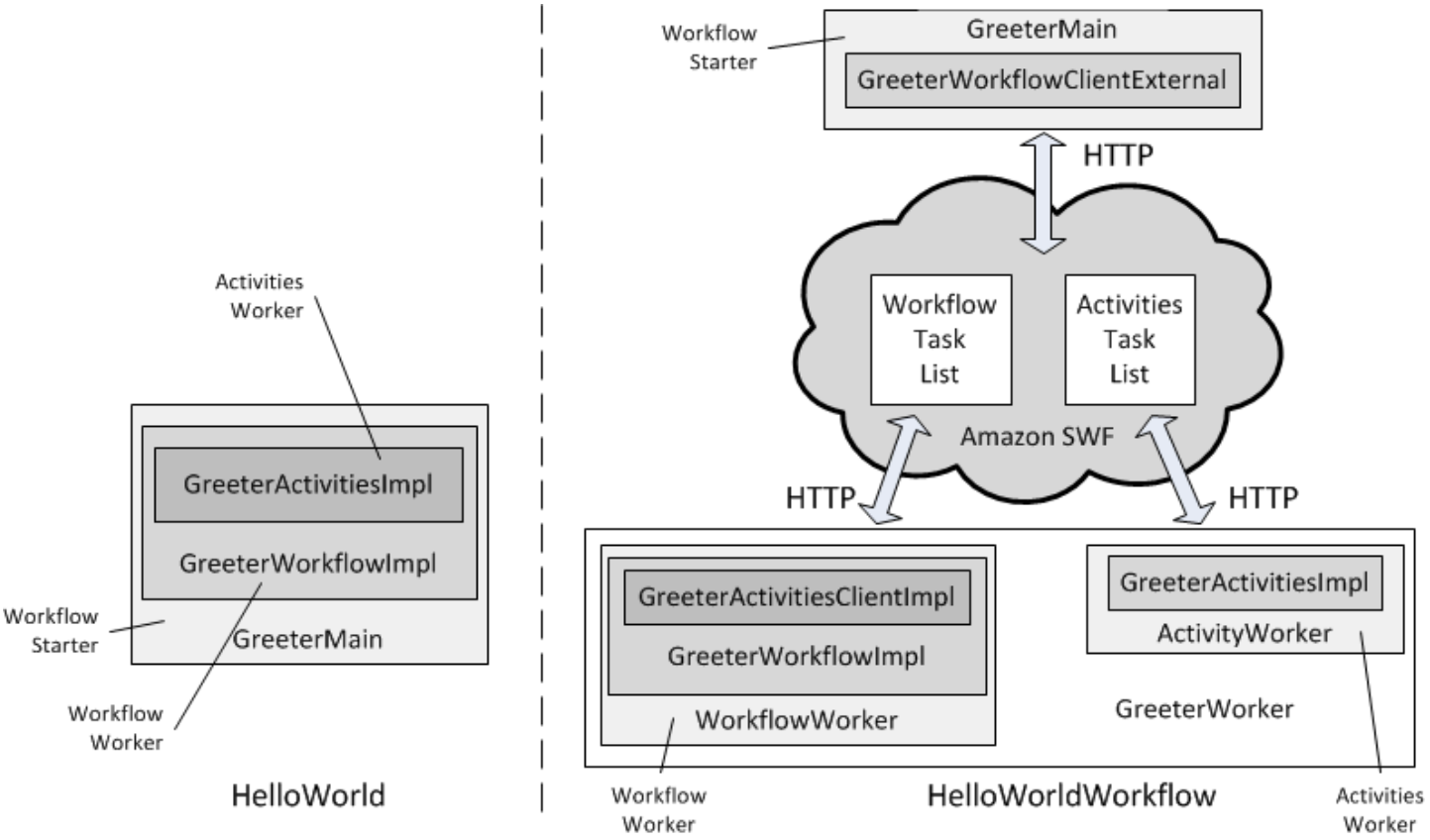
基本的な HelloWorld 例は、ワークフローのように構成されていますが、Amazon SWF ワークフローとはいくつかの重要な点で異なります。

従来のアプリケーションと Amazon SWF ワークフローアプリケーション

HelloWorld	Amazon SWF ワークフロー
単一のプロセスとしてローカルで実行されます。	複数のプロセスとして複数のシステム (Amazon EC2 インスタンス、プライベートデータセンター、クライアントコンピュータなど) に分散して実行できます。各プロセスで同じオペレーティングシステムを実行する必要もありません。
アクティビティは同期メソッドであり、完了するまで他はブロックされます。	アクティビティは非同期メソッドとして即座に戻り、完了するまで待つ間にワークフローで他のタスクを実行できます。
ワークフローワーカーは、アクティビティワーカーとやり取りするために、適切なメソッドを呼び出します。	ワークフローワーカーは、仲介として動作する Amazon SWF で HTTP リクエストを使用して、アクティビティワーカーとやり取りします。
ワークフロースターターは、ワークフローワーカーとやり取りするために、適切なメソッドを呼び出します。	ワークフロースターターは、仲介として動作する Amazon SWF で HTTP リクエストを使用して、ワークフローワーカーとやり取りします。

分散非同期ワークフローアプリケーションは、ワークフローワーカーとアクティビティワーカーのやり取りに直接ウェブサービスの呼び出しを使用することで、ゼロから実装することもできます。ただし、その場合は、複数のアクティビティの非同期実行を管理したり、データフローを処理したりするために必要なすべての複雑なコードを実装する必要があります。 AWS Flow Framework for Java と Amazon SWF はこれらの詳細をすべて処理するため、ビジネスロジックの実装に集中できます。

HelloWorldWorkflow は、HelloWorld の修正バージョンであり、Amazon SWF ワークフローとして実行されます。次の図は、2 つのアプリケーションの大まかな仕組みを示しています。



HelloWorld は単一のプロセスとして実行されます。スターター、ワークフローワーカー、およびアクティビティワーカーのやり取りには、従来のメソッド呼び出しが使用されます。`HelloWorldWorkflow`では、スターター、ワークフローワーカー、およびアクティビティワーカーは、HTTP リクエストを使用して Amazon SWF を介して相互作用する分散コンポーネントです。Amazon SWF は、ワークフローおよびアクティビティタスクのリストを維持することで相互作用を管理し、それぞれのコンポーネントにディスパッチします。このセクションでは、HelloWorldWorkflow を使用して、このフレームワークの仕組みを説明します。

HelloWorldWorkflow は AWS Flow Framework for Java API を使用して実装されます。この API は、バックグラウンドで Amazon SWF とやり取りする際の複雑な詳細を処理し、開発プロセスを大幅に簡素化します。Java アプリケーション用に設定されている HelloWorld AWS Flow Framework と同じプロジェクトを使用できます。ただし、アプリケーションを実行するには、Amazon SWF アカウントを以下のように設定する必要があります。

- AWS アカウントをお持ちでない場合は、Amazon Web Services でサインアップします。
- アカウントのアクセス ID とシークレットキー ID を、AWS_ACCESS_KEY_ID 環境変数と AWS_SECRET_KEY 環境変数にそれぞれ割り当てます。コードのリテラルキー値は公開しないようお勧めします。環境変数に保存すると、問題に対処しやすくなります。

- Amazon Simple Workflow Service で、Amazon SWF アカウントにサインアップします。
- にログイン AWS マネジメントコンソール し、Amazon SWF サービスを選択します。
- 右上隅の [ドメインの管理] を選択し、新しい Amazon SWF ドメインを登録します。ドメインは、アプリケーションリソース (ワークフロータイプ、アクティビティタイプ、ワークフロー実行など) の論理コンテナです。ドメイン名は任意に指定できます。このウォークスルーでは「helloWorldWalkthrough」を使用します。

HelloWorldWorkflow を実装するには、helloWorld.HelloWorld パッケージのコピーをプロジェクトディレクトリに作成し、名前を helloWorld.HelloWorldWorkflow とします。以下のセクションでは、元の HelloWorld コードを変更して AWS Flow Framework for Java を使用し、Amazon SWF ワークフローアプリケーションとして実行する方法について説明します。

HelloWorldWorkflow アクティビティワーカー

HelloWorld では、アクティビティワーカーを単一のクラスとして実装しました。 AWS Flow Framework for Java アクティビティワーカーには 3 つの基本コンポーネントがあります。

- アクティビティメソッド — 実際のタスクを実行するこのメソッドは、インターフェイスで定義され、関連するクラスに実装されます。
- ActivityWorker クラスは、アクティビティメソッドと Amazon SWF 間のやり取りを管理します。
- アクティビティホストアプリケーションは、アクティビティワーカーを登録して開始し、クリーンアップを処理します。

このセクションでは、アクティビティメソッドについて説明します。他の 2 つのクラスについては後で説明します。

HelloWorldWorkflow は、アクティビティインターフェイスを `GreeterActivities` で次のように定義します。

```
import com.amazonaws.services.simpleworkflow.flow.annotations.Activities;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.ActivityRegistrationOptions;

@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
                             defaultTaskStartToCloseTimeoutSeconds = 10)
@Activities(version="1.0")

public interface GreeterActivities {
```

```
    public String getName();
    public String getGreeting(String name);
    public void say(String what);
}
```

このインターフェイスは HelloWorld に厳密に必要なものではありませんが、 AWS Flow Framework for Java アプリケーション用です。インターフェイスの定義自体は変わっていないことに注意してください。ただし、Java 注釈 AWS Flow Framework には 2 つの @ActivityRegistrationOptionsと @Activitiesをインターフェイス定義に適用する必要があります。注釈は設定情報を提供し、 AWS Flow Framework for Java 注釈プロセッサにインターフェイス定義を使用してアクティビティクライアントクラスを生成するように指示します。これについては、後で説明します。

@ActivityRegistrationOptions には、いくつかの名前付きの値があり、アクティビティの動作を設定するために使用されます。HelloWorldWorkflow では 2 つのタイムアウトを指定します。

- defaultTaskScheduleToStartTimeoutSeconds は、アクティビティのタスクリストでタスクをキューイングできる時間を指定します。300 秒 (5 分) に設定されています。
- defaultTaskStartToCloseTimeoutSeconds は、アクティビティでタスクを実行する最大許容時間を指定します。10 秒に設定されています。

これらのタイムアウトにより、適切な時間内にアクティビティでタスクが完了されます。どちらかのタイムアウトを超えると、フレームワークでエラーが生成されます。ワークフローワーカーはエラーの処理方法を決定する必要があります。このようなエラーの処理方法については、「エラー処理」を参照してください。

@Activities には複数の値がありますが、通常はアクティビティのバージョン番号を指定するだけです。このバージョン番号により、世代が異なるアクティビティ実装を追跡できます。アクティビティインターフェイスを Amazon SWF に登録した後で変更した場合 (@ActivityRegistrationOptions の値を変更するなど) は、新しいバージョン番号を使用する必要があります。

HelloWorldWorkflow では、次のようにアクティビティメソッドを GreeterActivitiesImpl に実装します。

```
public class GreeterActivitiesImpl implements GreeterActivities {
    @Override
    public String getName() {
        return "World";
    }
```

```
    @Override
    public String getGreeting(String name) {
        return "Hello " + name;
    }
    @Override
    public void say(String what) {
        System.out.println(what);
    }
}
```

このコードは HelloWorld の実装と同じであることに注意してください。アクティビティの中核となるの AWS Flow Framework は、一部のコードを実行して結果を返すメソッドにすぎません。標準的なアプリケーションと Amazon SWF ワークフローアプリケーションの違いは、ワークフローでアクティビティを実行する方法、アクティビティの実行場所、および結果をワークフローワーカーに返す方法にあります。

HelloWorldWorkflow ワークフローワーカー

Amazon SWF ワークフローワーカーには 3 つの基本コンポーネントがあります。

- ワークフロー実装は、ワークフロー関連のタスクを実行するクラスです。
- アクティビティクライアントクラスは、基本的にアクティビティクラスのプロキシであり、アクティビティメソッドを非同期的に実行するためにワークフロー実装で使用されます。
- WorkflowWorker クラスは、ワークフローと Amazon SWF のやり取りを管理します。

このセクションでは、ワークフロー実装とアクティビティクライアントについて説明します。`WorkflowWorker`クラスについては後で説明します。

HelloWorldWorkflow では、次のようにワークフローインターフェイスを `GreeterWorkflow` で定義します。

```
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 3600)
public interface GreeterWorkflow {
    @Execute(version = "1.0")
```

```
    public void greet();
}
```

このインターフェイスは、HelloWorld に厳密に必要なわけではありませんが、 AWS Flow Framework for Java アプリケーションに不可欠です。ワークフローインターフェイス定義@WorkflowRegistrationOptionsには、Java 注釈 AWS Flow Framework に 2 つの @Workflowと を適用する必要があります。注釈は設定情報を提供し、後で説明するように、 AWS Flow Framework for Java 注釈プロセッサがインターフェイスに基づいてワークフロークライアントクラスを生成するように指示します。

@Workflow には、デフォルト値である dataConverter でよく使用される dataConverter というオプションのパラメータが 1 つあります。これは、JsonDataConverter を使用する必要があることを示します。 NullDataConverter

@WorkflowRegistrationOptions にも複数のオプションのパラメータがあります。これらは、ワークフローワーカーの設定に使用できます。ここでは、defaultExecutionStartToCloseTimeoutSeconds を設定し、ワークフローの実行時間を指定します。最大は 3600 秒 (1 時間) です。

GreeterWorkflow インターフェイス定義は、HelloWorld とは 1 つの重要な点 (@Execute 注釈) で異なります。ワークフローインターフェイスで指定するメソッドは、アプリケーション (ワークフロースターターなど) から呼び出すことができます。これらのメソッドの数は限られており、メソッドごとにロールが異なります。フレームワークはワークフローインターフェイスメソッドの名前やパラメータリストを指定しません。ユーザーがワークフローに適した名前やパラメータリストを使用し、 AWS Flow Framework for Java の注釈を適用してメソッドのロールを特定します。

@Execute には 2 つの目的があります。

- ワークフローのエントリポイントとして greet を識別します。これは、ワークフローを開始するために、ワークフロースターターで呼び出すメソッドです。通常、エントリポイントには 1 つ以上のパラメータを設定できます。これにより、スターターはワークフローを初期化できます。ただし、この例では初期化が不要です。
- ワークフローのバージョン番号を指定します。これにより、世代が異なるワークフロー実装を追跡できます。ワークフローインターフェイスを Amazon SWF に登録した後で変更する (タイムアウト値を変更するなど) には、新しいバージョン番号を使用する必要があります。

ワークフローインターフェイスに設定できる他のメソッドについては、「ワークフローコントラクトとアクティビティコントラクト」を参照してください。

HelloWorldWorkflow では、次のようにワークフローを GreeterWorkflowImpl に実装します。

```
import com.amazonaws.services.simpleworkflow.flow.core.Promise;

public class GreeterWorkflowImpl implements GreeterWorkflow {
   private GreeterActivitiesClient operations = new GreeterActivitiesClientImpl();

   public void greet() {
     Promise<String> name = operations.getName();
     Promise<String> greeting = operations.getGreeting(name);
     operations.say(greeting);
   }
}
```

このコードは HelloWorld に似ていますが、2 つの重要な相違点があります。

- GreeterWorkflowImpl は、GreeterActivitiesImpl の代わりにアクティビティクライアントとして GreeterActivitiesClientImpl のインスタンスを作成し、クライアントオブジェクトのメソッドを呼び出してアクティビティを実行します。
- 名前とグリーティングアクティビティは、String オブジェクトの代わりに Promise<String> オブジェクトを返します。

HelloWorld は単一のプロセスとしてローカルで実行される標準の Java アプリケーションです。そのため、GreeterWorkflowImpl でワークフロートポロジーを実装するには、単に GreeterActivitiesImpl のインスタンスを作成し、メソッドを順に呼び出して、戻り値をアクティビティ間で受け渡すだけです。Amazon SWF ワークフローの場合も同様に、アクティビティのタスクは GreeterActivitiesImpl のアクティビティメソッドで実行されます。ただし、このメソッドはワークフローと同じプロセスで実行されない場合があります。同じシステムで実行されない場合さえあります。したがって、ワークフローではアクティビティを非同期的に実行する必要があります。以上の要件に伴って以下の問題が生じします。

- 異なるプロセス (異なるシステムの場合もある) で実行されている可能性があるアクティビティメソッドを実行する方法。
- アクティビティメソッドを非同期的に実行する方法。
- アクティビティの入力と戻り値を管理する方法。たとえば、アクティビティ A の戻り値がアクティビティ B への入力である場合、アクティビティ A が完了するまでアクティビティ B の実行をブロックする必要があります。

使い慣れた Java フロー制御、アクティビティクライアント、および Promise<T> を組み合わせることで、さまざまなワークフロートポロジーをアプリケーションの制御フローで実装できます。

アクティビティクライアント

GreeterActivitiesClientImpl は、基本的に GreeterActivitiesImpl のプロキシであり、ワークフロー実装で GreeterActivitiesImpl のメソッドを非同期的に実行できます。

GreeterActivitiesClient クラスと GreeterActivitiesClientImpl クラスは自動生成されます。この自動生成には、GreeterActivities クラスに適用した注釈で提供される情報が使用されます。これらをユーザーが実装する必要はありません。

> **Note**
>
> プロジェクトを保存すると、これらのクラスが Eclipse で生成されます。生成されたコードは、プロジェクトディレクトリの .apt_generated サブディレクトリで確認できます。GreeterWorkflowImpl クラスでのコンパイルエラーを回避するには、.apt_generated ディレクトリを [Java Build Path] (Java ビルドパス) ダイアログボックスの [Order and Export] (オーダーとエクスポート) タブの一番上に移動することをお勧めします。

ワークフローワーカーは、対応するクライアントメソッドを呼び出すことで、アクティビティを実行します。このメソッドは非同期であり、Promise<T> オブジェクトを即座に返します。ここで T はアクティビティの戻り値の型です。返される Promise<T> オブジェクトは、基本的に、アクティビティメソッドから最終的に返される値のプレースホルダです。

- アクティビティクライアントメソッドが戻った最初の時点では、Promise<T> オブジェクトは準備ができていない状態です。この状態は、オブジェクトがまだ有効な戻り値を反映していないことを示します。
- 次に対応するアクティビティメソッドがタスクを完了して戻ると、フレームワークは Promise<T> オブジェクトに戻り値を割り当て、これを 準備完了状態にします。

プロミス <T>Type

Promise<T> の主目的は、非同期コンポーネント間のデータフローを管理し、これらのコンポーネントの実行時期を制御することです。これにより、アプリケーションでは、明示的に同期を管理したり、タイマーなどの機構に依存したりしなくても、非同期コンポーネントが予定より早く実行するのをブロックできます。アクティビティクライアントメソッドを呼び出すと、このメソッドは即座に戻

ります。ただし、フレームワークでは入力の Promise<T> オブジェクトが準備完了して有効なデータを反映するまで、対応するアクティビティメソッドの実行を遅らせます。

GreeterWorkflowImpl の観点からは、3 つすべてのアクティビティクライアントメソッドが即座に戻ります。GreeterActivitiesImpl の観点からは、フレームワークは name が完了するまで getGreeting を呼び出さず、getGreeting が完了するまで say を呼び出しません。

HelloWorldWorkflow では Promise<T> を使用してアクティビティ間でデータを受け渡すことで、アクティビティメソッドで無効なデータが使用されないようにします。さらに、アクティビティの実行時期を制御し、ワークフロートポロジーを暗黙で定義します。各アクティビティの Promise<T> の戻り値を次のアクティビティに渡すには、アクティビティを順に実行するためのリニアトポロジー (前述) を定義する必要があります。 AWS Flow Framework for Java では、特別なモデリングコードを使用して、標準の Java フロー制御と だけでなく、複雑なトポロジーも定義する必要はありませんPromise<T>。シンプルなパラレルトポロジーの実装方法を示す例については、「HelloWorldWorkflowParallel アクティビティワーカー」を参照してください。

> **Note**
>
> アクティビティメソッド (say など) が値を返さない場合は、対応するクライアントメソッドから Promise<Void> オブジェクトが返されます。オブジェクトをデータを反映しませんが、最初は準備未完了状態で、アクティビティが完了すると準備完了状態になります。したがって、Promise<Void> オブジェクトを他のアクティビティクライアントメソッドに渡し、元のアクティビティが完了するまで実行を遅らせることができます。

Promise<T> を使用すると、ワークフロー実装ではアクティビティクライアントメソッドとその戻り値を同期メソッドのように使用できます。ただし、Promise<T> オブジェクトの値にアクセスする場合は注意が必要です。Java の Future<T> タイプとは異なり、フレームワークで処理するのは Promise<T> の同期であり、アプリケーションではありません。Promise<T>.get を呼び出したときに、オブジェクトが準備未完了であれば、get から例外がスローされます。HelloWorldWorkflow が Promise<T> オブジェクトに直接アクセスすることはありません。オブジェクトをアクティビティ間で受け渡すだけです。オブジェクトが準備完了状態になると、フレームワークでは値を抽出し、その値を標準タイプとしてアクティビティメソッドに渡します。

Promise<T> オブジェクトは、必ず非同期コードでアクセスする必要があります。ここで、フレームワークは、オブジェクトを準備状態にし、有効な値を表します。HelloWorldWorkflow は、Promise<T> オブジェクトをアクティビティのクライアントメソッドにのみ渡して、問題を解決します。ワークフロー実装で Promise<T> オブジェクトの値にアクセスするには、オブジェクト

を 非同期ワークフローメソッドに渡します。このメソッドはアクティビティのように動作します。例については、HelloWorldWorkflowAsync アプリケーションを参照してください。

HelloWorldWorkflow のワークフロー実装とアクティビティ実装

ワークフロー実装とアクティビティ実装には、ワーカークラスとして ActivityWorker と WorkflowWorker に関連付けられています。これらのワーカークラスでは、Amazon SWF とアクティビティ/ワークフロー実装の間の通信を処理するために、適切な Amazon SWF タスクリストでタスクをポーリングし、各タスクの適切なメソッドを実行して、データフローを管理します。詳細については、「AWS Flow Framework 基本的な概念: アプリケーション構造」を参照してください。

アクティビティ/ワーカー実装と対応するワーカーオブジェクトを関連付けるには、以下の操作を行うワーカーアプリケーションを 1 つ以上実装します。

- ワークフローまたはアクティビティを Amazon SWF に登録する。
- ワーカーオブジェクトを作成してワークフローまたはアクティビティのワーカー実装に関連付ける。
- Amazon SWF との通信を開始するようワーカーオブジェクトに指示する。

ワークフローとアクティビティを別個のプロセスとして実行する場合は、ワークフローとアクティビティのワーカーホストを別個に実装する必要があります。例については、HelloWorldWorkflowDistributed アプリケーションを参照してください。シンプルにするために、HelloWorldWorkflow では以下のように 1 つのワーカーホストを実装し、アクティビティとワークフローのワーカーを同じプロセスで実行します。

```
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient;
import com.amazonaws.services.simpleworkflow.flow.ActivityWorker;
import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class GreeterWorker  {
   public static void main(String[] args) throws Exception {
     ClientConfiguration config = new ClientConfiguration().withSocketTimeout(70*1000);

     String swfAccessId = System.getenv("AWS_ACCESS_KEY_ID");
     String swfSecretKey = System.getenv("AWS_SECRET_KEY");
```

```
    AWSCredentials awsCredentials = new BasicAWSCredentials(swfAccessId,
 swfSecretKey);

    AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials,
 config);
    service.setEndpoint("https://swf.us-east-1.amazonaws.com");

    String domain = "helloWorldWalkthrough";
    String taskListToPoll = "HelloWorldList";

    ActivityWorker aw = new ActivityWorker(service, domain, taskListToPoll);
    aw.addActivitiesImplementation(new GreeterActivitiesImpl());
    aw.start();

    WorkflowWorker wfw = new WorkflowWorker(service, domain, taskListToPoll);
    wfw.addWorkflowImplementationType(GreeterWorkflowImpl.class);
    wfw.start();
  }
}
```

HelloWorld には `GreeterWorker` に対応するものがないため、`GreeterWorker` という Java クラスをプロジェクトに追加し、そのファイルにコード例をコピーする必要があります。

最初のステップとして、AmazonSimpleWorkflowClient オブジェクトを作成して設定します。このオブジェクトは基となる Amazon SWF サービスのメソッドを呼び出します。これを行うために、`GreeterWorker` で以下の操作を実行します。

1. ClientConfiguration オブジェクトを作成し、ソケットのタイムアウトとして 70 秒を指定します。この値により、ソケットが閉じるまでに、確立したオープン接続でデータを転送する時間を指定します。
2. BasicAWSCredentials オブジェクトを作成して AWS アカウントを識別し、アカウントキーをコンストラクタに渡します。利便性のためと、キーがコードでプレーンテキストとして公開されるのを避けるために、キーを環境変数として保存します。
3. ワークフローを表す AmazonSimpleWorkflowClient オブジェクトを作成し、`BasicAWSCredentials` オブジェクトと `ClientConfiguration` オブジェクトをコンストラクタに渡します。
4. クライアントオブジェクトのサービスエンドポイントの URL を設定します。Amazon SWF は現在、すべての AWS リージョンで利用可能です。

便宜上、GreeterWorker では 2 つの文字列定数を定義します。

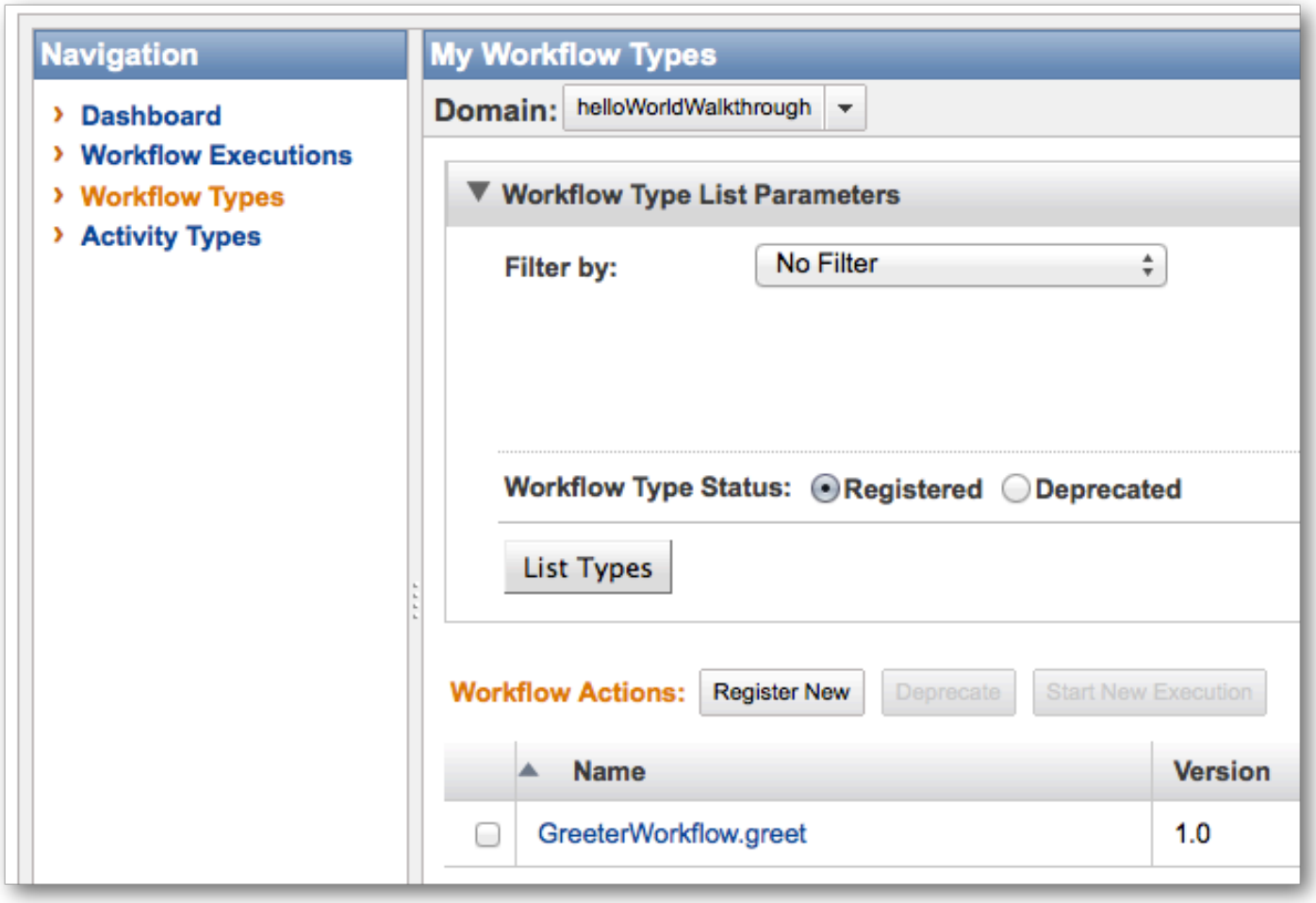
- domain は、ワークフローの Amazon SWF ドメイン名を設定したときに作成した Amazon SWF アカウントです。HelloWorldWorkflow では、「helloWorldWalkthrough」ドメインのワークフローが実行されていることを前提としています。
- taskListToPoll はタスクリストの名前です。ワークフローとアクティビティのワーカー間の通信を管理するために Amazon SWF で使用します。この名前として任意の便利な文字列を設定できます。HelloWorldWorkflow では、ワークフローとアクティビティの両方のタスクリストで「HelloWorldList」を使用します。バックグラウンドでは、これらの名前は最終的に別の名前空間に属するため、2 つのタスクリストは異なります。

GreeterWorker では、文字列定数と AmazonSimpleWorkflowClient オブジェクトを使用してワーカーオブジェクトを作成し、アクティビティ/ワーカー実装と Amazon SWF 間のやり取りを管理します。特に、ワーカーオブジェクトでは適切なタスクリストでタスクをポーリングするタスクを処理します。

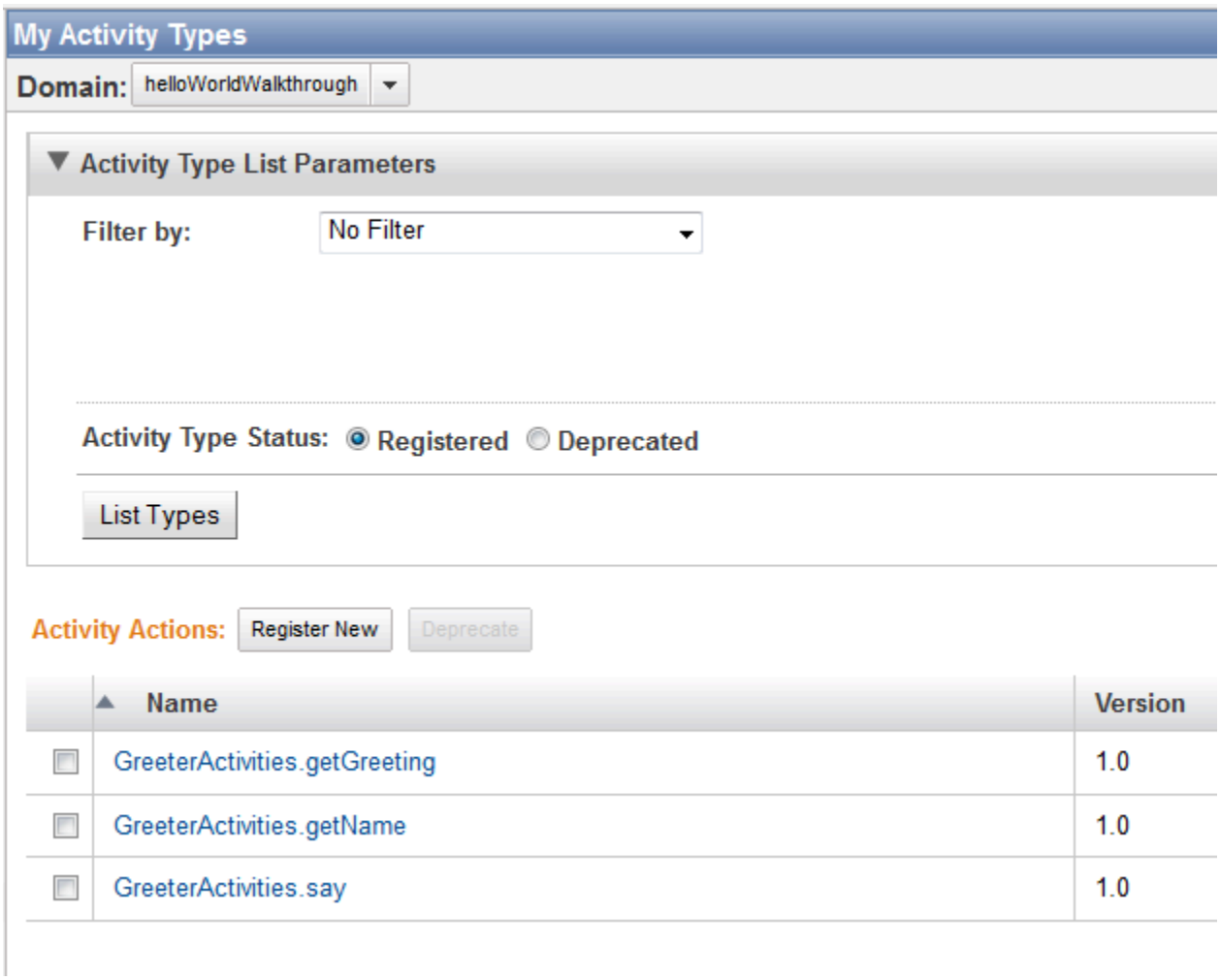
GreeterWorker は、ActivityWorker オブジェクトを作成し、新しいクラスインスタンスを追加して GreeterActivitiesImpl を処理するよう設定します。GreeterWorker はその後 ActivityWorker オブジェクトの start メソッドを呼び出します。これにより、指定されたアクティビティのタスクリストのポーリングを開始するようオブジェクトに指示します。

GreeterWorker では、WorkflowWorker オブジェクトを作成して設定し、クラスファイル名として GreeterWorkflowImpl.class を追加して GreeterWorkflowImpl を処理できるようにします。次に、WorkflowWorker オブジェクトの start メソッドを呼び出し、このオブジェクトに対して指定されたアクティビティタスクリストのポーリングを開始するよう指示します。

この時点で GreeterWorker を正常に実行できます。ワークフローとアクティビティが Amazon SWF に登録され、ワーカーオブジェクトによる各タスクリストのポーリングが開始されます。これを検証するには、GreeterWorker を実行し、Amazon SWF コンソールに移動して、ドメインのリストから helloWorldWalkthrough を選択します。[Navigation] (ナビゲーション) ペインで [Workflow Types] (ワークフロータイプ) を選択すると、GreeterWorkflow.greet が表示されるはずです。



[アクティビティタイプ] を選択すると、`GreeterActivities` メソッドが表示されます。



ただし、[Workflow Executions] (ワークフロー実行) を選択すると、アクティブな実行は表示されません。ワークフローワーカーとアクティビティワーカーはタスクをポーリングしますが、ワークフロー実行はまだ開始していないためです。

HelloWorldWorkflow スターター

仕上げとして、ワークフロースターターを実装します。スターターは、ワークフロー実行を開始するアプリケーションです。実行状態は Amazon SWF で保存され、その履歴と実行ステータスを確認できるようになります。HelloWorldWorkflow では、ワークフロースターターを実装するために、次のようにGreeterMain クラスを変更します。

```
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient;
```

```
public class GreeterMain {

   public static void main(String[] args) throws Exception {
     ClientConfiguration config = new ClientConfiguration().withSocketTimeout(70*1000);

     String swfAccessId = System.getenv("AWS_ACCESS_KEY_ID");
     String swfSecretKey = System.getenv("AWS_SECRET_KEY");
     AWSCredentials awsCredentials = new BasicAWSCredentials(swfAccessId,
 swfSecretKey);

     AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials,
 config);
     service.setEndpoint("https://swf.us-east-1.amazonaws.com");

     String domain = "helloWorldWalkthrough";

     GreeterWorkflowClientExternalFactory factory = new
 GreeterWorkflowClientExternalFactoryImpl(service, domain);
     GreeterWorkflowClientExternal greeter = factory.getClient("someID");
     greeter.greet();
   }
}
```

GreeterMain は、まず GreeterWorker と同じコードを使用して AmazonSimpleWorkflowClient オブジェクトを作成します。次に、GreeterWorkflowClientExternal オブジェクトを作成します。このオブジェクトは、ワークフローのプロキシとして機能しますが、その方法は GreeterWorkflowClientImpl で作成したアクティビティクライアントがアクティビティメソッドのプロキシとして機能する方法とほぼ同じです。ワークフロークライアントオブジェクトを作成するには、new を使用しないで、次の手順に従います。

1. 外部クライアントのファクトリオブジェクトを作成し、AmazonSimpleWorkflowClient オブジェクトと Amazon SWF ドメイン名をコンストラクタに渡します。クライアントのファクトリオブジェクトは、フレームワークの注釈プロセッサで作成します。このプロセッサでは、ワークフローインターフェイス名に「ClientExternalFactoryImpl」と付加するだけでオブジェクト名を作成します。
2. 外部クライアントオブジェクトを作成するために、ファクトリオブジェクトの getClient メソッドを呼び出し、ワークフローインターフェイス名に「ClientExternal」と付加してオブジェクト名を作成します。オプションとして、getClient に文字列を渡します。Amazon SWF では、

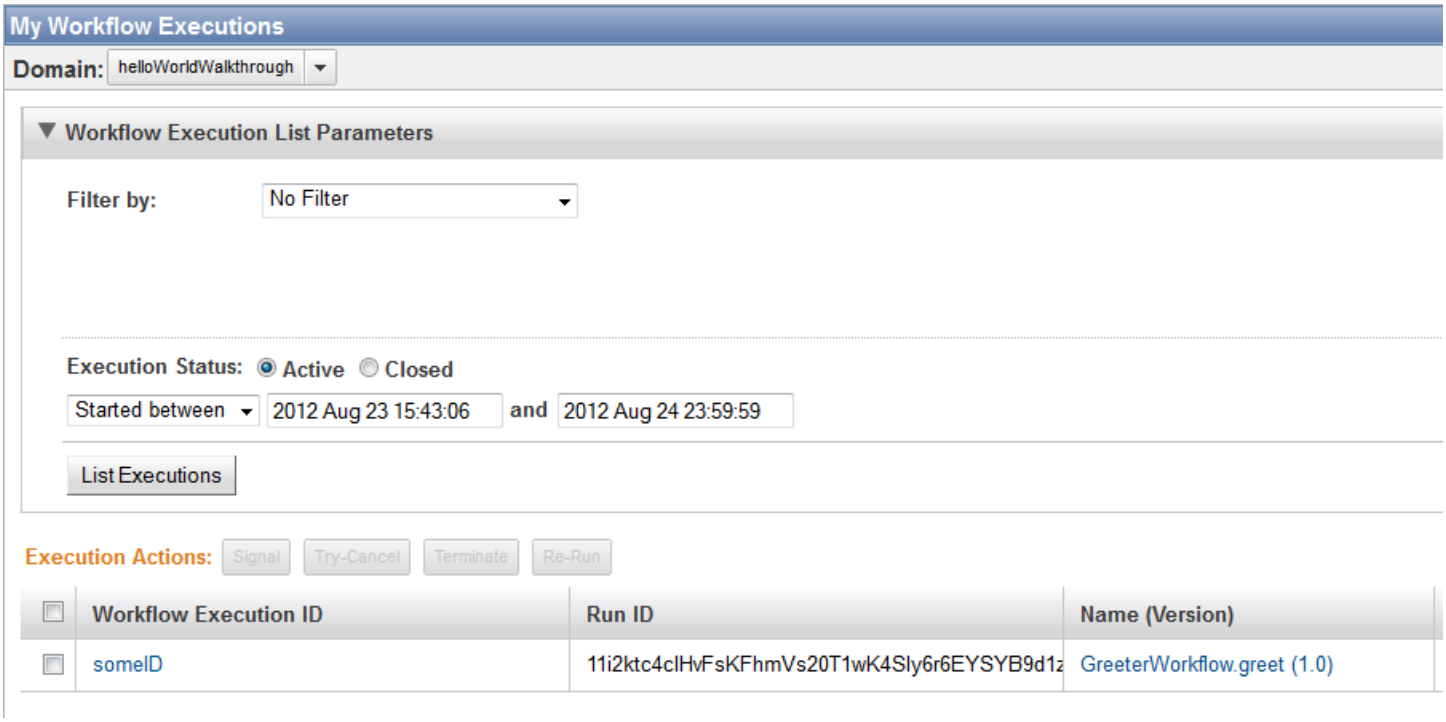
この文字列を使用して、このワークフローインスタンスを識別します。それ以外の場合、Amazon SWF は生成された GUID を使用してワークフローインスタンスを識別します。

ファクトリから返されるクライアントでは、getClient メソッドに渡された文字列を名前に使用するワークフローのみを作成します (ファクトリから返されるクライアントはすでに Amazon SWF 内に状態があります)。別の ID のワークフローを実行するには、ファクトリに戻り、別の ID を指定して新しいクライアントを作成する必要があります。

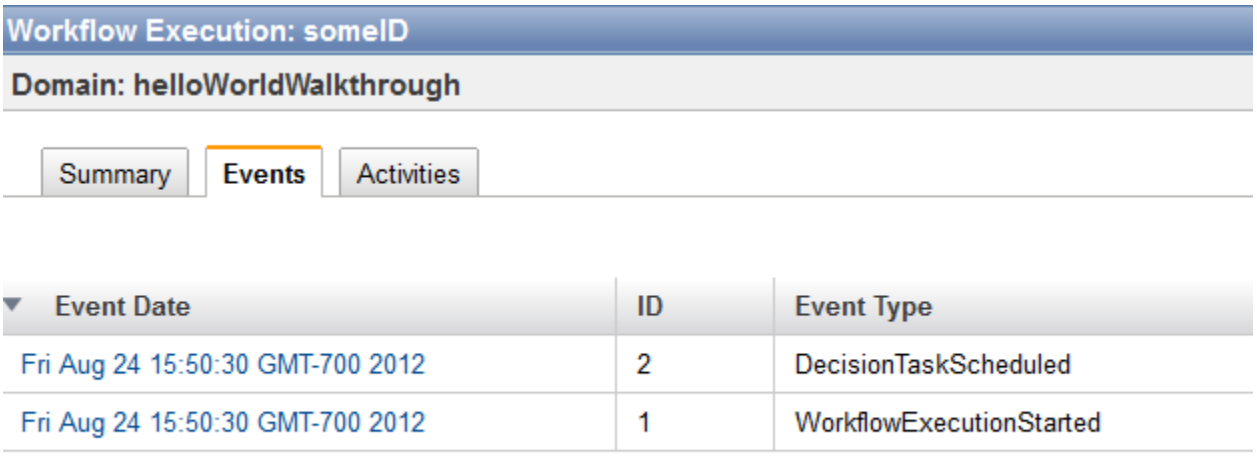
ワークフロークライアントは `greet` メソッドを公開します。`GreeterMain` では、このメソッドを呼び出してワークフローを開始します。`@Execute` 注釈で指定した `greet()` メソッドと同様です。

> **Note**
>
> 注釈プロセッサでは、内部クライアントのファクトリオブジェクトも作成されます。このオブジェクトは、子ワークフローの作成に使用されます。詳細については、「子ワークフロー実行」を参照してください。

`GreeterWorker` がまだ実行中の場合は、これを一時的にシャットダウンして `GreeterMain` を実行します。これで、Amazon SWF コンソールのアクティブなワークフロー実行のリストに someID が表示されます。



someID を選択して [イベント] タブを選択すると、イベントが表示されます。

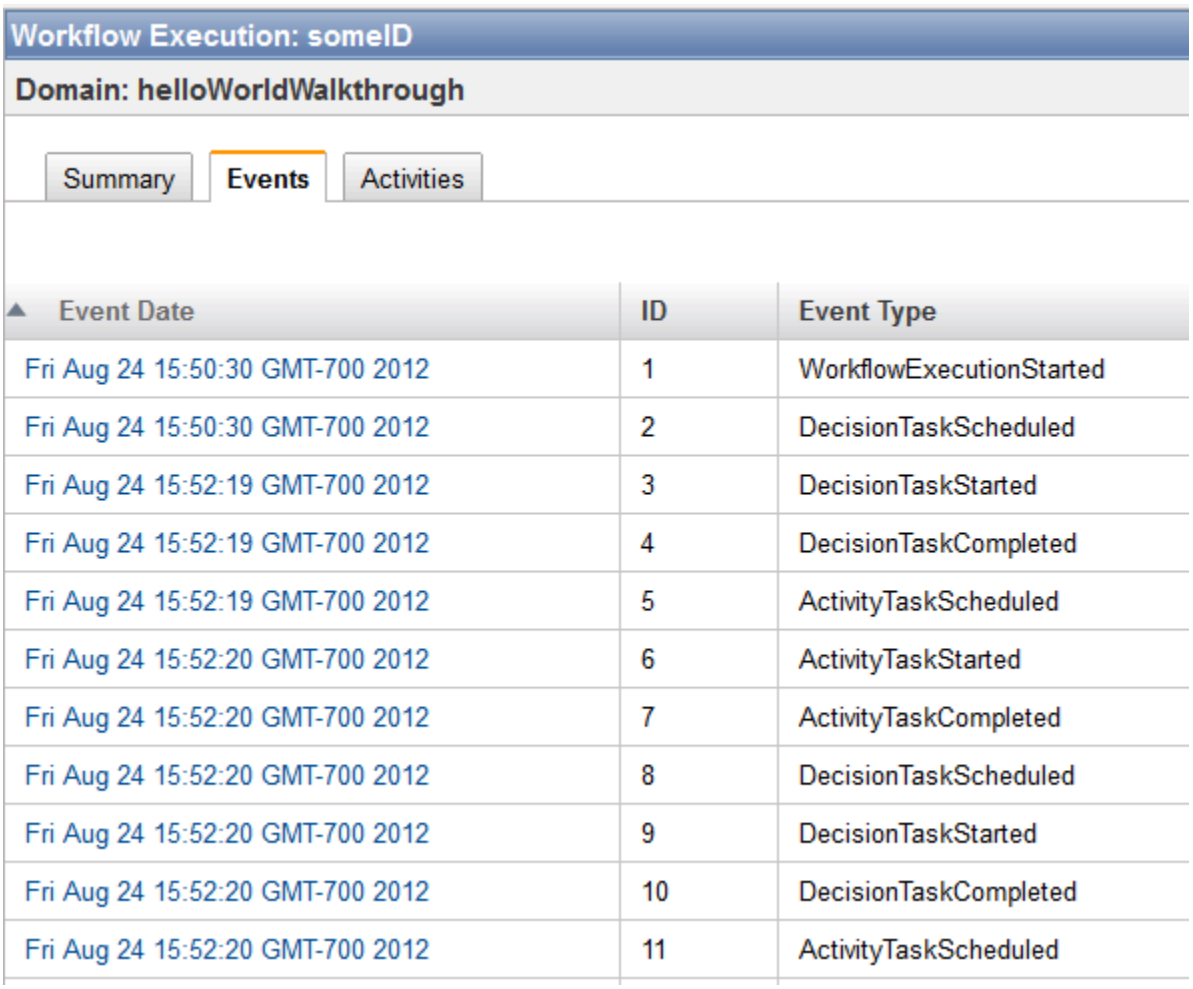


> **Note**
>
> GreeterWorker を早く開始して、まだ実行中の場合は、後で説明する理由のために表示されるイベントリストが長くなります。GreeterWorker を停止し、GreaterMain の実行を再試行してください。

[イベント] タブには 2 つのイベントのみが表示されます。

- WorkflowExecutionStarted は、ワークフローの実行を開始したことを示します。
- DecisionTaskScheduled は、Amazon SWF が最初の決定タスクをキューに入れたことを示します。

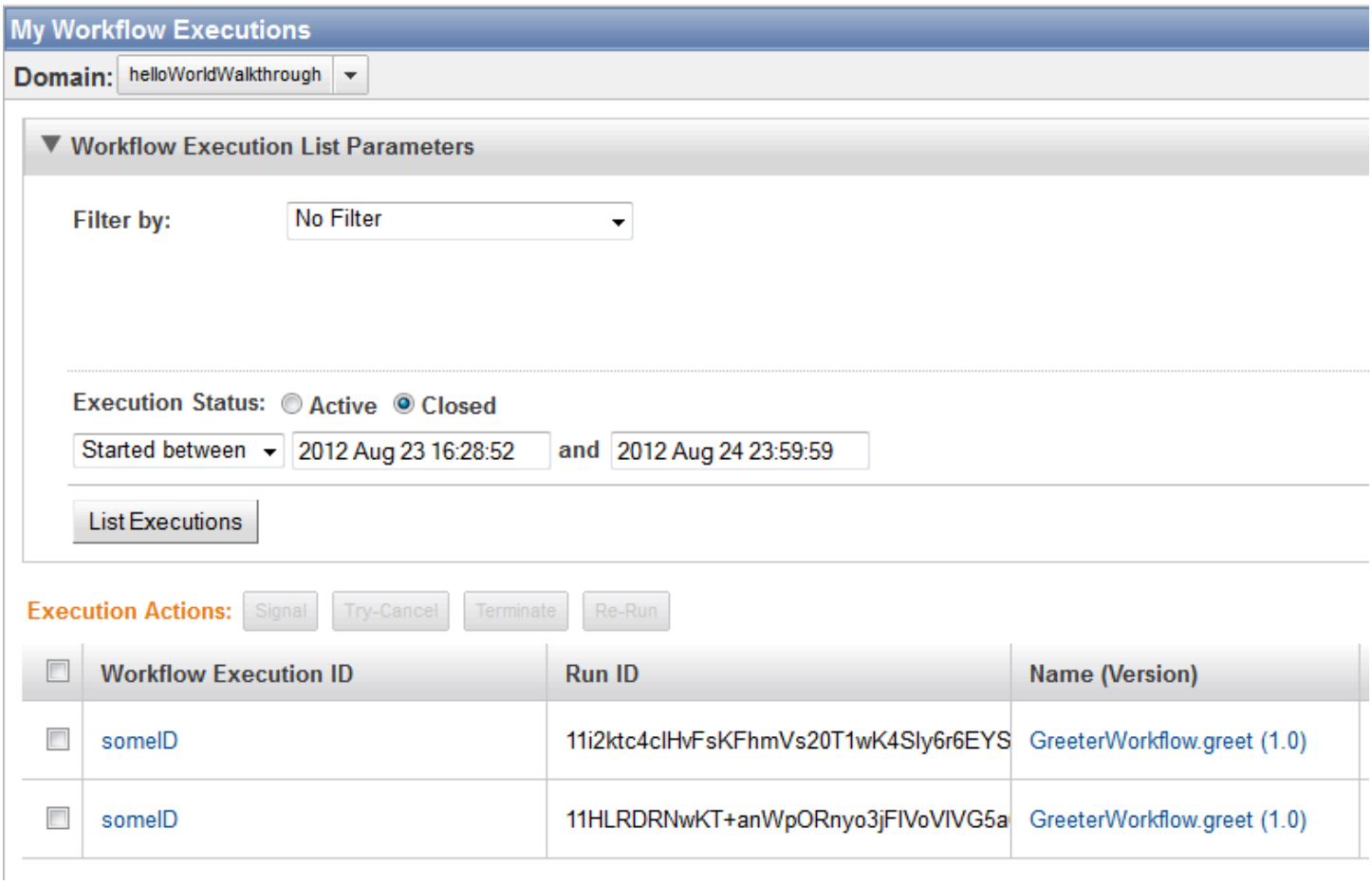
最初の決定タスクでワークフローがブロックされた理由は、ワークフローは、2 つのアプリケーション (GreeterMain と GreeterWorker) 間で分散されたからです。GreeterMain によってワークフロー実行が開始されましたが、GreeterWorker が実行されていないため、ワーカーはリストのポーリングとタスクの実行を行いません。各アプリケーションは個別に実行できますが、最初の決定タスクを超えてワークフロー実行を進めるには両方のアプリケーションが必要です。ここで GreeterWorker を実行すると、ワークフローとアクティビティのワーカーはポーリングを開始し、さまざまなタスクが迅速に完了されます。ここで [Events] (イベント) タブを確認すると、最初のイベントのバッチが表示されます。



Event Date	ID	Event Type
Fri Aug 24 15:50:30 GMT-700 2012	1	WorkflowExecutionStarted
Fri Aug 24 15:50:30 GMT-700 2012	2	DecisionTaskScheduled
Fri Aug 24 15:52:19 GMT-700 2012	3	DecisionTaskStarted
Fri Aug 24 15:52:19 GMT-700 2012	4	DecisionTaskCompleted
Fri Aug 24 15:52:19 GMT-700 2012	5	ActivityTaskScheduled
Fri Aug 24 15:52:20 GMT-700 2012	6	ActivityTaskStarted
Fri Aug 24 15:52:20 GMT-700 2012	7	ActivityTaskCompleted
Fri Aug 24 15:52:20 GMT-700 2012	8	DecisionTaskScheduled
Fri Aug 24 15:52:20 GMT-700 2012	9	DecisionTaskStarted
Fri Aug 24 15:52:20 GMT-700 2012	10	DecisionTaskCompleted
Fri Aug 24 15:52:20 GMT-700 2012	11	ActivityTaskScheduled

個別のイベントを選択して詳細情報を取得できます。確認が終わるまでに、ワークフローで「Hello World!」が コンソールに出力されます。

完了したワークフローは、アクティブな実行のリストに表示されなくなります。ただし、再確認する場合は、実行ステータスボタンの [Closed] (終了) を選択し、[List Executions] (実行の一覧表示) を選択できます。これにより、すべての完了したワークフローインスタンスのうち、ドメインの作成時に指定した保持期間を超えていないものが、指定したドメイン (`helloWorldWalkthrough`) に表示されます。



各ワークフローインスタンスには [Run ID] (実行 ID) として一意の値が割り当てられています。異なるワークフローインスタンスに同じワークフロー ID を使用できますが、一度に 1 つのアクティブな実行に対してのみ使用できます。

HelloWorldWorkflowAsync アプリケーション

ワークフローは、アクティビティを使用せずに、一定のタスクをローカルで実行する方が望ましい場合があります。ただし、多くの場合、ワークフロータスクは、`Promise<T>` オブジェクトで表す値を処理する必要があります。`Promise<T>` オブジェクトを非同期ワークフローメソッドに渡すと、ただちに実行されますが、オブジェクトの準備が完了するまで `Promise<T>` オブジェクトの値にアクセスすることはできません。`true` を返すまでは `Promise<T>.isReady` をポーリングできますが、非効率なだけでなく、メソッドによって長時間ブロックされる場合があります。この場合は、非同期メソッドを使用する方が適切です。

非同期メソッドは、標準メソッドとよく似た方法で (多くの場合、ワークフロー実装クラスのメンバーとして) 実装され、ワークフロー実装のコンテキストで実行されます。非同期メソッドとして指

定するには、@Asynchronous の注釈を適用します。これにより、アクティビティとよく似た方法で処理するようにフレームワークに指示されます。

- ワークフロー実装で非同期メソッドが呼び出されると、ただちに返ります。非同期メソッドでは、通常 Promise<T> オブジェクトが返り、メソッドが完了すると準備状態になります。
- 非同期メソッドを 1 つ以上の Promise<T> オブジェクトに渡すと、すべての入力オブジェクトが準備状態になるまで、実行は延期されます。そのため、非同期メソッドでは、例外のリスクなしに、入力値 Promise<T> にアクセスすることができます。

> **Note**
>
> AWS Flow Framework for Java がワークフローを実行する方法のため、非同期メソッドは通常複数回実行されるため、それらは簡単な低オーバーヘッドタスクにのみ使用する必要があります。また、大規模な計算などの長時間タスクを実行するには、アクティビティを使用します。詳細については、「AWS Flow Framework 基本的な概念: 分散実行」を参照してください。

このトピックは、HelloWorldWorkflowAsync のチュートリアルであり、アクティビティのいずれかを非同期メソッドに置き換える HelloWorldWorkflow の改訂版です。アプリケーションを実装するには、helloWorld.HelloWorldWorkflow パッケージのコピーをプロジェクトディレクトリに作成し、helloWorld.HelloWorldWorkflowAsync という名前を付けます。

> **Note**
>
> このトピックは、「HelloWorld アプリケーション」および「HelloWorldWorkflow アプリケーション」トピックで説明している概念とファイルに基づき、作成されています。ファイルや概念に精通して先に進む前にこれらのトピックに表示されます。

以下のセクションでは、元の HelloWorldWorkflow コードを変更して非同期メソッドを使用する方法について説明します。

HelloWorldWorkflowAsync アクティビティの実装

HelloWorldWorkflowAsync は、アクティビティワーカーインターフェイスを次のように GreeterActivities に実装します。

```
import com.amazonaws.services.simpleworkflow.flow.annotations.Activities;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.ActivityRegistrationOptions;

@Activities(version="2.0")
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
                             defaultTaskStartToCloseTimeoutSeconds = 10)
public interface GreeterActivities {
   public String getName();
   public void say(String what);
}
```

このインターフェイスは、以下を除き、HelloWorldWorkflow で使用するものと似ています。

- getGreeting アクティビティ。このタスクは、非同期メソッドで処理されるようになりました。
- バージョン番号は 2.0 に設定されます。Amazon SWF を使用してアクティビティインターフェイスを登録したら、バージョン番号を変更する場合を除き、変更することはできません。

その他のアクティビティメソッドは、HelloWorldWorkflow と同様に実装します。GreeterActivitiesImpl から getGreeting を削除します。

HelloWorldWorkflowAsync ワークフローの実装

HelloWorldWorkflowAsync は、ワークフローのインターフェイスを次のように定義します。

```
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 3600)
public interface GreeterWorkflow {

   @Execute(version = "2.0")
   public void greet();
}
```

このインターフェイスは、HelloWorldWorkflow と同じものを使用する (新しいバージョン番号を除く)。アクティビティと同様、登録されたワークフローを変更するには、バージョンを変更する必要があります。

HelloWorldWorkflowAsync は、ワークフローを次のように実装します。

```
import com.amazonaws.services.simpleworkflow.flow.annotations.Asynchronous;
import com.amazonaws.services.simpleworkflow.flow.core.Promise;

public class GreeterWorkflowImpl implements GreeterWorkflow {
   private GreeterActivitiesClient operations = new GreeterActivitiesClientImpl();

   @Override
   public void greet() {
      Promise<String> name = operations.getName();
      Promise<String> greeting = getGreeting(name);
      operations.say(greeting);
   }

   @Asynchronous
   private Promise<String> getGreeting(Promise<String> name) {
      String returnString = "Hello " + name.get() + "!";
      return Promise.asPromise(returnString);
   }
}
```

HelloWorldWorkflowAsync は、getGreeting アクティビティを getGreeting 非同期メソッドと置き換えますが、greet メソッドは以下とほとんど同じ方法で動作します。

1. getName アクティビティを実行します。これにより、すぐに Promise<String> オブジェクト (name) が返ります。このオブジェクトは、オブジェクトの名前を表します。
2. getGreeting 非同期メソッドを呼び出して、name オブジェクトを渡します。Promise<String> オブジェクト、greeting が getGreeting よりすぐに返ります。これは、あいさつを表します。
3. say アクティビティを実行し、greeting オブジェクトに渡します。
4. getName が完了すると、name は準備状態になり、getGreeting はその値を使用してあいさつを作成します。
5. getGreeting が完了すると、greeting は準備状態になり、say によって文字列がコンソールに出力されます。

この場合、アクティビティクライアントを呼び出して getGreeting アクティビティを実行せずに、あいさつで非同期の getGreeting メソッドを呼び出すという点が異なります。最終的な結果は同じですが、getGreeting メソッドの動作は、getGreeting アクティビティとは少し異なります。

- ワークフローワーカーは、標準の関数呼び出しのセマンティクスを使用して getGreeting を実行します。ただし、アクティビティの非同期実行は Amazon SWF を介して行われます。
- getGreeting は、ワークフロー実装のプロセスで実行されます。
- getGreeting では、String オブジェクトではなく、Promise<String> オブジェクトが返ります。Promise で保持される文字列値を取得するには、その get() メソッドを呼び出します。ただし、アクティビティは非同期的に実行されているため、その戻り値はすぐには準備できない可能性があります。 get() は、非同期メソッドの戻り値が使用可能になるまで例外を発生させます。

  Promise の詳細な仕組みについては、「AWS Flow Framework 基本的な概念: アクティビティとワークフロー間のデータ交換」を参照してください。

getGreeting では、あいさつの文字列を静的な Promise.asPromise メソッドに渡して戻り値を作成します。このメソッドでは、適切なタイプの Promise<T> オブジェクトを作成して値を設定し、準備状態に設定します。

HelloWorldWorkflowAsync のワークフローおよびアクティビティのホストおよびスターター

HelloWorldWorkflowAsync は、ワークフロー実装とアクティビティ実装のホストクラスとして GreeterWorker を実装します。このメソッドでは、taskListToPoll という名前を除き、HelloWorldWorkflow と同様に実装されます。名前は「HelloWorldAsyncList」に設定されます。

```
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient;
import com.amazonaws.services.simpleworkflow.flow.ActivityWorker;
import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class GreeterWorker {
```

```
    public static void main(String[] args) throws Exception {
        ClientConfiguration config = new
 ClientConfiguration().withSocketTimeout(70*1000);

        String swfAccessId = System.getenv("AWS_ACCESS_KEY_ID");
        String swfSecretKey = System.getenv("AWS_SECRET_KEY");
        AWSCredentials awsCredentials = new BasicAWSCredentials(swfAccessId,
 swfSecretKey);

        AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials,
 config);
        service.setEndpoint("https://swf.us-east-1.amazonaws.com");

        String domain = "helloWorldWalkthrough";
        String taskListToPoll = "HelloWorldAsyncList";

        ActivityWorker aw = new ActivityWorker(service, domain, taskListToPoll);
        aw.addActivitiesImplementation(new GreeterActivitiesImpl());
        aw.start();

        WorkflowWorker wfw = new WorkflowWorker(service, domain, taskListToPoll);
        wfw.addWorkflowImplementationType(GreeterWorkflowImpl.class);
        wfw.start();
    }
}
```

HelloWorldWorkflowAsync では、ワークフロースターターを `GreeterMain` で実装します。これは、HelloWorldWorkflow と同様に実装されます。

ワークフローを実行するには、HelloWorldWorkflow の場合と同様に `GreeterWorker` および `GreeterMain` を実行します。

HelloWorldWorkflowDistributed アプリケーション

HelloWorldWorkflow および HelloWorldWorkflowAsync では、Amazon SWF は、ワークフローとアクティビティ実装間の相互作用を仲介しますが、1 つのプロセスとしてローカルに実行されます。`GreeterMain` は、個別のプロセスですが、同じシステム上で実行されます。

Amazon SWF の主な機能は、分散アプリケーションをサポートすることです。例えば、ワークフローワーカーは Amazon EC2 インスタンス、ワークフロースターターはデータセンターコンピュータ、アクティビティはクライアントデスクトップコンピュータといったように実行できます。さまざまなシステムでさまざまなアクティビティを実行できます。

HelloWorldWorkflowDistributed アプリケーションは、2 つのシステムと 3 つのプロセスでアプリケーションを分散するように、HelloWorldWorkflowAsync を拡張します。

- ワークフローとワークフロースターターは、別々のプロセスを 1 つのシステムで実行します。
- アクティビティは、別々のシステムで実行されます。

アプリケーションを実装するには、helloWorld.HelloWorldWorkflowAsync パッケージのコピーをプロジェクトディレクトリに作成し、helloWorld.HelloWorldWorkflowDistributed という名前を付けます。以下のセクションでは、元の HelloWorldWorkflowAsync コードを変更し、2 つのシステムと 3 つのプロセスでアプリケーションを分散する方法について説明します。

別々のシステムで実行するために、ワークフロー実装やアクティビティ実装、バージョン番号を変更する必要はありません。また、`GreeterMain` の変更も不要です。必要なのは、アクティビティとワークフローのホストの変更のみです。

HelloWorldWorkflowAsync では、1 つのアプリケーションをワークフローとアクティビティのホストとして使用できます。ワークフロー実装とアクティビティの実装を別々のシステムで実行するには、アプリケーションを個別に実装する必要があります。GreeterWorker をプロジェクトから削除して、2 つの新しいクラスファイル (GreeterWorkflowWorker および GreeterActivitiesWorker) を追加します。

HelloWorldWorkflowDistributed は、アクティビティホストを次のように GreeterActivitiesWorker に実装します。

```
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient;
import com.amazonaws.services.simpleworkflow.flow.ActivityWorker;

public class GreeterActivitiesWorker {
   public static void main(String[] args) throws Exception {
      ClientConfiguration config = new
 ClientConfiguration().withSocketTimeout(70*1000);

      String swfAccessId = System.getenv("AWS_ACCESS_KEY_ID");
      String swfSecretKey = System.getenv("AWS_SECRET_KEY");
      AWSCredentials awsCredentials = new BasicAWSCredentials(swfAccessId,
 swfSecretKey);
```

```
      AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials,
 config);
      service.setEndpoint("https://swf.us-east-1.amazonaws.com");

      String domain = "helloWorldExamples";
      String taskListToPoll = "HelloWorldAsyncList";

      ActivityWorker aw = new ActivityWorker(service, domain, taskListToPoll);
      aw.addActivitiesImplementation(new GreeterActivitiesImpl());
      aw.start();
   }
}
```

HelloWorldWorkflowDistributed は、ワークフローホストを次のように GreeterWorkflowWorker に実装します。

```
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient;
import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class GreeterWorkflowWorker {
   public static void main(String[] args) throws Exception  {
      ClientConfiguration config = new
 ClientConfiguration().withSocketTimeout(70*1000);

      String swfAccessId = System.getenv("AWS_ACCESS_KEY_ID");
      String swfSecretKey = System.getenv("AWS_SECRET_KEY");
      AWSCredentials awsCredentials = new BasicAWSCredentials(swfAccessId,
 swfSecretKey);

      AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials,
 config);
      service.setEndpoint("https://swf.us-east-1.amazonaws.com");

      String domain = "helloWorldExamples";
      String taskListToPoll = "HelloWorldAsyncList";

      WorkflowWorker wfw = new WorkflowWorker(service, domain, taskListToPoll);
      wfw.addWorkflowImplementationType(GreeterWorkflowImpl.class);
```

```
        wfw.start();
    }
}
```

GreeterActivitiesWorker は、WorkflowWorker コードを使用しない GreeterWorker、GreeterWorkflowWorker は、ActivityWorker コードを使用しない GreeterWorker です。

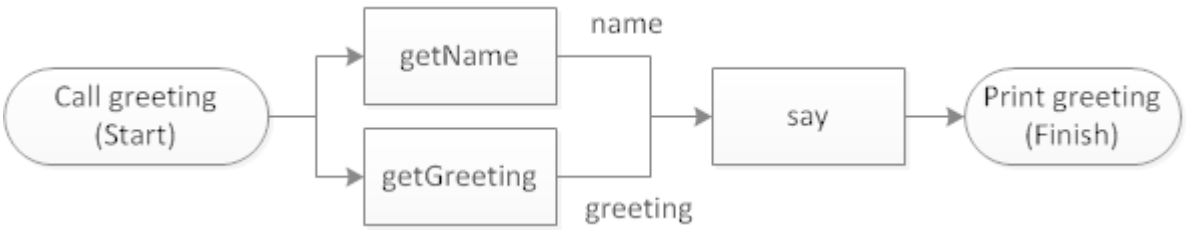
ワークフローを実行するには:

1. GreeterActivitiesWorker で実行できる JAR ファイルをエントリポイントとして作成します。
2. ステップ 1 から JAR ファイルを別のシステムにコピーします。これにより、Java をサポートするすべてのオペレーティングシステムを実行できます。
3. 同じ Amazon SWF ドメインにアクセスできる AWS 認証情報が他のシステムで使用可能であることを確認します。
4. JAR ファイルを実行します。
5. 開発システムで Eclipse を使用して、GreeterWorkflowWorker および GreeterMain を実行します。

アクティビティがワークフローワーカーやワークフロースターターとは異なるシステムで実行されているという事実を除き、ワークフローは HelloWorldAsync とまったく同じ方法で動作します。ただし、「Hello World!」という文字列をコンソールに出力する println コールは say アクティビティ内にあり、出力はアクティビティワーカーで実行されているシステムに表示されます。

HelloWorldWorkflowParallel アプリケーション

前述のバージョンの Hello World! 線形ワークフロートポロジーをすべて使用します。ただし、Amazon SWF はリニアトポロジーに制限されません。以下の図のように、HelloWorldWorkflowParallel アプリケーションは、変更されたバージョンの HelloWorldWorkflow であり、並列トポロジーが使用されています。



HelloWorldWorkflowParallel では、getName と getGreeting は並列に実行され、それぞれよりあいさつの一部が返ります。say はその後、2 つの文字列をグリーティングにマージし、コンソールに出力します。

アプリケーションを実装するには、helloWorld.HelloWorldWorkflow パッケージのコピーをプロジェクトディレクトリに作成し、helloWorld.HelloWorldWorkflowParallel という名前を付けます。以下のセクションでは、元の HelloWorldWorkflow コードを変更して、getName および getGreeting を並列に実行する方法について説明します。

HelloWorldWorkflowParallel アクティビティワーカー

HelloWorldWorkflowParallel アクティビティのインターフェイスは、以下の例に示すように、GreeterActivities で実装されます。

```
import com.amazonaws.services.simpleworkflow.flow.annotations.Activities;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.ActivityRegistrationOptions;

@Activities(version="5.0")
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
                             defaultTaskStartToCloseTimeoutSeconds = 10)
public interface GreeterActivities {
   public String getName();
   public String getGreeting();
   public void say(String greeting, String name);
}
```

このインターフェイスは、次の点を除き、HelloWorldWorkflow で使用するものと似ています。

- getGreeting は入力を一切受け付けません。あいさつの文字列のみ帰ります。
- say は、2 つの入力文字列 (あいさつと名前) を受け付けます。
- インターフェイスには新しいバージョン番号があります。登録済みのインターフェイスを変更する場合に必要です。

HelloWorldWorkflowParallel は、次のように GreeterActivitiesImpl でアクティビティを実装します。

```
public class GreeterActivitiesImpl implements GreeterActivities {

   @Override
```

```
    public String getName() {
        return "World!";
    }

    @Override
    public String getGreeting() {
        return "Hello ";
    }

    @Override
    public void say(String greeting, String name) {
        System.out.println(greeting + name);
    }
}
```

getName と getGreeting はあいさつの文字列の半分を返します。say は、その 2 つの部分を連結して完全なフレーズを生成し、コンソールに出力します。

HelloWorldWorkflowParallel ワークフローワーカー

HelloWorldWorkflowParallel ワークフローインターフェイスは、次のように GreeterWorkflow で実装されます。

```
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 3600)
public interface GreeterWorkflow {

    @Execute(version = "5.0")
    public void greet();
}
```

このクラスは、アクティビティワーカーに一致させるようにバージョン番号が変更されている点を除き、HelloWorldWorkflow バージョンと同じものです。

このワークフローは、次のように GreeterWorkflowImpl で実装されます。

```
import com.amazonaws.services.simpleworkflow.flow.core.Promise;
```

```
public class GreeterWorkflowImpl implements GreeterWorkflow {
   private GreeterActivitiesClient operations = new GreeterActivitiesClientImpl();

   public void greet() {
      Promise<String> name = operations.getName();
      Promise<String> greeting = operations.getGreeting();
      operations.say(greeting, name);
   }
}
```

ひと目で分かるように、この実装は、HelloWorldWorkflow にとてもよく似ています。3 つのアクティビティクライアントメソッドは順番に実行されます。ただし、アクティビティは実行されません。

- HelloWorldWorkflow は、name を getGreeting に渡しました。name は Promise<T> オブジェクトだったため、getGreeting は getName が完了するまでアクティビティを延期しました。その結果、2 つのアクティビティは順番に実行されました。
- HelloWorldWorkflowParallel では、入力 getName と getGreeting のいずれも渡されません。どちらのメソッドの実行も延期されず、そのアクティビティメソッドは並行ですぐに実行されます。

say アクティビティは、greeting と name のいずれも、入力パラメータとして取得します。これらは Promise<T> オブジェクトであるため、両方のアクティビティが完了するまで say で実行は延期されます。その後、あいさつが作成、出力されます。

HelloWorldWorkflowParallel では、ワークフロートポロジーの定義に特殊なモデル化コードは一切使用しません。これは、標準の Java フローコントロールを使用し、Promise<T>オブジェクトのプロパティを利用することによって暗黙的に行われます。 AWS Flow Framework for Java アプリケーションは、Promise<T>オブジェクトを従来の Java コントロールフロー構造と組み合わせて使用するだけで、複雑なトポロジーも実装できます。

HelloWorldWorkflowParallel ワークフローおよびアクティビティのホストおよびスターター

HelloWorldWorkflowParallel は、ワークフローとアクティビティ実装のホストクラスとして GreeterWorker を実装します。このメソッドでは、taskListToPoll という名前を除き、HelloWorldWorkflow と同様に実装されます。名前は「HelloWorldParallelList」に設定されます。

HelloWorldWorkflowParallel では、GreeterMain でワークフロースターターを実装します。この実装は、HelloWorldWorkflow と同様に行われます。

ワークフローを実行するには、HelloWorldWorkflow の場合と同様に GreeterWorker および GreeterMain を実行します。

Java AWS Flow Framework の理解

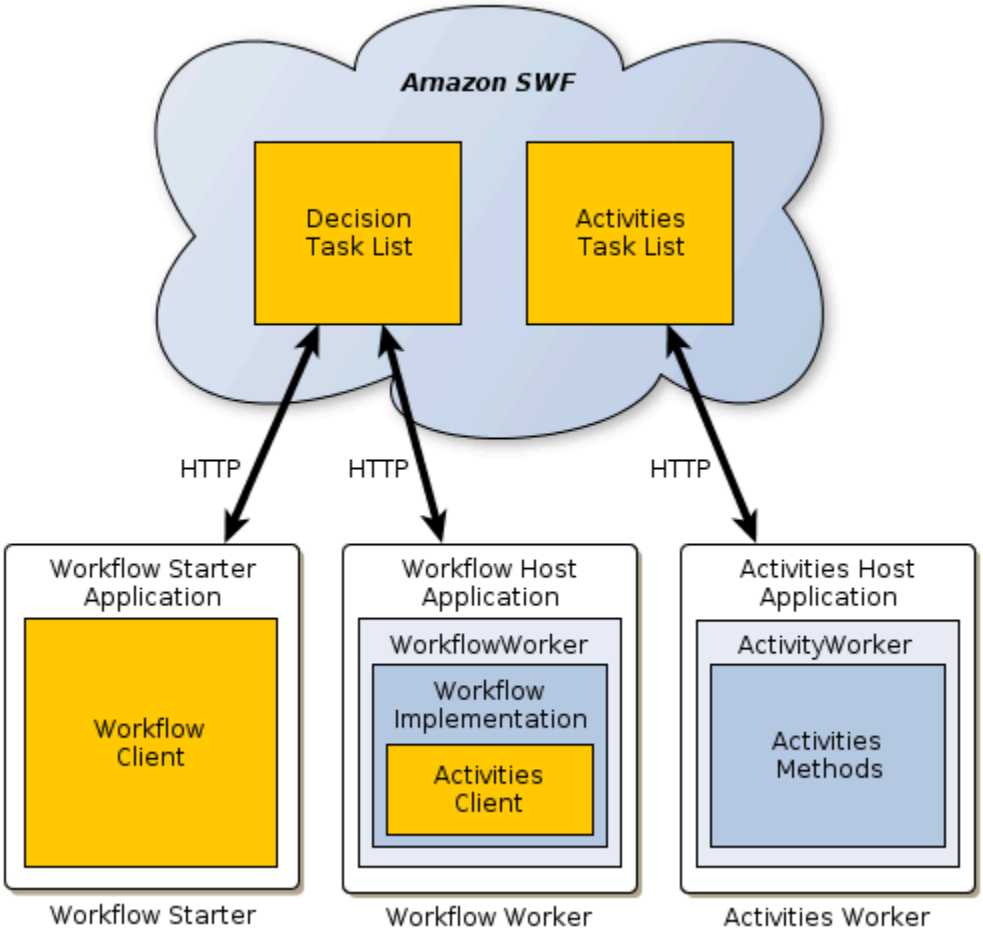
AWS Flow Framework for Java は Amazon SWF と連携して、スケーラブルで耐障害性のあるアプリケーションを簡単に作成し、長時間実行、リモート、またはその両方の非同期タスクを実行できます。「Hello World!」 の例では、 を使用して基本的なワークフローアプリケーション AWS Flow Framework を実装する方法の基本AWS Flow Framework for Java とはを紹介しました。このセクションでは、 AWS Flow Framework アプリケーションの仕組みに関する概念的な情報を提供します。最初のセクションでは AWS Flow Framework アプリケーションの基本構造を要約し、残りのセクションでは AWS Flow Framework アプリケーションの仕組みについて詳しく説明します。

トピック

AWS Flow Framework 基本的な概念: アプリケーション構造

概念的には、 AWS Flow Framework アプリケーションはワークフロースターター、ワークフローワーカー、アクティビティワーカーの 3 つの基本コンポーネントで構成されます。Amazon SWF を使用したワーカー (ワークフローとアクティビティ) の登録、ワーカーの開始、クリーンアップの処理を行うには、ホストアプリケーションを 1 つ以上使用します。ワーカーは、ワークフローを実行するメカニズムを処理します。これは、複数のホストに実装される場合があります。

この図は、基本的な AWS Flow Framework アプリケーションを示しています。



> **Note**
>
> これらのコンポーネントを 3 つのアプリケーションに別々に実装することは概念的には便利ですが、アプリケーションを作成して、さまざまな方法でこの機能を実装することができます。たとえば、アクティビティワーカーおよびワークフローワーカーに単一のホストアプリケーションを使用するか、アクティビティホストおよびワークフローホストを別々に使用します。さまざまなアクティビティセットを別々のホストで処理して、複数のアクティビティワーカーを使用することもできます。

3 つの AWS Flow Framework コンポーネントは、リクエストを管理する Amazon SWF に HTTP リクエストを送信することで間接的にやり取りします。Amazon SWF は、以下のことを行います。

- ディシジョンタスクを 1 つ以上管理します。これにより、次のステップがワークフローワーカーで実行されます。

- 1 つ以上のアクティビティタスクリストを管理します。これにより、アクティビティワーカーで実行されるタスクが判断されます。
- ワークフロー実行の詳細なステップバイステップ履歴を維持します。

を使用すると AWS Flow Framework、アプリケーションコードはAmazon SWF への HTTP リクエストの送信など、図に示す多くの詳細を直接処理する必要はありません。 AWS Flow Framework メソッドを呼び出すだけで、フレームワークはバックグラウンドの詳細を処理します。

アクティビティワーカーのロール

アクティビティワーカーは、ワークフローで行われるさまざまなタスクを実行します。アクティビティワーカーのタスク内容は以下のとおりです。

- アクティビティ実装。ワークフローで特定のタスクを実行する一連のアクティビティメソッドが含まれます。
- ActivityWorker オブジェクト。HTTP ロングポールリクエストを使用して、アクティビティタスクが実行されるように Amazon SWF をポーリングします。タスクが必要な場合、Amazon SWF は、タスクを実行する必要のある情報を送信して、リクエストに応答します。ActivityWorker オブジェクトは、適切なアクティビティメソッドを呼び出し、結果を Amazon SWF に返します。

ワークフローワーカーのロール

ワークフローワーカーは、さまざまなアクティビティの実行の調整、データフローの管理、失敗したアクティビティの処理を行います。アクティビティワーカーのタスク内容は以下のとおりです。

- ワークフローの実装。アクティビティ調整ロジックを含み、失敗したアクティビティなどを処理します。
- アクティビティクライアント。アクティビティワーカーのプロキシとして提供されており、ワークフローワーカーが、アクティビティが非同期的に実行されるようスケジュール設定します。
- WorkflowWorker オブジェクト、これは HTTP ロングポールリクエストを使用して、ディシジョンタスク向けに Amazon SWF をポーリングします。ワークフロータスクリストにタスクがある場合、Amazon SWF は、タスクの実行に必要な情報を返してリクエストに応答します。その後、フレームワークはワークフローを実行して、タスクを行い、その結果を Amazon SWF に返します。

ワークフロースターターのロール

ワークフロースターターがワークフローインスタンス (ワークフロー実行とも呼ばれる) を開始すると、実行時にインスタンスと通信し、ワークフローに追加データを渡したり、現在のワークフロー状態を取得したりできます。

また、ワークフロースターターは、ワークフロークライアントを使用してワークフローの実行を開始し、実行時に必要に応じてワークフローを操作して、クリーンアップを行います。ワークフロースターターは、ローカルで実行されるアプリケーション、ウェブアプリケーション、 AWS CLI 、または です AWS マネジメントコンソール。

Amazon SWF とアプリケーションの通信

Amazon SWF は、ワークフローコンポーネントにまたがる作業を調節し、詳細なワークフロー履歴を維持します。Amazon SWF は、コンポーネントとの通信を開始するのではなく、コンポーネントからの HTTP リクエストを待ち、必要に応じてリクエストを管理します。例:

- ワーカーからのリクエストの場合は、使用可能なタスクをポーリングすると、Amazon SWF はタスクが利用可能な場合にワーカーに直接応答します。ポーリングの仕組みの詳細については、「Amazon Simple Workflow Service デベロッパーガイド」の「Polling for Tasks」(タスクのポーリング) を参照してください。
- リクエストが、アクティビティワーカーからのタスクの完了通知である場合、Amazon SWF はその情報を実行履歴に記録して、ディシジョンタスクリストにタスクを追加し、タスクが完了したことをワークフローワーカーに通知することで、タスクを次のステップに進めます。
- リクエストが、アクティビティを実行するワークフローワーカーからである場合、Amazon SWF はその情報を実行履歴に記録し、アクティビティタスクリストにタスクを追加して、適切なアクティビティメソッドを実行するようアクティビティワーカーに指示します。

このアプローチにより、インターネット接続されているシステム (例: Amazon EC2 インスタンス、企業のデータセンター、クライアントコンピュータ) でワーカーを実行することができます。同じオペレーティングシステムを実行する必要はありません。HTTP リクエストはワーカーから開始されるため、外部から認識可能なポートは必要ありません。そのため、ワーカーはファイアウォールの内側で実行できます。

追加情報

Amazon SWF の仕組みの詳細については、「Amazon Simple Workflow Service Developer Guide」(Amazon Simple Workflow Service デベロッパーガイド) を参照してください。

AWS Flow Framework 基本的な概念: 信頼性の高い実行

非同期分散アプリケーションは、次のような従来のアプリケーションでは発生しない信頼性の問題に対処します。

- 非同期分散コンポーネント (例: リモートシステムでの長時間コンポーネント) 間で信頼性の高い通信を提供する方法。
- 特に長時間稼働アプリケーションなど、コンポーネントでエラーが発生するか、切断された場合に結果を紛失しない方法。
- エラーが発生した分散コンポーネントに対処する方法。

アプリケーションは、 AWS Flow Framework と Amazon SWF を使用してこれらの問題を管理することができます。長時間稼働しており、高い演算負荷や、ユーザーによる操作で非同期タスクが実行されている場合でも、信頼性が高く、予測可能な方法でワークフローを実行する Amazon SWF のメカニズムについて説明します。

信頼性の高い通信を保証する

AWS Flow Framework は、Amazon SWF を使用してタスクを分散アクティビティワーカーにディスパッチし、その結果をワークフローワーカーに返すことで、ワークフローワーカーとそのアクティビティワーカー間の信頼性の高い通信を提供します。Amazon SWF は、次の方法を使用して、ワーカーとそのアクティビティ間の信頼性の高い通信を確実にします。

- Amazon SWF は、スケジュールされたアクティビティとワークフロータスクを永続的に保存し、一度だけ実行されることを保証します。
- Amazon SWF では、アクティビティタスクを正常に実行して有効な結果を返すか、タスクが失敗したことをワークフローワーカーに通知することができます。
- また、Amazon SWF は、完了した各アクティビティの結果を永続的に保存するか、失敗したアクティビティでは、エラー情報を保存します。

AWS Flow Framework 次に、 は Amazon SWF からのアクティビティ結果を使用して、ワークフローの実行を続行する方法を決定します。

結果を保持する

ワークフロー履歴の維持

ペタバイトのデータのデータマイニングオペレーションを実行するアクティビティは、完了するのに数時間かかるのに対し、複雑なタスクを行うよう人間のワーカーに指示するアクティビティは数日間、場合によっては数週間かかる場合があります。

このようなシナリオに対応するために、 AWS Flow Framework ワークフローとアクティビティが完了するまでに任意の時間がかかる場合があります。ワークフローの実行には最大 1 年の制限があります。信頼性の高い方法で長時間稼働プロセスを実行するには、継続的なワークフローの実行履歴を永続的に保存するメカニズムが必要です。

は、各ワークフローインスタンスの実行履歴を保持する Amazon SWF に応じて、これ AWS Flow Framework を処理します。ワークフローの履歴には、すべてのワークフローやアクティビティタスクのスケジュールや完了など、ワークフローに関する信頼性の高い完全な進行状況の記録が含まれます。

AWS Flow Framework アプリケーションは通常、ワークフロー履歴を直接操作する必要はありませんが、必要に応じてアクセスできます。ほとんどの場合、アプリケーションは、フレームワークからバックグラウンドでワークフロー履歴にアクセスできます。ワークフロー履歴の詳細については、「Amazon Simple Workflow Service デベロッパーガイド」の「Workflow History」(ワークフロー履歴) を参照してください。

ステートレスな実行

実行履歴を使用することで、ワークフローワーカーをステートレスにできます。ワークフローワーカーまたはアクティビティワーカーの複数のインスタンスがある場合、どのワーカーも任意のタスクを実行できます。Amazon SWF からタスクを実行する必要のあるステート情報はすべてワーカーに送信されます。

このアプローチにより、ワークフローの信頼性は高まります。たとえば、アクティビティワーカーが失敗しても、ワークフローをやり直す必要はありません。ワーカーを再起動するだけで、障害が発生したタイミングに関係なく、タスクがリストに入るとそのタスクリストがポーリングされ、処理されます。2 つ以上のワークフロー、アクティビティワーカーを別々に使用して、ワークフロー全体の耐障害性を強化します。ワーカーのいずれかに失敗した場合、それ以外のワーカーは、ワークフローの進行を妨げることなく、スケジュールされたタスクに対応し続けます。

エラーが発生した分散コンポーネントに対処する

アクティビティは一時的な切断などの一時的な理由でエラーになることが多いため、失敗したアクティビティを処理する一般的な方法として、アクティビティを再試行します。再試行プロセスを処理せずに、複雑なメッセージを引き渡す戦略を実装することによって、アプリケーションで AWS Flow Frameworkを使用することができます。この機能では、失敗したアクティビティを再試行する複数のメカニズムや、ワークフローでタスクの分散的な非同期実行を使用する組み込みの例外処理を行うメカニズムを使用できます。

AWS Flow Framework 基本的な概念: 分散実行

ワークフローインスタンスは、本質的に、複数のリモートコンピュータ上で実行されるアクティビティやオーケストレーションロジックにまたがる仮想実行スレッドです。Amazon SWF と AWS Flow Framework 関数は、仮想 CPU 上のワークフローインスタンスを管理するオペレーティングシステムとして以下を実行します。

- 各インスタンスの実行状態を管理する。
- インスタンス間で切り替える。
- インスタンスの実行を切り替え時点で再開する。

ワークフローを再生する

アクティビティは長期的に実行される可能性があるため、完了するまでワークフローをブロックするのは望ましくありません。代わりに、 は再生メカニズムを使用してワークフロー実行 AWS Flow Framework を管理します。再生メカニズムは、Amazon SWF によって維持されるワークフロー履歴に依存して、エピソードでワークフローを実行します。

各エピソードは、各アクティビティを一度だけ実行するワークフローロジックを再生し、Promise オブジェクトが準備状態になるまで、アクティビティと非同期メソッドが実行されないようにします。

ワークフロースターターは、ワークフロー実行を開始する際、最初の再生エピソードを開始します。フレームワークは、ワークフローのエントリポイントメソッドを呼び出し、次のように動作します。

1. アクティビティ完了に依存しないすべてのワークフロータスク (例: すべてのアクティビティクライアントメソッドの呼び出し) を実行します。
2. 実行するためにスケジュールが設定されるアクティビティのリストを Amazon SWF に渡します。最初のエピソードでは、このリストは、Promise に依存しないアクティビティのみで構成されており、ただちに実行することができます。

3. エピソードが完了したことを Amazon SWF に通知します。

Amazon SWF は、ワークフロー履歴にアクティビティタスクを保存し、それらのタスクをアクティビティリストに配置して、実行用にスケジュールを設定します。アクティビティワーカーは、タスクリストをポーリングし、タスクを実行します。

タスクを完了すると、アクティビティワーカーはその結果を Amazon SWF に返します。これにより、結果がワークフロー実行履歴に記録された後、ワークフローワーカーの新しいワークフロータスクのスケジュールを設定するために、ワークフロータスクリストにその結果を配置します。ワークフローワーカーはタスクリストをポーリングし、タスクを受け取ると、次のように次の再生エピソードを実行します。

1. フレームワークは、ワークフローのエントリポイントメソッドを再度実行し、次の操作を行います。
   - アクティビティ完了に依存しないすべてのワークフロータスク (例: すべてのアクティビティクライアントメソッドの呼び出し) を実行します。ただし、フレームワークによって実行履歴がチェックされるため、アクティビティタスクが重複している場合はスケジュール設定されません。
   - 履歴から、完了したアクティビティタスクを確認し、そのようなアクティビティに依存する非同期ワークフローメソッドがある場合には実行します。
2. 実行できるワークフロータスクがすべて完了すると、フレームワークは、Amazon SWF にレポートを返します。
   - 最後のエピソード以降、入力 `Promise<T>` オブジェクトが準備状態になったアクティビティのリストが Amazon SWF に渡されるため、実行用にスケジュールを設定できます。
   - 追加のアクティビティタスクがエピソードで生成されていないが、未完了のアクティビティがある場合、フレームワークは、エピソードが完了したことを Amazon SWF に通知します。その後、もう 1 つのアクティビティが完了するまで待機し、次の再生エピソードを開始します。
   - エピソードで、追加のアクティビティタスクが生成されておらず、アクティビティがすべて完了している場合、フレームワークは、ワークフロー実行が完了したことを Amazon SWF に通知します。

再生の動作例については、「AWS Flow Framework for Java の再生動作」を参照してください。

再生と非同期ワークフローメソッド

メソッドでは、すべての入力 `Promise<T>` オブジェクトが準備状態になるまで実行が延期されるため、非同期ワークフローメソッドは、多くの場合、アクティビティとほとんど同じように使用されます。ただし、再生メカニズムでは、アクティビティとは異なる方法で非同期メソッドを処理します。

- 再生では、非同期メソッドが一度だけ実行されるとは限りません。入力 Promise オブジェクトの準備ができるまで非同期メソッドの実行は延期されますが、その後のエピソードのメソッド用に実行されます。
- 非同期メソッドが完了すると、新しいエピソードは開始しません。

非同期ワークフローの再生例については、「AWS Flow Framework for Java の再生動作」で確認できます。

再生とワークフロー実装

ほとんどの場合、再生メカニズムの詳細は気にする必要はありません。これらは基本的に裏側で起こる内容です。ただし、再生には、ワークフロー実装に 2 つの重要な意味があります。

- 再生では、タスクを複数回繰り返すため、ワークフローメソッドを使用して長期実行タスクを実行しないでください。非同期ワークフローメソッドの場合でも、通常は複数回実行されます。代わりに、長時間実行するタスクにはアクティビティを使用します。したがって、再生はアクティビティを 1 回だけ実行します。
- ワークフローロジックは、完全に決定論的でなければなりません。エピソードごとに同じ制御フローパスを使用する必要があります。たとえば、制御フローパスは、現在の時間に依存しないものにします。再生の詳細な説明および決定論の要件については、「非決定論」を参照してください。

AWS Flow Framework 基本的な概念: タスクリストとタスク実行

Amazon SWF は、ワークフローとアクティビティタスクを、名前付きリストに載せることで管理します。Amazon SWF は、少なくとも 2 つのタスクリストを保持します。ひとつはワークフローワーカー用、もうひとつはアクティビティワーカー用です。

> **Note**
>
> タスクリストは、必要なだけいくつでも指定し、リストごとに異なるワーカーを割り当てることができます。タスクリスト数に制限はありません。通常、ワーカーのタスクリストは、ワーカーオブジェクトの作成時にワーカーホストアプリケーションで指定します。

次の HelloWorldWorkflow ホストアプリケーションからの抜粋では、新しいアクティビティワーカーを作成し、それを HelloWorldList アクティビティタスクリストに割り当てます。

```
public class GreeterWorker  {
    public static void main(String[] args) throws Exception {
    ...
    String domain = " helloWorldExamples";
    String taskListToPoll = "HelloWorldList";

    ActivityWorker aw = new ActivityWorker(service, domain, taskListToPoll);
    aw.addActivitiesImplementation(new GreeterActivitiesImpl());
    aw.start();
    ...
  }
}
```

デフォルトでは、Amazon SWF はワーカーのタスクを HelloWorldList リストでスケジュールします。ワーカーは、このリストに対してタスクのポーリングを行います。タスクリストには任意の名前を指定できます。ワークフローリストとアクティビティリストの両方に同じ名前を使用することもできます。Amazon SWF 内部では、ワークフロー用とアクティビティ用のタスクリスト名が異なる名前空間に配置されるため、2 つのリストは区別されます。

タスクリストを指定しない場合、 はワーカーが Amazon SWF にタイプを登録するときにデフォルトリスト AWS Flow Framework を指定します。詳細については、「ワークフロータイプとアクティビティタイプの登録」を参照してください。

特定のワーカーやワーカーグループで実行するタスクを分担すると便利な場合があります。たとえば、イメージ処理ワークフローでイメージのダウンロードとイメージの処理を 2 つの異なるアクティビティで分担できます。両方のタスクを同じシステムで実行するほうが効率的です。大きなファイルのネットワーク転送に伴うオーバーヘッドを回避できます。

このようなシナリオをサポートするには、アクティビティメソッドを呼び出すときに、schedulingOptions パラメータを含むオーバーロードを使用してタスクリストを明示的に指

定できます。タスクリストを指定するには、適切に設定されたActivitySchedulingOptionsオブジェクトをメソッドに渡します。

例えば、HelloWorldWorkflow アプリケーションの say アクティビティをホストするアクティビティワーカーが getName や getGreeting とは異なるとします。次の例は、getName と getGreeting の割り当て先の元のタスクリストが異なる場合でも、この 2 つと同じタスクリストを say で使用する方法を示しています。

```
public class GreeterWorkflowImpl implements GreeterWorkflow {
  private GreeterActivitiesClient operations1 = new GreeterActivitiesClientImpl1(); //
getGreeting and getName
  private GreeterActivitiesClient operations2 = new GreeterActivitiesClientImpl2(); //
say
  @Override
  public void greet() {
    Promise<String> name = operations1.getName();
    Promise<String> greeting = operations1.getGreeting(name);
    runSay(greeting);
  }
  @Asynchronous
  private void runSay(Promise<String> greeting){
    String taskList = operations1.getSchedulingOptions().getTaskList();
    ActivitySchedulingOptions schedulingOptions = new ActivitySchedulingOptions();
    schedulingOptions.setTaskList(taskList);
    operations2.say(greeting, schedulingOptions);
  }
}
```

非同期の runSay メソッドは、getGreeting タスクリストをそのクライアントオブジェクトから取得します。次に、ActivitySchedulingOptions オブジェクトを作成し、say で getGreeting と同じタスクリストをポーリングするように設定します。

> **Note**
>
> schedulingOptions パラメータをアクティビティクライアントメソッドに渡すと、このアクティビティ実行に限り、元のタスクリストが上書きされます。タスクリストを指定しないで、このアクティビティクライアントメソッドを再び呼び出すと、Amazon SWF はタスクを元のリストに割り当て、このリストに対してアクティビティワーカーはポーリングを行います。

AWS Flow Framework 基本的な概念: スケーラブルなアプリケーション

Amazon SWF には、現在の負荷を処理できるように、ワークフローアプリケーションを簡単にスケーリングする 2 つの主な機能があります。

- 完全なワークフロー実行履歴。ステートレスなアプリケーションを実装できます。
- タスク実行に疎結合されているタスクスケジューリング。現在の需要を満たすように、アプリケーションを簡単にスケーリングできます。

Amazon SWF では、動的に割り当てられたタスクリストに投稿して、タスクをスケジュールします。ワークフローワーカーやアクティビティワーカーと直接通信することはありません。または、これらのワーカーは、HTTP リクエストを使用して、そのタスクのリストをポーリングします。このアプローチでは、タスクのスケジューリングはタスク実行に疎結合されるため、ワーカーは、Amazon EC2 インスタンス、企業のデータセンター、クライアントコンピュータなどの適切なシステムで実行されます。HTTP リクエストはワーカーから送信されるため、外部から見えるポートは必要ありません。これにより、ワーカーはファイアウォールの背後で実行することもできます。

タスクはロングポーリングの仕組みを使用してポーリングできるため、ワーカーが過負荷になることはありません。スケジュールされたタスクにスパイクがあっても、ワーカーはそれぞれのペースでタスクを引き出します。ただし、ワーカーはステートレスであるため、追加のワーカーインスタンスを起動することで、高い負荷を処理できるようにアプリケーションを動的にスケールできます。異なるシステムで実行されている場合でも、各インスタンスは同じタスクリストをポーリングし、最初に使用可能なワーカーインスタンスは、各タスクを実行します。ワーカーの場所や開始時間は関係ありません。負荷が低下した場合は、それに応じてワーカーの数も減らすことができます。

AWS Flow Framework 基本的な概念: アクティビティとワークフロー間のデータ交換

非同期アクティビティクライアントメソッドを呼び出すと、Promise (Future とも呼ばれる) オブジェクトがすぐに返ります。このオブジェクトはアクティビティメソッドの戻り値を表します。初期状態では、Promise は、準備未完了状態であり、戻り値は未定義です。アクティビティメソッドがタスクを完了して戻ると、フレームワークは、ネットワーク全体で戻り値をワークフローワーカーにマーシャリングします。これにより、値が Promise に割り当てられ、オブジェクトは準備状態になります。

アクティビティメソッドに戻り値がない場合でも、Promise を使用してワークフロー実行を管理できます。返された Promise をアクティビティクライアントメソッドまたは非同期ワークフローメソッドに渡すと、オブジェクトの準備が完了するまで実行は延期されます。

1 つ以上の Promise をアクティビティクライアントメソッドに渡すと、フレームワークは、タスクをキューに入れますが、すべてのオブジェクトの準備ができるまでスケジュールを延期します。次に、各 Promise からデータを抽出して、インターネット経由でアクティビティワーカーにマーシャリングし、アクティビティワーカーはそれを標準タイプとしてアクティビティメソッドに渡します。

> **Note**
>
> ワークフローワーカーとアクティビティワーカーの間で大量のデータを転送する必要がある場合は、データを都合のいい場所に保存し、検索情報を渡すことをお勧めします。例えば、Amazon S3 バケットにデータを保存して、その URL を渡すことができます。

Promise <T>Type

`Promise<T>` タイプは、いくつかの点で Java `Future<T>` タイプと似ています。どちらのタイプも、非同期メソッドより返る値を表し、最初の時点では定義されていません。オブジェクトの値にアクセスするには、その `get` メソッドを呼び出します。さらに、2 つのタイプの動作は全く異なります。

- `Future<T>` は同期構造のため、アプリケーションは非同期メソッドが完了するまで待機することができます。`get` を呼び出したが、オブジェクトの準備ができていない場合は、準備ができるまでブロックされます。
- `Promise<T>` を使用すると、フレームワークで同期が行われます。`get` を呼び出したときに、オブジェクトが準備未完了であれば、`get` から例外がスローされます。

`Promise<T>` の主な目的は、アクティビティ間のデータを管理することです。これにより、アクティビティは、入力データが有効になるまで実行されません。ワークフローワーカーが `Promise<T>` オブジェクトに直接アクセスすることはほとんどありません。アクティビティから次のアクティビティにオブジェクトを渡し、フレームワークとアクティビティワーカーに処理を指示します。ワークフローワーカーで、`Promise<T>` オブジェクトの値にアクセスするには、`get` メソッドを呼び出す前にオブジェクトが準備状態であることを確認します。

- Promise<T> オブジェクトを非同期ワークフローメソッドに渡して値を処理することをお勧めします。非同期メソッドでは、入力 Promise<T> オブジェクトの準備が完了するまで実行を延期します。これにより、安全に値にアクセスできるようになります。
- Promise<T> は、オブジェクトの準備が完了すると true を返す isReady メソッドを公開します。isReady を使用して Promise<T> オブジェクトをポーリングすることは推奨されていませんが、一部の状況では isReady が便利です。

AWS Flow Framework for Java には、 Settable<T>タイプも含まれています。これは から派生Promise<T>し、同様の動作を持ちます。違いは、フレームワークが通常Promise<T>オブジェクトの値を設定し、ワークフローワーカーが の値を設定することですSettable<T>。

ワークフローワーカーでは、Promise<T> オブジェクトを作成し、その値を設定する必要があります。たとえば、Promise<T> オブジェクトを返す非同期メソッドでは、戻り値を作成します。

- 入力された値を表すオブジェクトを作成するには、静的な Promise.asPromise メソッドを呼び出します。これにより、適切なタイプの Promise<T> オブジェクトを作成して値を設定し、準備状態になります。
- Promise<Void> オブジェクトを作成するには、静的な Promise.Void メソッドを呼び出します。

> **Note**
>
> Promise<T> は、有効なすべてのタイプを表すことができます。ただし、インターネット経由でデータをマーシャリングする必要がある場合、タイプとデータコンバータの間に互換性が必要です。詳細については、次のセクションを参照ください。

データコンバータとマーシャリング

は、データコンバーターを使用してインターネット経由でデータを AWS Flow Framework マーシャリングします。デフォルトでは、フレームワークは Jackson JSON プロセッサに基づき、データコンバータを使用します。ただし、このコンバータにはいくつかの制限があります。たとえば、文字列をキーとして使用しないマップをマーシャリングすることはできません。デフォルトのコンバータがアプリケーションにとって十分でない場合は、カスタムのデータコンバータを実装できます。詳細については、「DataConverters」を参照してください。

AWS Flow Framework 基本的な概念: アプリケーションとワークフロー実行間のデータ交換

ワークフローエントリポイントメソッドには、1 つ以上のパラメータを指定することができます。これにより、ワークフロースターターは初期データをワークフローに渡すことができます。また、実行時に追加データをワークフローに提供する上で有益です。たとえば、お客様が配送先を変更した場合、適切に変更されるように注文処理ワークフローを通知することができます。

Amazon SWF では、ワークフローでシグナルメソッドを実装できます。これにより、ワークフロースターターなどのアプリケーションは、いつでもワークフローにデータを渡すことができます。シグナルメソッドでは、使いやすい名前やパラメータを指定することができます。シグナルメソッドとして指定するには、ワークフローインターフェイスにそのメソッドを追加し、`@Signal` 注釈をメソッド宣言に適用します。

以下の例では、シグナルメソッド `changeOrder` を宣言する注文処理ワークフローインターフェイスを示します。これにより、ワークフロースターターは、ワークフロー開始後に元の注文を変更することができます。

```
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 300)
public interface WaitForSignalWorkflow {
    @Execute(version = "1.0")
    public void placeOrder(int amount);
    @Signal
    public void changeOrder(int amount);
}
```

フレームワークの注釈プロセッサでは、シグナルメソッドと同じ名前を使用して、ワークフロークライアントメソッドを作成し、ワークフロースターターはクライアントメソッドを呼び出して、データをワークフローに渡します。例については、「AWS Flow Framework レシピ」を参照してください。

Amazon SWF タイムアウトの種類

ワークフロー実行が正しく実行されるように、Amazon SWF でさまざまなタイプのタイムアウトを設定できます。一部のタイムアウトでは、ワークフローが完全に実行できる時間を指定します。その他のタイムアウトでは、ワーカーに割り当てる前にアクティビティタスクを実行できる時間と、スケジュールされてから完了までにかけることができる時間を指定します。Amazon SWF API のすべて

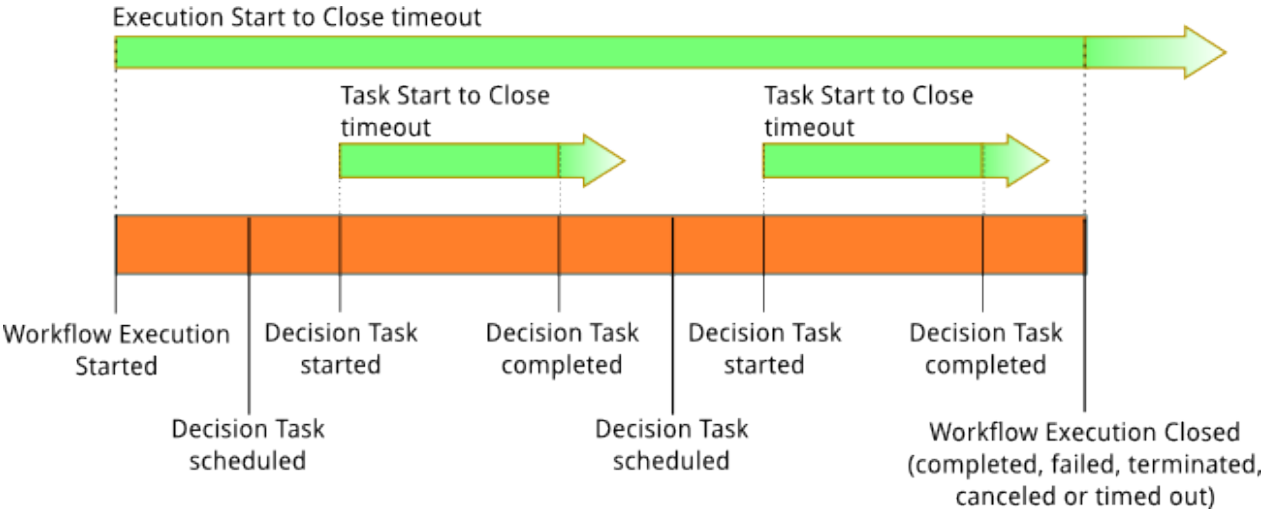
のタイムアウトは秒単位で指定されます。Amazon SWF は、タイムアウト値として文字列 NONE もサポートしています。これは、タイムアウトがないことを示します。

決定タスクとアクティビティタスクに関連するタイムアウトの場合、Amazon SWF はワークフロー実行履歴にイベントを追加します。イベントの属性により、発生したタイムアウトの種類と、影響を受けた決定タスクまたはアクティビティタスクに関する情報が提供されます。Amazon SWF は決定タスクもスケジュールします。ディサイダーは、新しい決定タスクを受け取ると、履歴でタイムアウトイベントを確認し、RespondDecisionTaskCompleted アクションを呼び出して適切なアクションを実行します。

タスクは、スケジュールされてからクローズされるまではオープン状態と見なされます。したがって、ワーカーの処理中はタスクはオープン状態と報告されます。タスクは、ワーカーによって完了済み、キャンセル済み、または失敗と報告されるとクローズされます。タスクは、タイムアウトの結果として Amazon SWF によってクローズされる場合もあります。

ワークフローと決定タスクのタイムアウト

次の図は、ワークフローと決定のタイムアウトがワークフローの有効期間にどのように関係するかを示しています。



ワークフローと決定タスクに関連して 2 つのタイムアウトの種類があります。

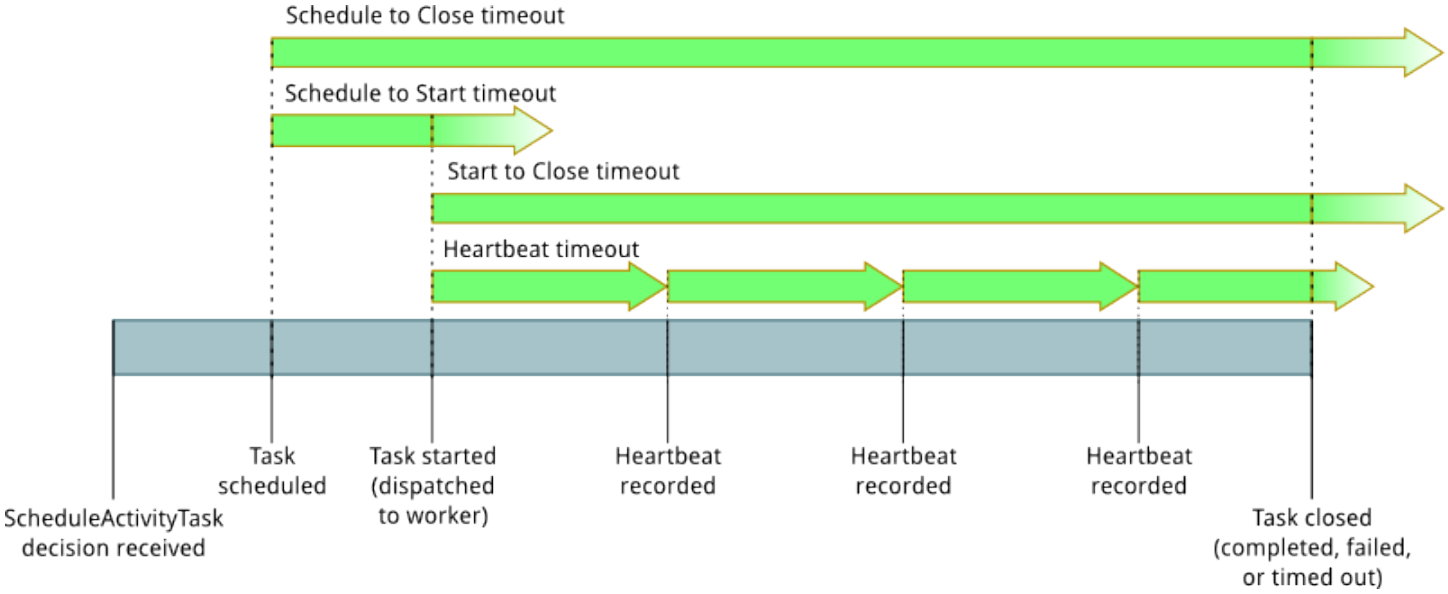
- ワークフローのスタートからクローズ (**timeoutType: START_TO_CLOSE**) - タイムアウトは、ワークフロー実行の完了までにかかる最大時間を指定します。これはワークフローの登録中にデフォルトとして設定されますが、ワークフローのスタート時に別の値でオーバーライドできます。このタイムアウトを超えると、Amazon SWF はワークフローの実行を終了し、WorkflowExecutionTimedOut タイプのイベントをワークフローの実行履歴に追加します。イベント属性は、timeoutType に加えて、このワークフロー実行に対して有効である

childPolicy を指定します。子ポリシーでは、親ワークフロー実行の回数がタイムアウトするか、それ以外に終了した場合に、子ワークフロー実行を処理する方法を指定します。たとえば、childPolicy が TERMINATE に設定された場合、子ワークフロー実行は終了します。ワークフロー実行がタイムアウトすると、表示の呼び出し以外にこれに対してアクションを実行することはできません。

- 決定タスクのスタートからクローズ (**timeoutType: START_TO_CLOSE**) - このタイムアウトにより、対応するディサイダーが決定タスクを完了するまでにかかる最大時間を指定します。これはワークフロータイプの登録中に設定されます。このタイムアウトを超えると、タスクはワークフロー実行履歴でタイムアウトとマークされ、Amazon SWF は DecisionTaskTimedOut 型のイベントをワークフロー履歴に追加します。イベントの属性には、この決定タスクがスケジュールされた日時 (scheduledEventId) およびスタートされた日時 (startedEventId) に対応するイベントの ID が含まれます。イベントの追加に加えて、Amazon SWF はこの決定タスクがタイムアウトしたことをディサイダーにアラートする新しい決定タスクをスケジュールします。このタイムアウトの発生後は、RespondDecisionTaskCompleted を使用してタイムアウトした決定タスクを完了する試みは失敗します。

アクティビティタスクのタイムアウト

次の図は、タイムアウトがアクティビティタスクの有効期間にどのように関係するかを示します。



アクティビティタスクに関連して 4 つのタイムアウトの種類があります。

- アクティビティタスクのスタートからクローズ (**timeoutType: START_TO_CLOSE**)
  – このタイムアウトは、ワーカーがタスクを受け取った後でアクティ

ビティワーカーがタスクを処理するためにかかる最大時間を指定します。RespondActivityTaskCanceled、RespondActivityTaskCompleted、および RespondActivityTaskFailed を使用してタイムアウトしたアクティビティタスクを閉じる試みは失敗します。

- アクティビティタスクのハートビート (**timeoutType: HEARTBEAT**) – このタイムアウトは、RecordActivityTaskHeartbeat アクションを通じて進捗状況を提供する前にタスクが実行できる最大時間を指定します。
- アクティビティタスクのスケジュールからスタート (**timeoutType: SCHEDULE_TO_START**) – このタイムアウトは、タスクを実行するワーカーを利用できない場合に、Amazon SWF がアクティビティタスクをタイムアウトするまでに待機する時間を指定します。タイムアウトした期限切れのタスクは、別のワーカーに割り当てられません。
- アクティビティタスクのスケジュールからクローズ (**timeoutType: SCHEDULE_TO_CLOSE**) – このタイムアウトは、タスクがスケジュールされてから完了するまでにかかる時間を指定します。ベストプラクティスとして、この値は、タスクのスケジュールからスタートまでのタイムアウトと、タスクのスタートからクローズまでのタイムアウトの合計よりも大きくしないでください。

> **Note**
>
> タイムアウトの種類ごとにデフォルト値があり、通常は NONE (無限) に設定されます。ただし、すべてのアクティビティの実行の最大時間は 1 年に制限されます。

アクティビティの種類を登録するときにこれらのデフォルト値を設定しますが、アクティビティタスクをスケジュールするときに新しい値でオーバーライドできます。これらのタイムアウトのいずれかが発生すると、Amazon SWF は ActivityTaskTimedOut 型の イベント をワークフロー履歴に追加します。このイベントの timeoutType 値属性では、これらのタイムアウトがいつ発生するかを指定します。それぞれのタイムアウトで、timeoutType の値は括弧内に示されます。イベントの属性には、アクティビティタスクがスケジュールされた日時 (scheduledEventId) およびスタートされた日時 (startedEventId) に対応するイベントの ID も含まれます。Amazon SWF は、イベントの追加に加えて、このタイムアウトが発生したことをディサイダーにアラートする新しい決定タスクをスケジュールします。

AWS Flow Framework for Java のタスクについて

トピック

タスク

AWS Flow Framework for Java が非同期コードの実行を管理するために使用する基盤となるプリミティブは Task クラスです。Task 型のオブジェクトは、非同期に実行する必要がある仕事を表します。非同期メソッドを呼び出すと、フレームワークでは、そのメソッドのコードを実行するための Task を作成し、それをリストに入れて後で実行できるようにします。同様に、Activity を呼び出すと、それに対する Task が作成されます。この後でメソッド呼び出しが戻り、通常、呼び出しの将来の結果として Promise<T> を返します。

Task クラスはパブリックであり、直接使用できます。たとえば、Hello World の例を書き換えて、非同期メソッドの代わりに Task を使用できます。

```
@Override
public void startHelloWorld(){
       final Promise<String> greeting = client.getName();
        new Task(greeting) {
        @Override
        protected void doExecute() throws Throwable {
         client.printGreeting("Hello " + greeting.get() +"!");
        }
    };
}
```

フレームワークでは、Task のコンストラクタに渡したすべての Promise が準備完了状態になると、doExecute() メソッドを呼び出します。Task クラスの詳細については、 AWS SDK for Java ドキュメントを参照してください。

また、フレームワークには Functor というクラスも含まれています。このクラスは、Promise<T> でもある Task を表します。Functor オブジェクトは、Task が完了すると、準備完了状態になります。次の例では、挨拶メッセージを取得するための Functor を作成しています。

```
Promise<String> greeting = new Functor<String>() {
    @Override
    protected Promise<String> doExecute() throws Throwable {
        return client.getGreeting();
    }
};
client.printGreeting(greeting);
```

実行順

タスクを実行できるのは、対応する非同期メソッドやアクティビティに渡したすべての Promise<T> で型指定されたパラメータが準備完了状態になった場合に限ります。実行準備が完了した Task は、準備完了キューに論理的に移動されます。つまり、実行が予定されます。ワーカークラスは、タスクを実行するために、ユーザーが非同期メソッドの本文を記述したコードを呼び出します。または、アクティビティメソッドの場合は、アクティビティタスクを Amazon Simple Workflow Service (AWS) でスケジュールします。

タスクが実行されて結果が生成されると、それに伴って他のタスクが準備完了状態となり、プログラムが続行されます。フレームワークでタスクを実行する方法は、非同期コードの実行順を理解するために重要です。コードは、プログラムに表示される順で実際に実行されるとは限りません。

```
Promise<String> name = getUserName();
printHelloName(name);
printHelloWorld();
System.out.println("Hello, Amazon!");

@Asynchronous
private Promise<String> getUserName(){
 return Promise.asPromise("Bob");
}
@Asynchronous
private void printHelloName(Promise<String> name){
 System.out.println("Hello, " + name.get() + "!");
}
@Asynchronous
private void printHelloWorld(){
```

```
 System.out.println("Hello, World!");
}
```

上のコードの出力は以下のとおりです。

```
Hello, Amazon!
Hello, World!
Hello, Bob
```

これは期待と異なる結果ですが、非同期メソッドのタスクがどのように実行されたかを考えると簡単に説明できます。

1. getUserName への呼び出しで Task が作成されます。これを Task1 とします。getUserName はパラメータを取らないため、 Task1はすぐに準備完了キューに入れられます。
2. 次に、printHelloName への呼び出しで作成される Task では、getUserName の結果を待つ必要があります。これを Task2 とします。必要な値はまだ準備されていないため、 Task2は待機リストに入れられます。
3. 次に、 printHelloWorld のタスクが作成されて、準備完了キューに追加されます。これを Task3 とします。
4. その後、println ステートメントで「Hello, Amazon!」が コンソールに出力されます。
5. この時点で、Task1 と Task3 は準備完了キューにあり、Task2 は待機リストにあります。
6. ワーカーによって Task1 が実行され、その結果によって Task2 が準備状態になります。Task2 は、Task3 の背景のキューに追加されます。
7. Task3 と Task2 が、この順序で実行されます。

アクティビティの実行も同じパターンに従います。アクティビティクライアントでメソッドを呼び出すと、Task が作成され、これが実行されると、Amazon SWF でアクティビティがスケジュールされます。

フレームワークは、コード生成や動的プロキシなどの機能に依存することで、メソッド呼び出しをアクティビティ呼び出しや非同期タスクに変換するロジックをプログラムに挿入します。

ワークフロー実行

ワークフロー実装の実行もワーカークラスによって管理されます。ワークフロークライアントでメソッドを呼び出すと、Amazon SWF が呼び出されてワークフローインスタンスが作成されます。Amazon SWF のタスクはフレームワークのタスクとは異なるので、混同しないでくださ

い。Amazon SWF のタスクは、アクティビティタスクまたは決定タスクです。アクティビティタスクの実行はシンプルです。アクティビティワーカークラスは、Amazon SWF からアクティビティタスクを受け取り、実装の適切なアクティビティメソッドを呼び出して、その結果を Amazon SWF に返します。

決定タスクの実行はもう少し複雑です。ワークフローワーカーは Amazon SWF から決定タスクを受け取ります。決定タスクは、実質的には、ワークフローロジックに次に何をするかを問い合わせるリクエストです。ワークフロークライアントを通じてワークフローインスタンスが開始されると、最初の決定タスクが生成されます。この決定タスクを受け取ると、フレームワークは `@Execute` 注釈が設定されたワークフローメソッドでコードの実行を開始します。このメソッドは、アクティビティをスケジュールする調整ロジックを実行します。ワークフローインスタンスの状態が変わると (例えばアクティビティが完了する)、さらに決定タスクがスケジュールされます。この時点で、ワークフローロジックはアクティビティの結果に基づいてアクションを実行することを決定できます。たとえば、別のアクティビティをスケジュールすることを決定できます。

フレームワークは、決定タスクをワークフローロジックにシームレスに変換することで、これらすべての詳細を開発者から隠します。開発者からは、コードが通常のプログラムのように見えます。表面下では、フレームワークは Amazon SWF に保持されている履歴を使用して、コードを Amazon SWF や決定タスクへの呼び出しにマッピングします。決定タスクが到着すると、フレームワークはプログラムの実行を再生し、その結果を現時点までの完了済みアクティビティの結果に追加します。これらの結果を待機していた非同期メソッドやアクティビティがブロック解除され、プログラムの実行が先に進みます。

イメージ処理ワークフロー例の実行および対応する履歴を次の表に示します。

サムネイルワークフローの実行

ワークフロープログラムの実行	Amazon SWF で管理される履歴
最初の実行	
1. ディスパッチループ 2. getImageUrls 3. downloadImage 4. createThumbnail (待機キューのタスク) 5. uploadImage (待機キューのタスク) 6. <ループの次のイテレーション>	1. ワークフローインスタンス開始、id= "1 " 2. downloadImage スケジュール

ワークフロープログラムの実行	Amazon SWF で管理される履歴
リプレイ	
1. ディスパッチループ 2. getImageUrls 3. downloadImage イメージ パス ="foo" 4. createThumbnail 5. uploadImage (待機キューのタスク) 6. <ループの次のイテレーション>	1. ワークフローインスタンス開始、id= "1 " 2. downloadImage スケジュール 3. downloadImage 完了、戻り値 ="foo" 4. createThumbnail スケジュール
リプレイ	
1. ディスパッチループ 2. getImageUrls 3. downloadImage イメージ パス ="foo" 4. createThumbnail サムネイル パス ="bar" 5. uploadImage 6. <ループの次のイテレーション>	1. ワークフローインスタンス開始、id= "1 " 2. downloadImage スケジュール 3. downloadImage 完了、戻り値 ="foo" 4. createThumbnail スケジュール 5. createThumbnail 完了、戻り値 ="bar" 6. uploadImage スケジュール
リプレイ	
1. ディスパッチループ 2. getImageUrls 3. downloadImage イメージ パス ="foo" 4. createThumbnail サムネイル パス ="bar" 5. uploadImage 6. <ループの次のイテレーション>	1. ワークフローインスタンス開始、id= "1 " 2. downloadImage スケジュール 3. downloadImage 完了、戻り値 ="foo" 4. createThumbnail スケジュール 5. createThumbnail 完了、戻り値 ="bar" 6. uploadImage スケジュール 7. uploadImage 完了 ...

`processImage` を呼び出すと、フレームワークは Amazon SWF に新しいワークフローインスタンスを作成します。これは、開始するワークフローインスタンスの永続的なレコードです。プロ

グラムは、downloadImage アクティビティへの呼び出しまで実行されます。この呼び出しでアクティビティをスケジュールすることを Amazon SWF にリクエストします。ワークフローはさらに実行され、後続のアクティビティのタスクを作成しますが、downloadImage アクティビティが完了するまで実行できません。したがって、このリプレイのエピソードは終了します。Amazon SWF は、実行のための downloadImage アクティビティのタスクをディスパッチし、タスクが完了すると、結果とともに履歴にレコードが作成されます。ワークフローは続行の準備が完了し、決定タスクが Amazon SWF で生成されます。フレームワークは決定タスクを受け取ってワークフローを再生し、その結果を履歴に記録されたダウンロード済みイメージの結果に追加します。これに伴って createThumbnail のタスクがブロック解除され、プログラムは Amazon SWF の createThumbnail アクティビティタスクがスケジュールされることで続行します。同じプロセスが uploadImage で繰り返されます。このようにプログラムの実行が継続され、最終的にワークフローですべてのイメージが処理され、保留中のタスクがなくなります。実行状態はローカルに保存されないため、各決定タスクは別のマシンで実行される可能性があります。これにより、フォールトトレラントでスケーラブルなプログラムを簡単に記述できます。

非決定論

フレームワークはリプレイに依存するため、オーケストレーションコード (アクティビティ実装を除くすべてのワークフローコード) が決定的であることが重要です。たとえば、プログラムの制御フローは乱数や現在の時間に依存すべきではありません。これらのモノは呼び出し間で変わるため、リプレイはオーケストレーションロジックの同じパスをたどらない場合があります。そのため、予期しない結果やエラーの原因となります。現在の時間を決定論的に取得するには、フレームワークが提供する WorkflowClock を使用できます。詳細については、「実行コンテキスト」セクションを参照してください。

> **Note**
>
> Spring のワークフロー実装オブジェクトのワイヤリングが不正確である場合にも非決定論につながることがあります。ワークフロー実装の Bean およびこれらが依存する Bean は、ワークフロースコープ (WorkflowScope) に存在する必要があります。たとえば、ワークフロー実装の Bean を、グローバルコンテキストで状態を保持する Bean にワイヤリングすると、予期しない動作が発生します。詳細については、「Spring との統合」セクションを参照してください。

AWS Flow Framework for Java プログラミングガイド

このセクションでは、 AWS Flow Framework for Java の機能を使用してワークフローアプリケーションを実装する方法について詳しく説明します。

トピック

を使用したワークフローアプリケーションの実装 AWS Flow Framework

を使用してワークフローを開発する一般的な手順 AWS Flow Framework は次のとおりです。

1. アクティビティとワークフローのコントラクトを定義します。アプリケーションの要件を分析し、必要なアクティビティとワークフロートポロジーを決定します。アクティビティでは必要な処理タスクを扱い、ワークフロートポロジーではワークフローの基本的な構造とビジネスロジックを定義します。

   たとえば、メディア処理アプリケーションでは、ファイルをダウンロードして処理し、処理済みのファイルを Amazon Simple Storage Service (S3) のバケットにアップロードする必要があります。これを以下の 4 つのアクティビティタスクに分割できます。

   1. ファイルをサーバーからダウンロードする
   2. ファイルを処理する (別のメディア形式に変換するなど)
   3. ファイルを S3 バケットにアップロードする
   4. ローカルファイルを削除してクリーンアップする

   このワークフローでは、エントリポイントメソッドを使用し、アクティビティを順に実行するシンプルなリニアトポロジーを実装します。HelloWorldWorkflow アプリケーション と非常に似ています。

2. アクティビティインターフェイスとワークフローインターフェイスを実装します。ワークフローコントラクトとアクティビティコントラクトを定義するには、Java インターフェイスを使用して、呼び出し規則を SWF で予測可能なものにし、ワークフローロジックとアクティビティタスクを柔軟に実装できるようにします。プログラムの各部分は、他の部分のデータのコンシューマーとして機能できますが、他のいずれの部分についても実装の詳細を知る必要はありません。

   たとえば、`FileProcessingWorkflow` を定義し、動画エンコーディング、圧縮、サムネイルなどのそれぞれに異なるワークフロー実装を提供できます。これらのワークフロー別に異なる制御フローを使用し、異なるアクティビティメソッドを呼び出すことができます。ワークフロースターターでは、これらの詳細を知る必要がありません。インターフェイスを使用すると、モック実装を使用してワークフローを簡単にテストすることもできます。モック実装は、後で実際のコードに置き換えることができます。

3. アクティビティクライアントとワークフロークライアントを生成します。を使用すると、非同期実行の管理、HTTP リクエストの送信、データのマーシャリングなどの詳細を実装する必要 AWS Flow Framework がなくなります。代わりに、ワークフロースターターでは、ワークフロークライアントのメソッドを呼び出してワークフローインスタンスを実行します。また、ワークフロー実装では、アクティビティクライアントのメソッドを呼び出してアクティビティを実行します。フレームワークでは、これらのやり取りの詳細をバックグラウンドで処理します。

Eclipse を使用していて、 のようにプロジェクトを設定している場合AWS Flow Framework for Java のセットアップ、 AWS Flow Framework 注釈プロセッサはインターフェイス定義を使用して、対応するインターフェイスと同じメソッドのセットを公開するワークフロークライアントとアクティビティクライアントを自動的に生成します。

4. アクティビティとワークフローのホストアプリケーションを実装します。ワークフローとアクティビティの実装は、タスクの Amazon SWF をポーリングし、データをマーシャリングして、適切な実装方法を呼び出すホストアプリケーションに埋め込む必要があります。Java AWS Flow Framework 用 には、ホストアプリケーションの実装を簡単かつ簡単にする WorkflowWorker クラスと ActivityWorker クラスが含まれています。
5. ワークフローをテストします。Java AWS Flow Framework 用 は、ワークフローをインラインおよびローカルでテストするために使用できる JUnit 統合を提供します。
6. ワーカーをデプロイします。ワーカーは必要に応じてデプロイできます。例えば、Amazon EC2 インスタンスにデプロイしたり、データセンターのコンピュータにデプロイしたりできます。ワーカーをデプロイして起動すると、ワーカーは Amazon SWF に対してタスクのポーリングをスタートし、必要に応じてタスクを処理します。
7. 実行をスタートします。アプリケーションは、ワークフローインスタンスをスタートするために、ワークフロークライアントを使用してワークフローのエントリポイントを呼び出します。ワークフローをスタートするには、Amazon SWF コンソールを使用することもできます。ワークフローインスタンスのスタート方法を問わず、Amazon SWF コンソールを使用して実行中のワークフローインスタンスを監視し、インスタンスの実行、完了、失敗に関するワークフロー履歴を確認できます。

AWS SDK for Java には、ルートディレクトリの readme.html ファイルの指示に従って参照および実行できる AWS Flow Framework for Java サンプルセットが含まれています。また、様々なプログラミング問題の対処方法を示すレシピ (シンプルなアプリケーション) のセットもあります。「AWS Flow Framework Recipes」より使用できます。

ワークフローコントラクトとアクティビティコントラクト

Java インターフェイスを使用してワークフローとアクティビティの署名を宣言します。このインターフェイスは、ワークフロー (またはアクティビティ) の実装と、そのワークフロー (またはアクティビティ) のクライアントとの間でコントラクトを構成します。たとえば、ワークフロータイプ `MyWorkflow` は、`@Workflow` 注釈を設定したインターフェイスを使用して定義します。

```
@Workflow
```

```
@WorkflowRegistrationOptions(
   defaultExecutionStartToCloseTimeoutSeconds = 60,
   defaultTaskStartToCloseTimeoutSeconds = 10)
public interface MyWorkflow
{
    @Execute(version = "1.0")
    void startMyWF(int a, String b);

    @Signal
    void signal1(int a, int b, String c);

    @GetState
    MyWorkflowState getState();
}
```

コントラクトには、実装固有の設定がありません。この実装に依存しないコントラクトを使用することで、クライアントを実装から分離し、クライアントを損なうことなく実装の詳細を柔軟に変更できます。逆に、クライアントの変更に伴って関連するワークフローやアクティビティを変更する必要がありません。たとえば、アクティビティ実装の変更を必要とせずに、Promise (Promise<T>) を使用してアクティビティを非同期的に呼び出すようにクライアントを変更できます。同様に、アクティビティの実装を変更して、アクティビティのクライアントの変更を必要とせずに、アクティビティ実装を (E メールの送信者などが) 非同期で完了するようにすることができます。

上の例では、ワークフローインターフェイス MyWorkflow に、新しい実行を開始するためのメソッドとして startMyWF が含まれています。このメソッドには @Execute 注釈が設定されており、戻り値の型として void または Promise<> を使用する必要があります。特定のワークフローインターフェイスで、この注釈を設定できるメソッドは最大 1 つです。このメソッドはワークフローロジックのエントリポイントであり、フレームワークでは、決定タスクを受け取ったときに、このメソッドを呼び出してワークフローロジックを実行します。

ワークフローインターフェイスは、ワークフローに送信するシグナルも定義します。シグナルメソッドは、一致する名前のシグナルがワークフロー実装で受信されたときに呼び出されます。例えば、MyWorkflow インターフェイスは、@Signal 注釈が設定されたシグナルメソッド (signal1) を宣言します。

@Signal注釈は、シグナルメソッドに必須です。シグナルメソッドの戻り値の型は、void であることが必要です。ワークフローインターフェイスには、ゼロ個以上のシグナルメソッドが定義されている場合があります。@Execute メソッドなしのワークフローインターフェイスと、いくつかの @Signal メソッドを宣言し、実行は開始できないが進行中の実行にシグナルを送信できるクライアントを生成できます。

@Execute 注釈と @Signal 注釈を設定したメソッドでは、Promise<T> やその派生型を除くあらゆる型のパラメータをいくつでも使用できます。これにより、ワークフロー実装の開始時と実行中に、厳密に型指定された入力を渡すことができます。@Execute メソッドの戻り値の型は、void または Promise<> であることが必要です。

さらに、ワークフロー実行の最新状態を報告するためのメソッド (前の例の getState メソッドなど) をワークフローインターフェイスで宣言することもできます。この状態は、ワークフローのアプリケーション状態を全面的に反映するものではありません。この機能の用途は、最大 32 KB のデータを保存して、実行の最新ステータスを示すことにあります。たとえば、注文処理ワークフローでは、受注、注文処理、またはキャンセルを示す文字列を保存できます。このメソッドは、決定タスクが完了するたびに、フレームワークから呼び出されて最新状態を取得します。状態は Amazon Simple Workflow Service (Amazon SWF) に保存され、生成された外部クライアントを使用して取得できます。これにより、ワークフロー実行の最新状態をチェックできます。@GetState 注釈を設定したメソッドは、引数を取ることはできず、戻り値の型として void を使用することはできません。このメソッドからは、必要に応じて任意の型を返すことができます。上の例では、MyWorkflowState (下の定義を参照) のオブジェクトがメソッドから返され、文字列状態と進捗状況 (%) の保存に使用されます。このメソッドは、ワークフロー実装オブジェクトの読み取り専用アクセスを実行するものであり、同期的に呼び出されます。これにより、@Asynchronous 注釈を設定したメソッドを呼び出すなどの非同期操作は禁止されます。@GetState 注釈を設定できるメソッドは、ワークフローインターフェイスで最大 1 つです。

```
public class MyWorkflowState {
   public String status;
   public int percentComplete;
}
```

同様に、アクティビティのセットを定義するには、@Activities 注釈を設定したインターフェイスを使用します。インターフェイスの各メソッドは、アクティビティに対応しています。以下に例を挙げます。

```
@Activities(version = "1.0")
@ActivityRegistrationOptions(
     defaultTaskScheduleToStartTimeoutSeconds = 300,
     defaultTaskStartToCloseTimeoutSeconds = 3600)
public interface MyActivities {
    // Overrides values from annotation found on the interface
    @ActivityRegistrationOptions(description = "This is a sample activity",
         defaultTaskScheduleToStartTimeoutSeconds = 100,
         defaultTaskStartToCloseTimeoutSeconds = 60)
```

```
    int activity1();

    void activity2(int a);
}
```

このインターフェイスでは、複数の関連するアクティビティをグループ化できます。アクティビティインターフェイス内には、アクティビティをいくつでも定義できます。また、アクティビティインターフェイスは、必要に応じていくつでも定義できます。@Execute メソッドや @Signal メソッドと同様に、アクティビティメソッドでは Promise<T> やその派生型を除くあらゆる型の引数をいくつでも使用できます。アクティビティの戻り値の型として、Promise<T> やその派生型は使用できません。

ワークフロータイプとアクティビティタイプの登録

Amazon SWF では、使用する前に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する必要があります。ワークフローおよびアクティビティは、ワーカーに追加する実装で、フレームワークによって自動的に登録されます。このフレームワークは、ワークフローおよびアクティビティを実装するタイプを検索し、Amazon SWF でそれらを登録します。デフォルトでは、フレームワークはインターフェイス定義を使用して、ワークフロータイプおよびアクティビティタイプの登録オプションを推測します。@WorkflowRegistrationOptions 注釈または @SkipRegistration 注釈を使用するには、ワークフローインターフェイスが必要です。ワークフローワーカーは、@WorkflowRegistrationOptions 注釈を使用して設定されているすべてのワークフロータイプを登録します。同様に、@ActivityRegistrationOptions、または @SkipRegistration を使用して各メソッドに注釈を入れるか、これらのいずれかの注釈が @Activities インターフェイスに存在する必要があります。アクティビティワーカーは、@ActivityRegistrationOptions 注釈を適用して設定されているアクティビティタイプをすべて登録します。ワーカーのいずれかを開始すると、自動的に登録されます。@SkipRegistration 注釈を持つワークフローおよびアクティビティタイプは登録されていません。@ActivityRegistrationOptions および @SkipRegistration 注釈にはオーバーライドセマンティクスが含まれており、最も具体的なものはアクティビティタイプに適用されます。

Amazon SWF では、登録後にタイプを再登録または変更することはできません。フレームワークは、すべてのタイプを登録しようとしますが、そのタイプが既に登録されている場合は、再登録されず、エラーも報告されません。

登録済みの設定を変更する必要がある場合は、そのタイプの新しいバージョンを登録する必要があります。新しい実行を開始する際、または生成されたクライアントを使用するアクティビティを呼び出す際、登録された設定を上書きすることもできます。

登録するには、タイプ名といくつかの登録オプションが必要です。デフォルトの実装では、これらの項目を次のように判断します。

ワークフロータイプ名とバージョン

フレームワークは、ワークフローインターフェイスからワークフロータイプ名を決定します。デフォルトのワークフロータイプ名の形式は、{*prefix*}{*name*} です。{*prefix* は、@Workflow インターフェイスの名前に「.」がついたもの、{*name*} は、@Execute メソッドの名前に設定されます。前述の例のデフォルトのワークフロータイプ名は MyWorkflow.startMyWF です。デフォルト名は、@Execute メソッドの名前パラメータを使用して上書きできます。次の例のデフォルトのワークフロータイプ名は startMyWF です。名前の文字列を空欄にすることはできません。@Execute を使用して名前を上書きすると、フレームワークでプレフィックスは自動的に付加されません。独自の命名スキームを使用することができます。

ワークフローバージョンは、@Execute 注釈の version パラメータを使用して指定されます。デフォルトの version はないため、明示的に指定する必要があります。version は自由形式の文字列で、独自のバージョニングスキームを使用することができます。

通知名

シグナルの名前は、@Signal 注釈の名前パラメータを使用して指定されます。指定しない場合は、シグナルメソッドの名前にデフォルトで設定されます。

アクティビティタイプ名とバージョン

フレームワークは、アクティビティインターフェイスからアクティビティタイプの名前を決定します。デフォルトのアクティビティタイプ名の形式は、{*prefix*}{*name*} です。{*prefix*} は、@Activities インターフェイスの名前に「.」がついたもの、{*name*} は、メソッド名に設定されます。デフォルトの {{*prefix*} は、アクティビティインターフェイスの @Activities 注釈で上書きできます。アクティビティメソッドで @Activity 注釈を使用して、アクティビティタイプ名を指定することもできます。@Activity を使用して名前を上書きすると、フレームワークでプレフィックスが付加されることはありません。独自の命名スキームを使用することができます。

アクティビティバージョンは、@Activities 注釈のバージョンパラメータを使用して指定されます。このバージョンは、インターフェイスで定義されているすべてのアクティビティとして使用されているため、@Activity 注釈を使用してアクティビティ単位で上書きすることができます。

デフォルトのタスクリスト

デフォルトのタスクリストは、`@WorkflowRegistrationOptions` および `@ActivityRegistrationOptions` 注釈と、`defaultTaskList` パラメータを使用して設定できます。デフォルトでは、`USE_WORKER_TASK_LIST` に設定されます。これは、アクティビティタイプまたはワークフロータイプを登録するために使用されるワーカーオブジェクトで構成されたタスクリストを使用するようにフレームワークに指示する特殊な値です。これらの注釈を使用して、デフォルトタスクリストを `NO_DEFAULT_TASK_LIST` に設定し、デフォルトタスクリストを登録しないように選択することもできます。この設定は、実行時にタスクを指定する必要がある場合に使用できます。デフォルトタスクが登録されていない場合は、ワークフローの開始時にタスクリストを指定するか、生成されたクライアントの各メソッドオーバーロードで `StartWorkflowOptions` および `ActivitySchedulingOptions` パラメータを使用して、アクティビティメソッドを呼び出す必要があります。

他の登録オプション

Amazon SWF API で許可されているすべてのワークフロータイプおよびアクティビティタイプの登録オプションは、フレームワークを通じて指定できます。

ワークフローの登録オプションの詳細な一覧については、以下を参照してください。

- @Workflow
- @Execute
- @WorkflowRegistrationOptions
- @Signal

アクティビティの登録オプションの詳細な一覧については、以下を参照してください。

- @Activity
- @Activities
- @ActivityRegistrationOptions

タイプの登録を完全に制御するには、「ワーカーの拡張機能」を参照してください。

アクティビティクライアントとワークフロークライアント

ワークフロークライアントとアクティビティクライアントは、`@Workflow` インターフフェイスと `@Activities` インターフェイスに基づいて、フレームワークで生成されます。クライアントインターフェイスは、クライアント別のメソッドと設定を反映して個別に生成されます。Eclipse を使用して開発している場合は、該当するインターフェイスを含むファイルを保存するたびに Amazon SWF Eclipse プラグインで生成されます。生成されたコードは、インターフェイスと同じパッケージのプロジェクトの生成されたソースディレクトリに配置されます。

> **Note**
>
> Eclipse で使用されるデフォルトのディレクトリ名は .apt_generated です。Eclipse' で始まる名前のディレクトリは表示されません。Package Explorer を参照してください。生成されたファイルをプロジェクトエクスプローラで表示するには、別のディレクトリ名を使用してください。Eclipse のパッケージエクスプローラで、パッケージを右クリックし、[プロパティ]、[Java Compiler] (Java コンパイラ)、[Annotation processing] (注釈処理) の順に選択して、[Generate source directory] (ソースディレクトリの生成) の設定を変更します。

ワークフロークライアント

ワークフローの生成されたアーティファクトには、3 つのクライアント側インターフェイスとこれらを実装するクラスが含まれています。生成されたクライアントは以下のとおりです。

- 非同期クライアント。ワークフロー実装内からワークフロー実行の開始とシグナルの送信を行うための非同期メソッドを提供します。
- 外部クライアント。ワークフロー実装のスコープ外から、実行の開始、シグナルの送信、ワークフロー状態の取得を行うことができます。
- セルフクライアント。継続的なワークフローを作成できます。

たとえば、サンプルの `MyWorkflow` インターフェイスで生成されたクライアントインターフェイスは以下のとおりです。

```
//Client for use from within a workflow
public interface MyWorkflowClient extends WorkflowClient
{
    Promise<Void> startMyWF(
       int a, String b);
```

```
    Promise<Void> startMyWF(
       int a, String b,
       Promise<?>... waitFor);

    Promise<Void> startMyWF(
       int a, String b,
       StartWorkflowOptions optionsOverride,
       Promise<?>... waitFor);

    Promise<Void> startMyWF(
       Promise<Integer> a,
       Promise<String> b);

    Promise<Void> startMyWF(
       Promise<Integer> a,
       Promise<String> b,
       Promise<?>... waitFor);

    Promise<Void> startMyWF(
       Promise<Integer> a,
       Promise<String> b,
       StartWorkflowOptions optionsOverride,
       Promise<?>... waitFor);

    void signal1(
      int a, int b, String c);
}

//External client for use outside workflows
public interface MyWorkflowClientExternal extends WorkflowClientExternal
{
    void startMyWF(
       int a, String b);

    void startMyWF(
       int a, String b,
       StartWorkflowOptions optionsOverride);

    void signal1(
       int a, int b, String c);

    MyWorkflowState getState();
}
```

```
//self client for creating continuous workflows
public interface MyWorkflowSelfClient extends WorkflowSelfClient
{
    void startMyWF(
       int a, String b);

    void startMyWF(
       int a, String b,
       Promise<?>... waitFor);

    void startMyWF(
       int a, String b,
       StartWorkflowOptions optionsOverride,
       Promise<?>... waitFor);

    void startMyWF(
       Promise<Integer> a,
       Promise<String> b);

    void startMyWF(
       Promise<Integer> a,
       Promise<String> b,
       Promise<?>... waitFor);

    void startMyWF(
       Promise<Integer> a,
       Promise<String> b,
       StartWorkflowOptions optionsOverride,
       Promise<?>... waitFor);
```

これらのインターフェイスには、宣言した @Workflow インターフェイスの各メソッドに対応するオーバーロードメソッドがあります。

外部クライアントは、@Workflow インターフェイスのメソッドをミラーリングし、StartWorkflowOptions を取る @Execute メソッドのオーバーロードを 1 つ追加しています。新しいワークフロー実行を開始する場合、このオーバーロードを使用して追加のオプションを渡すことができます。これらのオプションにより、デフォルトのタスクリストやタイムアウト設定を上書きし、ワークフロー実行にタグを関連付けることができます。

一方、非同期クライアントには、@Execute メソッドの非同期呼び出しを許可するメソッドがあります。次のメソッドオーバーロードは、ワークフローインターフェイスの @Execute メソッドに対して、クライアントインターフェイスで生成されます。

1. 元の引数をそのまま取るオーバーロード。このオーバーロードの戻り値の型は、元のメソッドから void が返された場合は Promise<Void> になります。それ以外の場合は、元のメソッドに宣言されている Promise<> になります。例:

   元のメソッド:

   ```
   void startMyWF(int a, String b);
   ```

   生成されたメソッド:

   ```
   Promise<Void> startMyWF(int a, String b);
   ```

   ワークフローのすべての引数が使用可能で、待機する必要がない場合は、このオーバーロードを使用します。

2. 元の引数 (そのまま) と追加の Promise<?> 型の変数引数を取るオーバーロード。このオーバーロードの戻り値の型は、元のメソッドから void が返された場合は Promise<Void> になります。それ以外の場合は、元のメソッドに宣言されている Promise<> になります。例:

   元のメソッド:

   ```
   void startMyWF(int a, String b);
   ```

   生成されたメソッド:

   ```
   Promise<void> startMyWF(int a, String b, Promise<?>...waitFor);
   ```

   ワークフローのすべての引数が使用可能で、待機する必要がないが、他のいくつかの Promise の準備が完了するのを待機する場合は、このオーバーロードを使用します。この変数引数を使用して、引数として宣言されていないが、呼び出しを実行するまで待機する Promise<?> オブジェクトを渡すことができます。

3. 元の引数 (そのまま)、追加の StartWorkflowOptions 型の引数、および追加の Promise<?> 型の変数引数を取るオーバーロード。このオーバーロードの戻り値の型は、元のメソッドから

void が返された場合は Promise<Void> になります。それ以外の場合は、元のメソッドに宣言されている Promise<> になります。例:

元のメソッド:

```
void startMyWF(int a, String b);
```

生成されたメソッド:

```
Promise<void> startMyWF(
   int a,
   String b,
   StartWorkflowOptions optionOverrides,
   Promise<?>...waitFor);
```

このオーバーロードは、ワークフローのすべての引数が使用可能で待機する必要がない場合、ワークフロー実行のスタート用のデフォルト設定を上書きする場合、または他の Promise の準備が完了するまで待機する場合に使用します。この変数引数を使用して、引数として宣言されていないが、呼び出しを実行するまで待機する Promise<?> オブジェクトを渡すことができます。

4. 元のメソッドの各引数が Promise<> ラッパーに置き換えられたオーバーロード。このオーバーロードの戻り値の型は、元のメソッドから void が返された場合は Promise<Void> になります。それ以外の場合は、元のメソッドに宣言されている Promise<> になります。例:

   元のメソッド:

   ```
   void startMyWF(int a, String b);
   ```

   生成されたメソッド:

   ```
   Promise<Void> startMyWF(
      Promise<Integer> a,
      Promise<String> b);
   ```

   このオーバーロードは、ワークフロー実行に渡す引数を非同期的に評価する場合に使用します。このメソッドオーバーロードに対する呼び出しは、オーバーロードに渡したすべての引数の準備が完了するまで、実行されません。

一部の引数が準備完了済みである場合は、これらを Promise.asPromise(*value*) メソッドを通じて準備完了済み状態の Promise に変換します。例:

```
Promise<Integer> a = getA();
String b = getB();
startMyWF(a, Promise.asPromise(b));
```

5. 元のメソッドの各引数が Promise<> ラッパーに置き換えられたオーバーロード。このオーバーロードには、追加の Promise<?> 型の変数引数もあります。このオーバーロードの戻り値の型は、元のメソッドから void が返された場合は Promise<Void> になります。それ以外の場合は、元のメソッドに宣言されている Promise<> になります。例:

   元のメソッド:

   ```
   void startMyWF(int a, String b);
   ```

   生成されたメソッド:

   ```
   Promise<Void> startMyWF(
      Promise<Integer> a,
      Promise<String> b,
      Promise<?>...waitFor);
   ```

   ワークフロー実行に渡す引数を非同期的に評価する必要があり、他のいくつかの Promise の準備が完了するのを待機する場合は、このオーバーロードを使用します。このメソッドオーバーロードに対する呼び出しは、オーバーロードに渡したすべての引数の準備が完了するまで、実行されません。

6. 元のメソッドの各引数が Promise<?> ラッパーに置き換えられたオーバーロード。このオーバーロードには、追加の StartWorkflowOptions 型の引数と Promise<?> 型の変数引数があります。このオーバーロードの戻り値の型は、元のメソッドから void が返された場合は Promise<Void> になります。それ以外の場合は、元のメソッドに宣言されている Promise<> になります。例:

   元のメソッド:

   ```
   void startMyWF(int a, String b);
   ```

   生成されたメソッド:

```
Promise<Void> startMyWF(
   Promise<Integer> a,
   Promise<String> b,
   StartWorkflowOptions optionOverrides,
   Promise<?>...waitFor);
```

ワークフロー実行に渡す引数を非同期的に評価し、ワークフロー実行をスタートするために使用するデフォルト設定を上書きする場合は、このオーバーロードを使用します。このメソッドオーバーロードに対する呼び出しは、オーバーロードに渡したすべての引数の準備が完了するまで、実行されません。

また、ワークフローインターフェイスの各シグナルに対応するメソッドも生成されます。以下に例を挙げます。

元のメソッド:

```
void signal1(int a, int b, String c);
```

生成されたメソッド:

```
void signal1(int a, int b, String c);
```

この非同期クライアントには、元のインターフェイスの @GetState 注釈が設定されたメソッドに対応するメソッドがありません。状態の取得にはウェブサービス呼び出しが必要なため、ワークフロー内での使用は適していません。したがって、これは外部クライアントを通じてのみ提供されます。

セルフクライアントは、現在の実行が完了したときに、ワークフロー内から新しい実行を開始するために使用します。このクライアントのメソッドは、非同期クライアントのものと似ていますが、void を返します。このクライアントには、@Signal 注釈と @GetState 注釈が設定されたメソッドに対応するメソッドがありません。詳細については、「継続的なワークフロー」を参照してください。

生成されたクライアントは、基本インターフェイスの WorkflowClient と WorkflowClientExternal からそれぞれ派生されます。これらのインターフェイスが提供するメソッドを使用して、ワークフロー実行をキャンセルまたは終了できます。これらのインターフェイスの詳細については、 AWS SDK for Java のドキュメントを参照してください。

生成されたクライアントでは、厳密に型指定された形式でワークフロー実行とやり取りできます。生成されたクライアントのインスタンスは、作成後、特定のワークフロー実行に関連付けられ、その実行でのみ使用できます。また、フレームワークは、ワークフローのタイプや実行に固有でない動的クライアントも提供します。生成されたクライアントは、この動的クライアントに表面下で依存します。これらのクライアントを直接使用することもできます。「動的クライアント」のセクションを参照してください。

フレームワークでは、厳密に型指定されたクライアントを作成するためのファクトリも生成します。MyWorkflow インターフェイスの例で生成されたクライアントファクトリは以下のとおりです。

```
//Factory for clients to be used from within a workflow
public interface MyWorkflowClientFactory
   extends WorkflowClientFactory<MyWorkflowClient>
{
}

//Factory for clients to be used outside the scope of a workflow
public interface MyWorkflowClientExternalFactory
{
   GenericWorkflowClientExternal getGenericClient();
   void setGenericClient(GenericWorkflowClientExternal genericClient);
   DataConverter getDataConverter();
   void setDataConverter(DataConverter dataConverter);
   StartWorkflowOptions getStartWorkflowOptions();
   void setStartWorkflowOptions(StartWorkflowOptions startWorkflowOptions);
   MyWorkflowClientExternal getClient();
   MyWorkflowClientExternal getClient(String workflowId);
   MyWorkflowClientExternal getClient(WorkflowExecution workflowExecution);
   MyWorkflowClientExternal getClient(
      WorkflowExecution workflowExecution,
      GenericWorkflowClientExternal genericClient,
      DataConverter dataConverter,
      StartWorkflowOptions options);
}
```

WorkflowClientFactory基本インターフェイスは次のとおりです。

```
public interface WorkflowClientFactory<T> {
    GenericWorkflowClient getGenericClient();
    void setGenericClient(GenericWorkflowClient genericClient);
    DataConverter getDataConverter();
```

```
    void setDataConverter(DataConverter dataConverter);
    StartWorkflowOptions getStartWorkflowOptions();
    void setStartWorkflowOptions(StartWorkflowOptions startWorkflowOptions);
    T getClient();
    T getClient(String workflowId);
    T getClient(WorkflowExecution execution);
    T getClient(WorkflowExecution execution,
                StartWorkflowOptions options);
    T getClient(WorkflowExecution execution,
                StartWorkflowOptions options,
                DataConverter dataConverter);
}
```

クライアントのインスタンスを作成するには、これらのファクトリを使用する必要があります。ファクトリでは、一般的なクライアント (カスタムクライアント実装を提供するために使用)、DataConverter (データをマーシャリングするためにクライアントで使用)、およびオプション (ワークフロー実行を開始するために使用) を設定できます。詳細については、「DataConverters」セクションと「子ワークフロー実行」セクションを参照してください。StartWorkflowOptions には、登録時に指定されたデフォルト (タイムアウトなど) をオーバーライドするために使用できる設定が含まれています。StartWorkflowOptions クラスの詳細については、 AWS SDK for Java ドキュメントを参照してください。

非同期クライアントではワークフロー内のコードからワークフロー実行を開始できますが、外部クライアントではワークフローのスコープ外からワークフロー実行を開始できます。実行を開始するには、生成されたクライアントを使用して、ワークフローインターフェイスの @Execute 注釈が設定されたメソッドに対応するメソッドを呼び出すだけです。

フレームワークでは、クライアントインターフェイスの実装クラスも生成します。これらのクライアントでは、該当するアクションを実行するためのリクエストを作成して Amazon SWF に送信します。クライアントバージョンの @Execute メソッドでは、新しいワークフロー実行をスタートするか、Amazon SWF API を使用して子ワークフロー実行を作成します。同様に、クライアントバージョンの @Signal メソッドでは、Amazon SWF API を使用してシグナルを送信します。

> **Note**
>
> 外部ワークフロークライアントは、Amazon SWF クライアントとドメインを使用して設定する必要があります。これらをパラメータとして取るクライアントファクトリコンストラクタを使用するか、Amazon SWF クライアントとドメインが設定済みの一般的なクライアント実装を渡すことができます。

フレームワークでは、ワークフローインターフェイスの型階層を確認し、さらに親ワークフローインターフェイスのクライアントインターフェイスを生成して、これらから子を派生させます。

アクティビティクライアント

ワークフロークライアントと同様に、@Activities 注釈が設定されたインターフェイスごとにクライアントを生成します。生成されたアーティファクトには、クライアント側インターフェイスとクライアントクラスが含まれています。上の @Activities インターフェイス例 (MyActivities) で生成されたインターフェイスは以下のとおりです。

```
public interface MyActivitiesClient extends ActivitiesClient
{
  Promise<Integer> activity1();
  Promise<Integer> activity1(Promise<?>... waitFor);
  Promise<Integer> activity1(ActivitySchedulingOptions optionsOverride,
                             Promise<?>... waitFor);
  Promise<Void> activity2(int a);
  Promise<Void> activity2(int a,
                          Promise<?>... waitFor);
  Promise<Void> activity2(int a,
                          ActivitySchedulingOptions optionsOverride,
                          Promise<?>... waitFor);
  Promise<Void> activity2(Promise<Integer> a);
  Promise<Void> activity2(Promise<Integer> a,
                          Promise<?>... waitFor);
  Promise<Void> activity2(Promise<Integer> a,
                          ActivitySchedulingOptions optionsOverride,
                          Promise<?>... waitFor);
}
```

このインターフェイスには、@Activities インターフェイスの各アクティビティメソッドに対応するオーバーロードメソッドのセットが含まれています。これらのオーバーロードは、利便性のために提供されており、アクティビティを非同期的に呼び出すために使用できます。@Activities インターフェイスのアクティビティメソッドごとに、以下のメソッドオーバーロードがクライアントインターフェイスに生成されます。

1. 元の引数をそのまま取るオーバーロード。このオーバーロードの戻り値の型は Promise<*T*> であり、ここで *T* は元のメソッドの戻り値の型です。例:

元のメソッド:

```
void activity2(int foo);
```

生成されたメソッド:

```
Promise<Void> activity2(int foo);
```

ワークフローのすべての引数が使用可能で、待機する必要がない場合は、このオーバーロードを使用します。

2. 元の引数 (そのまま)、ActivitySchedulingOptions 型の引数、および追加の Promise<?> 型の変数引数を取るオーバーロード。このオーバーロードの戻り値の型は Promise<*T*> であり、ここで *T* は元のメソッドの戻り値の型です。例:

   元のメソッド:

   ```
   void activity2(int foo);
   ```

   生成されたメソッド:

   ```
   Promise<Void> activity2(
     int foo,
     ActivitySchedulingOptions optionsOverride,
     Promise<?>... waitFor);
   ```

   このオーバーロードを使用するのは、ワークフローのすべての引数が使用可能で待機する必要がない場合、デフォルト設定を上書きする場合、または追加の Promise の準備が完了するまで待機する場合です。この変数引数を使用して、引数として宣言されていないが、呼び出しを実行するまで待機する追加の Promise<?> オブジェクトを渡すことができます。

3. 元のメソッドの各引数が Promise<> ラッパーに置き換えられたオーバーロード。このオーバーロードの戻り値の型は Promise<*T*> であり、ここで *T* は元のメソッドの戻り値の型です。例:

   元のメソッド:

   ```
   void activity2(int foo);
   ```

   生成されたメソッド:

```
Promise<Void> activity2(Promise<Integer> foo);
```

このオーバーロードは、アクティビティに渡す引数を非同期的に評価する場合に使用します。このメソッドオーバーロードに対する呼び出しは、オーバーロードに渡したすべての引数の準備が完了するまで、実行されません。

4. 元のメソッドの各引数が Promise<> ラッパーに置き換えられたオーバーロード。このオーバーロードには、追加の ActivitySchedulingOptions 型の引数と Promise<?> 型の変数引数があります。このオーバーロードの戻り値の型は Promise<*T*> であり、ここで *T* は元のメソッドの戻り値の型です。例:

   元のメソッド:

   ```
   void activity2(int foo);
   ```

   生成されたメソッド:

   ```
   Promise<Void> activity2(
      Promise<Integer> foo,
      ActivitySchedulingOptions optionsOverride,
      Promise<?>...waitFor);
   ```

   このオーバーロードを使用するのは、アクティビティに渡す引数を非同期的に評価する場合、タイプに登録したデフォルト設定を上書きする場合、または追加の Promise の準備が完了するまで待機する場合です。このメソッドオーバーロードに対する呼び出しは、オーバーロードに渡したすべての引数の準備が完了するまで、実行されません。生成されたクライアントクラスでは、このインターフェイスを実装します。各インターフェイスメソッドの実装では、Amazon SWF API を使用した適切な型のアクティビティタスクをスケジュールするためのリクエストを作成し、Amazon SWF に送信します。

5. 元の引数 (そのまま) と追加の Promise<?> 型の変数引数を取るオーバーロード。このオーバーロードの戻り値の型は Promise<*T*> であり、ここで *T* は元のメソッドの戻り値の型です。例:

   元のメソッド:

   ```
   void activity2(int foo);
   ```

   生成されたメソッド:

```
Promise< Void > activity2(int foo,
                              Promise<?>...waitFor);
```

アクティビティのすべての引数が使用可能で、待機する必要がないが、他の Promise オブジェクトの準備が完了するのを待機する場合は、このオーバーロードを使用します。

6. 元のメソッドの各引数が Promise ラッパーに置き換えられ、さらに追加の Promise<?> 型の変数引数を取るオーバーロード。このオーバーロードの戻り値の型は Promise<*T*> であり、ここで *T* は元のメソッドの戻り値の型です。例:

   元のメソッド:

   ```
   void activity2(int foo);
   ```

   生成されたメソッド:

   ```
   Promise<Void> activity2(
     Promise<Integer> foo,
     Promise<?>... waitFor);
   ```

   アクティビティのすべての引数を非同期的に待機し、さらに他のいくつかの Promise の準備が完了するのを待機する場合は、このオーバーロードを使用します。このメソッドオーバーロードに対する呼び出しは、渡したすべての Promise オブジェクトが準備完了状態になると、非同期的に実行されます。

また、生成されたアクティビティクライアントには、各アクティビティメソッドに対応する保護されたメソッド ({*activity method name*}Impl()) があり、すべてのアクティビティオーバーロードから呼び出されます。このメソッドを上書きして、モッククライアント実装を作成できます。このメソッドは引数として、ActivitySchedulingOptions ラッパーの元のメソッドに対するすべての引数、Promise<>、および Promise<?> 型の変数引数を取ります。例:

元のメソッド:

```
void activity2(int foo);
```

生成されたメソッド:

```
Promise<Void> activity2Impl(
```

```
    Promise<Integer> foo,
    ActivitySchedulingOptions optionsOverride,
    Promise<?>...waitFor);
```

スケジュールオプション

生成されたアクティビティクライアントでは、ActivitySchedulingOptions を引数として渡すことができます。ActivitySchedulingOptions 構造内の設定により、Amazon SWF でフレームワークがスケジュールするアクティビティタスクの構成を決定します。これらの設定は、登録オプションで指定したデフォルトを上書きします。スケジュールオプションを動的に指定するには、ActivitySchedulingOptions オブジェクトを作成し、必要に応じて設定した上で、アクティビティメソッドに渡します。次の例では、アクティビティタスク用のタスクリストを指定しています。これにより、このアクティビティの呼び出し用に登録時に設定したデフォルトのタスクリストが上書きされます。

```
public class OrderProcessingWorkflowImpl implements OrderProcessingWorkflow {

    OrderProcessingActivitiesClient activitiesClient
            = new OrderProcessingActivitiesClientImpl();

    // Workflow entry point
    @Override
    public void processOrder(Order order) {
        Promise<Void> paymentProcessed = activitiesClient.processPayment(order);
        ActivitySchedulingOptions schedulingOptions
             = new ActivitySchedulingOptions();
        if (order.getLocation() == "Japan") {
            schedulingOptions.setTaskList("TasklistAsia");
        } else {
            schedulingOptions.setTaskList("TasklistNorthAmerica");
        }

        activitiesClient.shipOrder(order,
                                   schedulingOptions,
                                   paymentProcessed);
    }
}
```

動的クライアント

生成されたクライアントに加えて、フレームワークには、汎用クライアントとして DynamicWorkflowClient と DynamicActivityClient もあります。これらを使用してワークフロー実行の動的なスタート、シグナルの送信、アクティビティのスケジュールなどを行うことができます。たとえば、設計時には未知である型のアクティビティをスケジュールできます。このようなアクティビティタスクは、DynamicActivityClient を使用してスケジュールできます。同様に、DynamicWorkflowClient を使用して子ワークフロー実行を動的にスケジュールできます。次の例では、ワークフローでデータベースからアクティビティを探し、これを動的アクティビティクライアントを使用してスケジュールします。

```
//Workflow entrypoint
@Override
public void start() {
   MyActivitiesClient client = new MyActivitiesClientImpl();
  Promise<ActivityType> activityType
      = client.lookUpActivityFromDB();
  Promise<String> input = client.getInput(activityType);
  scheduleDynamicActivity(activityType,
                          input);
}
@Asynchronous
void scheduleDynamicActivity(Promise<ActivityType> type,
                             Promise<String> input){
  Promise<?>[] args = new Promise<?>[1];
   args[0] = input;
   DynamicActivitiesClient activityClient
       = new DynamicActivitiesClientImpl();
   activityClient.scheduleActivity(type.get(),
                                   args,
                                   null,
                                   Void.class);
}
```

詳細については、 AWS SDK for Java ドキュメントを参照してください。

ワークフロー実行のシグナル送信とキャンセル

生成されたワークフロークライアントには、ワークフローに送信できる各シグナルに対応するメソッドがあります。これらのメソッドをワークフロー内から使用して、他のワークフロー実行にシグナルを送信できます。これにより、シグナル信号の型指定された機構が提供されます。ただし、シグナル

名をメッセージで受信した場合など、シグナル名を動的に決定する必要がある場合があります。動的ワークフロークライアントを使用すると、任意のワークフロー実行にシグナルを動的に送信できます。同様に、このクライアントを使用して、別のワークフロー実行の取り消しをリクエストできます。

次の例では、ワークフローはデータベースからシグナルを送信する先の実行を見つけ、動的ワークフロークライアントを使ってシグナルを動的に送信します。

```
//Workflow entrypoint
public void start()
{
  MyActivitiesClient client = new MyActivitiesClientImpl();
  Promise<WorkflowExecution> execution = client.lookUpExecutionInDB();
  Promise<String> signalName = client.getSignalToSend();
  Promise<String> input = client.getInput(signalName);
  sendDynamicSignal(execution, signalName, input);
}

@Asynchronous
void sendDynamicSignal(
   Promise<WorkflowExecution> execution,
   Promise<String> signalName,
   Promise<String> input)
{
   DynamicWorkflowClient workflowClient
      = new DynamicWorkflowClientImpl(execution.get());
   Object[] args = new Promise<?>[1];
   args[0] = input.get();
   workflowClient.signalWorkflowExecution(signalName.get(), args);
}
```

ワークフロー実装

ワークフローを実装するには、必要な @Workflow インターフェイスを実装するクラスを記述します。たとえば、ワークフローインターフェイス例 (MyWorkflow) を実装する場合は以下のとおりです。

```
public class MyWFImpl implements MyWorkflow
{
  MyActivitiesClient client = new MyActivitiesClientImpl();
  @Override
```

```
  public void startMyWF(int a, String b){
    Promise<Integer> result = client.activity1();
    client.activity2(result);
  }
  @Override
  public void signal1(int a, int b, String c){
    //Process signal
     client.activity2(a + b);
  }
}
```

このクラスの @Execute メソッドは、ワークフローロジックのエントリポイントです。フレームワークは、決定タスクが処理されるときにリプレイを使用してオブジェクトの状態を再構築するため、決定タスクごとに新しいオブジェクトが作成されます。

Promise<*T*> をパラメータとして使用することは、@Workflow インターフェイス内の @Execute メソッドでは禁止されています。これは非同期呼び出しが純粋に呼び出し元の決定に基づくためです。ワークフロー実装自体は、呼び出しが同期であるか非同期であるかに依存しません。したがって、生成されたクライアントインターフェイスのオーバーロードでは、これらのメソッドを非同期的に呼び出せるように Promise<*T*> パラメータを取ります。

@Execute メソッドの戻り値の型は、void または Promise<*T*> に限られます。対応する外部クライアントの戻り値の型は、void であり、Promise<> ではないことに注意してください。外部クライアントは非同期コードから使用することを意図していないため、外部クライアントはPromiseオブジェクトを返しません。外部で記述されているワークフロー実行の結果を取得するには、アクティビティを通じて外部データストアで状態を更新するようワークフローを設計できます。Amazon SWF の可視性 API は、診断目的でワークフローの結果を取得するためにも使用できます。これらの APIs コールは Amazon SWF によってスロットリングされる可能性があるため、一般的なプラクティスとして、可視性 API を使用してワークフロー実行の結果を取得することはお勧めしません。可視性 API では、WorkflowExecution 構造を使用してワークフロー実行を識別する必要があります。この構造は、getWorkflowExecution メソッドを呼び出して、生成されたワークフロークライアントから取得できます。このメソッドにより、クライアントがバインドされているワークフロー実行に対応する WorkflowExecution 構造が返されます。可視性 API に関する詳細については、「Amazon Simple Workflow Service API Reference」(Amazon Simple Workflow Service API リファレンス) を参照してください。

ワークフロー実装からアクティビティを呼び出す場合は、生成されたアクティビティクライアントを使用します。同様に、シグナルを送信するには、生成されたワークフロークライアントを使用します。

決定コンテキスト

フレームワークでは、ワークフローコードがフレームワークで実行されるたびに環境コンテキストを提供します。このコンテキスト固有の機能をワークフロー実装 (タイマーの作成など) で使用できます。詳細については、「実行コンテキスト」セクションを参照してください。

実行状態の公開

Amazon SWF では、カスタム状態をワークフロー履歴に追加できます。ワークフロー実行から報告される最新の状態は、可視性呼び出しを通じて Amazon SWF サービスと Amazon SWF コンソールに返されます。たとえば、注文処理ワークフローで「受注」、「注文出荷」などのさまざまなステージで注文のステータスを報告できます。 AWS Flow Framework for Java では、これを `@GetState` 注釈が設定された、ワークフローインターフェイスのメソッドを通じて達成できます。ディサイダーは、決定タスクの処理を完了すると、このメソッドを呼び出してワークフロー実装から最新の状態を取得します。可視性呼び出しとは別に、生成された外部クライアント (内部で可視性 API コールを使用する) を使用して状態を取得することもできます。

次の例は、実行コンテキストの設定方法を示しています。

```
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
                defaultTaskStartToCloseTimeoutSeconds = 10)
public interface PeriodicWorkflow {

    @Execute(version = "1.0")
    void periodicWorkflow();

    @GetState
    String getState();
}

@Activities(version = "1.0")
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
                             defaultTaskStartToCloseTimeoutSeconds = 3600)
public interface PeriodicActivity {
    void activity1();

}

public class PeriodicWorkflowImpl implements PeriodicWorkflow {

    private DecisionContextProvider contextProvider
```

```
                = new DecisionContextProviderImpl();

    private WorkflowClock clock
                = contextProvider.getDecisionContext().getWorkflowClock();

    private PeriodicActivityClient activityClient
                = new PeriodicActivityClientImpl();

    private String state;

    @Override
    public void periodicWorkflow() {
        state = "Just Started";
        callPeriodicActivity(0);
    }

    @Asynchronous
    private void callPeriodicActivity(int count,
                                      Promise<?>... waitFor)
    {
        if(count == 100) {
            state = "Finished Processing";
            return;
        }

        // call activity
        activityClient.activity1();

        // Repeat the activity after 1 hour.
        Promise<Void> timer = clock.createTimer(3600);
        state = "Waiting for timer to fire. Count = "+count;
        callPeriodicActivity(count+1, timer);
    }

    @Override
    public String getState() {
        return state;
    }
}

public class PeriodicActivityImpl implements PeriodicActivity
{
@Override
      public static void activity1()
```

```
   {
      ...
    }
}
```

生成された外部クライアントを使用して、ワークフロー実行の最新状態をいつでも取得できます。

```
PeriodicWorkflowClientExternal client
        = new PeriodicWorkflowClientExternalFactoryImpl().getClient();
System.out.println(client.getState());
```

上の例で、実行状態はさまざまなステージで報告されます。ワークフローインスタンスが開始すると、periodicWorkflow は最初の状態を「Just Started」として報告します。次に、callPeriodicActivity に対する呼び出しごとにワークフロー状態が更新されます。activity1 が 100 回呼び出されると、このメソッドは戻り、ワークフローインスタンスが完了します。

ワークフローのローカル

ワークフロー実装で静的変数の使用が必要になる場合があります。たとえば、さまざまな場所 (さまざまなクラスなど) からアクセスされるカウンターをワークフローの実装に保存すると便利な場合があります。ただし、ワークフローの静的変数には依存できません。静的変数はスレッド間で共有されているため、ワーカーは複数の異なるスレッドで複数の異なる決定タスクを同時に処理することがあり、問題が生じます。または、そのような状態をワークフロー実装のフィールドに保存することもできますが、その場合は実装オブジェクトを多方面に渡す必要があります。このニーズに対処するには、フレームワークで WorkflowExecutionLocal<?> クラスを提供します。セマンティクスなどの静的変数を必要とする状態は、WorkflowExecutionLocal<?> を使用してインスタンスローカルとして保持します。このタイプの静的変数を宣言して使用できます。たとえば、次のスニペットでは、WorkflowExecutionLocal<String> を使用してユーザー名を保存しています。

```
public class MyWFImpl implements MyWF {
  public static WorkflowExecutionLocal<String> username
      = new WorkflowExecutionLocal<String>();

  @Override
  public void start(String username){
    this.username.set(username);
    Processor p = new Processor();
    p.updateLastLogin();
```

```
    p.greetUser();
   }

  public static WorkflowExecutionLocal<String> getUsername() {
    return username;
  }

  public static void setUsername(WorkflowExecutionLocal<String> username) {
    MyWFImpl.username = username;
  }
}

public class Processor {
  void updateLastLogin(){
    UserActivitiesClient c = new UserActivitiesClientImpl();
    c.refreshLastLogin(MyWFImpl.getUsername().get());
  }
   void greetUser(){
    GreetingActivitiesClient c = new GreetingActivitiesClientImpl();
    c.greetUser(MyWFImpl.getUsername().get());
  }
}
```

アクティビティ実装

アクティビティを実装するには、@Activities インターフェイスの実装を提供します。 AWS Flow Framework for Java は、ワーカーで設定されたアクティビティ実装インスタンスを使用して、実行時にアクティビティタスクを処理します。ワーカーは適切なタイプのアクティビティ実装を自動的に見つけます。

アクティビティインスタンスにリソース (データベース接続など) を渡すには、プロパティやフィールドを使用できます。アクティビティ実装オブジェクトは複数のスレッドからアクセスできる可能性があるため、共有リソースはスレッドセーフである必要があります。

アクティビティ実装は、Promise<> 型のパラメータや、この型を戻り値の型として使用しないことに注意してください。アクティビティの実装は、呼び出し方法 (同期または非同期) に依存しないことが求められるためです。

前に示したアクティビティインターフェイスは次のように実装できます。

```
public class MyActivitiesImpl implements MyActivities {
```

```
    @Override
    @ManualActivityCompletion
    public int activity1(){
       //implementation
    }

    @Override
    public void activity2(int foo){
      //implementation
    }
}
```

アクティビティ実装では、スレッドローカルコンテキストを使用してタスクオブジェクトや使用するデータコンバータオブジェクトなどを取得できます。現在のコンテキストには、`ActivityExecutionContextProvider.getActivityExecutionContext()` を通じてアクセスできます。詳細については、「」の AWS SDK for Java ドキュメントActivityExecutionContextと「」セクションを参照してください実行コンテキスト。

マニュアルでのアクティビティの完了

上の例の `@ManualActivityCompletion` 注釈は、オプションの注釈です。この注釈は、アクティビティを実装するメソッドでのみ使用できます。これにより、アクティビティメソッドが戻ったときにアクティビティが自動的に完了しないように設定します。これは、アクティビティを非同期的に完了する場合 (人間のアクションが完了した後でマニュアルで完了する場合など) に役立ちます。

デフォルトでは、アクティビティメソッドが戻ると、フレームワークではアクティブティを完了済みとみなします。つまり、アクティビティワーカーは Amazon SWF に対してアクティビティタスクの完了を報告し、その結果 (ある場合) を提供します。ただし、アクティビティメソッドが戻ったときに、アクティビティタスクを完了済みとしてマークしないように設定することもできます。これは人間のタスクをモデリングするときに特に便利です。たとえば、アクティビティタスクが完了する前に、仕事を完了する必要がある人に対して、アクティビティメソッドから E メールを送信する場合があります。このような場合、`@ManualActivityCompletion` 注釈を使用してアクティビティメソッドに注釈を設定し、アクティビティを自動的に完了しないようにアクティビティワーカーに指示できます。アクティビティをマニュアルで完了するには、フレームワークに用意されている `ManualActivityCompletionClient` を使用するか、Amazon SWF SDK に用意されている Amazon SWF Java クライアントの `RespondActivityTaskCompleted` メソッドを使用できます。詳細については、 AWS SDK for Java ドキュメントを参照してください。

アクティビティタスクを完了するには、タスクトークンを指定する必要があります。タスクトークンは、タスクを一意に識別するために Amazon SWF によって使用さ

れます。このトークンには、アクティビティ実装の ActivityExecutionContext からアクセスできます。タスクを完了する当事者に、このトークンを渡す必要があります。このトークンを ActivityExecutionContext から取得するには、ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken() を呼び出します。

Hello World 例の getName アクティビティを実装することで、挨拶メッセージを提供するように誰かに依頼する E メールを送信できます。

```
@ManualActivityCompletion
@Override
public String getName() throws InterruptedException {
    ActivityExecutionContext executionContext
         = contextProvider.getActivityExecutionContext();
    String taskToken = executionContext.getTaskToken();
    sendEmail("abc@xyz.com",
         "Please provide a name for the greeting message and close task with token: " +
 taskToken);
    return "This will not be returned to the caller";
}
```

次のコードスニペットでは、挨拶を提供した後で、ManualActivityCompletionClient を使用してタスクを終了できます。または、タスクを失敗させることもできます。

```
public class CompleteActivityTask {

    public void completeGetNameActivity(String taskToken) {

        AmazonSimpleWorkflow swfClient
           = new AmazonSimpleWorkflowClient(...); // use AWS access keys
        ManualActivityCompletionClientFactory manualCompletionClientFactory
           = new ManualActivityCompletionClientFactoryImpl(swfClient);
        ManualActivityCompletionClient manualCompletionClient
           = manualCompletionClientFactory.getClient(taskToken);
        String result = "Hello World!";
        manualCompletionClient.complete(result);
    }

    public void failGetNameActivity(String taskToken, Throwable failure) {
        AmazonSimpleWorkflow swfClient
           = new AmazonSimpleWorkflowClient(...); // use AWS access keys
        ManualActivityCompletionClientFactory manualCompletionClientFactory
```

```
            = new ManualActivityCompletionClientFactoryImpl(swfClient);
        ManualActivityCompletionClient manualCompletionClient
            = manualCompletionClientFactory.getClient(taskToken);
        manualCompletionClient.fail(failure);
    }
}
```

AWS Lambda タスクの実装

トピック

- について AWS Lambda
- Lambda タスクを使用する利点と制限
- AWS Flow Framework for Java ワークフローでの Lambda タスクの使用
- HelloLambda サンプルを表示する

について AWS Lambda

AWS Lambda は、カスタムコードまたは Amazon S3、DynamoDB DynamoDB 、Amazon Kinesis、Amazon SNS、Amazon Cognito などのさまざまな AWS サービスによって生成されたイベントに応答してコードを実行するフルマネージド型のコンピューティングサービスです。Lambda の詳細については、 デベロッパーガイドAWS Lambda を参照してください。

Amazon Simple Workflow Service は Lambda タスクを提供し、従来の Amazon SWF アクティビティの代わりに、またはそれと一緒に Lambda 関数を実行できるようにします。

> **Important**
>
> AWS アカウントは、Amazon SWF がユーザーに代わって実行した Lambda 実行 (リクエスト) に対して課金されます。Lambda の料金の詳細については、https://aws.amazon.com/lambda/pricing/ を参照してください。

Lambda タスクを使用する利点と制限

従来の Amazon SWF アクティビティの代わりに Lambda タスクを使用することには、多くの利点があります。

- Lambda タスクは、Amazon SWF アクティビティタイプのように登録またはバージョン管理する必要はありません。
- 既にワークフローで定義している既存の Lambda 関数を使用することができます。
- Lambda 関数は Amazon SWF によって直接呼び出されます。従来のアクティビティのように実行するためのワーカープログラムを実装する必要はありません。
- Lambda では、関数の実行を追跡し分析するためのメトリクスとログが用意されています。

Lambda タスクには注意すべきいくつかの制限があります。

- Lambda タスクは、Lambda をサポートする AWS リージョンでのみ実行できます。Lambda で現在サポートされているリージョンの詳細については、「Amazon Web Services General Reference」(Amazon Web Services 全般リファレンス) の「Lambda Regions and Endpoints」(Lambda リージョンとエンドポイント) を参照してください。
- Lambda タスクは現在、ベース SWF HTTP API と AWS Flow Framework for Java でのみサポートされています。現在、 AWS Flow Framework for Ruby では Lambda タスクはサポートされていません。

AWS Flow Framework for Java ワークフローでの Lambda タスクの使用

AWS Flow Framework for Java ワークフローで Lambda タスクを使用するには、次の 3 つの要件があります。

- 実行するための Lambda 関数。定義した任意の Lambda 関数を使用できます。Lambda関数を作成する方法の詳細については、「AWS Lambda デベロッパーガイド」を参照してください。
- Amazon SWF ワークフローから Lambda 関数を実行するアクセス権限を付与する IAM ロール。
- ワークフロー内から Lambda タスクをスケジュールするコード。

IAM ロールのセットアップ

Amazon SWF から Lambda 関数を呼び出す前に、Amazon SWF から Lambda へのアクセス権を付与する IAM ロールを準備する必要があります。次のいずれかを行うことができます。

- あらかじめ定義されたロール、AWSLambdaRole を選択して、ワークフローにアカウントに関連する Lambda 関数を呼び出すアクセス許可を付与します。
- 独自のポリシーと関連付けられたロールを定義して、Amazon リソースネーム (ARN) で指定された特定の Lambda 関数を呼び出すためのワークフローのアクセス許可を付与します。

IAM ロールのアクセス許可を制限する

Amazon SWF に提供する IAM ロールに対するアクセス許可を制限するには、リソースの信頼ポリシーの SourceArn および SourceAccount コンテキストキーを使用します。これらのキーは、指定されたドメイン ARN に属する Amazon Simple Workflow Service の実行からのみ使用されるように、IAM ポリシーの使用を制限します。これらのグローバル条件コンテキストキーの両方を、同じポリシーステートメントで使用する場合、aws:SourceAccount 値と aws:SourceArn 値の中の参照されるアカウントには、同じアカウント ID を使用する必要があります。

次の例では、SourceArnコンテキストキーは、アカウント の に属する Amazon Simple Workflow Service の実行でのみ IAM サービスロールを使用するように制限someDomainします123456789012。

- ステートメント 1

  プリンシパル : "Service": "swf.amazonaws.com"

  アクション: sts:AssumeRole

```
"Condition": {
   "ArnLike": {
     "aws:SourceArn": "arn:aws:swf:*:123456789012:/domain/someDomain"
   }
}
```

次の例では、SourceAccountコンテキストキーは、アカウント の Amazon Simple Workflow Service 実行でのみ IAM サービスロールを使用するように制限します123456789012。

```
"Condition": {
   "StringLike": {
     "aws:SourceAccount": "123456789012"
   }
}
```

Amazon SWF に Lambda ロールを呼び出すためのアクセスを提供する

あらかじめ定義されたロール、AWSLambdaRole を使用して、Amazon SWF ワークフローがアカウントに関連する Lambda 関数を呼び出せるようにできます。

AWSLambdaRole を使用して、Lambda 関数を呼び出すアクセス権を Amazon SWF に付与するには

1. Amazon IAM コンソールを開きます。
2. [Roles] (ロール) を選択してから [Create New Role] (ロールの新規作成) を選びます。
3. ロールに `swf-lambda` などの名前を付け、[Next Step] (次のステップ) を選択します。
4. [AWS サービスロール] で、[Amazon SWF]、[次のステップ] の順に選択します。
5. [ポリシーのアタッチ] 画面で、リストから [AWSLambdaRole] を選択します。
6. ロールを確認したら、[Next Step] (次のステップ) を選択してから、[Create Role] (ロールの作成) を選択します。

特定の Lambda 関数を呼び出すためのアクセス権を付与する IAM ロールの定義

ワークフローから特定の Lambda 関数を呼び出すためのアクセスを提供する場合は、独自の IAM ポリシーを定義する必要があります。

特定の Lambda 関数へのアクセスを提供する IAM ポリシーを作成するには

1. Amazon IAM コンソールを開きます。
2. [Policies] (ポリシー) を選択して、[Create Policy] (ポリシーの作成) を選択します。
3. AWS 管理ポリシーのコピーを選択し、リストから AWSLambdaRole を選択します。ポリシーが生成されます。必要に応じて名前と説明を編集することもできます。
4. [Policy Document] (ポリシードキュメント) の [Resource] (リソース) フィールドに、Lambda 関数の ARN を追加します。例:

   - リソース: `arn:aws:lambda:us-east-1:111122223333:function:hello_lambda_function`

   > **Note**
   >
   > IAM ロールでリソースを指定する方法の詳細については、「Using IAM」(IAM の使用) の「Overview of IAM Policies」(IAM ポリシーの概要) を参照してください。

5. [Create Policy] (ポリシーの作成) を選択してポリシーの作成を完了します。

新しい IAM ロールを作成するときにこのポリシーを選択し、そのロールを使用して Amazon SWF ワークフローへのアクセスを呼び出すことができます。この手順は、AWSLambdaRole ポリシーを

使用してロールを作成する場合と非常によく似ています。代わりに、ロールを作成するときに独自のポリシーを選択します。

Lambda ポリシーを使用して Amazon SWF ロールを作成するには

1. Amazon IAM コンソールを開きます。
2. [Roles] (ロール) を選択してから [Create New Role] (ロールの新規作成) を選びます。
3. ロールに `swf-lambda-function` などの名前を付け、[Next Step] (次のステップ) を選択します。
4. [AWS サービスロール] で、[Amazon SWF]、[次のステップ] の順に選択します。
5. [ポリシーのアタッチ] 画面で、リストから Lambda 関数固有のポリシーを選択します。
6. ロールを確認したら、[Next Step] (次のステップ) を選択してから、[Create Role] (ロールの作成) を選択します。

Lambda タスクの実行をスケジュール設定する

Lambda 関数の呼び出しを許可する IAM ロールを定義したら、ワークフローの一部として、実行スケジュールを設定できます。

> **Note**
>
> このプロセスは、 AWS SDK for Javaに含まれる HelloLambda サンプルで完全に実証されています。

Lambda タスクの実行をスケジュール設定するには

1. ワークフロー実装で、`LambdaFunctionClient` のインスタンスを取得するには、`DecisionContext` インスタンスで `getLambdaFunctionClient()` を呼び出します。

```
// Get a LambdaFunctionClient instance
DecisionContextProvider decisionProvider = new DecisionContextProviderImpl();
DecisionContext decisionContext = decisionProvider.getDecisionContext();
LambdaFunctionClient lambdaClient = decisionContext.getLambdaFunctionClient();
```

2. LambdaFunctionClient の scheduleLambdaFunction() メソッドを使用してタスクをスケジュール設定します。その際、作成した Lambda 関数の名前と、Lambda タスクの入力データを渡します。

```
// Schedule the Lambda function for execution, using your IAM role for access.
String lambda_function_name = "The name of your Lambda function.";
String lambda_function_input = "Input data for your Lambda task.";

lambdaClient.scheduleLambdaFunction(lambda_function_name, lambda_function_input);
```

3. ワークフロー実行スターターで、IAM lambda ロールをデフォルトワークフローオプションに追加するには、StartWorkflowOptions.withLambdaRole() を使用して、ワークフロー開始時にオプションを渡します。

```
// Workflow client classes are generated for you when you use the @Workflow
// annotation on your workflow interface declaration.
MyWorkflowClientExternalFactory clientFactory =
  new MyWorkflowClientExternalFactoryImpl(sdk_swf_client, swf_domain);

MyWorkflowClientExternal workflow_client = clientFactory.getClient();

// Give the ARN of an IAM role that allows SWF to invoke Lambda functions on
// your behalf.
String lambda_iam_role = "arn:aws:iam::111111000000:role/swf_lambda_role";

StartWorkflowOptions workflow_options =
  new StartWorkflowOptions().withLambdaRole(lambda_iam_role);

// Start the workflow execution
workflow_client.helloWorld("User", workflow_options);
```

HelloLambda サンプルを表示する

Lambda タスクを使用するワークフローを実装するサンプルは、 AWS SDK for Javaで入手できます。表示して実行するには、ソースをダウンロードします。

HelloLambda サンプルを構築して実行する方法の詳細については、 AWS Flow Framework for Java サンプルに付属の README ファイルを参照してください。

AWS Flow Framework for Java で記述されたプログラムの実行

トピック

このフレームワークでは、ワーカークラスを使用して、 AWS Flow Framework for Java ランタイムを初期化し、Amazon SWF と通信することができます。ワークフローまたはアクティビティワーカーを実装するには、ワーカークラスのインスタンスを作成して起動する必要があります。これらのワーカークラスは、進行中の非同期オペレーションの管理、ブロック解除される非同期メソッドの呼び出し、Amazon SWF との通信を行います。これらは、ワークフローおよびアクティビティ実装、スレッド数、ポーリングするタスクリストなどを使用して設定できます。

フレームワークには 2 つのワーカークラスが用意されており、アクティビティとワークフローに使用します。ワークフローロジックを実行するには、`WorkflowWorker` クラスを使用します。アクティビティと同様に、`ActivityWorker` クラスを使用します。これらのクラスは、アクティビティタスク用に Amazon SWF をポーリングし、実装に適切なメソッドを呼び出します。

次の例では、`WorkflowWorker` をインスタンス化し、タスクのポーリングを開始する方法について解説します。

```
AmazonSimpleWorkflow swfClient = new AmazonSimpleWorkflowClient(awsCredentials);
WorkflowWorker worker = new WorkflowWorker(swfClient, "domain1", "tasklist1");
// Add workflow implementation types
worker.addWorkflowImplementationType(MyWorkflowImpl.class);

// Start worker
worker.start();
```

`ActivityWorker` のインスタンスを作成し、タスクのポーリングを開始する基本ステップは次のとおりです。

```
AmazonSimpleWorkflow swfClient
      = new AmazonSimpleWorkflowClient(awsCredentials);
ActivityWorker worker = new ActivityWorker(swfClient,
                                           "domain1",
```

```
                                                "tasklist1");
worker.addActivitiesImplementation(new MyActivitiesImpl());

// Start worker
worker.start();
```

アクティビティまたはディサイダーをシャットダウンする場合、使用されているワーカークラスのインスタンスと、Amazon SWF Java クライアントインスタンスをアプリケーションからシャットダウンします。これにより、ワーカークラスで使用されるリソースはすべて、正式にリリースされます。

```
worker.shutdown();
worker.awaitTermination(1, TimeUnit.MINUTES);
```

実行を開始するには、生成した外部クライアントのインスタンスを作成し、@Execute メソッドを呼び出すだけです。

```
MyWorkflowClientExternalFactory factory = new MyWorkflowClientExternalFactoryImpl();
MyWorkflowClientExternal client = factory.getClient();
client.start();
```

WorkflowWorker

変数名が示すように、このワーカークラスは、ワークフローの実装を目的としています。これは、タスクリストとワークフローの実装タイプで設定されます。ワーカークラスは、ループを実行して、指定のタスクリストでディシジョンタスクをポーリングします。ディシジョンタスクが送信されると、ワークフロー実装のインスタンスを作成し、@Execute メソッドを呼び出してタスクを処理します。

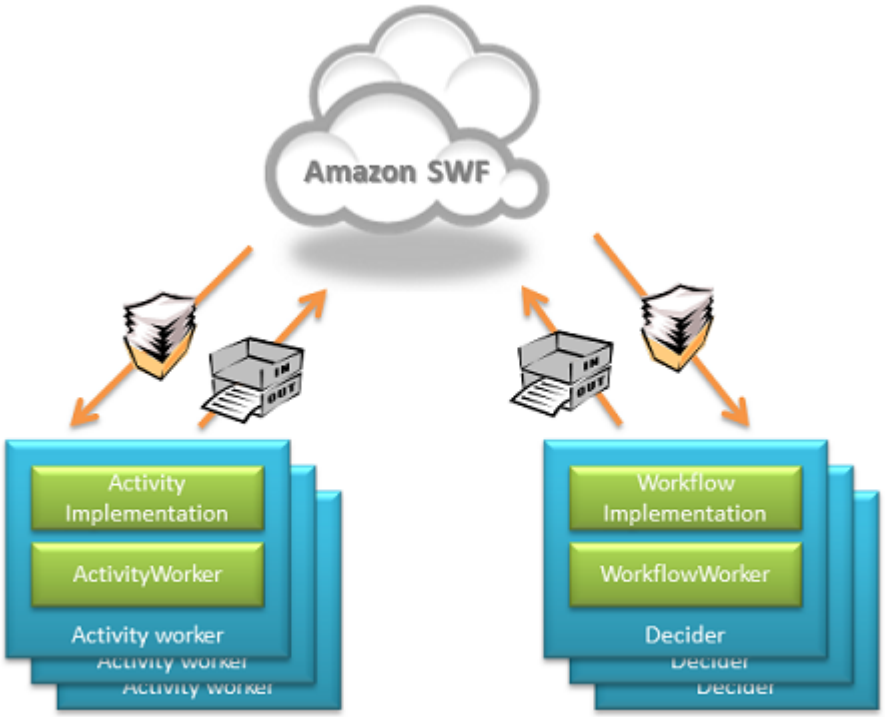
ActivityWorker

アクティビティワーカーを実装するには、ActivityWorker クラスを使用して、アクティビティタスクのタスクリストを簡単にポーリングできます。アクティビティ実装オブジェクトを使用して、アクティビティワーカーを設定します。このワーカークラスは、ループを実行して、指定のタスクリストでアクティビティタスクをポーリングします。アクティビティタスクが送信されると、指定した適切な実装を検索し、アクティビティメソッドを呼び出して、タスクを処理します。ファクトリを呼び出して、ディシジョンタスクごとに新しいインスタンスを作成する WorkflowWorker とは異なり、ActivityWorker は、指定したオブジェクトを単純に使用します。

`ActivityWorker` クラスは AWS Flow Framework for Java 注釈を使用して、登録オプションと実行オプションを決定します。

ワーカースレッディングモデル

AWS Flow Framework for Java では、アクティビティまたはディサイダーの実装はワーカークラスのインスタンスです。アプリケーションは、各マシンのワーカーオブジェクト、およびワーカーとして動作するプロセスの設定およびインスタンス化を行います。ワーカーオブジェクトは、Amazon SWF から自動的にタスクを受け取り、アクティビティおよびワークフロー実装に割り当て、結果を Amazon SWF に報告します。1 つのワークフローインスタンスで多数のワーカーをカバーできます。Amazon SWF に保留中のアクティビティタスクが 1 つ以上ある場合、タスクは利用可能な最初のワーカーに割り当てられ、次のワーカーと続きます。これにより、同じワークフローインスタンスに属するタスクを、異なるワーカーで同時に処理することができます。

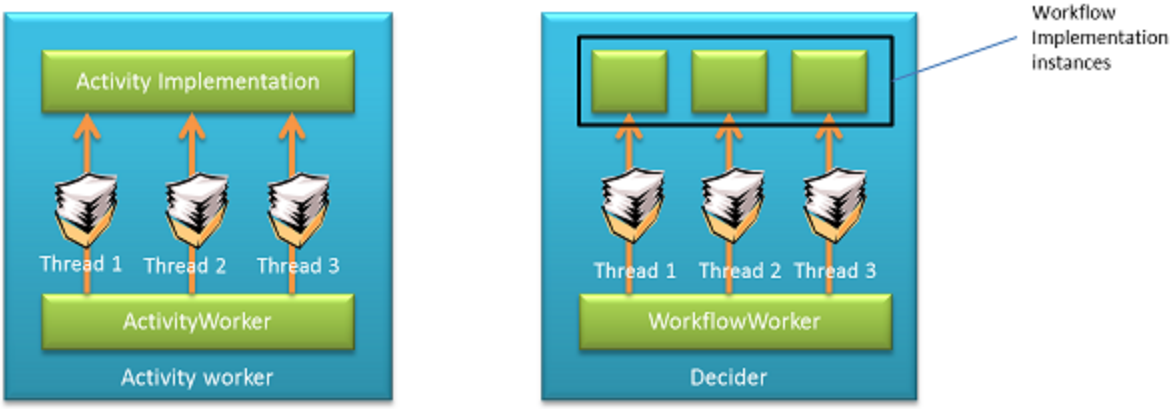


さらに、各ワーカーを設定して、複数のスレッドでタスクを処理することができます。そのため、ワークフローインスタンスのアクティビティタスクでは、ワーカーが 1 つでも同時に実行することができます。

ディシジョンタスクは、所定のワークフロー実行では、一度に 1 つの実行できないことを Amazon SWF で保証される例外と同様に動作します。1 つのワークフロー実行では、通常複数のディシジョンタスクが必要です。従って、プロセスやスレッドが複数の場合は実行が終了する場合があります。

ディサイダーは、ワークフロー実装のタイプで設定されます。ディサイダーよりディシジョンタスクを受け取ると、ワークフロー実装のインスタンス (オブジェクト) がこのタスクで作成されます。このフレームワークでは、これらのインスタンスを作成する拡張可能なファクトリパターンが用意されています。デフォルトのワークフローファクトリでは、毎回新しいオブジェクトが作成されます。この動作を上書きするには、カスタムのファクトリを指定します。

ワークフロー実装タイプで設定されているディサイダーとは異なり、アクティビティワーカーは、アクティビティ実装のインスタンス (オブジェクト) で設定されます。アクティビティワーカーよりアクティビティタスクを受け取ると、適切なアクティビティ実装オブジェクトに割り当てられます。



ワークフローワーカーは、1 つのスレッドプールを保持し、タスクの Amazon SWF をポーリングするために使用したのと同じスレッドでワークフローを実行します。アクティビティは長時間実行されるため (少なくともワークフローロジックと比較すると)、アクティビティワーカークラスは 2 つのスレッドプールを保持します。1 つはアクティビティタスクの Amazon SWF をポーリングするため、もう 1 つはアクティビティ実装を実行してタスクを処理するためです。これにより、実行するスレッドの数とは別に、タスク用にポーリングするスレッドの数を設定できます。たとえば、少数のスレッドをポーリングし、多数のスレッドでタスクを実行することができます。アクティビティワーカークラスは、無制限のポーリングスレッドと無制限のスレッドでタスクを処理する場合にのみ、タスクの Amazon SWF をポーリングします。

このスレッディングおよびインスタンスの動作は、次のことを意味します。

1. アクティビティタスクはステートレスである必要がある。インスタンス変数を使用して、アクティビティオブジェクトにアプリケーション状態を保存しないでください。ただし、フィールドを使用して、データベース接続などのリソースを保存する場合があります。
2. アクティビティタスクはスレッドセーフである必要がある。同じインスタンスを使用して異なるスレッドからのタスクを同時に処理できるため、アクティビティコードから共有リソースへのアクセスを同期する必要があります。

3. ワークフロー実装はステートフルの場合があり、インスタンス変数を使用して状態を保存する場合があります。ワークフロー実装の新しいインスタンスを作成して、ディシジョンタスクごとに処理していますが、フレームワークでは、状態が正式に再作成されます。ただし、ワークフロー実装は決定的である必要があります。詳細については、「AWS Flow Framework for Java のタスクについて」セクションを参照してください。
4. ワークフローの実装では、デフォルトファクトリを使用する際、スレッドセーフにする必要はありません。デフォルトの実装では、1 つのスレッドで一度にワークフロー実装のインスタンスが使用されます。

ワーカーの拡張機能

AWS Flow Framework for Java には、きめ細かな制御と拡張性を実現するいくつかの低レベルワーカークラスも含まれています。これにより、実装オブジェクトを作成する際、ワークフローやアクティビティタイプの登録を完全にカスタマイズし、ファクトリを設定できます。これらのワーカーは `GenericWorkflowWorker` と `GenericActivityWorker` です。

ワークフロー定義ファクトリを作成するためにファクトリを使用して、`GenericWorkflowWorker` を設定できます。ワークフロー定義ファクトリは、ワークフロー実装のインスタンスを作成し、登録オプションなどの構成設定を提供する役割を担います。通常の場合、`WorkflowWorker` クラスを直接使用する必要があります。フレームワークで提供されるファクトリ実装 (`POJOWorkflowDefinitionFactoryFactory` および `POJOWorkflowDefinitionFactory`) を自動的に作成および設定できます。ファクトリでは、ワークフロー実装クラスに引数コンストラクタは必要ありません。このコンストラクタは、実行時にワークフローオブジェクトのインスタンスを作成するために使用されます。ファクトリは、ワークフローインターフェイスおよび実装で使用した注釈を参照して、適切な登録および実行オプションを作成します。

ファクトリを独自に実装するには、`WorkflowDefinitionFactory`、`WorkflowDefinitionFactoryFactory`、および `WorkflowDefinition` を実装します。`WorkflowDefinition` クラスは、ディシジョンタスクおよびシグナルを割り当てるためにワーカークラスで使用されます。これらの基本クラスを実装することで、ファクトリと、ワークフロー実装へのリクエストの割り当てを完全にカスタマイズできます。たとえば、独自の注釈に基づき、ワークフローを記述するか、またはフレームワークで使用されるコードの最初のアプローチではなく、WSDL から生成するには、これらの拡張ポイントを使用して、カスタムプログラミングモデルを作成します。カスタムファクトリを使用するには、`GenericWorkflowWorker` クラスを使用する必要があります。これらのクラスの詳細については、 AWS SDK for Java ドキュメントを参照してください。

同様に、GenericActivityWorker を使用して、カスタムアクティビティ実装ファクトリを作成できます。ActivityImplementationFactory および ActivityImplementation クラスを実装することで、アクティビティのインスタンス化を完全に制御できるだけでなく、登録および実行オプションをカスタマイズできます。これらのクラスの詳細については、 AWS SDK for Java ドキュメントを参照してください。

実行コンテキスト

トピック

フレームワークは、環境コンテキストをワークフローおよびアクティビティの実装に渡します。このコンテキストは、処理されているタスク固有であり、実装で使用できるいくつかのユーティリティが用意されています。コンテキストオブジェクトは、新しいタスクがワーカーに処理される度に作成されます。

決定コンテキスト

決定タスクが実行されると、フレームワークは DecisionContext クラスを介してワークフロー実装にコンテキストを提供します。DecisionContext は、ワークフロー実行 ID、クロック、タイマー機能など、状況に応じた情報を提供します。

ワークフロー実装の DecisionContext にアクセスする

ワークフロー実装の DecisionContext にアクセスするには、DecisionContextProviderImpl クラスを使用します。または、テスト容易性と依存関係に示す Spring を使用して、フィールドのコンテキスト、またはワークフロー実装のプロパティを挿入することもできます。

```
DecisionContextProvider contextProvider
    = new DecisionContextProviderImpl();
DecisionContext context = contextProvider.getDecisionContext();
```

クロックおよびタイマーの作成

DecisionContext には、タイマーおよびクロック機能を提供する WorkflowClock タイプのプロパティが含まれます。ワークフローロジックは決定的である必要があるため、ワークフロー実装でシ

ステムクロックを直接使用しないでください。WorkflowClock の currentTimeMills メソッドは、処理されているディシジョンの開始イベントの時間を返します。これにより、再生時に同じ時間の値を取得できるため、ワークフローロジックを決定的にすることができます。

WorkflowClock にも createTimer メソッドがあります。このメソッドでは、指定された間隔が過ぎると、準備完了状態になる Promise オブジェクトが返ります。この値を他の非同期メソッドのパラメータとして使用し、指定の時間まで実行を遅らせることができます。このように、非同期メソッドまたはアクティビティを効率的にスケジュールして、後で実行することができます。

次の例では、アクティビティを定期的に呼び出す方法について説明します。

```
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
               defaultTaskStartToCloseTimeoutSeconds = 10)
public interface PeriodicWorkflow {

    @Execute(version = "1.0")
    void periodicWorkflow();
}

@Activities(version = "1.0")
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 300,
                             defaultTaskStartToCloseTimeoutSeconds = 3600)
public interface PeriodicActivity {
    void activity1();
}

public class PeriodicWorkflowImpl implements PeriodicWorkflow {

    private DecisionContextProvider contextProvider
         = new DecisionContextProviderImpl();

    private WorkflowClock clock
         = contextProvider.getDecisionContext().getWorkflowClock();

    @Override
    public void periodicWorkflow() {
        callPeriodicActivity(0);
    }

    @Asynchronous
    private void callPeriodicActivity(int count,
                                      Promise<?>... waitFor) {
```

```
        if (count == 100) {
            return;
        }
        PeriodicActivityClient client = new PeriodicActivityClientImpl();
        // call activity
        Promise<Void> activityCompletion = client.activity1();

        Promise<Void> timer = clock.createTimer(3600);

        // Repeat the activity either after 1 hour or after previous activity run
        // if it takes longer than 1 hour
        callPeriodicActivity(count + 1, timer, activityCompletion);
    }
}


public class PeriodicActivityImpl implements PeriodicActivity
{
@Override
   public void activity1() {
      ...
      }
}
```

上記の例では、callPeriodicActivity 非同期メソッドは activity1 を呼び出した後、現在の AsyncDecisionContext を使用してタイマーを作成します。これにより、返った Promise を引数として、それ自体に再帰的な呼び出しを行います。この再帰的呼び出しは、実行前にタイマーが鳴るまで (この例では 1 時間) 待機します。

アクティビティ実行コンテキスト

ディシジョンタスクが処理されると DecisionContext でコンテキストが渡されるように、ActivityExecutionContext では、アクティビティタスクが処理されると同様のコンテキスト情報が渡されます。このコンテキストでは、ActivityExecutionContextProviderImpl クラスを介して、アクティビティコードを利用できます。

```
ActivityExecutionContextProvider provider
    = new ActivityExecutionContextProviderImpl();
ActivityExecutionContext aec = provider.getActivityExecutionContext();
```

以下のアクションは、ActivityExecutionContext を使用して実行できます。

長時間実行アクティビティのハートビート

アクティビティが長時間実行されている場合は、その進行状況を定期的に Amazon SWF に報告し、タスクが進行中であることを知らせます。このようなハートビートが不明な場合に、タスクのハートビートのタイムアウトが、アクティビティタイプの登録時、またはアクティビティのスケジュール時に設定されていると、タイムアウトが発生する場合があります。ハートビートを送信するには、`ActivityExecutionContext` の `recordActivityHeartbeat` メソッドを使用できます。また、ハートビートには、実行中のアクティビティをキャンセルするメカニズムがあります。詳細と例については、「エラー処理」セクションを参照してください。

アクティビティタスクの詳細を取得する

必要に応じて、エグゼキューターがタスクを取得した際に Amazon SWF によって渡されたアクティビティタスクの詳細をすべて取得することができます。この詳細には、タスク、タスクタイプ、タスクトークンなどの入力に関する情報が含まれます。手動で行われているアクティビティ (例: 人間によるアクション) を実装する場合は、`ActivityExecutionContext` を使用して、タスクトークンを取得し、アクティビティタスクを少しずつ完了するプロセスに渡す必要があります。詳細については、「マニュアルでのアクティビティの完了」セクションを参照してください。

エグゼキューターによって使用されている Amazon SWF クライアントオブジェクトを取得する

エグゼキューターによって使用されている Amazon SWF クライアントオブジェクトは、`ActivityExecutionContext` の `getService` メソッドを呼び出して取得できます。このメソッドは、Amazon SWF サービスを直接呼び出す場合に便利です。

子ワークフロー実行

これまでの例では、ワークフロー実行をアプリケーションから直接開始しました。しかし、ワークフローの実行は、生成されたクライアント上でワークフローエントリポイントメソッドを呼び出すことによって、ワークフロー内から開始される場合があります。ワークフローの実行が別のワークフロー実行のコンテキストから開始された場合、その実行は、子ワークフロー実行と呼ばれます。これにより、複雑なワークフローをより小さい単位にリファクタリングし、各ワークフロー間で共有できるようになります。たとえば、支払い処理ワークフローを作成し、注文処理ワークフローから呼び出すことができます。

意味的に、子ワークフロー実行は、次の違いを除き、スタンドアロンワークフローと同じように動作します。

1. ユーザーの明示的なアクションにより親ワークフローが終了した場合 (例: TerminateWorkflowExecution Amazon SWF API を呼び出す)、またはタイムアウトにより終了した場合、子ワークフロー実行は、子ポリシーによって変わります。子ワークフローの実行を終了、キャンセル、または中止 (実行中の場合) するには、この子ポリシーを設定します。
2. 子ワークフローの出力 (エントリポイントメソッドの戻り値) は、非同期メソッドによって返る Promise<T> と同じように、親ワークフロー実行で使用できます。この実行は、アプリケーションで Amazon SWF API を使用して出力を取得するスタンドアロン実行とは異なります。

以下の例では、OrderProcessor ワークフローで PaymentProcessor 子ワークフローを作成します。

```
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
              defaultTaskStartToCloseTimeoutSeconds = 10)
public interface OrderProcessor {

    @Execute(version = "1.0")
    void processOrder(Order order);
}

public class OrderProcessorImpl implements OrderProcessor {
    PaymentProcessorClientFactory factory
         = new PaymentProcessorClientFactoryImpl();

    @Override
    public void processOrder(Order order) {
        float amount = order.getAmount();
        CardInfo cardInfo = order.getCardInfo();

        PaymentProcessorClient childWorkflowClient = factory.getClient();
        childWorkflowClient.processPayment(amount, cardInfo);
    }

}

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
                 defaultTaskStartToCloseTimeoutSeconds = 10)
public interface PaymentProcessor {

    @Execute(version = "1.0")
```

```
    void processPayment(float amount, CardInfo cardInfo);

}

public class PaymentProcessorImpl implements PaymentProcessor {
    PaymentActivitiesClient activitiesClient = new PaymentActivitiesClientImpl();

    @Override
    public void processPayment(float amount, CardInfo cardInfo) {
        Promise<PaymentType> payType = activitiesClient.getPaymentType(cardInfo);
        switch(payType.get()) {
        case Visa:
            activitiesClient.processVisa(amount, cardInfo);
            break;
        case Amex:
            activitiesClient.processAmex(amount, cardInfo);
            break;
      default:
         throw new UnSupportedPaymentTypeException();
        }
    }

}

@Activities(version = "1.0")
@ActivityRegistrationOptions(defaultTaskScheduleToStartTimeoutSeconds = 3600,
                             defaultTaskStartToCloseTimeoutSeconds = 3600)
public interface PaymentActivities {

    PaymentType getPaymentType(CardInfo cardInfo);

    void processVisa(float amount, CardInfo cardInfo);

    void processAmex(float amount, CardInfo cardInfo);

}
```

継続的なワークフロー

一部のユースケースでは、永続的に実行されるワークフロー、または長期間実行するワークフロー (例: サーバー群の状況を監視するワークフロー) が必要な場合があります。

> **Note**
>
> Amazon SWF はワークフロー実行の履歴全体を保持するため、履歴は時間の経過とともに増加し続けます。再生を実行すると、フレームワークは、Amazon SWF からこの履歴を取得します。そのため、履歴サイズが大きすぎると、コストが高くなります。このような長時間稼働ワークフローや継続的なワークフローでは、現在の実行を定期的に終了し、新しい実行を開始して処理を続行する必要があります。

これは、論理的なワークフロー実行の続きです。生成した独自クライアントは、この目的のために使用することができます。ワークフロー実装では、独自クライアントで `@Execute` メソッドを呼び出します。現在の実行が完了したら、同じワークフロー ID を使用して、フレームワークで新しい実行を開始します。

また、現在の `DecisionContext` から取得できる `GenericWorkflowClient` の `continueAsNewOnCompletion` メソッドを呼び出して、実行を続けることもできます。たとえば、次のワークフロー実装では、タイマーが 1 日後に設定されており、独自のエントリポイントを呼び出して、新しい実行を開始します。

```
public class ContinueAsNewWorkflowImpl implements ContinueAsNewWorkflow {

    private DecisionContextProvider contextProvider
         = new DecisionContextProviderImpl();

    private ContinueAsNewWorkflowSelfClient selfClient
         = new ContinueAsNewWorkflowSelfClientImpl();

    private WorkflowClock clock
         = contextProvider.getDecisionContext().getWorkflowClock();

    @Override
    public void startWorkflow() {
        Promise<Void> timer = clock.createTimer(86400);
        continueAsNew(timer);
    }

    @Asynchronous
    void continueAsNew(Promise<Void> timer) {
        selfClient.startWorkflow();
    }
```

```
}
```

ワークフローが再帰的に呼び出された場合、保留中のタスクがすべて完了し、新しいワークフロー実行が開始されると、フレームワークは現在のワークフローを終了します。保留中のタスクがある限り、現在のワークフローの実行は終了しません。新しい実行では、履歴やデータが元の実行から自動的に継承されることはありません。新しい実行の状態を引き継ぐ場合は、入力として明示的に渡す必要があります。

Amazon SWF でのタスク優先度の設定

デフォルトでは、タスクリストのタスクは到着時間に基づいて提供されます。つまり、最初にスケジュールされたタスクは、通常は可能な限り最初に実行されます。オプションのタスクの優先順位を設定することで、特定のタスクに優先順位を与えることができます。Amazon SWF は、タスクリストで優先順位が低いものよりも先に、優先順位の高いタスクの提供を試みます。

ワークフローとアクティビティの両方のタスクの優先順位を設定できます。ワークフローのタスクの優先順位は、スケジュールされるいずれのアクティビティタスクの優先順位にも影響しません。また、起動されるいずれの子ワークフローにも影響しません。アクティビティまたはワークフローのデフォルトの優先順位は、登録中に (ユーザーまたは Amazon SWF によって) 設定され、アクティビティのスケジュール中またはワークフロー実行の開始中にオーバーライドされない限り、登録されたタスクの優先順位が常に使用されます。

タスクの優先順位の値は、"-2147483648" から "2147483647" の範囲であり、値が高いほど優先順位も高くなります。アクティビティまたはワークフローのタスク優先順位を設定しない場合、優先順位としてゼロ ("0") が割り当てられます。

トピック

- ワークフローのタスクの優先順位の設定
- アクティビティのタスクの優先順位の設定

ワークフローのタスクの優先順位の設定

登録または起動時に、ワークフローのタスクの優先順位を設定できます。ワークフロータイプの登録時に設定されるタスクの優先順位は、ワークフロー実行の開始時にオーバーライドされない限り、その種類の任意のワークフロー実行のデフォルトとして使用されます。

ワークフロータイプをデフォルトのタスク優先度で登録するには、宣言するときに WorkflowRegistrationOptions で defaultTaskPriority オプションを設定します。

```
@Workflow
@WorkflowRegistrationOptions(
    defaultTaskPriority = 10,
    defaultTaskStartToCloseTimeoutSeconds = 240)
public interface PriorityWorkflow
{
    @Execute(version = "1.0")
    void startWorkflow(int a);
}
```

ワークフローの開始時に taskPriority を設定し、登録済み (デフォルト) のタスクの優先順位を上書きすることもできます。

```
StartWorkflowOptions priorityWorkflowOptions
    = new StartWorkflowOptions().withTaskPriority(10);

PriorityWorkflowClientExternalFactory cf
    = new PriorityWorkflowClientExternalFactoryImpl(swfService, domain);

priority_workflow_client = cf.getClient();

priority_workflow_client.startWorkflow(
        "Smith, John", priorityWorkflowOptions);
```

さらに、子ワークフローを開始するときや、ワークフローを新規として継続するときに、タスクの優先順位を設定できます。例えば、ContinueAsNewWorkflowExecutionParameters または StartChildWorkflowExecutionParameters で taskPriority オプションを設定できます。

アクティビティのタスクの優先順位の設定

登録時またはスケジュール時に、アクティビティのタスクの優先順位を設定できます。アクティビティタイプを登録するときに設定されるタスクの優先順位は、アクティビティのスケジュール時にオーバーライドされない限り、アクティビティ実行時のデフォルトの優先順位として使用されます。

アクティビティタイプをデフォルトのタスク優先度で登録するには、宣言するときに ActivityRegistrationOptions で defaultTaskPriority オプションを設定します。

```
@Activities(version = "1.0")
@ActivityRegistrationOptions(
    defaultTaskPriority = 10,
```

```
    defaultTaskStartToCloseTimeoutSeconds = 120)
public interface ImportantActivities {
    int doSomethingImportant();
}
```

アクティビティのスケジュール時に taskPriority を設定し、登録済み (デフォルト) のタスクの優先順位を上書きすることもできます。

```
ActivitySchedulingOptions activityOptions = new
 ActivitySchedulingOptions.withTaskPriority(10);

ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();

activityClient.doSomethingImportant(activityOptions);
```

DataConverters

ワークフロー実装でリモートアクティビティを呼び出す場合、そのアクティビティに渡す入力とアクティビティの実行結果をシリアル化して無線で送信できるようにする必要があります。フレームワークでは、この目的のために DataConverter クラスを使用します。この抽象クラスを実装して、独自のシリアライザを提供できます。フレームワークには、デフォルトの Jackson シリアライザに基づく実装 (JsonDataConverter) が用意されています。詳細については、「AWS SDK for Java ドキュメント」を参照してください。Jackson でシリアル化を行う方法および関連する Jackson 注釈の詳細については、JSON プロセッサのドキュメントを参照してください。使用されるワイヤ形式はコントラクトの一部と見なされます。したがって、@Activities 注釈と @Workflow 注釈の DataConverter プロパティを設定することで、アクティビティインターフェイスとワークフローインターフェイスで DataConverter を指定できます。

フレームワークでは、@Activities 注釈で指定した DataConverter 型のオブジェクトを作成し、アクティビティへの入力をシリアル化して、その結果を逆シリアル化します。同様に、@Workflow 注釈で指定した DataConverter 型のオブジェクトを使用して、ワークフローに渡すパラメータをシリアル化します。子ワークフローの場合は、その結果を逆シリアル化します。フレームワークでは、入力に加えて、追加のデータ (例外の詳細など) も Amazon SWF に渡します。このデータもワークフローシリアライザを使用してシリアル化します。

フレームワークで DataConverter を自動的に作成しない場合は、このインスタンスを独自に指定することもできます。生成されたクライアントには、DataConverter を取るコンストラクタオーバーロードがあります。

DataConverter 型を指定せず、DataConverter オブジェクトも渡さない場合は、デフォルトで JsonDataConverter が使用されます。

非同期メソッドにデータを渡す

トピック

Promise<*T*> の使用については、前のセクションを参照してください。Promise<*T*> の高度なユースケースはこちらで確認できます。

コレクションとマップを非同期メソッドに渡す

フレームワークは、非同期メソッドの Promise タイプとして、配列、コレクション、マップを渡すことができます。たとえば、以下のリストに示すように、非同期メソッドでは、Promise<ArrayList<String>> を引数として使用することができます。

```
@Asynchronous
public void printList(Promise<List<String>> list) {
    for (String s: list.get()) {
        activityClient.printActivity(s);
    }
}
```

意味的に、これは Promise タイプの他のパラメータとして動作し、非同期メソッドは、コレクションが実行前に利用可能になるまで待機します。コレクションのメンバーが Promise オブジェクトの場合は、以下のスニペットで示すように、すべてのメンバーが準備状態になるまでフレームワークで待機させることができます。これにより、非同期メソッドは、コレクションの各メンバーが利用可能になるまで待機します。

```
@Asynchronous
public void printList(@Wait List<Promise<String>> list) {
  for (Promise<String> s: list) {
```

```
        activityClient.printActivity(s);
    }
}
```

Promise オブジェクトを含むことを示すには、パラメータで @Wait 注釈を使用する必要があります。

このアクティビティ printActivity では、String 引数を使用しますが、生成されたクライアントの一致メソッドでは、Promise<String> が使用されます。クライアントでメソッドを呼び出します。アクティビティメソッドを直接呼び出すことはありません。

設定可能 <T>

Settable<*T*> は、Promise<T> から派生したもので、Promise の値を手動で設定できる設定メソッドです。たとえば、次のワークフローは、Settable<?> で待機して、シグナルを受け取るのを待ちます。このシグナルは、シグナルメソッドで設定されます。

```
public class MyWorkflowImpl implements MyWorkflow{
   final Settable<String> result = new Settable<String>();

   //@Execute method
   @Override
   public Promise<String> start() {
      return done(result);
   }

   //Signal
   @Override
   public void manualProcessCompletedSignal(String data) {
      result.set(data);
   }

   @Asynchronous
   public Promise<String> done(Settable<String> result){
       return result;
   }
}
```

Settable<?> を一度に別の Promise に連鎖することもできます。Promise をグループ化するには、AndPromise および OrPromise を使用します。連鎖された Settable を解除するには、unchain() を呼び出します。連鎖されている場合は、連鎖されている Promise が準備状態にな

ると、Settable<?> も自動的に準備状態になります。連鎖は、プログラムの他の部分の doTry() の範囲内から返った Promise を使用する場合に特に便利です。TryCatchFinally はネストされたクラスとして使用されるため、親のスコープPromise<>で を宣言して で設定することはできませんdoTry()。これは、Java では、変数が親スコープで宣言され、ネステッドクラスで final とマークされる必要があるためです。例:

```
@Asynchronous
public Promise<String> chain(final Promise<String> input) {
    final Settable<String> result = new Settable<String>();

    new TryFinally() {

        @Override
        protected void doTry() throws Throwable {
            Promise<String> resultToChain = activity1(input);
            activity2(resultToChain);

            // Chain the promise to Settable
            result.chain(resultToChain);
        }

        @Override
        protected void doFinally() throws Throwable {
            if (result.isReady()) { // Was a result returned before the exception?
                // Do cleanup here
            }
        }
    };

    return result;
}
```

Settable を一度に 1 つの Promise に連鎖できます。連鎖された Settable を解除するには、unchain() を呼び出します。

@NoWait

Promise を非同期メソッドに渡すと、デフォルトでは、フレームワークは、メソッドを実行する前に Promise が準備状態になるまで待機します (コレクションタイプの場合を除く)。この動作を上書きするには、非同期メソッドの宣言のパラメータに @NoWait 注釈を使用します。この方法は、Settable<T> で渡す場合に便利です。これにより、非同期メソッドが再帰的に設定されます。

プロミス <Void>

非同期メソッドの依存関係は、他のメソッドに渡す引数として、メソッドで返る Promise を渡して実装されます。ただし、メソッドから void を返しても、完了後に他の非同期メソッドが実行される場合があります。このような場合は、Promise<Void> をメソッドの戻り値として使用できます。Promise クラスでは、Promise<Void> オブジェクトの作成に使用する静的メソッド Void が指定されます。この Promise により、非同期メソッドで実行されると準備状態になります。この Promise は、他の Promise オブジェクトのように他の非同期メソッドに渡すことができます。Settable<Void> を使用している場合は、準備するために null で set メソッドを呼び出します。

AndPromise と OrPromise

AndPromise と OrPromise では、複数の Promise<> オブジェクトを論理的な 1 つの Promise にグループ化できます。AndPromise は、構築に使用する Promise がすべて準備できると、準備状態になります。OrPromise は、構築するために使用する Promise のコレクション内の Promise が準備できたら、準備状態になります。構成要素の Promise の値のリストを取得するには、AndPromise および OrPromise で getValues() を呼び出すことができます。

テストの容易性と依存関係の挿入

トピック

フレームワークは、制御の反転 (IoC) をフレンドリーに実現する設計になっています。アクティビティ/ワークフロー実装とフレームワーク提供のワーカー/コンテキストオブジェクトは、Spring などのコンテナを使用して設定しインスタンス化できます。追加設定なしで、フレームワークでは Spring フレームワークとの統合を利用できます。さらに、JUnit との統合を利用してワークフロー実装とアクティビティ実装の単体テストを行うこともできます。

Spring との統合

com.amazonaws.services.simpleworkflow.flow.spring パッケージに含まれているクラスを使うと、Spring フレームワークをアプリケーションで簡単に使用できます。たとえば、カスタムスコープ、Spring 対応のアクティビティワーカーとワークフローワーカーとして WorkflowScope、SpringWorkflowWorker、SpringActivityWorker が含まれています。こ

れらのクラスでは、Spring を通じてワークフロー/アクティビティ実装、ワークフロー/アクティビティワーカー全体を設定できます。

WorkflowScope

WorkflowScope は、フレームワークが提供するカスタム Spring スコープです。このスコープで Spring コンテナに作成されるオブジェクトの有効期間は、決定タスクの有効期間にスコープ指定されます。このスコープの Bean は、新しい決定タスクがワーカーで受信されるたびにインスタンス化されます。このスコープをワークフロー実装の Bean とそれが依存する他のすべての Bean に使用する必要があります。ワークフロー実装の Bean には、Spring 提供のシングルトンスコープとプロトタイプスコープを使用しません。これらを使用すると、決定タスクごとに新しい Bean を作成することをフレームワークから要求されます。作成できないと、予期しない動作が発生します。

次の例で示す Spring 設定のスニペットでは、WorkflowScope を登録し、これを使用してワークフロー実装の Bean とアクティビティクライアントの Bean を設定します。

```
<!-- register AWS Flow Framework for Java WorkflowScope -->
   <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
      <property name="scopes">
       <map>
         <entry key="workflow">
          <bean
 class="com.amazonaws.services.simpleworkflow.flow.spring.WorkflowScope" />
         </entry>
       </map>
      </property>
   </bean>

   <!-- activities client -->
   <bean id="activitiesClient" class="aws.flow.sample.MyActivitiesClientImpl"
 scope="workflow">
   </bean>

   <!-- workflow implementation -->
   <bean id="workflowImpl" class="aws.flow.sample.MyWorkflowImpl" scope="workflow">
      <property name="client" ref="activitiesClient"/>
      <aop:scoped-proxy proxy-target-class="false" />
   </bean>
```

workflowImpl Bean の設定で使用されている設定行 <aop:scoped-proxy proxy-target-class="false" /> は必須です。WorkflowScope では、CGLIB を使用したプロキシがサ

ポートされないためです。この設定を、別のスコープの別の Bean にワイヤリングされている WorkflowScope のすべての Bean に使用する必要があります。この場合、workflowImpl Bean をシングルトンスコープのワークフローワーカー Bean にワイヤリングする必要があります (以下の詳細な例を参照)。

カスタムスコープの詳細については、Spring フレームワークのドキュメントを参照してください。

Spring 対応のワーカー

Spring を使用する場合は、フレームワークが提供する Spring 対応のワーカークラス (SpringWorkflowWorker と SpringActivityWorker) を使用してください。これらのワーカーは、次の例に示すように、Spring を使用してアプリケーションに挿入できます。Spring 対応のワーカーは、Spring の SmartLifecycle インターフェイスを実装し、Spring コンテキストが初期化されると、デフォルトでタスクのポーリングを自動的に開始します。この機能を無効にするには、ワーカーの disableAutoStartup プロパティを true に設定します。

次の例は、ディサイダーの設定方法を示しています。この例では、インターフェイスとして MyActivities と MyWorkflow (非表示)、対応する実装として MyActivitiesImpl と MyWorkflowImpl を使用しています。生成されるクライアントインターフェイスと実装は、MyWorkflowClient/MyWorkflowClientImpl と MyActivitiesClient/MyActivitiesClientImpl (非表示) です。

アクティビティクライアントは、Spring の自動ワイヤリング機能を使用してワークフロー実装に挿入されます。

```
public class MyWorkflowImpl implements MyWorkflow {
   @Autowired
   public MyActivitiesClient client;

   @Override
   public void start() {
      client.activity1();
   }
}
```

ディサイダーの Spring 設定は以下のとおりです。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
```

```
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/
spring-aop-2.5.xsd
   http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <!-- register custom workflow scope -->
   <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
      <property name="scopes">
       <map>
         <entry key="workflow">
          <bean
 class="com.amazonaws.services.simpleworkflow.flow.spring.WorkflowScope" />
         </entry>
       </map>
      </property>
   </bean>
   <context:annotation-config/>

   <bean id="accesskeys" class="com.amazonaws.auth.BasicAWSCredentials">
      <constructor-arg value="{AWS.Access.ID}"/>
      <constructor-arg value="{AWS.Secret.Key}"/>
   </bean>

   <bean id="clientConfiguration" class="com.amazonaws.ClientConfiguration">
      <property name="socketTimeout" value="70000" />
   </bean>

   <!-- Amazon SWF client -->
   <bean id="swfClient"
      class="com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient">
      <constructor-arg ref="accesskeys" />
      <constructor-arg ref="clientConfiguration" />
      <property name="endpoint" value="{service.url}" />
   </bean>

   <!-- activities client -->
   <bean id="activitiesClient" class="aws.flow.sample.MyActivitiesClientImpl"
 scope="workflow">
   </bean>

   <!-- workflow implementation -->
```

```
   <bean id="workflowImpl" class="aws.flow.sample.MyWorkflowImpl" scope="workflow">
      <property name="client" ref="activitiesClient"/>
      <aop:scoped-proxy proxy-target-class="false" />
   </bean>

   <!-- workflow worker -->
   <bean id="workflowWorker"
      class="com.amazonaws.services.simpleworkflow.flow.spring.SpringWorkflowWorker">
      <constructor-arg ref="swfClient" />
      <constructor-arg value="domain1" />
      <constructor-arg value="tasklist1" />
      <property name="registerDomain" value="true" />
      <property name="domainRetentionPeriodInDays" value="1" />
      <property name="workflowImplementations">
         <list>
            <ref bean="workflowImpl" />
         </list>
      </property>
   </bean>
</beans>
```

SpringWorkflowWorker は Spring で完全に設定され、Spring コンテキストが初期化されると自動的にポーリングを開始するため、ディサイダーのホストプロセスはシンプルです。

```
public class WorkflowHost {
   public static void main(String[] args){
      ApplicationContext context
          = new FileSystemXmlApplicationContext("resources/spring/
WorkflowHostBean.xml");
      System.out.println("Workflow worker started");
   }
}
```

同様に、アクティビティワーカーは以下のように設定できます。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans.xsd
```

```
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/
spring-aop-2.5.xsd
   http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <!-- register custom scope -->
   <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
      <property name="scopes">
         <map>
            <entry key="workflow">
               <bean

 class="com.amazonaws.services.simpleworkflow.flow.spring.WorkflowScope" />
            </entry>
         </map>
      </property>
   </bean>

   <bean id="accesskeys" class="com.amazonaws.auth.BasicAWSCredentials">
      <constructor-arg value="{AWS.Access.ID}"/>
      <constructor-arg value="{AWS.Secret.Key}"/>
   </bean>

   <bean id="clientConfiguration" class="com.amazonaws.ClientConfiguration">
      <property name="socketTimeout" value="70000" />
   </bean>

   <!-- Amazon SWF client -->
   <bean id="swfClient"
      class="com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient">
      <constructor-arg ref="accesskeys" />
      <constructor-arg ref="clientConfiguration" />
      <property name="endpoint" value="{service.url}" />
   </bean>

   <!-- activities impl -->
   <bean name="activitiesImpl" class="asadj.spring.test.MyActivitiesImpl">
   </bean>

   <!-- activity worker -->
   <bean id="activityWorker"
      class="com.amazonaws.services.simpleworkflow.flow.spring.SpringActivityWorker">
      <constructor-arg ref="swfClient" />
      <constructor-arg value="domain1" />
```

```
      <constructor-arg value="tasklist1" />
      <property name="registerDomain" value="true" />
      <property name="domainRetentionPeriodInDays" value="1" />
      <property name="activitiesImplementations">
         <list>
            <ref bean="activitiesImpl" />
         </list>
      </property>
   </bean>
</beans>
```

アクティビティワーカーのホストプロセスはディサイダーに似ています。

```
public class ActivityHost {
   public static void main(String[] args) {
      ApplicationContext context = new FileSystemXmlApplicationContext(
      "resources/spring/ActivityHostBean.xml");
      System.out.println("Activity worker started");
   }
}
```

決定コンテキストの挿入

ワークフロー実装がコンテキストオブジェクトに依存する場合、これらも Spring を通じて簡単に挿入できます。フレームワークでは、コンテキスト関連の Bean を Spring コンテナに自動的に登録します。たとえば、次のスニペットでは、さまざまなコンテキストオブジェクトが自動ワイヤリングされています。コンテキストオブジェクトの他の Spring 設定は不要です。

```
public class MyWorkflowImpl implements MyWorkflow {
   @Autowired
   public MyActivitiesClient client;
   @Autowired
   public WorkflowClock clock;
   @Autowired
   public DecisionContext dcContext;
   @Autowired
   public GenericActivityClient activityClient;
   @Autowired
   public GenericWorkflowClient workflowClient;
   @Autowired
   public WorkflowContext wfContext;
   @Override
```

```
    public void start() {
       client.activity1();
    }
}
```

Spring XML 設定を通じてワークフロー実装でコンテキストオブジェクトを設定する場合は、com.amazonaws.services.simpleworkflow.flow.spring パッケージの `WorkflowScopeBeanNames` クラスに宣言されている Bean 名を使用します。例:

```
<!-- workflow implementation -->
<bean id="workflowImpl" class="asadj.spring.test.MyWorkflowImpl" scope="workflow">
   <property name="client" ref="activitiesClient"/>
   <property name="clock" ref="workflowClock"/>
   <property name="activityClient" ref="genericActivityClient"/>
   <property name="dcContext" ref="decisionContext"/>
   <property name="workflowClient" ref="genericWorkflowClient"/>
   <property name="wfContext" ref="workflowContext"/>
   <aop:scoped-proxy proxy-target-class="false" />
</bean>
```

または、`DecisionContextProvider` をワークフロー実装の Bean に挿入し、これを使用してコンテキストを作成することもできます。これは、プロバイダーとコンテキストのカスタム実装を提供する場合に便利です。

アクティビティへのリソースの挿入

制御の反転 (IoC) コンテナを使用してアクティビティ実装をインスタンス化して設定し、データベース接続などのリソースを簡単に挿入できます。そのためには、アクティビティ実装クラスのプロパティとしてリソースを宣言します。通常、このようなリソースはシングルトンとしてスコープ指定されます。アクティビティ実装は、複数のスレッドでアクティビティワーカーから呼び出されることに注意してください。そのため、共有リソースへのアクセスは同期する必要があります。

JUnit との統合

フレームワークでは、JUnit 拡張とコンテキストオブジェクトのテスト実装 (テストクロックなど) を提供します。これらを使用して JUnit で単体テストを記述して実行できます。これらの拡張を使用して、ワークフロー実装のインラインテストをローカルで実行できます。

シンプルな単体テストの記述

ワークフローのテストを記述するには、com.amazonaws.services.simpleworkflow.flow.junit パッケージの WorkflowTest クラスを使用します。このクラスは、フレームワーク固有の JUnit MethodRule 実装であり、ワークフローコードをローカルで実行し、Amazon SWF を経ることなく、アクティビティをインラインで呼び出します。これにより、料金を発生させることなく、テストを必要なだけ何回でも実行できます。

このクラスを使用するには、WorkflowTest 型のフィールドを宣言し、これに @Rule 注釈を設定します。テストを実行する前に、新しい WorkflowTest オブジェクトを作成し、これにアクティビティ実装とワークフロー実装を追加します。次に、生成されたワークフロークライアントファクトリを使用してクライアントを作成し、ワークフローの実行を開始できます。フレームワークに用意されているカスタム JUnit ランナーの FlowBlockJUnit4ClassRunner もワークフローテストに使用する必要があります。例:

```
@RunWith(FlowBlockJUnit4ClassRunner.class)
public class BookingWorkflowTest {

    @Rule
    public WorkflowTest workflowTest = new WorkflowTest();

    List<String> trace;

    private BookingWorkflowClientFactory workflowFactory
         = new BookingWorkflowClientFactoryImpl();

    @Before
    public void setUp() throws Exception {
        trace = new ArrayList<String>();
        // Register activity implementation to be used during test run
        BookingActivities activities = new BookingActivitiesImpl(trace);
        workflowTest.addActivitiesImplementation(activities);
        workflowTest.addWorkflowImplementationType(BookingWorkflowImpl.class);
    }

    @After
    public void tearDown() throws Exception {
        trace = null;
    }

    @Test
    public void testReserveBoth() {
```

```
        BookingWorkflowClient workflow = workflowFactory.getClient();
        Promise<Void> booked = workflow.makeBooking(123, 345, true, true);
        List<String> expected = new ArrayList<String>();
        expected.add("reserveCar-123");
        expected.add("reserveAirline-123");
        expected.add("sendConfirmation-345");
        AsyncAssert.assertEqualsWaitFor("invalid booking", expected, trace, booked);
    }
}
```

また、WorkflowTest に追加するアクティビティ実装ごとに個別のタスクリストを指定することもできます。たとえば、ワークフロー実装でアクティビティをホスト固有のタスクリストにスケジュールする場合、アクティビティを各ホストのタスクリストに登録できます。

```
for (int i = 0; i < 10; i++) {
    String hostname = "host" + i;
    workflowTest.addActivitiesImplementation(hostname,
                                             new ImageProcessingActivities(hostname));
}
```

@Test のコードは非同期であることに注意してください。したがって、実行を開始するには非同期ワークフロークライアントを使用する必要があります。テストの結果を検証するために、AsyncAssert ヘルプクラスも用意されています。このクラスでは、Promise が準備完了になるまで待った上で、結果を検証できます。この例では、ワークフロー実行の結果が準備完了になるまで待った上で、テストの出力を検証します。

Spring を使用している場合は、WorkflowTest クラスの代わりに SpringWorkflowTest クラスを使用できます。SpringWorkflowTest では、Spring 構成を介してアクティビティとワークフローを簡単に実装できるプロパティを提供します。Spring 対応のワーカーと同様に、WorkflowScope を使用してワークフロー実装の Bean を設定する必要があります。これにより、決定タスクごとに新しいワークフロー実装 Bean が作成されます。これらの Bean を設定する場合は、scoped-proxy proxy-target-class 設定を必ず false に設定してください。詳細については、「Spring Integration」(Spring との統合) セクションを参照してください。「Spring との統合」セクションに示されている Spring 設定例を変更し、SpringWorkflowTest を使用してワークフローをテストできます。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://
www.springframework.org/schema/aop"
  xmlns:context="http://www.springframework.org/schema/context"
```

```
  xsi:schemaLocation="http://www.springframework.org/schema/beans ht
tp://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop http://www.springframe
work.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <!-- register custom workflow scope -->
  <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
    <property name="scopes">
      <map>
        <entry key="workflow">
          <bean
            class="com.amazonaws.services.simpleworkflow.flow.spring.WorkflowScope" />
        </entry>
      </map>
    </property>
  </bean>
  <context:annotation-config />
  <bean id="accesskeys" class="com.amazonaws.auth.BasicAWSCredentials">
    <constructor-arg value="{AWS.Access.ID}" />
    <constructor-arg value="{AWS.Secret.Key}" />
  </bean>
  <bean id="clientConfiguration" class="com.amazonaws.ClientConfiguration">
    <property name="socketTimeout" value="70000" />
  </bean>

  <!-- Amazon SWF client -->
  <bean id="swfClient"
    class="com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClient">
    <constructor-arg ref="accesskeys" />
    <constructor-arg ref="clientConfiguration" />
    <property name="endpoint" value="{service.url}" />
  </bean>

  <!-- activities client -->
  <bean id="activitiesClient" class="aws.flow.sample.MyActivitiesClientImpl"
    scope="workflow">
  </bean>

  <!-- workflow implementation -->
  <bean id="workflowImpl" class="aws.flow.sample.MyWorkflowImpl"
    scope="workflow">
    <property name="client" ref="activitiesClient" />
```

```
    <aop:scoped-proxy proxy-target-class="false" />
  </bean>

  <!-- WorkflowTest -->
  <bean id="workflowTest"
    class="com.amazonaws.services.simpleworkflow.flow.junit.spring.SpringWorkflowTest">
    <property name="workflowImplementations">
      <list>
        <ref bean="workflowImpl" />
      </list>
    </property>
    <property name="taskListActivitiesImplementationMap">
      <map>
        <entry>
          <key>
            <value>list1</value>
          </key>
          <ref bean="activitiesImplHost1" />
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

アクティビティ実装のモッキング

テストでは実際のアクティビティ実装を使用できますが、ワークフローロジックの単体テストのみを行う場合は、モックアクティビティを使用してください。これを行うには、アクティビティインターフェイスのモック実装を WorkflowTest クラスに提供します。例:

```
@RunWith(FlowBlockJUnit4ClassRunner.class)
public class BookingWorkflowTest {

    @Rule
    public WorkflowTest workflowTest = new WorkflowTest();

    List<String> trace;

    private BookingWorkflowClientFactory workflowFactory
         = new BookingWorkflowClientFactoryImpl();

    @Before
    public void setUp() throws Exception {
```

```
        trace = new ArrayList<String>();
        // Create and register mock activity implementation to be used during test run
        BookingActivities activities = new BookingActivities() {

            @Override
            public void sendConfirmationActivity(int customerId) {
                trace.add("sendConfirmation-" + customerId);
            }

            @Override
            public void reserveCar(int requestId) {
                trace.add("reserveCar-" + requestId);
            }

            @Override
            public void reserveAirline(int requestId) {
                trace.add("reserveAirline-" + requestId);
            }
        };
        workflowTest.addActivitiesImplementation(activities);
        workflowTest.addWorkflowImplementationType(BookingWorkflowImpl.class);
    }

    @After
    public void tearDown() throws Exception {
        trace = null;
    }

    @Test
    public void testReserveBoth() {
        BookingWorkflowClient workflow = workflowFactory.getClient();
        Promise<Void> booked = workflow.makeBooking(123, 345, true, true);
        List<String> expected = new ArrayList<String>();
        expected.add("reserveCar-123");
        expected.add("reserveAirline-123");
        expected.add("sendConfirmation-345");
        AsyncAssert.assertEqualsWaitFor("invalid booking", expected, trace, booked);
    }
}
```

または、アクティビティクライアントのモック実装を提供し、これをワークフロー実装に挿入することもできます。

テストコンテキストオブジェクト

ワークフロー実装がフレームワークのコンテキストオブジェクト (DecisionContext など) に依存している場合、このようなワークフローをテストするには特に何も行う必要がありません。WorkflowTest を通じてテストを実行すると、テストコンテキストオブジェクトが自動的に挿入されます。ワークフロー実装がコンテキストオブジェクト (DecisionContextProviderImpl を使用するなど) にアクセスすると、テスト実装が取得されます。これらのテストコンテキストオブジェクトをテストコード (@Test メソッド) で操作して有益なテストケースを作成できます。例えば、ワークフローでタイマーを作成する場合、WorkflowTest クラスで clockAdvanceSeconds メソッドを呼び出すことでタイマーを始動し、クロックの時間を進めることができます。また、WorkflowTest の ClockAccelerationCoefficient プロパティを使用してクロックを加速させ、通常よりも早くタイマーを始動することもできます。たとえば、ワークフローで 1 時間のタイマーを作成する場合、ClockAccelerationCoefficient を 60 に設定してタイマーを 1 分で始動できます。ClockAccelerationCoefficient は、デフォルトで「1」に設定されます。

com.amazonaws.services.simpleworkflow.flow.test パッケージと com.amazonaws.services.simpleworkflow.flow.junit パッケージの詳細については、 AWS SDK for Java のドキュメントを参照してください。

エラー処理

トピック

- TryCatchFinally のセマンティクス
- キャンセル
- ネストされた TryCatchFinally

Java の try/catch/finally コンストラクトは、エラー処理の簡単な方法として広く利用されています。このコンストラクトでは、エラー処理をコードのブロックと関連付けることができます。内部的には、エラー処理に関する追加のメタデータがコールスタックに追加されます。例外がスローされると、ランタイムでは、関連付けられたエラー処理をコールスタックで見つけて呼び出します。適切なエラー処理が見つからない場合は、コールチェーンの上に例外を伝播させます。

この方法は、同期コードには適していますが、非同期プログラムや分散プログラムでのエラー処理には他の問題が伴います。非同期呼び出しはすぐに戻るため、非同期コードの実行時に呼び出し元は呼び出しスタックにありません。つまり、非同期コードの未処理の例外は、呼び出し元が通常の方法で処理できません。一般的に、非同期コードでの例外を処理するには、エラー状態をコールバックに渡

し、それを非同期メソッドに渡します。または、Future<?> を使用している場合は、これにアクセスしようとするとエラーが報告されます。これは理想的な状態ではありません。例外を受け取るコード (Future<?> を使用するコードまたはコールバック) は元の呼び出しのコンテキストを参照できず、例外を適切に処理できない可能性があるためです。さらに、分散非同期システムでは、複数のコンポーネントが同時に実行されるため、複数のエラーが同時に発生する可能性があります。これらのエラーは、タイプと深刻度が異なる可能性があり、適切な処理が必要になります。

非同期呼び出し後のリソースのクリーンアップも、簡単ではありません。同期コードとは異なり、呼び出しコードで try/catch/finally を使用してリソースをクリーンアップすることはできません。これは、最後のブロックの実行時に try ブロックで開始された作業がまだ進行中の可能性があるためです。

フレームワークが提供する機構では、分散非同期コードでのエラー処理が Java の try/catch/finally と類似しており、同じようにシンプルです。

```
ImageProcessingActivitiesClient activitiesClient
     = new ImageProcessingActivitiesClientImpl();

public void createThumbnail(final String webPageUrl) {

  new TryCatchFinally() {

    @Override
    protected void doTry() throws Throwable {
      List<String> images = getImageUrls(webPageUrl);
      for (String image: images) {
        Promise<String> localImage
            = activitiesClient.downloadImage(image);
        Promise<String> thumbnailFile
            = activitiesClient.createThumbnail(localImage);
        activitiesClient.uploadImage(thumbnailFile);
      }
    }

    @Override
    protected void doCatch(Throwable e) throws Throwable {

      // Handle exception and rethrow failures
      LoggingActivitiesClient logClient = new LoggingActivitiesClientImpl();
      logClient.reportError(e);
      throw new RuntimeException("Failed to process images", e);
    }
```

```
    @Override
    protected void doFinally() throws Throwable {
      activitiesClient.cleanUp();
    }
  };
}
```

TryCatchFinally クラスとそのバリアント (TryFinally と TryCatch) は、Java の try/catch/finally と同じように動作します。これを使用して、例外ハンドラをワークフローコードのブロックと関連付け、非同期のリモートタスクとして実行できます。doTry() メソッドは、論理的に try ブロックと同じです。フレームワークでは、doTry() のコードを自動的に実行します。Promise オブジェクトのリストを TryCatchFinally のコンストラクタに渡すことができます。コンストラクタに渡したすべての Promise オブジェクトが準備完了状態になると、doTry メソッドが実行されます。doTry() 内から非同期的に呼び出されたコードによって例外がスローされると、doTry() で保留中の仕事がすべてキャンセルされ、doCatch() が呼び出されて例外が処理されます。たとえば、上のリスティングで、downloadImage から例外がスローされると、createThumbnail と uploadImage がキャンセルされます。最後に、すべての非同期の仕事が終了する (完了、失敗、またはキャンセルになる) と、doFinally() が呼び出されます。これは、リソースのクリーンアップに使用できます。必要に応じて、これらのクラスをネストすることもできます。

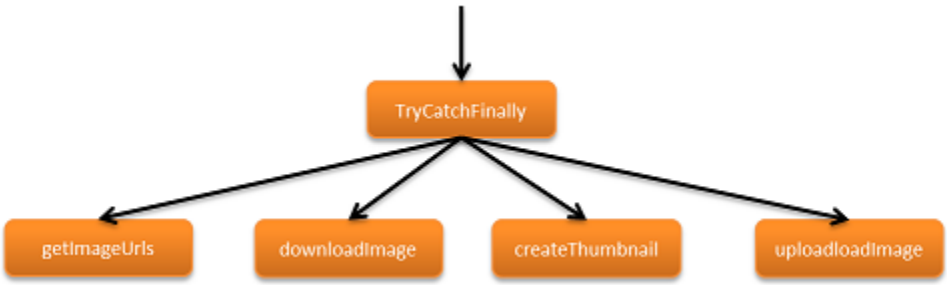
例外が doCatch() で報告されると、フレームワークでは、非同期呼び出しやリモート呼び出しを含む完全な論理コールスタックを提供します。これはデバッグするときに便利です。特に、非同期メソッドから他の非同期メソッドを呼び出している場合に役立ちます。たとえば、downloadImage からは、次のような例外が生成されます。

```
RuntimeException: error downloading image
  at downloadImage(Main.java:35)
  at ---continuation---.(repeated:1)
  at errorHandlingAsync$1.doTry(Main.java:24)
  at ---continuation---.(repeated:1)
…
```

TryCatchFinally のセマンティクス

AWS Flow Framework for Java プログラムの実行は、ブランチを同時に実行するツリーとして視覚化できます。非同期メソッド、アクティビティ、および TryCatchFinally 自体への呼び出しによ

り、この実行ツリーに新しいブランチが作成されます。たとえば、イメージ処理ワークフローは、次の図に示すツリーとして表示できます。



特定の実行ブランチでのエラーに伴って、そのブランチのアンワインドが生じます。例外に伴って、Java プログラムのコールスタックのアンワインドが生じるのと同じです。アンワインドに伴って、エラーが処理されるか、ツリーのルートに達してワークフロー実行が終了するまで、実行ブランチが上に移動し続けます。

フレームワークでは、タスクの処理中に発生したエラーを例外として報告します。`TryCatchFinally` で定義した例外ハンドラ (`doCatch()` メソッド) は、対応する `doTry()` のコードで作成されたすべてのタスクに関連付けられます。タスクが (タイムアウトや未処理の例外など) 失敗すると、該当する例外がスローされ、これを処理するために対応する `doCatch()` が呼び出されます。これを達成するために、フレームワークでは Amazon SWF と連携してリモートエラーを伝播し、呼び出し元のコンテキストで例外を再現します。

キャンセル

同期コードで例外が発生すると、制御は `catch` ブロックに直接ジャンプし、`try` ブロックの残存コードをスキップします。例:

```
try {
    a();
    b();
    c();
}
catch (Exception e) {
    e.printStackTrace();
}
```

このコードでは、`b()` で例外がスローされると、`c()` はまったく呼び出されません。これを次のワークフローと比較します。

```
new TryCatch() {
```

```
    @Override
    protected void doTry() throws Throwable {
        activityA();
        activityB();
        activityC();
    }

    @Override
    protected void doCatch(Throwable e) throws Throwable {
        e.printStackTrace();
    }
};
```

この場合、activityA、activityB、および activityC に対する呼び出しのすべてが正常に戻り、3 つのタスクが作成されて非同期に実行されます。後で、activityB のタスクがエラーになったとします。このエラーは Amazon SWF によって履歴に記録されます。これを処理するために、フレームワークでは、まず同じ doTry( ) のスコープに属する他のすべてのタスク (この例では activityA と activityC) をキャンセルしようとします。すべての該当するタスクが完了 (キャンセル、失敗、または正常に終了) すると、このエラーを処理するために適切な doCatch( ) メソッドが呼び出されます。

同期の例 (c( ) がまったく実行されない) とは異なり、activityC が呼び出されてタスクの実行がスケジュールされています。したがって、フレームワークではこれをキャンセルしようとします。ただし、キャンセルされる保証はありません。キャンセルが保証されない理由としては、アクティビティが完了済みであるか、キャンセルリクエストが無視されるか、エラーで失敗することが考えられます。ただし、フレームワークでは、対応する doTry( ) で開始されたすべてのタスクが完了した後でのみ doCatch( ) を呼び出すことを保証します。また、doTry( ) と doCatch( ) で開始されたすべてのタスクが完了した後でのみ doFinally( ) を呼び出すことを保証します。例えば、上の例でアクティビティが相互に依存している場合 (activityB が activityA に依存し、activityC が activityB に依存している場合など)、activityC のキャンセルは activityB が完了するまで Amazon SWF でスケジュールされないため、すぐに行われます。

```
new TryCatch() {

    @Override
    protected void doTry() throws Throwable {
        Promise<Void> a = activityA();
        Promise<Void> b = activityB(a);
        activityC(b);
```

```
    }

    @Override
    protected void doCatch(Throwable e) throws Throwable {
        e.printStackTrace();
    }
};
```

アクティビティのハートビート

AWS Flow Framework for Java の協調キャンセルメカニズムにより、実行中のアクティビティタスクを適切にキャンセルできます。キャンセルがトリガーされると、ワーカーをブロックしたタスクや、ワーカーへの割り当てを待機しているタスクは自動的にキャンセルされます。ただし、タスクがワーカーに割り当て済みである場合、フレームワークはアクティビティにキャンセルをリクエストします。アクティビティ実装では、このようなキャンセルリクエストを明示的に処理する必要があります。これを行うには、アクティビティのハートビートを報告します。

ハートビートを報告することで、アクティビティ実装では、進行中のアクティビティタスクの進行状況を報告できます。これはモニタリングに役立ちます。また、アクティビティでは、キャンセルリクエストを確認できます。recordActivityHeartbeat メソッドは、キャンセルがリクエストされると、CancellationException をスローします。アクティビティ実装では、この例外をキャッチし、キャンセルリクエストに対応します。または、キャンセルリクエストを無視して、例外を ”飲み込み” ます。キャンセルリクエストに対応する場合、アクティビティでは、必要に応じて目的のクリーンアップを実施し、CancellationException を再スローします。この例外がアクティビティ実装からスローされると、アクティビティタスクはキャンセル済みで完了したものとしてフレームワークに記録されます。

次の例は、イメージをダウンロードして処理するアクティビティを示しています。各イメージの処理後にハートビートを送信します。キャンセルがリクエストされると、クリーンアップを行い、例外を再スローしてキャンセルを確認します。

```
@Override
public void processImages(List<String> urls) {
    int imageCounter = 0;
    for (String url: urls) {
        imageCounter++;
        Image image = download(url);
        process(image);
        try {
            ActivityExecutionContext context
                 = contextProvider.getActivityExecutionContext();
```

```
            context.recordActivityHeartbeat(Integer.toString(imageCounter));
        } catch(CancellationException ex) {
            cleanDownloadFolder();
            throw ex;
        }
    }
}
```

アクティビティのハートビートを報告することは必須ではありませんが、アクティビティの実行時間が長引いている場合や、高価な操作の実行中にエラーが発生したときにキャンセルする場合には推奨されます。アクティビティ実装から heartbeatActivityTask を定期的に呼び出す必要があります。

アクティビティがタイムアウトすると、ActivityTaskTimedOutException がスローされ、例外オブジェクトの getDetails からデータが返されます。これは、対応するアクティビティタスクの heartbeatActivityTask に対する最後の正常な呼び出しに渡されたデータです。ワークフロー実装では、この情報に基づいてアクティビティタスクがタイムアウトするまでの進行状況を判断できます。

> **Note**
>
> ハートビートの送信が多すぎると、Amazon SWF でハートビートリクエストがスロットリングされるため、調整する必要があります。Amazon SWF の制限については、「Amazon Simple Workflow Service Developer Guide」(Amazon Simple Workflow Service デベロッパーガイド) を参照してください。

タスクの明示的なキャンセル

エラー条件とは別に、タスクを明示的にキャンセルする場合もあります。たとえば、クレジットカードで支払いを処理するアクティビティは、ユーザーが注文を取り消した場合に、キャンセルが必要になることがあります。フレームワークでは、TryCatchFinally のスコープで作成されたタスクを明示的にキャンセルできます。次の例では、支払いの処理中にシグナルを受信すると、支払いタスクがキャンセルされます。

```
public class OrderProcessorImpl implements OrderProcessor {
    private PaymentProcessorClientFactory factory
        = new PaymentProcessorClientFactoryImpl();
    boolean processingPayment = false;
    private TryCatchFinally paymentTask = null;
```

```
    @Override
    public void processOrder(int orderId, final float amount) {
        paymentTask = new TryCatchFinally() {

            @Override
            protected void doTry() throws Throwable {
                processingPayment = true;

                PaymentProcessorClient paymentClient = factory.getClient();
                paymentClient.processPayment(amount);
            }

            @Override
            protected void doCatch(Throwable e) throws Throwable {
                if (e instanceof CancellationException) {
                    paymentClient.log("Payment canceled.");
                } else {
                    throw e;
                }
            }

            @Override
            protected void doFinally() throws Throwable {
                processingPayment = false;
            }
        };

    }

    @Override
    public void cancelPayment() {
        if (processingPayment) {
            paymentTask.cancel(null);
        }
    }
}
```

キャンセル済みタスクの通知の受信

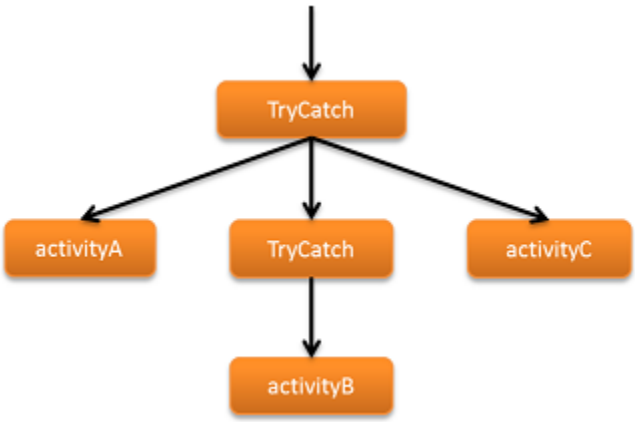
タスクがキャンセル済みとして完了すると、フレームワークでは、CancellationException をスローしてワークフローロジックに通知します。アクティビティがキャンセル済みとして完了すると、レコードが履歴に作成され、フレームワークでは CancellationException を使用して適切

な doCatch() を呼び出します。前の例で示したように、支払い処理タスクがキャンセルされ、ワークフローは CancellationException を受け取ります。

処理されない CancellationException は、他の例外と同じように、実行ブランチの上に伝播されます。ただし、doCatch() メソッドが CancellationException を受け取るのは、スコープに他の例外がない場合に限られます。他の例外があると、キャンセルより優先されます。

ネストされた TryCatchFinally

必要に応じて TryCatchFinally をネストできます。それぞれが実行ツリーに新しいブランチTryCatchFinallyを作成するため、ネストされたスコープを作成できます。親スコープで例外が発生すると、親スコープ内にネストされた TryCatchFinally で開始されたすべてのタスクに対してキャンセルが試行されます。ただし、ネストされた TryCatchFinally での例外は自動的に親に伝播されません。ネストされた TryCatchFinally の例外を親の TryCatchFinally に伝播する場合は、doCatch() で例外を再スローする必要があります。つまり、Java の try/catch と同じように、未処理の例外のみがバブルアップされます。キャンセルメソッドを呼び出してネストされた TryCatchFinally をキャンセルすると、ネストされた TryCatchFinally はキャンセルされますが、親の TryCatchFinally は自動的にキャンセルされません。




```
new TryCatch() {
    @Override
    protected void doTry() throws Throwable {
        activityA();

        new TryCatch() {
            @Override
            protected void doTry() throws Throwable {
                activityB();
            }
```

```
            @Override
            protected void doCatch(Throwable e) throws Throwable {
                reportError(e);
            }
        };

        activityC();
    }

    @Override
    protected void doCatch(Throwable e) throws Throwable {
        reportError(e);
    }
};
```

失敗したアクティビティを再試行する

一時的な接続の切断など、一過性の理由でアクティビティが失敗することがあります。アクティビティが成功するかどうかは場合によって異なるため、アクティビティが失敗した場合は、そのアクティビティを何度かやり直します。

アクティビティを再試行する戦略はさまざまです。最適な方法は、ワークフローの詳細によって異なります。この戦略は、3 つの基本カテゴリーに分類されます。

- 成功するまで再試行する戦略では、完了するまでアクティビティを再試行します。
- 指数的な再試行戦略では、アクティビティが完了するまで、またはプロセスが指定の停止ポイント (最大試行回数など) に達するまで、再試行の間隔を指数的に大きくします。
- カスタムの再試行戦略では、試行が失敗する度にアクティビティを再試行するかどうか、またはその方法を決定します。

次のセクションでは、このような戦略を実装する方法について説明します。ワークフローワーカーの例ではすべて、1 つのアクティビティ unreliableActivity を使用します。これにより、以下のいずれかがランダムに実行されます。

- ただちに完了する
- タイムアウト値の超過により、意図的に失敗する
- IllegalStateException をスローして、意図的に失敗する

成功するまで再試行する戦略

最もシンプルな再試行戦略は、最終的に成功するまで、失敗する度にアクティビティを再試行し続けることです。基本的なパターンは次のとおりです。

1. ワークフローのエントリポイントメソッドで、ネステッドクラス (TryCatch または TryCatchFinally) を実装する
2. doTry でアクティビティを実行する
3. アクティビティに失敗すると、フレームワークは doCatch を呼び出します。これにより、エントリポイントメソッドが再度実行されます。
4. アクティビティメソッドが正常に完了するまで、ステップ 2~3 を繰り返します。

次のワークフロー実装では、成功するまで再試行する戦略が実装されます。このワークフローインターフェイスは、RetryActivityRecipeWorkflow で実装されており、1 つのメソッド runUnreliableActivityTillSuccess が含まれます。これが、ワークフローのエンドポイントです。ワークフローワーカーは、次のように RetryActivityRecipeWorkflowImpl で実装されます。

```
public class RetryActivityRecipeWorkflowImpl
    implements RetryActivityRecipeWorkflow {

    @Override
    public void runUnreliableActivityTillSuccess() {
        final Settable<Boolean> retryActivity = new Settable<Boolean>();

        new TryCatch() {
            @Override
            protected void doTry() throws Throwable {
                Promise<Void> activityRanSuccessfully
                    = client.unreliableActivity();
                setRetryActivityToFalse(activityRanSuccessfully, retryActivity);
            }

            @Override
            protected void doCatch(Throwable e) throws Throwable {
                retryActivity.set(true);
            }
        };
        restartRunUnreliableActivityTillSuccess(retryActivity);
    }
```

```
    @Asynchronous
    private void setRetryActivityToFalse(
            Promise<Void> activityRanSuccessfully,
            @NoWait Settable<Boolean> retryActivity) {
        retryActivity.set(false);
    }

    @Asynchronous
    private void restartRunUnreliableActivityTillSuccess(
            Settable<Boolean> retryActivity) {
        if (retryActivity.get()) {
            runUnreliableActivityTillSuccess();
        }
    }
}
```

ワークフローの動作は次のとおりです。

1. runUnreliableActivityTillSuccess は、アクティビティが失敗し、再試行する必要があるかどうかを示すために使用される、retryActivity という名前の Settable<Boolean> オブジェクトを作成します。Settable<T> は Promise<T> から派生し、同じように動作しますが、Settable<T> オブジェクトの値は手動で設定します。
2. runUnreliableActivityTillSuccess は、匿名のネステッドクラス TryCatch を実装して、unreliableActivity アクティビティによってスローされるすべての例外に対処します。非同期コードでスローされる例外の対処方法の詳細については、「エラー処理」を参照してください。
3. doTry では、unreliableActivity アクティビティが実行され、activityRanSuccessfully という名前の Promise<Void> オブジェクトが返ります。
4. doTry は非同期メソッド setRetryActivityToFalse を呼び出します。このメソッドには、次の 2 つのパラメータが含まれます。
   - activityRanSuccessfully は、unreliableActivity アクティビティによって返る Promise<Void> オブジェクトを取得します。
   - retryActivity は、retryActivity オブジェクトを取得します。

   unreliableActivity が完了すると、activityRanSuccessfully は準備状態になり、setRetryActivityToFalse によって、retryActivity が FALSE に設定されます。それ以外の場合、activityRanSuccessfully は準備状態になり、setRetryActivityToFalse は実行されません。

5. unreliableActivity によって例外がスローされると、フレームワークは doCatch を呼び出し、例外オブジェクトを渡します。doCatch は retryActivity を true に設定します。
6. runUnreliableActivityTillSuccess は、非同期メソッド restartRunUnreliableActivityTillSuccess を呼び出し、retryActivity オブジェクトに渡します。retryActivity は Promise<T> タイプのため、restartRunUnreliableActivityTillSuccess は、retryActivity が準備状態になるまで実行を延期します。そのため、TryCatch が完了すると実行されます。
7. retryActivity が準備状態になると、restartRunUnreliableActivityTillSuccess によって値が抽出されます。
   - 値が false の場合、再試行は成功です。restartRunUnreliableActivityTillSuccess は問題ではないため、再試行シーケンスは終了します。
   - 値が true の場合、再試行は失敗です。アクティビティを再度実行するには、restartRunUnreliableActivityTillSuccess で runUnreliableActivityTillSuccess を呼び出します。
8. unreliableActivity が完了するまで、ステップ 1~7 を繰り返します。

> **Note**
>
> doCatch では、例外は処理されません。retryActivity オブジェクトは、アクティビティが失敗したことを示す true に設定されます。この再試行は、非同期メソッド restartRunUnreliableActivityTillSuccess で処理されます。これにより、TryCatch が完了するまで、実行は延期されます。このアプローチの理由は、doCatch でアクティビティを再試行する場合はキャンセルできないことです。restartRunUnreliableActivityTillSuccess でアクティビティを再試行すると、キャンセル可能なアクティビティを実行できます。

指数的再試行戦略

指数的再試行戦略を使用して、フレームワークは、指定された時間 (N 秒) が経過すると、失敗したアクティビティを実行します。再試行が失敗した場合、フレームワークは 2N 秒、4N 秒と経過すると、再度アクティビティを実行します。待機時間は大きくなる可能性があるため、通常は無期限に続けずに、再試行を一定のポイントで停止します。

フレームワークには、指数的再試行戦略を実装する 3 つの方法があります。

- @ExponentialRetry 注釈は、最もシンプルなアプローチですが、コンパイル時に再試行設定オプションを設定する必要があります。
- RetryDecorator クラスでは、実行時の再試行を設定し、必要に応じて変更することができます。
- AsyncRetryingExecutor クラスでは、実行時の再試行を設定し、必要に応じて変更することができます。また、フレームワークは、ユーザー実装の AsyncRunnable.run メソッドを呼び出して、ひとつずつ再試行を行います。

次の設定オプションはすべてのアプローチでサポートされており、時刻値は秒単位で計測されます。

- 最初の再試行待機時間。
- バックオフ係数。次のように、再試行間隔の計算に使用されます。

```
retryInterval = initialRetryIntervalSeconds * Math.pow(backoffCoefficient,
 numberOfTries - 2)
```

  デフォルト値は 2.0 です。
- 再試行の最大数。デフォルト値の制限はありません。
- 最大再試行間隔。デフォルト値の制限はありません。
- 有効期限。再試行は、プロセスの合計期間がこの値を超過すると停止します。デフォルト値の制限はありません。
- 再試行プロセスをトリガーする例外。デフォルトでは、例外ごとに再試行プロセスがトリガーされます。
- 再試行プロセスをトリガーしない例外。デフォルトでは、除外される例外はありません。

次のセクションでは、指数的再試行戦略を実装するさまざまな方法について説明します。

@ExponentialRetry を使用した指数的再試行

アクティビティの指数的再試行戦略を実装する最もシンプルな方法は、@ExponentialRetry 注釈をインターフェイスの定義のアクティビティに適用することです。アクティビティが失敗すると、フレームワークは、特定のオプション値に基づき、自動的に再試行プロセスに対処します。基本的なパターンは次のとおりです。

1. @ExponentialRetry を適切なアクティビティに適用し、再試行設定を指定します。

2. 注釈されたアクティビティが失敗した場合、フレームワークは、注釈の引数で指定された設定に従って、自動的にアクティビティを再試行します。

ExponentialRetryAnnotationWorkflow ワークフローワーカーは、@ExponentialRetry 注釈を使用して、再試行戦略を実装します。次のように、インターフェイス定義が ExponentialRetryAnnotationActivities に実装されている unreliableActivity アクティビティを使用します。

```
@Activities(version = "1.0")
@ActivityRegistrationOptions(
    defaultTaskScheduleToStartTimeoutSeconds = 30,
    defaultTaskStartToCloseTimeoutSeconds = 30)
public interface ExponentialRetryAnnotationActivities {
    @ExponentialRetry(
        initialRetryIntervalSeconds = 5,
        maximumAttempts = 5,
        exceptionsToRetry = IllegalStateException.class)
    public void unreliableActivity();
}
```

@ExponentialRetry オプションでは、以下の戦略を指定します。

- アクティビティで IllegalStateException がスローされる場合のみ、再試行します。
- 最初の待機時間として 5 秒を使用します。
- 6 回以上再試行することはできません。

このワークフローインターフェイスは、RetryWorkflow で実装されており、1 つのメソッド process が含まれます。これが、ワークフローのエンドポイントです。ワークフローワーカーは、次のように ExponentialRetryAnnotationWorkflowImpl で実装されます。

```
public class ExponentialRetryAnnotationWorkflowImpl implements RetryWorkflow {
    public void process() {
        handleUnreliableActivity();
    }

    public void handleUnreliableActivity() {
        client.unreliableActivity();
    }
}
```

ワークフローの動作は次のとおりです。

1. process では、非同期メソッド handleUnreliableActivity が実行されます。
2. handleUnreliableActivity は、unreliableActivity アクティビティを実行します。

IllegalStateException のスローによりアクティビティが失敗した場合、フレームワークは、ExponentialRetryAnnotationActivities で指定されている再試行戦略を自動的に実行します。

RetryDecorator クラスを使用した指数的再試行

@ExponentialRetry の使用は簡単です。ただし、設定は静的で、コンパイル時に設定されるため、フレームワークは、アクティビティが失敗する度に、同じ再試行戦略を使用します。より複雑な再試行戦略を実装するには、RetryDecorator クラスを使用します。これにより、実行時に設定を指定し、必要に応じて変更することができます。基本的なパターンは次のとおりです。

1. 再試行設定を指定する ExponentialRetryPolicy オブジェクトを作成し、設定します。
2. RetryDecorator オブジェクトを作成し、ステップ 1 の ExponentialRetryPolicy オブジェクトをコンストラクタに渡します。
3. アクティビティクライアントのクラス名を RetryDecorator オブジェクトのデコレートメソッドに渡し、デコレーターオブジェクトをアクティビティに適用します。
4. アクティビティを実行します。

アクティビティが失敗した場合、フレームワークは、ExponentialRetryPolicy オブジェクトの設定に従って、アクティビティを再試行します。必要に応じて、設定を変更して再試行するには、このオブジェクトを変更します。

> **Note**
>
> @ExponentialRetry 注釈および RetryDecorator クラスは、相互に排他的です。@ExponentialRetry 注釈で指定されている再試行ポリシーを動的に上書きするために RetryDecorator を使用することはできません。

以下のワークフロー実装では、指数的再試行戦略を実装するために RetryDecorator クラスの使用方法を示します。@ExponentialRetry 注釈を含まない unreliableActivity アクティビティ

を使用します。このワークフローインターフェイスは、RetryWorkflow で実装されており、1 つのメソッド process が含まれます。これが、ワークフローのエントポイントです。ワークフローワーカーは、次のように DecoratorRetryWorkflowImpl で実装されます。

```
public class DecoratorRetryWorkflowImpl implements RetryWorkflow {
   ...
  public void process() {
      long initialRetryIntervalSeconds = 5;
      int maximumAttempts = 5;
      ExponentialRetryPolicy retryPolicy = new ExponentialRetryPolicy(
              initialRetryIntervalSeconds).withMaximumAttempts(maximumAttempts);

      Decorator retryDecorator = new RetryDecorator(retryPolicy);
      client = retryDecorator.decorate(RetryActivitiesClient.class, client);
      handleUnreliableActivity();
  }

  public void handleUnreliableActivity() {
      client.unreliableActivity();
  }
}
```

ワークフローの動作は次のとおりです。

1. process は、次の方法で、ExponentialRetryPolicy オブジェクトを作成し設定します。
   - 最初の再試行間隔をコンストラクタに渡します。
   - オブジェクトの withMaximumAttempts メソッドの呼び出し試行回数の上限を 5 に設定します。ExponentialRetryPolicy は、他の設定オプションを指定するために使用できる他の with オブジェクトを公開します。
2. process は、retryDecorator という名前の RetryDecorator オブジェクトを作成し、ステップ 1 の ExponentialRetryPolicy オブジェクトをコンストラクタに渡します。
3. process は、retryDecorator.decorate メソッドを呼び出して、アクティビティクライアントのクラス名を渡すことで、デコレーターをアクティビティに適用します。
4. handleUnreliableActivity は、アクティビティを実行します。

アクティビティが失敗した場合、フレームワークは、ステップ 1 で指定した設定に従って、アクティビティを再試行します。

> **Note**
>
> ExponentialRetryPolicy クラスの with メソッドの中には、set メソッドが含まれる場合があります。このメソッドを呼び出して、次の設定オプションをいつでも変更することができます: setBackoffCoefficient、setMaximumAttempts、setMaximumRetryIntervalSeconds、se

AsyncRetryingExecutor クラスを使用した指数的再試行

RetryDecorator クラスを使用して、再試行プロセスを設定すると、@ExponentialRetry よりも柔軟性は高くなりますが、フレームワークは、ExponentialRetryPolicy オブジェクトの現在の設定に基づき、自動的に再試行を行います。AsyncRetryingExecutor クラスを使用するアプローチの方が柔軟性は高くなります。フレームワークは、実行時の再試行プロセスの設定を可能にするだけでなく、ユーザー実装の AsyncRunnable.run メソッドを呼び出して、アクティビティを単純に実行せずに、再試行ごとに実行します。

基本的なパターンは次のとおりです。

1. ExponentialRetryPolicy オブジェクトを作成、設定して、再試行設定を指定します。
2. AsyncRetryingExecutor オブジェクトを作成し、ExponentialRetryPolicy オブジェクトと、ワークフロークロックのインスタンスを渡します。
3. 匿名のネステッドクラス (TryCatch または TryCatchFinally) を実装します。
4. 匿名のクラス (AsyncRunnable) を実装して、run メソッドを上書きし、アクティビティの実行に使用するカスタムコードを実装します。
5. doTry を上書きして AsyncRetryingExecutor オブジェクトの execute メソッドを呼び出し、ステップ 4 の AsyncRunnable クラスを渡します。AsyncRetryingExecutor オブジェクトは、AsyncRunnable.run を呼び出してこのアクティビティを実行します。
6. アクティビティが失敗した場合、AsyncRetryingExecutor オブジェクトは、ステップ 1 の再試行ポリシーに従って、AsyncRunnable.run メソッドを再度呼び出します。

以下のワークフローでは、指数的再試行戦略を実装するために AsyncRetryingExecutor クラスの使用方法を示します。このクラスでは、先ほどの DecoratorRetryWorkflow ワークフローと同じ unreliableActivity アクティビティが使用されます。このワークフローインターフェイスは、RetryWorkflow で実装されており、1 つのメソッド process が含まれ

ます。これが、ワークフローのエンドポイントです。ワークフローワーカーは、次のように AsyncExecutorRetryWorkflowImpl で実装されます。

```
public class AsyncExecutorRetryWorkflowImpl implements RetryWorkflow {
  private final RetryActivitiesClient client = new RetryActivitiesClientImpl();
  private final DecisionContextProvider contextProvider = new
 DecisionContextProviderImpl();
  private final WorkflowClock clock =
 contextProvider.getDecisionContext().getWorkflowClock();

  public void process() {
      long initialRetryIntervalSeconds = 5;
      int maximumAttempts = 5;
      handleUnreliableActivity(initialRetryIntervalSeconds, maximumAttempts);
  }
  public void handleUnreliableActivity(long initialRetryIntervalSeconds, int
 maximumAttempts) {

      ExponentialRetryPolicy retryPolicy = new
 ExponentialRetryPolicy(initialRetryIntervalSeconds).withMaximumAttempts(maximumAttempts);
      final AsyncExecutor executor = new AsyncRetryingExecutor(retryPolicy, clock);

      new TryCatch() {
          @Override
          protected void doTry() throws Throwable {
              executor.execute(new AsyncRunnable() {
                  @Override
                  public void run() throws Throwable {
                      client.unreliableActivity();
                  }
              });
          }
          @Override
          protected void doCatch(Throwable e) throws Throwable {
          }
      };
  }
}
```

ワークフローの動作は次のとおりです。

1. process は、handleUnreliableActivity メソッドを呼び出し、構成設定を渡します。

2. handleUnreliableActivity では、ステップ 1 の設定を使用して、ExponentialRetryPolicy オブジェクト、retryPolicy を作成します。
3. handleUnreliableActivity は、AsyncRetryExecutor オブジェクト、executor を作成し、ステップ 2 のExponentialRetryPolicy オブジェクトと、ワークフロークロックのインスタンスをコンストラクタに渡します。
4. handleUnreliableActivity は、非同期のネステッドクラス TryCatch を実装し、doTry および doCatch メソッドを上書きして、再試行を実行し、例外に対処します。
5. doTry は匿名の AsyncRunnable クラスを作成して、run メソッドを上書きし、unreliableActivity を実行するためのカスタムコードを実装します。分かりやすいように、run ではアクティビティを実行しますが、必要に応じて、洗練されたアプローチを実装することもできます。
6. doTry は executor.execute を呼び出して、AsyncRunnable オブジェクトに渡します。execute は、AsyncRunnable オブジェクトの run メソッドを呼び出して、アクティビティを実行します。
7. アクティビティが実行した場合、エグゼキューターは、retryPolicy オブジェクトの設定に従って、再度 run を呼び出します。

TryCatch クラスを使用して、エラーに対応する詳細な方法については、「AWS Flow Framework for Java の例外」を参照してください。

カスタムの再試行戦略

失敗したアクティビティを再試行する最も柔軟な方法はカスタム戦略です。この方法では、成功するまで再試行する戦略とほとんど同じように、再試行を実行する非同期メソッドを再帰的に呼び出します。ただし、アクティビティを再度実行するだけではなく、連続的な再試行それぞれの実行有無および実行方法を決めるカスタムロジックを実装します。基本的なパターンは次のとおりです。

1. Settable<T> ステータスオブジェクトを作成します。このオブジェクトは、アクティビティが失敗したかどうかを示します。
2. ネステッドクラス (TryCatch または TryCatchFinally) クラスを実装します。
3. doTry は、アクティビティを実行します。
4. アクティビティが失敗した場合、doCatch は、アクティビティが失敗したことを示すステータスオブジェクトを設定します。
5. 非同期障害処理メソッドを呼び出し、それにステータスオブジェクトを渡します。このメソッドでは、TryCatch または TryCatchFinally が完了するまで、実行が延期されます。

6. 障害処理メソッドは、アクティビティの再試行有無を判断し、再試行する場合は再試行日時を決定します。

以下のワークフローは、カスタムの再試行戦略を実行する方法を示します。この戦略では、DecoratorRetryWorkflow および AsyncExecutorRetryWorkflow ワークフローと同じ unreliableActivity が使用されます。このワークフローインターフェイスは、RetryWorkflow で実装されており、1 つのメソッド process が含まれます。これが、ワークフローのエントポイントです。ワークフローワーカーは、次のように CustomLogicRetryWorkflowImpl で実装されます。

```
public class CustomLogicRetryWorkflowImpl implements RetryWorkflow {
  ...
  public void process() {
      callActivityWithRetry();
  }
  @Asynchronous
  public void callActivityWithRetry() {
      final Settable<Throwable> failure = new Settable<Throwable>();
      new TryCatchFinally() {
          protected void doTry() throws Throwable {
              client.unreliableActivity();
          }
          protected void doCatch(Throwable e) {
              failure.set(e);
          }
          protected void doFinally() throws Throwable {
              if (!failure.isReady()) {
                  failure.set(null);
              }
          }
      };
      retryOnFailure(failure);
  }
  @Asynchronous
  private void retryOnFailure(Promise<Throwable> failureP) {
      Throwable failure = failureP.get();
      if (failure != null && shouldRetry(failure)) {
          callActivityWithRetry();
      }
  }
  protected Boolean shouldRetry(Throwable e) {
      //custom logic to decide to retry the activity or not
```

```
        return true;
    }
}
```

ワークフローの動作は次のとおりです。

1. process は、非同期メソッド callActivityWithRetry を呼び出します。
2. callActivityWithRetry は、アクティビティが失敗したかどうかを示すために使用される failure という名前の Settable<Throwable> オブジェクトを作成します。Settable<T> は Promise<T> から派生して同じように動作しますが、Settable<T> オブジェクトの値は手動で設定します。
3. callActivityWithRetry では、非同期のネステッドクラス TryCatchFinally を実装して、unreliableActivity によってスローされるすべての例外に対処します。非同期コードでスローされる例外の対処方法の詳細については、「AWS Flow Framework for Java の例外」を参照してください。
4. doTry は、unreliableActivity を実行します。
5. unreliableActivity によって例外がスローされると、フレームワークは doCatch を呼び出し、例外オブジェクトに渡します。doCatch は failure を例外オブジェクトに設定します。これは、アクティビティが失敗し、オブジェクトが準備状態になったことを表します。
6. doFinally は、failure が準備状態かどうかを確認します。failure が doCatch で設定されている場合のみ、true になります。
   - failure の準備ができたら、 doFinallyは何もしません。
   - failure の準備ができていない場合、アクティビティは完了し、doFinally によってエラーが null に設定されます。
7. callActivityWithRetry は、非同期メソッド retryOnFailure を呼び出し、障害に渡します。障害は Settable<T> タイプのため、callActivityWithRetry は、障害が準備状態になるまで実行を延期します。そのため、TryCatchFinally が完了すると実行されます。
8. retryOnFailure は、障害から値を取得します。
   - failure が null に設定されると、再試行は成功です。retryOnFailure では何も行われないため、再試行プロセスは終了します。
   - 障害が例外オブジェクトに設定されており、shouldRetry より true が返る場合、retryOnFailure は callActivityWithRetry を呼び出して、そのアクティビティを再試行します。

shouldRetry はカスタムロジックを実装して、失敗したアクティビティを再試行するかどうかを決定します。分かりやすいように、shouldRetry は、常に true を返し、retryOnFailure はアクティビティをただちに実行しますが、必要に応じて洗練されたロジックを実装することができます。

9. unreliableActivity が完了するか、 shouldRetryでプロセスの停止が決定するまで、ステップ 2 ~ 8 を繰り返します。

> **Note**
>
> doCatch では、再試行プロセスは処理されません。この場合は、アクティビティが失敗したことを示す障害に設定されます。この再試行プロセスは、非同期メソッド retryOnFailure で処理されます。これにより、TryCatch が完了するまで、実行は延期されます。このアプローチの理由は、doCatch でアクティビティを再試行する場合はキャンセルできないことです。retryOnFailure でアクティビティを再試行すると、キャンセル可能なアクティビティを実行できます。

デーモンタスク

AWS Flow Framework for Java では、特定のタスクを としてマークできますdaemon。これにより、バックグラウンド処理を行うタスクを作成し、他のすべての処理が完了したら、これらのタスクをキャンセルできます。たとえば、稼動状況をモニタリングするタスクは、ワークフローの残りが完了したら、キャンセルする必要があります。これを行うには、非同期メソッドまたは TryCatchFinally のインスタンスに daemon フラグを設定します。次の例では、非同期メソッドの monitorHealth() を daemon としてマークしています。

```
public class MyWorkflowImpl implements MyWorkflow {
  MyActivitiesClient activitiesClient = new MyActivitiesClientImpl();

  @Override
  public void startMyWF(int a, String b) {
    activitiesClient.doUsefulWorkActivity();
    monitorHealth();
  }

  @Asynchronous(daemon=true)
```

```
  void monitorHealth(Promise<?>... waitFor) {
    activitiesClient.monitoringActivity();
  }
}
```

上の例では、doUsefulWorkActivity が完了すると、monitoringHealth が自動的にキャンセルされます。これに伴って、この非同期メソッドに基づく実行ブランチ全体がキャンセルされます。キャンセルのセマンティクスは、TryCatchFinally の場合と同じです。同様に、コンストラクタにブール型フラグを渡して TryCatchFinally をデーモンとしてマークできます。

```
public class MyWorkflowImpl implements MyWorkflow {
    MyActivitiesClient activitiesClient = new MyActivitiesClientImpl();

    @Override
    public void startMyWF(int a, String b) {
        activitiesClient.doUsefulWorkActivity();
        new TryFinally(true) {
            @Override
            protected void doTry() throws Throwable {
                activitiesClient.monitoringActivity();
            }

            @Override
            protected void doFinally() throws Throwable {
                // clean up
            }
        };
    }
}
```

TryCatchFinally 内で開始したデーモンタスクは、その作成元のコンテキストにスコープ指定されます。つまり、doTry()、doCatch()、または doFinally() のいずれかのメソッドにスコープされます。次の例では、startMonitoring 非同期メソッドがデーモンとしてマークされ、doTry() から呼び出されます。これに対して作成されたタスクは、doTry() 内で開始した他のタスク (この例では doUsefulWorkActivity) が終了すると同時にキャンセルされます。

```
public class MyWorkflowImpl implements MyWorkflow {
    MyActivitiesClient activitiesClient = new MyActivitiesClientImpl();

    @Override
```

```
    public void startMyWF(int a, String b) {
        new TryFinally() {
            @Override
            protected void doTry() throws Throwable {
                activitiesClient.doUsefulWorkActivity();
                startMonitoring();
            }

            @Override
            protected void doFinally() throws Throwable {
                // Clean up
            }
        };
    }

    @Asynchronous(daemon = true)
    void startMonitoring(){
      activitiesClient.monitoringActivity();
    }
```

AWS Flow Framework for Java の再生動作

このトピックでは、「AWS Flow Framework for Java とは」セクションの例を使用して、再生動作の例を説明します。同期シナリオと非同期シナリオの両方について説明します。

例 1: 同期再生

同期ワークフローでの再生動作の例では、HelloWorldWorkflow ワークフローおよびアクティビティ実装を変更します。そのために、各実装内に次のような println 呼び出しを追加します。

```
public class GreeterWorkflowImpl implements GreeterWorkflow {
...
   public void greet() {
      System.out.println("greet executes");
      Promise<String> name = operations.getName();
      System.out.println("client.getName returns");
      Promise<String> greeting = operations.getGreeting(name);
      System.out.println("client.greeting returns");
      operations.say(greeting);
      System.out.println("client.say returns");
   }
}
```

```
**************
public class GreeterActivitiesImpl implements GreeterActivities {
   public String getName() {
      System.out.println("activity.getName completes");
      return "World";
   }

   public String getGreeting(String name) {
      System.out.println("activity.getGreeting completes");
      return "Hello " + name + "!";
   }

   public void say(String what) {
      System.out.println(what);
   }
}
```

このコードの詳細については、「HelloWorldWorkflow アプリケーション」を参照してください。以下は、出力を編集したバージョンであり、各再生エピソードの開始を示すコメントが付いています。

```
//Episode 1
greet executes
client.getName returns
client.greeting returns
client.say returns

activity.getName completes
//Episode 2
greet executes
client.getName returns
client.greeting returns
client.say returns

activity.getGreeting completes
//Episode 3
greet executes
client.getName returns
client.greeting returns
client.say returns

Hello World! //say completes
//Episode 4
greet executes
```

```
client.getName returns
client.greeting returns
client.say returns
```

この例の再生プロセスは次のようになります。

- 最初のエピソードでは getName アクティビティタスクをスケジュールします。このタスクには依存関係がありません。
- 2 番目のエピソードでは getGreeting アクティビティタスクをスケジュールします。このタスクは getName に依存します。
- 3 番目のエピソードでは say アクティビティタスクをスケジュールします。このタスクは getGreeting に依存します。
- 最後のエピソードでは、追加のタスクをスケジュールしません。未完了のアクティビティがなければ、ワークフロー実行を終了します。

> **Note**
>
> 各エピソードに対して 3 つのアクティビティクライアントメソッドが 1 回呼び出されます。ただし、これらの呼び出しの 1 つのみがアクティビティタスクになるため、各タスクは 1 回のみ実行されます。

例 2: 非同期再生

同期再生の例と同じように、HelloWorldWorkflowAsync アプリケーション を変更して非同期再生の動作を確認できます。次の出力が生成されます。

```
//Episode 1
greet executes
client.name returns
workflow.getGreeting returns
client.say returns

activity.getName completes
//Episode 2
greet executes
client.name returns
workflow.getGreeting returns
```

```
client.say returns
workflow.getGreeting completes

Hello World! //say completes
//Episode 3
greet executes
client.name returns
workflow.getGreeting returns
client.say returns
workflow.getGreeting completes
```

HelloWorldAsync は 3 つの再生エピソードを使用します。2 つのアクティビティしかないためです。getGreeting アクティビティは getGreeting 非同期ワークフローメソッドに置き換えられています。このメソッドの完了に伴って再生エピソードは開始されません。

最初のエピソードでは、名前アクティビティの完了に依存するため、getGreeting を呼び出しません。ただし、getName が完了すると、再生は後続のエピソードごとに getGreeting を呼び出します。

以下の資料も参照してください。

- AWS Flow Framework 基本的な概念: 分散実行

ベストプラクティス

これらのベストプラクティスを使用して、 AWS Flow Framework for Java を最大限に活用します。

トピック

- ディサイダーコードの変更: バージョニングと機能フラグ

ディサイダーコードの変更: バージョニングと機能フラグ

このセクションでは、次の 2 つのメソッドを使用して、ディサイダーに対する後方互換性のない変更を回避する方法について説明します。

- バージョニングは、基本的な解決策です。
- 機能フラグ付きバージョニングは、バージョニングソリューションで構築されます。新しいバージョンのワークフローは導入されないため、バージョンを更新するために新しいコードをプッシュする必要はありません。

これらのソリューションを行う前に、「シナリオの例」セクションを確認します。このセクションでは、後方互換性のない変更の原因と影響について記載されています。

再生プロセスとコード変更

AWS Flow Framework for Java ディサイダーワーカーが決定タスクを実行する場合、ステップを追加する前に、まず実行の現在の状態を再構築する必要があります。ディサイダーは、再生と呼ばれるプロセスを使用してこの作業を行います。

再生プロセスでは、ディサイダーコードが最初から再実行されるのに対し、すでに実行されたイベントの履歴も同時に更新されます。イベント履歴を進めることで、フレームワークでシグナルやタスクの完了に反応し、コードの `Promise` オブジェクトのブロックを解除します。

ディサイダーコードが実行されると、フレームワークはカウンターを増分し、スケジュールされた各タスク (アクティビティ、Lambda 関数、タイマー、子ワークフロー、送信中のシグナル) に ID を割り当てます。フレームワークは、この ID を Amazon SWF に送信し、`ActivityTaskCompleted` などの履歴イベントにその ID を追加します。

再生プロセスを成功させるには、ディサイダーコードが決定的であり、すべてのワークフロー実行の各ディシジョンにおいて、同じタスクを同じ順序でスケジュールすることが重要です。この要件に

準拠しない場合、フレームワークは、ActivityTaskCompleted イベントの ID を既存の Promise オブジェクトと一致させることができない場合があります。

シナリオの例

後方互換性のない変更とみなされるコードのクラスの変更があります。このような変更には、スケジュールされたタスクの数値、型、または順序を変更する更新などがあります。次の例を考えます。

2 つのタイマータスクをスケジュールするには、ディサイダーコードを記述します。実行をスタートし、ディシジョンを実行します。その結果、2 つのタイマータスクは、ID 1 と 2 でスケジュールされます。

次のディシジョンが実行される前に、ディサイダーコードを更新して 1 つのタイマーのみスケジュールする場合、次のディシジョンタスク時、ID 2 は、コードが生成されたどのタイマータスクとも一致しないため、フレームワークで 2 番目の TimerFired イベントを再生することはできません。

シナリオの概要

このシナリオのステップを以下の概要に示します。シナリオの最終目標は、システムに移行して 1 つのタイマーのみスケジュールすることですが、移行前にスタートした実行が失敗することはありません。

1. ディサイダーの初期バージョン
   a. ディサイダーを記述します。
   b. ディサイダーをスタートします。
   c. ディサイダーによって、2 つのタイマーがスケジュールされます。
   d. ディサイダーは、5 つの実行をスタートします。
   e. ディサイダーを停止します。
2. 後方互換性のないディサイダーの変更
   a. ディサイダーを変更します。
   b. ディサイダーをスタートします。
   c. ディサイダーによって、1 つのタイマーがスケジュールされます。
   d. ディサイダーは、5 つの実行をスタートします。

次のセクションには、このシナリオの実装方法を示す Java コードの例が含まれます。ソリューション セクションのコード例は、後方互換性のない変更を修正するさまざまな方法を示します。

> **Note**
>
> このコードを実行するには、最新バージョンの AWS SDK for Java を使用します。

共通コード

次の Java コードによって、このシナリオの例が変更されることはありません。

SampleBase.java

```
package sample;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.flow.JsonDataConverter;
import com.amazonaws.services.simpleworkflow.model.DescribeWorkflowExecutionRequest;
import com.amazonaws.services.simpleworkflow.model.DomainAlreadyExistsException;
import com.amazonaws.services.simpleworkflow.model.RegisterDomainRequest;
import com.amazonaws.services.simpleworkflow.model.Run;
import com.amazonaws.services.simpleworkflow.model.StartWorkflowExecutionRequest;
import com.amazonaws.services.simpleworkflow.model.TaskList;
import com.amazonaws.services.simpleworkflow.model.WorkflowExecution;
import com.amazonaws.services.simpleworkflow.model.WorkflowExecutionDetail;
import com.amazonaws.services.simpleworkflow.model.WorkflowType;

public class SampleBase {

    protected String domain = "DeciderChangeSample";
    protected String taskList = "DeciderChangeSample-" + UUID.randomUUID().toString();
    protected AmazonSimpleWorkflow service =
 AmazonSimpleWorkflowClientBuilder.defaultClient();
    {
        try {
            AmazonSimpleWorkflowClientBuilder.defaultClient().registerDomain(new
 RegisterDomainRequest().withName(domain).withDescription("desc").withWorkflowExecutionRetentio
        } catch (DomainAlreadyExistsException e) {
        }
    }
```

```
    protected List<WorkflowExecution> workflowExecutions = new ArrayList<>();

    protected void startFiveExecutions(String workflow, String version, Object input) {
        for (int i = 0; i < 5; i++) {
            String id = UUID.randomUUID().toString();
            Run startWorkflowExecution = service.startWorkflowExecution(
                    new
StartWorkflowExecutionRequest().withDomain(domain).withTaskList(new
TaskList().withName(taskList)).withInput(new JsonDataConverter().toData(new
Object[] { input })).withWorkflowId(id).withWorkflowType(new
WorkflowType().withName(workflow).withVersion(version)));
            workflowExecutions.add(new
WorkflowExecution().withWorkflowId(id).withRunId(startWorkflowExecution.getRunId()));
            sleep(1000);
        }
    }

    protected void printExecutionResults() {
        waitForExecutionsToClose();
        System.out.println("\nResults:");
        for (WorkflowExecution wid : workflowExecutions) {
            WorkflowExecutionDetail details = service.describeWorkflowExecution(new
DescribeWorkflowExecutionRequest().withDomain(domain).withExecution(wid));
            System.out.println(wid.getWorkflowId() + " " +
details.getExecutionInfo().getCloseStatus());
        }
    }

    protected void waitForExecutionsToClose() {
        loop: while (true) {
            for (WorkflowExecution wid : workflowExecutions) {
                WorkflowExecutionDetail details = service.describeWorkflowExecution(new
DescribeWorkflowExecutionRequest().withDomain(domain).withExecution(wid));
                if ("OPEN".equals(details.getExecutionInfo().getExecutionStatus())) {
                    sleep(1000);
                    continue loop;
                }
            }
            return;
        }
    }

    protected void sleep(int millis) {
```

```
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

}
```

Input.java

```
package sample;

public class Input {

    private Boolean skipSecondTimer;

    public Input() {
    }

    public Input(Boolean skipSecondTimer) {
        this.skipSecondTimer = skipSecondTimer;
    }

    public Boolean getSkipSecondTimer() {
        return skipSecondTimer != null && skipSecondTimer;
    }

    public Input setSkipSecondTimer(Boolean skipSecondTimer) {
        this.skipSecondTimer = skipSecondTimer;
        return this;
    }

}
```

最初のディサイダーコードを記述する

ディサイダーの最初の Java コードは以下のとおりです。これは、バージョン 1 として登録され、5 秒のタイマータスクが 2 つスケジュールされます。

InitialDecider.java

```
package sample.v1;
```

```
import com.amazonaws.services.simpleworkflow.flow.DecisionContext;
import com.amazonaws.services.simpleworkflow.flow.DecisionContextProviderImpl;
import com.amazonaws.services.simpleworkflow.flow.WorkflowClock;
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
 defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {

    @Execute(version = "1")
    public void sample(Input input);

    public static class Impl implements Foo {

        private DecisionContext decisionContext = new
 DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

        @Override
        public void sample(Input input) {
            System.out.println("Decision (V1) WorkflowId: " +
 decisionContext.getWorkflowContext().getWorkflowExecution().getWorkflowId());
            clock.createTimer(5);
            clock.createTimer(5);
        }

    }
}
```

後方互換性のない変更のシミュレーション

次のディサイダーの変更された Java コードは、後方互換性のない変更を示す良い例です。コードは、バージョン 1 として登録されますが、スケジュールされるタイマーは 1 つのみです。

ModifiedDecider.java

```
package sample.v1.modified;

import com.amazonaws.services.simpleworkflow.flow.DecisionContext;
import com.amazonaws.services.simpleworkflow.flow.DecisionContextProviderImpl;
import com.amazonaws.services.simpleworkflow.flow.WorkflowClock;
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
 defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {

    @Execute(version = "1")
    public void sample(Input input);

    public static class Impl implements Foo {

        private DecisionContext decisionContext = new
 DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

        @Override
        public void sample(Input input) {
            System.out.println("Decision (V1 modified) WorkflowId: " +
 decisionContext.getWorkflowContext().getWorkflowExecution().getWorkflowId());
            clock.createTimer(5);
        }

    }
}
```

次の Java コードでは、変更後のディサイダーを実行して、後方互換性のない変更が行われる問題をシミュレートします。

RunModifiedDecider.java

```
package sample;
```

```
import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class BadChange extends SampleBase {

    public static void main(String[] args) throws Exception {
        new BadChange().run();
    }

    public void run() throws Exception {
        // Start the first version of the decider
        WorkflowWorker before = new WorkflowWorker(service, domain, taskList);
        before.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        before.start();

        // Start a few executions
        startFiveExecutions("Foo.sample", "1", new Input());

        // Stop the first decider worker and wait a few seconds
        // for its pending pollers to match and return
        before.suspendPolling();
        sleep(2000);

        // At this point, three executions are still open, with more decisions to make

        // Start the modified version of the decider
        WorkflowWorker after = new WorkflowWorker(service, domain, taskList);
        after.addWorkflowImplementationType(sample.v1.modified.Foo.Impl.class);
        after.start();

        // Start a few more executions
        startFiveExecutions("Foo.sample", "1", new Input());

        printExecutionResults();
    }

}
```

このプログラムを実行すると、失敗する 3 つの実行は、初期バージョンのディサイダーでスタートされ、移行後も継続します。

ソリューション

後方互換性のない変更を回避するには、次のソリューションを使用します。詳細については、「Making Changes to Decider Code」(ディサイダーコードの変更) および「シナリオの例」を参照してください。

バージョニングの使用

このソリューションでは、ディサイダーを新しいクラスにコピーして、ディサイダーを変更し、新しいワークフローバージョンでそのディサイダーを登録します。

VersionedDecider.java

```
package sample.v2;

import com.amazonaws.services.simpleworkflow.flow.DecisionContext;
import com.amazonaws.services.simpleworkflow.flow.DecisionContextProviderImpl;
import com.amazonaws.services.simpleworkflow.flow.WorkflowClock;
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
 defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {

    @Execute(version = "2")
    public void sample(Input input);

    public static class Impl implements Foo {

        private DecisionContext decisionContext = new
 DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

        @Override
        public void sample(Input input) {
            System.out.println("Decision (V2) WorkflowId: " +
 decisionContext.getWorkflowContext().getWorkflowExecution().getWorkflowId());
```

```
            clock.createTimer(5);
        }

    }

}
```

更新後の Java コードでは、2 番目のディサイダーワーカーによって、いずれのバージョンのワークフローも実行されるため、バージョン 2 の変更とは別に、処理中の実行を続けることができます。

RunVersionedDecider.java

```
package sample;

import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class VersionedChange extends SampleBase {

    public static void main(String[] args) throws Exception {
        new VersionedChange().run();
    }

    public void run() throws Exception {
        // Start the first version of the decider, with workflow version 1
        WorkflowWorker before = new WorkflowWorker(service, domain, taskList);
        before.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        before.start();

        // Start a few executions with version 1
        startFiveExecutions("Foo.sample", "1", new Input());

        // Stop the first decider worker and wait a few seconds
        // for its pending pollers to match and return
        before.suspendPolling();
        sleep(2000);

        // At this point, three executions are still open, with more decisions to make

        // Start a worker with both the previous version of the decider (workflow
 version 1)
        // and the modified code (workflow version 2)
        WorkflowWorker after = new WorkflowWorker(service, domain, taskList);
        after.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
```

```
        after.addWorkflowImplementationType(sample.v2.Foo.Impl.class);
        after.start();

        // Start a few more executions with version 2
        startFiveExecutions("Foo.sample", "2", new Input());

        printExecutionResults();
    }

}
```

プログラムを実行すると、実行はすべて正常に完了します。

機能フラグの使用

後方互換性の問題のもう 1 つのソリューションは、ワークフローバージョンではなく、入力データに基づき分岐するために、同じクラスの 2 つの実装をサポートするコードを分岐することです。

このアプローチを使用する場合は、機密的な変更を行う度に、入力オブジェクトにフィールドを追加するか、既存フィールドを変更します。移行前にスタートする実行の場合、入力オブジェクトにフィールドは作成されません (または異なる値が含まれます)。このように、バージョン番号を増やす必要はありません。

> **Note**
>
> 新しいフィールドを追加する場合は、JSON の逆シリアル化プロセスに後方互換性があることを確認します。フィールドの導入前にシリアル化されたオブジェクトは、移行すると正常に逆シリアル化されます。フィールドがない場合、null 値が JSON に設定されるため、必ず boxed 型 (boolean ではなく Boolean) を使用し、値が null のケースに対応します。

`FeatureFlagDecider.java`

```
package sample.v1.featureflag;

import com.amazonaws.services.simpleworkflow.flow.DecisionContext;
import com.amazonaws.services.simpleworkflow.flow.DecisionContextProviderImpl;
import com.amazonaws.services.simpleworkflow.flow.WorkflowClock;
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
```

```
import
 com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
 defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {

    @Execute(version = "1")
    public void sample(Input input);

    public static class Impl implements Foo {

        private DecisionContext decisionContext = new
 DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

        @Override
        public void sample(Input input) {
            System.out.println("Decision (V1 feature flag) WorkflowId: " +
 decisionContext.getWorkflowContext().getWorkflowExecution().getWorkflowId());
            clock.createTimer(5);
            if (!input.getSkipSecondTimer()) {
                clock.createTimer(5);
            }
        }

    }
}
```

更新後の Java コードでは、ワークフローのコードはいずれのバージョンも、バージョン 1 として登録されます。ただし、移行後、新しい実行は、true に設定されている入力データの skipSecondTimer フィールドからスタートします。

RunFeatureFlagDecider.java

```
package sample;

import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class FeatureFlagChange extends SampleBase {
```

```
    public static void main(String[] args) throws Exception {
        new FeatureFlagChange().run();
    }

    public void run() throws Exception {
        // Start the first version of the decider
        WorkflowWorker before = new WorkflowWorker(service, domain, taskList);
        before.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        before.start();

        // Start a few executions
        startFiveExecutions("Foo.sample", "1", new Input());

        // Stop the first decider worker and wait a few seconds
        // for its pending pollers to match and return
        before.suspendPolling();
        sleep(2000);

        // At this point, three executions are still open, with more decisions to make

        // Start a new version of the decider that introduces a change
        // while preserving backwards compatibility based on input fields
        WorkflowWorker after = new WorkflowWorker(service, domain, taskList);
        after.addWorkflowImplementationType(sample.v1.featureflag.Foo.Impl.class);
        after.start();

        // Start a few more executions and enable the new feature through the input
 data
        startFiveExecutions("Foo.sample", "1", new Input().setSkipSecondTimer(true));

        printExecutionResults();
    }

}
```

プログラムを実行すると、実行はすべて正常に完了します。

AWS Flow Framework for Java のトラブルシューティングとデバッグのヒント

トピック

このセクションでは、 AWS Flow Framework for Java を使用してワークフローを開発する際に発生する可能性がある一般的な落とし穴について説明します。また、問題の診断とデバッグに役立つヒントも紹介します。

コンパイルエラー

AspectJ のコンパイル時ウィービングオプションを使用している場合、コンパイラでワークフローおよびアクティビティ用に生成されたクライアントクラスを発見できないコンパイル時エラーが発生する場合があります。このようなコンパイルエラーは、コンパイル時に生成されたクライアントを AspectJ ビルダーが無視したことが原因です。この問題を解決するには、AspectJ の機能をプロジェクトから削除して再度有効にします。この操作は、ワークフローまたはアクティビティのインターフェイスが変更される度に行う必要があります。この問題を回避するために、ロード時間ウィービングオプションを使用することをお勧めします。詳細については、「AWS Flow Framework for Java のセットアップ」セクションを参照してください。

不明なリソース障害

使用できないリソースに対して操作を実行しようとすると、Amazon SWF より未知のリソース障害が返ります。この障害に対する一般的な原因は次のとおりです。

- ドメインが存在しないワーカーを設定している。この問題を修正するには、まず Amazon SWF コンソール または Amazon SWF サービスAPI を使用してドメインを登録します。
- ワークフロー実行または未登録タイプのアクティビティタスクを作成しようとしている。この問題は、ワーカーが実行される前にワークフロー実行を作成しようとすると発生する場合があります。ワーカーは初めて実行するときにタイプを登録するため、実行を開始する前に少なくとも 1 回実行する必要があります (または、コンソールまたはサービス API を使用して手動でタイプを登録します)。タイプが登録されると、実行中のワーカーがなくても実行を作成できます。
- ワーカーが、すでにタイムアウトしているタスクを実行しようとしている。たとえば、処理に時間がかかりすぎてワーカーがタイムアウトした場合は、タスクの完了時または失敗時に UnknownResource 障害が発生します。 AWS Flow Framework ワーカーは引き続き Amazon SWF をポーリングし、追加のタスクを処理します。ただし、タイムアウトの調整を検討する必要があります。タイムアウトを調整するには、新しいバージョンのアクティビティタイプを登録する必要があります。

Promise で get() を呼び出すときの例外

Java 機能とは異なり、`Promise` は、ノンブロッキング構築であり、準備状態になっていない `Promise` で `get()` を呼び出すと、ブロックされずに例外がスローされます。`Promise` を使用する正しい方法は、これを非同期タスク (またはタスク) に渡して非同期メソッドで値にアクセスすることです。 AWS Flow Framework for Java により、すべての `Promise` 引数が渡され準備が整った場合にのみ、非同期メソッドが呼び出されます。 コードが正しいと思われる場合、または AWS Flow Framework いずれかのサンプルの実行中にこれを実行した場合、AspectJ が正しく設定されていないことが原因である可能性があります。詳細については、「AWS Flow Framework for Java のセットアップ」セクションを参照してください。

非決定的なワークフロー

「非決定論」セクションに記載されているように、ワークフローの実装は決定的である必要があります。非決定論につながる一般的な間違いには、システムクロックの使用、乱数の使用、および GUID の生成などがあります。これらのコンストラクトは異なる時間に異なる値を返す可能性があるため、ワークフローの実行ごとにワークフローの制御フローに異なるパスがかかる場合があります (詳細については、「」セクションAWS Flow Framework 基本的な概念: 分散実行とAWS Flow Framework for Java のタスクについて「」セクションを参照)。ワークフローの実行中にフレームワークで非決定性が検出されると、例外がスローされます。

バージョニングによる問題

新しいバージョンのワークフローやアクティビティを実装する場合 (例: 新しい機能を追加する場合)、適切な注釈 (`@Workflow`、`@Activites`、または `@Activity`) を使用してタイプのバージョンを増やす必要があります。新しいバージョンのワークフローがデプロイされると、すでに実行されている既存のバージョンが実行されることがあります。そのため、適切なバージョンのワークフローとアクティビティを持つワーカーがタスクを取得できることを確認します。これを行うには、バージョンごとに異なるタスクリストを使用します。たとえば、タスクリストの名前にバージョン番号を追加することができます。これにより、異なるバージョンのワークフローおよびアクティビティに属するタスクが適切なワーカーに割り当てられます。

ワークフロー実行のトラブルシューティングとデバッグ

ワークフロー実行のトラブルシューティングでは、まず Amazon SWF コンソールを使用してワークフロー履歴を確認します。ワークフロー履歴は、ワークフロー実行の実行状態を変更したすべてのイベントの完全かつ信頼できるレコードです。この履歴は、Amazon SWF で管理されます。また、問題の診断に非常に役立ちます。Amazon SWF コンソールを使用すると、ワークフロー実行を検索し、各履歴イベントを詳細に確認できます。

AWS Flow Framework には、ワークフロー実行をローカルで再生してデバッグするために使用できる`WorkflowReplayer`クラスが用意されています。このクラスを使用して、クローズ済みおよび実行中のワークフロー実行をデバッグできます。`WorkflowReplayer` は Amazon SWF に保存されている履歴に依存して再生を実行します。Amazon SWF アカウントのワークフロー実行をポイントするか、履歴イベントを使用できます (例えば、Amazon SWF から履歴を取得し、後で使用できるようにローカルでシリアル化することができます)。`WorkflowReplayer` を使用してワークフロー実行を再生しても、アカウントで実行されているワークフロー実行には影響ありません。再生はクライアント上で完全に行われます。デバッグツールを使用して、ワークフローのデバッグ、ブレークポイントの作成、コーディングを行うことができます。Eclipse を使用している場合は、ステップフィルターを追加して AWS Flow Framework パッケージをフィルタリングすることを検討してください。

たとえば、次のコードスニペットは、ワークフロー実行の再生に使用できます。

```
String workflowId = "testWorkflow";
String runId = "<run id>";
Class<HelloWorldImpl> workflowImplementationType = HelloWorldImpl.class;
WorkflowExecution workflowExecution = new WorkflowExecution();
workflowExecution.setWorkflowId(workflowId);
workflowExecution.setRunId(runId);
```

```
WorkflowReplayer<HelloWorldImpl> replayer = new WorkflowReplayer<HelloWorldImpl>(
    swfService, domain, workflowExecution, workflowImplementationType);

System.out.println("Beginning workflow replay for " + workflowExecution);
Object workflow = replayer.loadWorkflow();
System.out.println("Workflow implementation object:");
System.out.println(workflow);
System.out.println("Done workflow replay for " + workflowExecution);
```

AWS Flow Framework では、ワークフロー実行の非同期スレッドダンプを取得することもできます。このスレッドダンプでは、オープンのすべての非同期タスクの呼び出しスタックが割り当てられます。この情報は、実行中に保留されており、スタック状態になっているタスクを判断する上で便利です。例:

```
String workflowId = "testWorkflow";
String runId = "<run id>";
Class<HelloWorldImpl> workflowImplementationType = HelloWorldImpl.class;
WorkflowExecution workflowExecution = new WorkflowExecution();
workflowExecution.setWorkflowId(workflowId);
workflowExecution.setRunId(runId);

WorkflowReplayer<HelloWorldImpl> replayer = new WorkflowReplayer<HelloWorldImpl>(
    swfService, domain, workflowExecution, workflowImplementationType);

try {
    String flowThreadDump = replayer.getAsynchronousThreadDumpAsString();
    System.out.println("Workflow asynchronous thread dump:");
    System.out.println(flowThreadDump);
}
catch (WorkflowException e) {
    System.out.println("No asynchronous thread dump available as workflow has failed: "
 + e);
}
```

失われたタスク

タスクが古いワーカーに引き渡されることを確認する場合に限り、ワーカーをシャットダウンし、続けて新しいワーカーをすぐに開始することがあります。この操作は、システム内の競合状態が原因で行う場合があり、複数のプロセスに分散されます。また、この問題は、短い周期でユニットテストを実行している場合にも発生することがあります。Eclipse のテストを停止して、シャットダウンハンドラが呼び出されない場合にも、この現象が起こることもあります。

古いワーカーでタスクが取得されていることによる問題であることを確認するには、ワークフロー履歴を確認し、新しいワーカーが受け取る予定のタスクを受け取ったプロセスを判断する必要があります。たとえば、履歴の DecisionTaskStarted には、タスクを受け取ったワークフローワーカーの識別子が含まれます。フローフレームワークによって使用されているこの ID の形式は、次のとおりです: {*processId*}@{*host name*} 例えば、サンプル実行における Amazon SWF コンソールの DecisionTaskStarted イベントの詳細は次のとおりです。

イベントのタイムスタンプ	Mon Feb 20 11:52:40 GMT-800 2012
アイデンティティ	2276@ip-0A6C1DF5
スケジュールされたイベント ID	33

この状況を回避するには、テストごとに異なるタスクリストを使用します。また、古いワーカーのシャットダウンと、新しいワーカーの起動の間に、遅延を追加することを検討してください。

API パラメータの長さの制約による検証の失敗

Amazon SWF は、API パラメータに長さの制約を適用します。ワークフローまたはアクティビティの実装が制約を超えると、HTTP 400エラーが発生します。たとえば、実行中のアクティビティのハートビートを送信ActivityExecutionContextするために recordActivityHeartbeat を呼び出す場合、文字列は 2048 文字以下にする必要があります。

もう 1 つの一般的なシナリオは、例外が原因でアクティビティが失敗した場合です。フレームワークは、シリアル化された例外を詳細として RespondActivityTaskFailed を呼び出して、アクティビティの失敗を Amazon SWF に報告します。シリアル化された例外の長さが 32,768 バイトを超える場合、API コールは 400 エラーを報告します。この状況を軽減するために、例外メッセージまたはが長さの制約に準拠する原因を切り捨てることができます。

AWS Flow Framework for Java リファレンス

トピック

AWS Flow Framework for Java の注釈

トピック

@Activities

この注釈をインターフェイスに設定してアクティビティタイプのセットを宣言できます。この注釈を設定したインターフェイスの各メソッドはアクティビティタイプを表します。インターフェイスには、`@Workflow` 注釈と `@Activities` 注釈の両方は設定できません

この注釈では、以下のパラメータを指定できます。

`activityNamePrefix`

インターフェイスで宣言したアクティビティタイプの名前のプレフィックスを指定します。空の文字列 (デフォルト) に設定すると、インターフェースの名前の後に「.」がプレフィックスとして使用されます。

`version`

インターフェイスで宣言したアクティビティタイプのデフォルトバージョンを指定します。デフォルト値は `1.0` です。

`dataConverter`

このアクティビティタイプとその結果を作成するときに、データのシリアル化/逆シリアル化に使用する `DataConverter` の型を指定します。デフォルトでは `NullDataConverter` に設定されます。この場合、`JsonDataConverter` を使用することを指示します。

@Activity

この注釈は、`@Activities` で注釈を設定したインターフェイス内のメソッドで使用できます。

この注釈では、以下のパラメータを指定できます。

`name`

アクティビティタイプの名前を指定します。デフォルトの空の文字列では、デフォルトのプレフィックスとアクティビティメソッドの名前を使用して、アクティビティタイプの名前 ({{prefix}}{{name} 形式) を決定します。`@Activity` 注釈で名前を指定する場合、フレームワークでは自動的にプレフィックスを先頭に追加しないことに注意してください。独自の命名スキームを使用することができます。

`version`

アクティビティタイプのバージョンを指定します。これにより、含むインターフェイスで `@Activities` 注釈に指定したデフォルトバージョンは上書きされます。デフォルトは空の文字列です。

@ActivityRegistrationOptions

アクティビティタイプの登録オプションを指定します。この注釈は、`@Activities` で注釈を設定したインターフェイスまたはインターフェイス内のメソッドで使用できます。両方の場所に指定すると、メソッドに設定した注釈が適用されます。

この注釈では、以下のパラメータを指定できます。

`defaultTasklist`

このアクティビティタイプに対して Amazon SWF に登録されているデフォルトのタスクリストを指定します。このデフォルトは、生成されたクライアントでアクティビティメソッドを呼び出すときに、`ActivitySchedulingOptions` パラメータを使用して上書きできます。デフォルトでは `USE_WORKER_TASK_LIST` に設定されます。この特別な値では、登録を担当するワーカーで使用しているタスクリストを使用することを指定します。

`defaultTaskScheduleToStartTimeoutSeconds`

このアクティビティタイプに対して Amazon SWF に登録されている defaultTaskScheduleToStartTimeout を指定します。このアクティビティタイプのタスクをワーカーに割り当てるまでに許容される最大の待機時間を示します。詳細については、「Amazon Simple Workflow Service API Reference」(Amazon Simple Workflow Service API リファレンス) を参照してください。

`defaultTaskHeartbeatTimeoutSeconds`

このアクティビティタイプに対して Amazon SWF に登録されている `defaultTaskHeartbeatTimeout` を指定します。アクティビティワーカーは、この時間内にハートビートを提供する必要があります。提供しない場合、タスクはタイムアウトになります。デフォルトでは -1 に設定されます。この特別な値では、このタイムアウトを無効化することを指定します。詳細については、「Amazon Simple Workflow Service API Reference」(Amazon Simple Workflow Service API リファレンス) を参照してください。

`defaultTaskStartToCloseTimeoutSeconds`

このアクティビティタイプに対して Amazon SWF に登録されている defaultTaskStartToCloseTimeout を指定します。このタイムアウトは、このタイプのアクティビティタスクをワーカーで処理する最大許容時間を決定します。詳細については、「Amazon Simple Workflow Service API Reference」(Amazon Simple Workflow Service API リファレンス) を参照してください。

`defaultTaskScheduleToCloseTimeoutSeconds`

このアクティビティタイプに対して Amazon SWF に登録されている `defaultScheduleToCloseTimeout` を指定します。このタイムアウトは、タスクをオープン状態に保持できる総時間を決定します。デフォルトでは -1 に設定されます。この特別な値では、このタイムアウトを無効化することを指定します。詳細については、「Amazon Simple Workflow

Service API Reference」(Amazon Simple Workflow Service API リファレンス) を参照してください。

@Asynchronous

ワークフロー調整ロジックのメソッドで使用した場合、メソッドを非同期的に実行することを指定します。メソッドへの呼び出しは即座に戻りますが、実際の実行は、メソッドに渡したすべての Promise<> パラメータが準備完了状態になったときに非同期的に行われます。@Asynchronous で注釈を設定したメソッドは、戻り値の型として Promise<> または void を使用する必要があります。

`daemon`

非同期メソッド用に作成したタスクをデーモンタスクにするかどうかを指定します。デフォルトでは `False` です。

@Execute

`@Workflow` で注釈を設定したインターフェイスのメソッドで使用した場合、ワークフローのエントリポイントを識別します。

> **Important**
>
> `@Execute` で修飾できるインターフェイスのメソッドは 1 つのみです。

この注釈では、以下のパラメータを指定できます。

`name`

ワークフロータイプの名前を指定します。設定しない場合は、デフォルトで {prefix}{name} に設定されます。ここで、{prefix} は、ワークフローインターフェイスの名前に「.」がついたもの、{name} は、ワークフロー内で `@Execute` で修飾されたメソッドの名前を表します。

`version`

ワークフロータイプのバージョンを指定します。

@ExponentialRetry

アクティビティまたは非同期メソッドで使用した場合、処理されない例外がメソッドからスローされると、指数再試行ポリシーを設定します。再試行は、試行数の累乗によって計算されるバックオフ期間の後に行われます。

この注釈では、以下のパラメータを指定できます。

`intialRetryIntervalSeconds`

最初の再試行までに待機する時間を指定します。この値は、`maximumRetryIntervalSeconds` および `retryExpirationSeconds` 以下にする必要があります。

`maximumRetryIntervalSeconds`

再試行間の最大時間を指定します。最大時間に達すると、この値が再試行間隔の上限となります。デフォルトの -1 に設定すると、時間は無制限になります。

`retryExpirationSeconds`

指数再試行を停止する期限を指定します。デフォルトの -1 に設定すると、有効期限切れはありません。

`backoffCoefficient`

再試行間隔の計算に使用する係数を指定します。「指数的再試行戦略」を参照してください。

`maximumAttempts`

指数再試行を停止するまでの試行数を指定します。デフォルトの -1 に設定すると、再試行数に制限はありません。

`exceptionsToRetry`

再試行をトリガーする例外タイプのリストを指定します。これらのタイプの処理されない例外は以後伝播されず、計算された再試行間隔の後でメソッドが再試行されます。デフォルトでは、リストに `Throwable` が含まれます。

`excludeExceptions`

再試行をトリガーしない例外タイプのリストを指定します。このタイプの処理されない例外は伝播できます。デフォルトでは、リストは空です。

@GetState

@Workflow 注釈を設定したインターフェイスのメソッドで使用した場合、このメソッドを使用してワークフロー実行の最新状態を取得することを指定します。@Workflow 注釈を設定したインターフェイスでは、この注釈を最大 1 つのメソッドに設定できます。この注釈を設定したメソッドは、パラメータを取ることができず、戻り値の型として void 以外を使用する必要があります。

@ManualActivityCompletion

この注釈をアクティビティメソッドで使用して、メソッドが戻ったときにアクティビティタスクを未完了状態にすることを指定できます。アクティビティタスクは自動的に完了しないため、Amazon SWF API を使用して直接手動で完了する必要があります。これが役立つユースケースとしては、アクティビティタスクを外部システムに委任する際に、委任先が自動化されていないか、人間が介入して完了させる必要がある場合が挙げられます。

@Signal

@Workflow 注釈を設定したインターフェイスのメソッドで使用した場合、インターフェイスで宣言したワークフロータイプの実行で受信できるシグナルを識別します。シグナルメソッドを定義するには、この注釈を使用する必要があります。

この注釈では、以下のパラメータを指定できます。

name

シグナル名の名前部分を指定します。指定しない場合は、メソッド名が使用されます。

@SkipRegistration

@Workflow 注釈を設定したインターフェイスで使用した場合、ワークフロータイプを Amazon SWF に登録しないことを指定します。@Workflow で注釈を設定したインターフェイスでは、@WorkflowRegistrationOptions と @SkipRegistrationOptions のいずれかを使用します。両方を使用することはできません。

@Wait と @NoWait

これらの注釈は、 タイプのパラメータで使用Promise<>して、 AWS Flow Framework for Java がメソッドを実行する前に準備が完了するのを待つ必要があるかどうかを示すことができます。デフォルトでは、@Asynchronous メソッドに渡した Promise<> パラメータが準備完了状態になった後

で、メソッドが実行されます。特定のシナリオでは、このデフォルトの動作を上書きする必要があります。@Asynchronous メソッドに渡され、@NoWait 注釈の Promise<> パラメータは待機されません。

List<Promise<Int>> などの Promise を含むコレクションパラメータ (またはサブクラス) には @Wait 注釈を設定する必要があります。デフォルトでは、フレームワークはコレクションのメンバーを待機しません。

@Workflow

この注釈をインターフェイスで使用して workflow タイプを宣言します。ワークフローのエントリポイントを宣言するには、この注釈で修飾したインターフェイスに、@Execute 注釈で修飾したメソッドを正確に 1 つ含める必要があります。

> **Note**
>
> インターフェイスでは、@Workflow 注釈と @Activities 注釈の両方を同時に宣言できません。この 2 つは相互に排他的です。

この注釈では、以下のパラメータを指定できます。

dataConverter

このワークフロータイプのワークフロー実行に対してリクエストの送信と結果の受信を行う場合、どの DataConverter を使用するかを指定します。

デフォルトでは、NullDataConverter を使用します。これに伴って JsonDataConverter ですべてのリクエストおよびレスポンスデータが JavaScript Object Notation (JSON) として処理されます。

例

```
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 3600)
```

```
public interface GreeterWorkflow {
   @Execute(version = "1.0")
   public void greet();
}
```

@WorkflowRegistrationOptions

@Workflow 注釈を設定したインターフェイスで使用した場合は、ワークフロータイプの登録時に Amazon SWF で使用されるデフォルト設定を提供します。

> **Note**
>
> @Workflow 注釈を設定したインターフェイスでは、@WorkflowRegistrationOptions と @SkipRegistrationOptions のいずれかを使用します。両方を指定することはできません。

この注釈では、以下のパラメータを指定できます。

説明

ワークフロータイプのテキスト説明 (オプション)。

defaultExecutionStartToCloseTimeoutSeconds

このワークフロータイプに対して Amazon SWF に登録されている defaultExecutionStartToCloseTimeout を指定します。このタイプのワークフロー実行を完了するまでの総許容時間です。

ワークフローのタイムアウトの詳細については、「Amazon SWF タイムアウトの種類」を参照してください。

defaultTaskStartToCloseTimeoutSeconds

このワークフロータイプに対して Amazon SWF に登録されている defaultTaskStartToCloseTimeout を指定します。このタイプのワークフロー実行で 1 つの決定タスクを完了するまでの総許容時間を指定します。

defaultTaskStartToCloseTimeout を指定しない場合は、デフォルトで 30 秒になります。

ワークフローのタイムアウトの詳細については、「Amazon SWF タイムアウトの種類」を参照してください。

`defaultTaskList`

このワークフロータイプの実行で決定タスクに使用するデフォルトのタスクリスト。ここで設定したデフォルト値は、ワークフロー実行の開始時に `StartWorkflowOptions` を使用して上書きできます。

`defaultTaskList` を指定しない場合は、デフォルトで `USE_WORKER_TASK_LIST` に設定されます。これは、ワークフローの登録を担当するワーカーで使用されているタスクリストを使用することを指定します。

`defaultChildPolicy`

このタイプの実行が終了した場合に、子ワークフローで使用するポリシーを指定します。デフォルト値は `ABANDON` です。指定できる値は以下のとおりです。

- `ABANDON` – 子ワークフロー実行を続行できます
- `TERMINATE` – 子ワークフロー実行を終了します
- `REQUEST_CANCEL` – 子ワークフロー実行のキャンセルをリクエストします

AWS Flow Framework for Java の例外

for Java AWS Flow Framework では、次の例外が使用されます。このセクションでは、例外の概要を説明します。詳細については、個々の例外の AWS SDK for Java ドキュメントを参照してください。

トピック

- ActivityFailureException
- ActivityTaskException
- ActivityTaskFailedException
- ActivityTaskTimedOutException
- ChildWorkflowException
- ChildWorkflowFailedException
- ChildWorkflowTerminatedException
- ChildWorkflowTimedOutException
- DataConverterException
- DecisionException
- ScheduleActivityTaskFailedException

- SignalExternalWorkflowException
- StartChildWorkflowFailedException
- StartTimerFailedException
- TimerException
- WorkflowException

ActivityFailureException

この例外は、アクティビティの失敗を通知するためにフレームワークによって内部的に使用されます。例外が処理されないことが原因で、アクティビティに失敗した場合は、ActivityFailureException でラッピングされ、Amazon SWF にレポートされます。この例外は、アクティビティワーカー拡張ポイントを使用する場合にのみ、処理する必要があります。アプリケーションコードで、この例外を処理する必要はありません。

ActivityTaskException

アクティビティタスクの失敗の例外の基本クラスは次のとおりです: ScheduleActivityTaskFailedException、ActivityTaskFailedException、ActivityTaskTim
失敗したタスクのタスク ID およびアクティビティタスクタイプが含まれます。ワークフロー実装でこの例外をキャッチし、一般的な方法でアクティビティの失敗を処理できます。

ActivityTaskFailedException

アクティビティの処理されない例外は、ActivityTaskFailedException でスローして、ワークフロー実装にレポートされます。元の例外は、この例外の cause プロパティから取得できます。この例外には、履歴内の一意のアクティビティ識別子などのデバッグ目的に役立つその他の情報も含まれます。

このフレームワークで、リモート例外を実行するには、元の例外をアクティビティワーカーからシリアル化します。

ActivityTaskTimedOutException

この例外は、アクティビティが Amazon SWF によってタイムアウトされた場合にスローされます。これは、アクティビティタスクが所要時間内にワーカーに割り当てられなかった場合、またはその所要時間にワーカーが完了できなかった場合に発生することがあります。アクティビティのこれらのタイムアウトは、@ActivityRegistrationOptions 注釈、またはアクティビティメソッドを呼び出す際に、ActivitySchedulingOptions パラメータを使用して設定できます。

ChildWorkflowException

子ワークフロー実行の失敗を報告するために使用される例外の基本クラス。例外には、子ワークフロー実行の ID とそのワークフロータイプが含まれます。この例外をキャッチし、一般的な方法で子ワークフロー実行の失敗を処理できます。

ChildWorkflowFailedException

子ワークフローの処理されない例外は、`ChildWorkflowFailedException` でスローして、親ワークフロー実装にレポートされます。元の例外は、この例外の `cause` プロパティから取得できます。この例外には、子の実行の一意の識別子などのデバッグ目的に役立つその他の情報も含まれます。

ChildWorkflowTerminatedException

この例外は、親ワークフロー実行時にスローされ、子ワークフロー実行の終了を報告します。たとえば、クリーンアップまたは補償を実行するなど、子ワークフローの終了を処理する場合は、この例外をキャッチする必要があります。

ChildWorkflowTimedOutException

この例外は、親ワークフロー実行時にスローされ、子ワークフロー実行がタイムアウトし、Amazon SWF で閉じられたことを報告します。たとえば、クリーンアップまたは補償を実行するなど、子ワークフローを強制終了する場合は、この例外をキャッチする必要があります。

DataConverterException

フレームワークでは、`DataConverter` コンポーネントを使用して、送信されるデータをマーシャルおよびアンマーシャルできます。この例外は、`DataConverter` で、データをマーシャルまたはアンマーシャルできない場合にスローされます。これは、たとえば、データをマーシャルおよびアンマーシャルするために使用される `DataConverter` コンポーネントの不一致など、さまざまな理由で発生する可能性があります。

DecisionException

これは、Amazon SWF によってディシジョン適用の失敗を表す例外の基本クラスです。このような例外を一般的に処理するには、この例外をキャッチします。

ScheduleActivityTaskFailedException

この例外は、Amazon SWF でアクティビティタスクをスケジュール設定できない場合にスローされます。これは、さまざまな理由で発生する可能性があります。例えば、アクティビティが廃止された場合、アカウントの Amazon SWF 上限に達した場合などがあります。例外の `failureCause` プロパティは、失敗の正確な原因を指定して、アクティビティをスケジュール設定します。

SignalExternalWorkflowException

この例外は、ワークフロー実行によるリクエストを Amazon SWF で処理して、別のワークフロー実行を通知できない場合にスローされます。これは、対象ワークフローの実行が見つからない場合、つまり、指定したワークフロー実行が存在しないか、閉じた状態になっている場合に発生します。

StartChildWorkflowFailedException

この例外は、Amazon SWF で子ワークフロー実行を開始できない場合にスローされます。これは、さまざまな理由で発生する可能性があります。例えば、指定した子ワークフローのタイプが廃止された場合、アカウントの Amazon SWF 上限に達した場合などがあります。例外の `failureCause` プロパティは、失敗の正確な原因を指定して、子ワークフロー実行を開始します。

StartTimerFailedException

この例外は、ワークフロー実行でリクエストされたタイマーを Amazon SWF で開始できない場合にスローされます。これは、指定されたタイマー ID がすでに使用されているか、アカウントの Amazon SWF 上限に達している場合に発生する可能性があります。例外の `failureCause` プロパティは、失敗の正確な原因を指定します。

TimerException

これは、タイマーに関連する例外の基本クラスです。

WorkflowException

この例外は、ワークフロー実行の失敗を報告するためにフレームワークによって内部的に使用されます。この例外は、ワークフローワーカー拡張ポイントを使用している場合にのみ、処理する必要があります。

AWS Flow Framework for Java パッケージ

このセクションでは、 AWS Flow Framework for Java に含まれるパッケージの概要を説明します。各パッケージの詳細については、「AWS SDK for Java API リファレンス」の com.amazonaws.services.simpleworkflow.flow を参照してください。

com.amazonaws.services.simpleworkflow.flow

Amazon SWF と統合するコンポーネントが含まれています。

com.amazonaws.services.simpleworkflow.flow.annotations

AWS Flow Framework for Java プログラミングモデルで使用される注釈が含まれています。

com.amazonaws.services.simpleworkflow.flow.aspectj

@Asynchronous や などの機能に必要な AWS Flow Framework for Java コンポーネントが含まれています@ExponentialRetry。

com.amazonaws.services.simpleworkflow.flow.common

フレームワーク定義の定数などの一般的なユーティリティが含まれています。

com.amazonaws.services.simpleworkflow.flow.core

`Task` や `Promise` などのコア機能が含まれています。

com.amazonaws.services.simpleworkflow.flow.generic

他の機能を構築する基盤となるコアコンポーネント (汎用クライアントなど) が含まれています。

com.amazonaws.services.simpleworkflow.flow.interceptors

フレームワークが提供するデコレーター (`RetryDecorator` など) の実装が含まれています。

com.amazonaws.services.simpleworkflow.flow.junit

Junit の統合を提供するコンポーネントが含まれています。

com.amazonaws.services.simpleworkflow.flow.pojo

注釈ベースのプログラミングモデルのアクティビティおよびワークフロー定義を実装するクラスが含まれています。

com.amazonaws.services.simpleworkflow.flow.spring

Spring の統合を提供するコンポーネントが含まれています。

com.amazonaws.services.simpleworkflow.flow.test

ワークフロー実装の単体テスト用のヘルパークラス (TestWorkflowClock) が含まれています。

com.amazonaws.services.simpleworkflow.flow.worker

アクティビティワーカーとワークフローワーカーの実装が含まれています。

ドキュメント履歴

次の表に、「AWS Flow Framework for Java のデベロッパーガイド」の前回のリリース以後に行われた、ドキュメントの重要な変更を示します。

- API バージョン: 2012-01-25
- 文書の最終更新: 2018 年 6 月 25 日

変更	説明	変更日
更新	`@ExponentialRetry` の `backoffCoefficient` の説明のエラーを修正しました。「@ExponentialRetry」を参照してください。	2018 年 6 月 25 日
更新	このガイド全体でコード例をクリーンアップしました。	2017 年 6 月 5 日
更新	このガイドの構成や内容を簡素化し、改善しました。	2017 年 5 月 19 日
更新	「ディサイダーコードの変更: バージョニングと機能フラグ」セクションを簡素化および改善しました。	2017 年 4 月 10 日
更新	ディサイダーコードの変更に関する新しいガイダンスを盛り込んだ「ベストプラクティス」セクションを新しく追加しました。	2017 年 3 月 3 日
新機能	ワークフローで従来のアクティビティタスクに加えて Lambda タスクを指定できます。詳細については、「AWS Lambda タスクの実装」を参照してください。	2015 年 7 月 21 日
新機能	Amazon SWF では、タスクの優先順位を反映したタスクリストを設定し、タスクを優先順位の高い順に提供するように試みます。詳細については、「Amazon SWF でのタスク優先度の設定」を参照してください。	2014 年 12 月 17 日

変更	説明	変更日
更新	更新と修正を行いました。	2013 年 8 月 1 日
更新	• Eclipse4.3 および AWS SDK for Java 1.4.7 のセットアップ手順の更新を含む、更新と修正を行いました。 • スターターのシナリオを構築するための新しいチュートリアルのセットを追加しました。	2013 年 6 月 28 日
新機能	AWS Flow Framework for Java の初期リリース。	2012 年 2 月 27 日

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。