aws

开发人员指南

AWS Flow Framework 适用于 Java

API 版本 2021-04-28

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Flow Framework 适用于 Java: 开发人员指南

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商标和商业外观不得用于任何非 Amazon 的商品或服务，也不得以任何可能引起客户混淆或者贬低或诋毁 Amazon 的方式使用。所有非 Amazon 拥有的其他商标均为各自所有者的财产，这些所有者可能附属于 Amazon、与 Amazon 有关联或由 Amazon 赞助，也可能不是如此。

Table of Contents

Java 的 AWS Flow Framework 用法是什么？

借助 AWS Flow Framework，您可以专注于实现工作流程逻辑。在幕后，该框架使用 Amazon SWF 的调度、路由和状态管理功能来管理您的工作流程的执行并使其具有可扩展性、可靠性和可审计性。基于框架的工作流程高度并发。工作流可以分布在多个组件上，这些组件可以作为单独的进程在不同的计算机上运行，并且可以独立扩展。如果应用程序的任何组件在运行，则该应用程序可以继续运行，因此具有很高的容错能力。

本指南的内容?

本指南包含有关如何安装、设置和使用该框架构建 Amazon SWF 应用程序的信息。

开始使用适用于 Java 的 AWS Flow Framework

如果您刚开始使用 Java AWS Flow Framework 版，请阅读本开始使用适用于 Java 的 AWS Flow Framework节。它将指导您下载和安装 AWS Flow Framework 适用于 Java 的，如何设置开发环境，并引导您完成创建工作流程的简单示例。

适用于 Java 的 AWS Flow Framework 的工作原理

介绍基本的 Amazon SWF 和框架概念，描述框架应用程序的基本结构以及分布式工作流程各部分之间如何交换数据。

适用于 Java 的 AWS Flow Framework 编程指南

本章提供了有关使用适用于 Java 的 AWS Flow Framework 开发工作流应用程序的基本编程指南，包括如何注册活动和工作流类型、实施工作流客户端、创建子工作流、处理错误等。

深入剖析

本章更深入地介绍了 for Java 的工作方式，为您提供了有关异步工作流程执行顺序和标准工作流程执行的逻辑步骤的其他信息。 AWS Flow Framework

故障排除和调试提示

本章提供了有关常见错误的信息，这些信息可用于排查工作流程的问题或了解如何避免常见错误。

适用于 Java 的 AWS Flow Framework 参考

本章引用了 for Java 在 Java SDK 中添加 AWS Flow Framework 的注释、异常和软件包。

文档历史记录

本章提供有关对文档进行的主要更改的详细信息。此处列出了新的部分和主题以及大幅修改后的主题。

开始使用适用于 Java 的 AWS Flow Framework

本部分通过演练一系列介绍基本编程模型和 API 的示例应用程序来介绍 AWS Flow Framework。示例应用程序基于用于介绍 C 和相关编程语言的标准 Hello World 应用程序。下面是 Hello World 的典型 Java 实现：

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

以下是对示例应用程序的简要说明。它们包含完整源代码，因此您可以自行实现和运行应用程序。在开始之前，您应首先配置开发环境并创建一个适用于 Java 的 AWS Flow Framework 项目，如 设置适用于 Java 的 AWS Flow Framework 中所述。

- HelloWorld 应用程序通过将 Hello World 实现为标准 Java 应用程序，但使其结构类似工作流应用程序，来介绍工作流应用程序。
- HelloWorldWorkflow 应用程序 使用适用于 Java 的 AWS Flow Framework 将 HelloWorld 转换为 Amazon SWF 工作流。
- HelloWorldWorkflowAsync 应用程序 修改 `HelloWorldWorkflow` 以使用异步工作流方法。
- HelloWorldWorkflowDistributed 应用程序 修改 `HelloWorldWorkflowAsync`，使工作流和活动工作线程可在不同系统上运行。
- HelloWorldWorkflowParallel 应用程序 修改 `HelloWorldWorkflow` 以并行运行两个活动。

设置适用于 Java 的 AWS Flow Framework

适用于 Java 的 AWS Flow Framework 包含在AWS SDK for Java 中。如果您尚未设置适用于 Java 的 AWS SDK for Java，请访问《AWS SDK for Java 开发人员指南》中的 Getting Started，了解有关安装和配置该 SDK 的信息。

本主题提供了有关使用适用于 Java 的 AWS Flow Framework 所需的其他步骤的信息。这些步骤是为 Eclipse 和 Maven 提供的。

主题

- 为 Maven 安装

- 为 Eclipse 安装

为 Maven 安装

Amazon 在 Maven Central 存储库中提供了 Amazon SWF 构建工具，以帮助在您的 Maven 项目中设置适用于 Java 的 AWS Flow Framework。

要为 Maven 安装流程框架，请在项目的 pom.xml 文件中添加以下依赖项：

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-swf-build-tools</artifactId>
  <version>1.0</version>
</dependency>
```

Amazon SWF 构建工具是开源的，如需查看或下载代码或自行构建工具，请访问该存储库：https://github.com/aws/aws-swf-build-tools。

为 Eclipse 安装

如果您使用 Eclipse IDE，请使用 AWS Toolkit for Eclipse 安装适用于 Java的 AWS Flow Framework。

主题

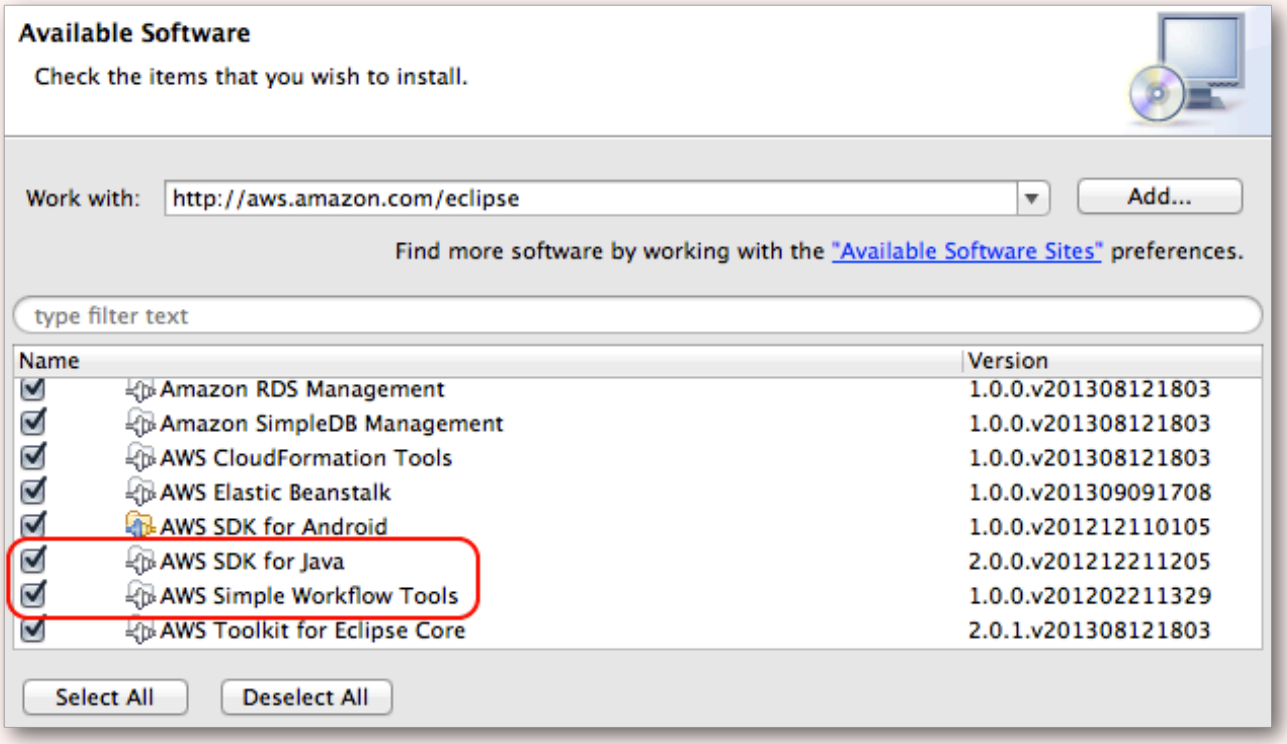
- 安装 AWS Toolkit for Eclipse
- 创建适用于 Java 的 AWS Flow Framework 项目

安装 AWS Toolkit for Eclipse

安装 Toolkit for Eclipse 是开始使用适用于 Java 的 AWS Flow Framework 的最简单方法。要安装 Toolkit for Eclipse，请参阅《AWS Toolkit for Eclipse 入门指南》中的 Setting Up the AWS Toolkit for Eclipse。

Important

在 Eclipse 的可用软件对话框中选择要安装的软件包时，请确保同时包含 AWS SDK for Java 和 AWS Simple Workflow Tools：



如果安装了所有 可用的程序包（选择 AWS Toolkit for Eclipse 顶级节点或选择 Select All (全选)），则会自动选择并安装这两个程序包。

创建适用于 Java 的 AWS Flow Framework 项目

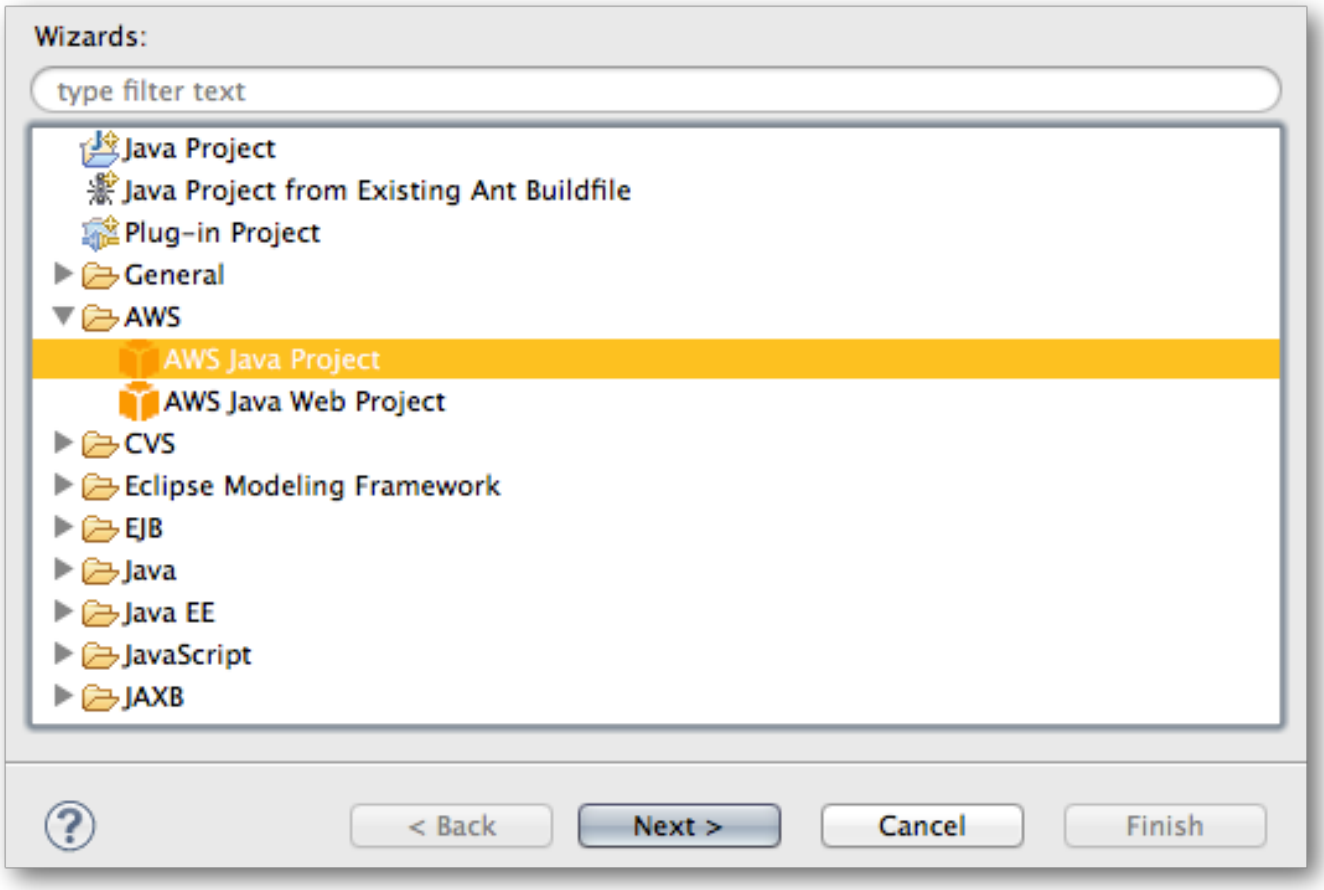
在 Eclipse 中创建正确配置的适用于 Java 的 AWS Flow Framework 项目包含多个步骤：

1. 创建适用于 Java 的 AWS 项目。
2. 为您的项目启用注释处理。
3. 启用并配置 AspectJ。

现在将详细介绍其中的每个步骤。

创建适用于 Java 的 AWS 项目。

1. 启动 Eclipse。
2. 要选择 Java 透视图，请选择 Window (窗口)、Open Perspective (打开透视图) 和 Java。
3. 依次选择文件、新建、适用于 Java 的 AWS项目。



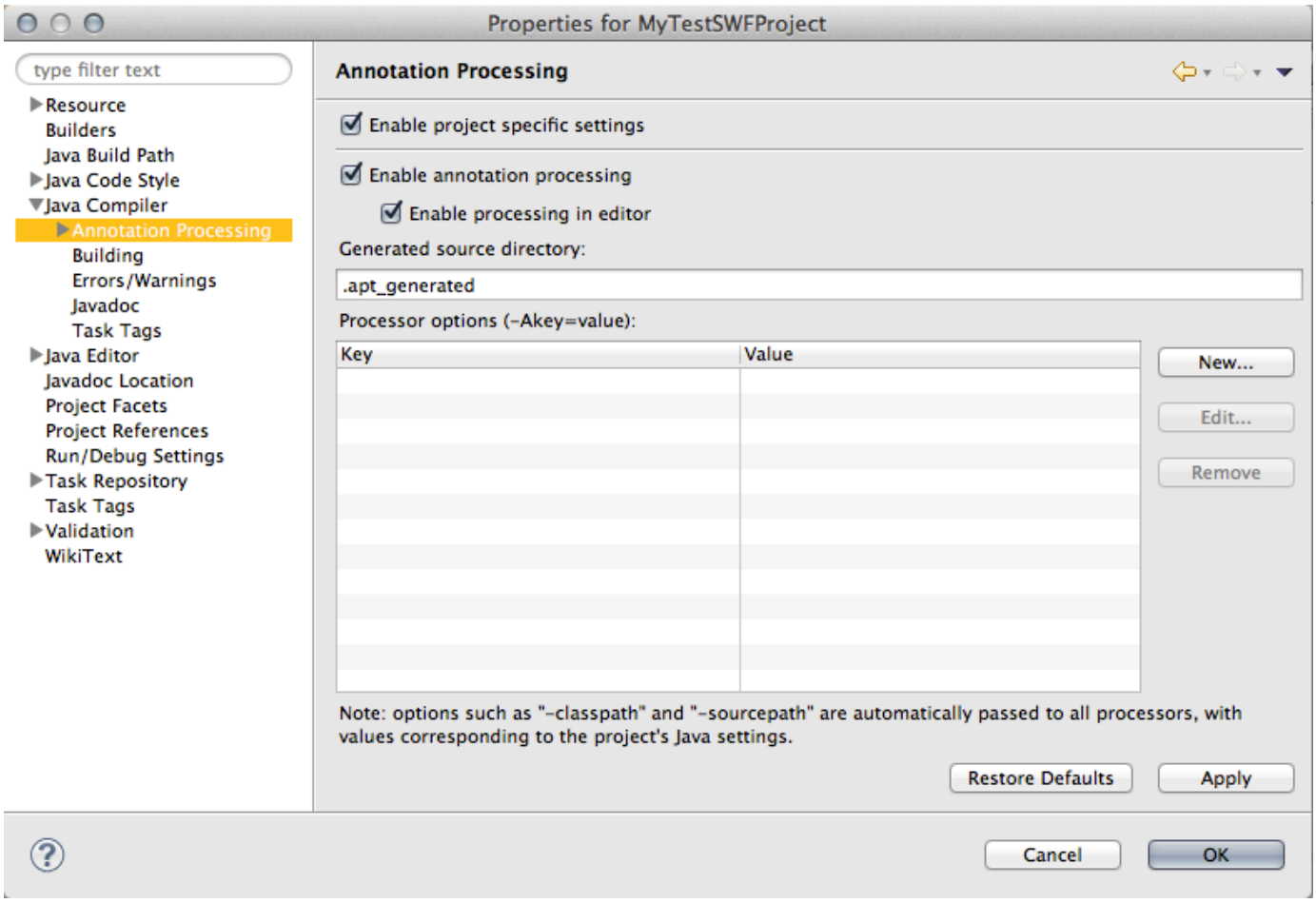
4. 使用适用于 Java 的 AWS 项目向导创建新的项目。

> **Note**
>
> 在您首次使用 Eclipse 创建适用于 Java 的 AWS 项目时，系统会在项目向导启动时自动下载并安装适用于 Java 的 SDK。

在创建适用于 Java 的 AWS 项目后，请为该项目启用注释处理。适用于 Java 的 AWS Flow Framework 包含一个注释处理器，可根据注释的源代码生成几个关键类。

启用注释处理

1. 在 Project Explorer (项目资源管理器) 中，右键单击您的项目，然后选择 Properties (属性)。
2. 在 Properties (属性) 对话框中，导航到 Java Compiler (Java 编译器) > Annotation Processing (注释处理)。
3. 选中 Enable project specific settings (启用项目特定的设置) (还会启用注释处理，但如果未启用，请确保还要选中该选项)。然后选择 OK（确定）。



> **Note**
>
> 在启用注释处理后，您需要重新构建您的项目。

启用并配置 AspectJ

接下来，您应该启用并配置 AspectJ。某些适用于 Java 的 AWS Flow Framework 注释（如 `@Asynchronous`）需要 AspectJ。您不需要直接使用 AspectJ，但必须使用加载时 织入或编译时 织入启用它。

> **Note**
>
> 建议的方法是使用加载时 织入。

主题

先决条件

在配置 AspectJ 之前，您需要使用与您的 Java 版本匹配的 AspectJ 版本：

- 如果使用 Java 8，请下载最新的 AspectJ 1.8.X 版本。
- 如果使用 Java 7，请下载最新的 AspectJ 1.7.X 版本。
- 如果使用 Java 6，请下载最新的 AspectJ 1.6.X 版本。

您可以从 Eclipse 下载页面下载其中的任一 AspectJ 版本。

在下载完 AspectJ 后，请执行下载的 `.jar` 文件以安装 AspectJ。AspectJ 安装将询问要在何处安装二进制文件，并在最终屏幕上提供建议的步骤以完成安装。记住 aspectjweaver.jar 文件的位置；您需要使用该文件在 Eclipse 中配置 AspectJ。

配置 AspectJ 加载时织入

要为适用于 Java 的 AWS Flow Framework 项目配置 AspectJ 加载时织入，首先要将 AspectJ JAR 文件指定为 Java 代理，然后通过将 `aop.xml` 文件添加到项目进行配置。

将 AspectJ 添加为 Java 代理

1. 要打开 Preferences (首选项) 对话框，请选择 Window (窗口) > Preferences (首选项)。

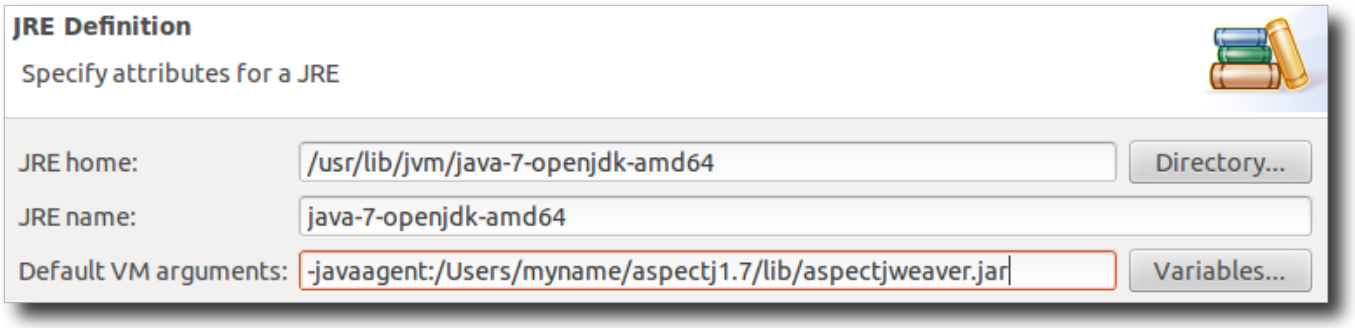
2. 导航到 Java > Installed JREs (安装的 JRE)。
3. 选择相应的 JRE，然后选择 Edit (编辑)。
4. 在 Default VM arguments (默认虚拟机参数) 框中，输入安装的 AspectJ 二进制文件的路径。这是一个路径 (如 `/home/user/aspectj1.7/lib/aspectjweaver.jar`)，具体取决于您的操作系统和下载的 AspectJ 版本。

   在 Linux、macOS 或 Unix 上，请使用：

   `-javaagent:`*`/your_path/aspectj/lib/aspectjweaver.jar`*

   在 Windows 上，请使用标准 Windows 格式的路径：

   `-javaagent:`*`C:\your_path\aspectj\lib\aspectjweaver.jar`*



要为适用于 Java 的 AWS Flow Framework 配置 AspectJ，请将 `aop.xml` 文件添加到项目。

添加 aop.xml 文件

1. 在项目的 src 目录中，添加一个名为 `META-INF` 的目录。
2. 在 `META-INF` 中添加一个名为 `aop.xml` 且包含以下内容的文件。

```
<aspectj>
   <aspects>
      <aspect
 name="com.amazonaws.services.simpleworkflow.flow.aspectj.AsynchronousAspect"/>
      <aspect
 name="com.amazonaws.services.simpleworkflow.flow.aspectj.ExponentialRetryAspect"/>
   </aspects>
   <weaver options="-verbose">
     <include within="MySimpleWorkflow.*"/>
   </weaver>
```

```
</aspectj>
```

`<include within=""/>` 值取决于命名项目的程序包的方式。以上示例假定项目的程序包采用 MySimpleWorkflow.* 形式。请使用适用于您自己的项目的程序包的值。

AspectJ 编译时织入

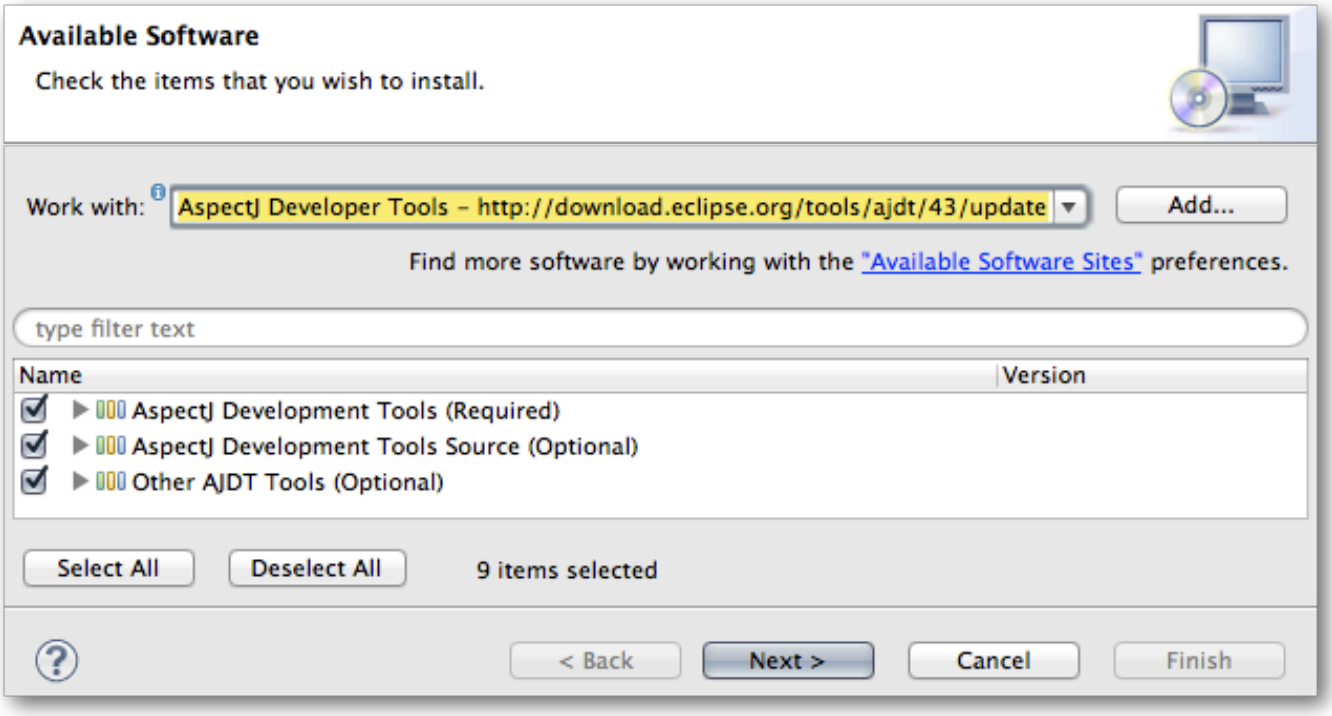
要启用并配置 AspectJ 编译时织入，您必须先安装适用于 Eclipse 的 AspectJ 开发人员工具，可以从 http://www.eclipse.org/aspectj/downloads.php 下载这些工具。

在 Eclipse 中安装 AspectJ 开发人员工具

1. 在 Help (帮助) 菜单中，选择 Install New Software (安装新软件)。
2. 在 Available Software (可用的软件) 对话框中，输入 `http://download.eclipse.org/tools/ajdt/`*`version`*`/dev/update`，其中 version 表示您的 Eclipse 版本号。例如，如果您使用的是 Eclipse 4.6，则应输入：`http://download.eclipse.org/tools/ajdt/`*`46`*`/dev/update`

   > **Important**
   >
   > 确保 AspectJ 版本与您的 Eclipse 版本匹配，否则，AspectJ 安装将失败。

3. 选择 Add (添加) 以添加位置。在添加位置后，将列出 AspectJ 开发人员工具。



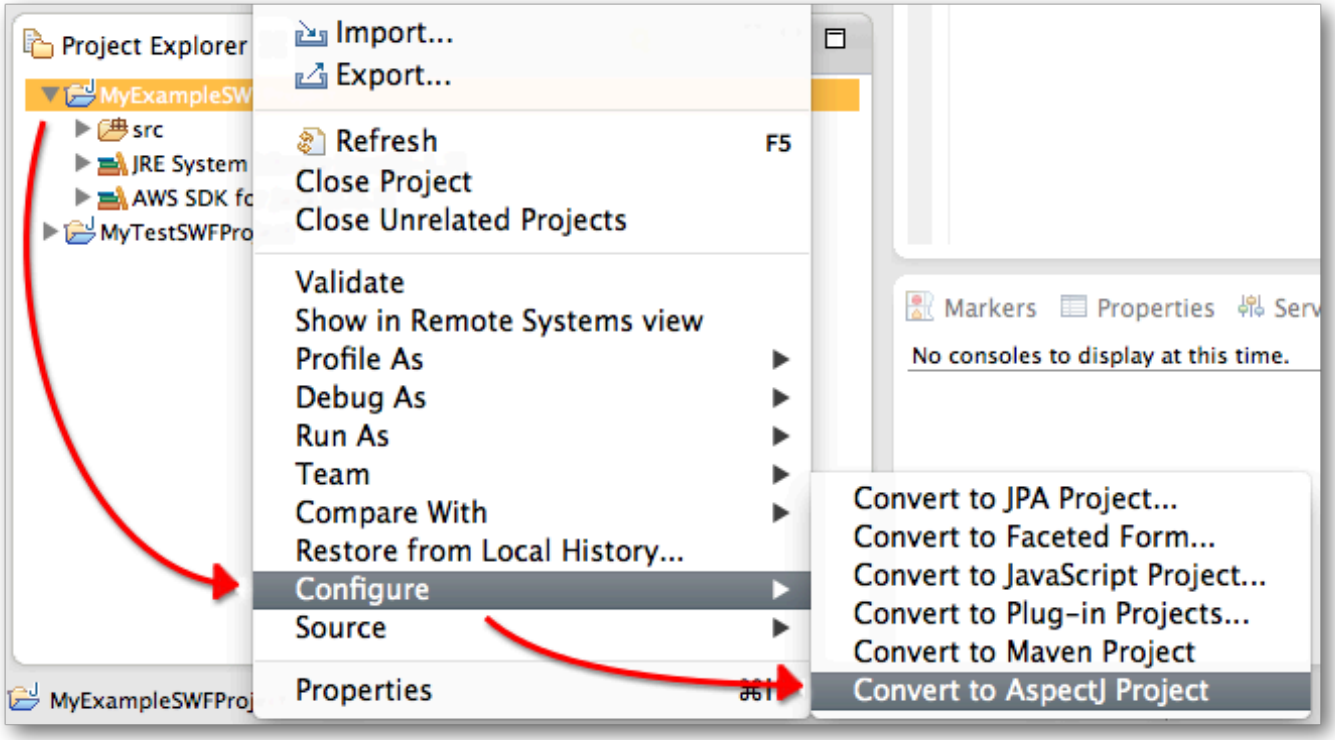
4. 选择 Select All (全选) 以选择所有 AspectJ 开发人员工具，然后选择 Next (下一步) 以安装这些工具。

> **Note**
>
> 您需要重新启动 Eclipse 以完成安装。

然后，必须配置您的项目。

为您的项目配置 AspectJ 编译时织入

1. 在 Project Explorer (项目资源管理器) 中，右键单击您的项目，然后选择 Configure (配置) > Convert to AspectJ Project (转换为 AspectJ 项目)。



将在您的项目中添加 AspectJ 运行时库。

2. 再次右键单击您的项目，然后选择 Properties (属性)。
3. 选择 AspectJ Build (AspectJ 构建)，然后选择 Aspect Path (Aspect 路径) 选项卡。
4. 选择 Add External JARs (添加外部 JAR)，然后将AWS SDK for Java JAR 文件添加到您的项目的 Aspect 路径。

> **Note**
>
> AWS Toolkit for Eclipse 会将AWS SDK for Java JAR 文件安装到工作区中的 `.metadata/.plugins/com.amazonaws.eclipse.core/aws-java-sdk/`*`AWS Version`*`/lib` 目录中，在该目录中，您将 *`AWS Version`* 替换为安装的 AWS SDK 版本号。否则，您可以使用常规 AWS SDK 安装中包含的 JAR 文件，它位于 `lib` 目录中。

解决 AspectJ 和 Eclipse 问题。

AspectJ Eclipse 插件存在一个问题，它可能妨碍编译生成的代码。要在重新编译后强制识别生成的代码，最快的方法是在 Java 生成路径设置页面的排序和导出选项卡上更改包含生成代码的源目录的顺序（例如，您可以将默认设置设为 `apt/java`）。

HelloWorld 应用程序

为了说明 Amazon SWF 应用程序的构建方式，我们将创建一个 Java 应用程序，它的行为与工作流类似，但在单个进程中本地运行。不需要连接到 Amazon Web Services。

> **Note**
>
> HelloWorldWorkflow 示例以该应用程序为基础，连接到 Amazon SWF 来处理工作流管理。

工作流程应用程序包含三个基本组件：

- 活动工作线程 支持一组活动，每个活动是一个独立执行的方法以执行特定的任务。
- 工作流程工作线程 协调活动执行并管理数据流。这是工作流程拓扑 的编程实现，这基本上是一个流程图，它定义了何时执行各种活动，这些活动是按顺序执行还是同时执行，等等。
- 工作流程启动程序 启动工作流程实例 (称为执行)，并且可以在执行期间与其进行交互。

HelloWorld 是作为三个类和两个相关接口实现的，如以下几节中所述。在启动之前，您应按照设置 AWS 中的说明设置开发环境并新建一个 设置适用于 Java 的 AWS Flow Framework Java 项目。用于以下演练的程序包名称均为 `helloWorld.`*`XYZ`*。要使用这些名称，请在 aop.xml 中设置 `within` 属性，如下所示：

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

要实现 HelloWorld，请在 AWS SDK 项目中创建一个名为 `helloWorld.HelloWorld` 的新 Java 软件包，并添加以下文件：

- 名为 `GreeterActivities.java` 的接口文件。
- 名为 `GreeterActivitiesImpl.java` 的类文件，它实现活动工作线程。
- 名为 `GreeterWorkflow.java` 的接口文件。
- 名为 `GreeterWorkflowImpl.java` 的类文件，它实现工作流程工作线程。
- 名为 `GreeterMain.java` 的类文件，它实现工作流程启动程序。

在以下几节中提供了详细信息，并包含每个组件的完整代码 (可添加到相应的文件中)。

HelloWorld 活动实现

HelloWorld 将在控制台中输出 "Hello World!" 问候语的整个任务拆分为三个任务，每个任务由一个活动方法 执行。活动方法是在 GreeterActivities 接口中定义的，如下所示。

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

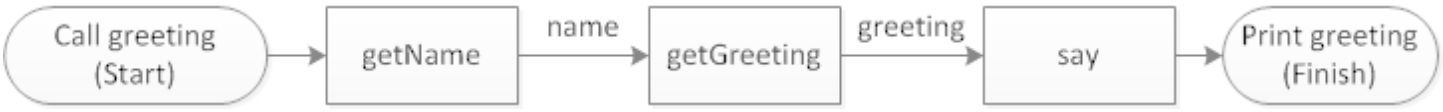
HelloWorld 具有一个活动实现 (GreeterActivitiesImpl)，它提供 GreeterActivities 方法，如下所示：

```
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);
   }
}
```

活动是相互独立的，通常可由不同的工作流程使用。例如，任何工作流程可以使用 say 活动将字符串输出到控制台。工作流程也可能具有多个活动实现，每个实现执行一组不同的任务。

HelloWorld 工作流程工作线程

要将“Hello World!” 输出到控制台，必须按正确的顺序使用正确的数据依次执行活动任务。HelloWorld 工作流程工作线程根据简单的线性工作流程拓扑 协调活动执行，如下图所示。



三个活动按顺序执行，并将数据从一个活动传输到下一个活动。

HelloWorld 工作流程工作线程具有单个方法 (工作流程的入口点)，它是在 GreeterWorkflow 接口中定义的，如下所示：

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

GreeterWorkflowImpl 类实现该接口，如下所示：

```
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 方法创建一个 GreeterActivitiesImpl 实例，按正确顺序调用每个活动方法，然后将相应数据传递到每个方法以实现 HelloWorld 拓扑。

HelloWorld 工作流程启动程序

工作流程启动程序 是一个启动工作流程执行的应用程序，可能还会在工作流程执行时与其进行通信。GreeterMain 类实现 HelloWorld 工作流程启动程序，如下所示：

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

GreeterMain 创建一个 GreeterWorkflowImpl 实例并调用 greet 以运行工作流程工作线程。将 GreeterMain 作为 Java 应用程序运行，您应该会在控制台输出中 看到“Hello World!”。

HelloWorldWorkflow 应用程序

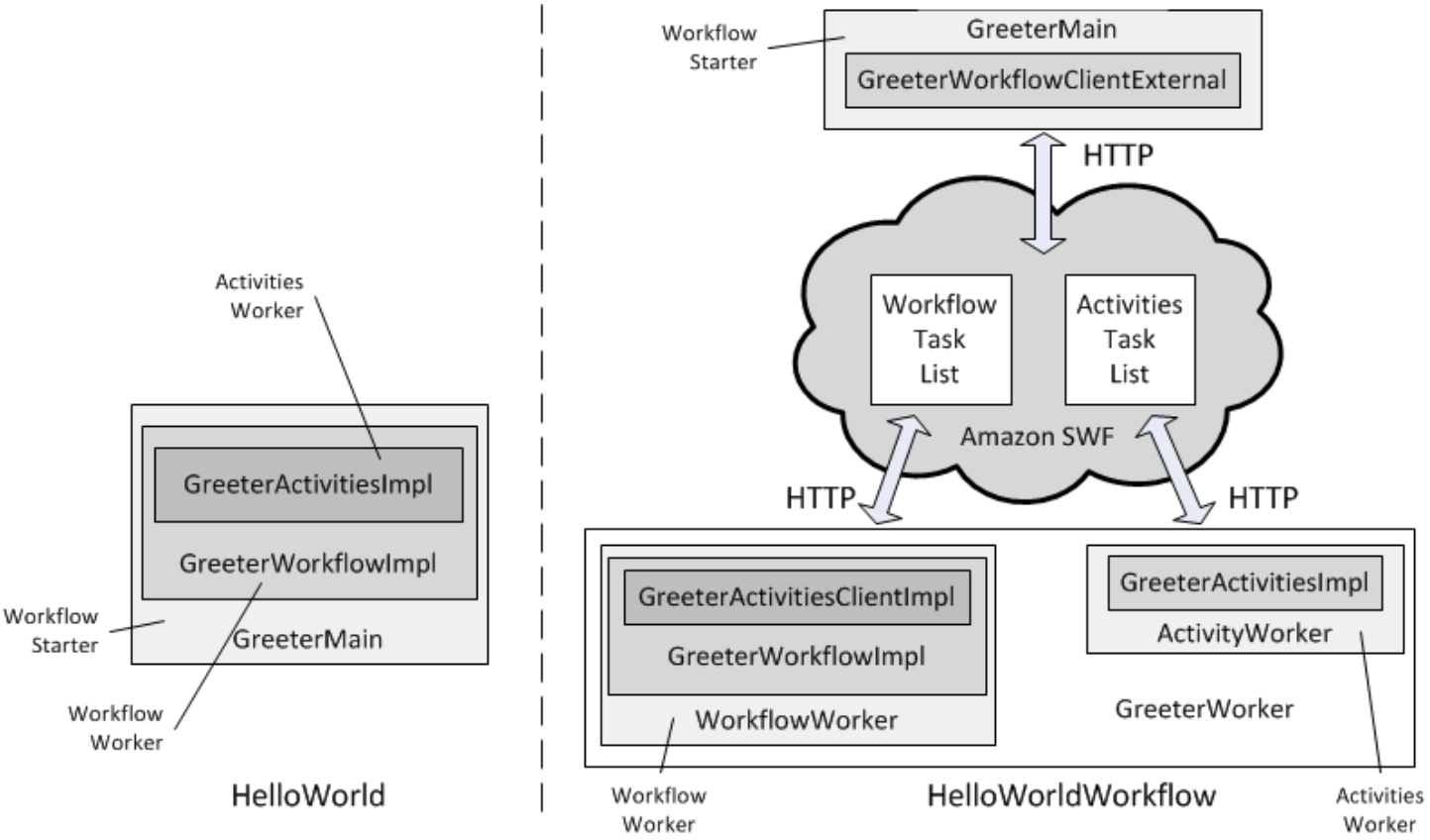
尽管基本HelloWorld示例的结构类似于工作流程，但它在几个关键方面与 Amazon SWF 工作流程有所不同：

传统工作流应用程序和 Amazon SWF 工作流应用程序

HelloWorld	Amazon SWF Workflow
在本地作为单个进程运行。	以多个进程的形式运行，可以分布在多个系统中，包括 Amazon EC2 实例、私有数据中心、客户端计算机等。这些进程甚至不必运行在相同的操作系统上。
活动是同步方法，在完成前保持阻止状态。	活动表示为异步方法，立即返回，并允许工作流程在等待活动完成时执行其他任务。
工作流程工作线程通过调用适当的方法与活动工作线程交互。	工作流工作线程使用 HTTP 请求与活动工作线程交互，而 Amazon SWF 则充当中介。
工作流程启动程序通过调用相应的方法与工作流程工作线程交互。	工作流启动程序使用 HTTP 请求与工作流工作线程交互，而 Amazon SWF 则充当中介。

您可以从头开始实施分布式异步工作流程应用程序，例如，让您的工作流程工作线程直接通过 Web 服务调用与活动工作线程交互。不过，随后您必须实施管理异步执行多个活动、处理数据流等所需的所有复杂代码。f AWS Flow Framework or Java 和 Amazon SWF 负责所有这些细节，这使您可以专注于实现业务逻辑。

HelloWorldWorkflow 是作为 Amazon SWF 工作流程运行的修改版。 HelloWorld 下图总结了两个应用程序的工作原理。



HelloWorld 作为单个进程运行，启动者、工作流程工作人员和活动工作者使用传统的方法调用进行交互。对于 HelloWorldWorkflow，启动程序、工作流工作线程和活动工作线程是分布式组件，通过 Amazon SWF 使用 HTTP 请求进行交互。Amazon SWF 通过维护工作流和活动任务列表来管理交互，并将这些工作流和任务分派给相应的组件。本节介绍该框架的工作原理 HelloWorldWorkflow。

HelloWorldWorkflow 是通过使用 for Java API 实现的，该API可以处理有时在后台与 Amazon SWF 交互的复杂细节，并大大简化了开发过程。 AWS Flow Framework 您可以使用与您为之相同的项目 HelloWorld，该项目已针对 Java 应用程序进行了配置。 AWS Flow Framework 但是，要运行该应用程序，您必须按如下所示设置 Amazon SWF 账户：

- 如果您还没有 AWS 账户，请在 Amazon Web Services 上注册一个账户。
- 将您的账户的访问 ID 和密钥 ID 分别分配到 AWS_ACCESS_KEY_ID 和 AWS_SECRET_KEY 环境变量。最好不要在代码中暴露文本密钥值。将它们存储在环境变量中是一种方便处理问题的方法。
- 在 Amazon Simple Workflow Service 注册 Amazon SWF 账户。
- 登录 AWS Management Console 并选择 Amazon SWF 服务。
- 选择右上角的管理域，然后注册一个新的 Amazon SWF 域。域 是一个应用程序资源 (例如工作流程和活动类型) 和工作流程执行的逻辑容器。您可以使用任何方便的域名，但演练使用“helloWorldWalkthrough”。

要实现 HelloWorldWorkflow，请创建 HelloWorld 的副本。 HelloWorld 打包到你的项目目录中然后把它命名为 HelloWorld。 HelloWorldWorkflow。以下各节介绍如何修改原始 HelloWorld代码以使用 AWS Flow Framework 适用于 Java 并作为 Amazon SWF 工作流程应用程序运行。

HelloWorldWorkflow 活动工作者

HelloWorld 将其活动工人作为一个班级实施。 AWS Flow Framework 适用于 Java 的活动工作程序有三个基本组件：

- 活动方法：用于执行实际任务，在接口中定义，并在相关类中实现。
- ActivityWorker类管理活动方法与 Amazon SWF 之间的交互。
- 活动宿主 应用程序注册并启动活动工作线程，以及处理清除操作。

此部分讨论活动方法；另外两个类在后文中讨论。

HelloWorldWorkflow 定义了中的活动接口`GreeterActivities`，如下所示：

```
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);
}
```

这个接口并不是绝对必要的 HelloWorld，但它是用于 Java AWS Flow Framework 的应用程序的。请注意，接口定义本身并未更改。但是，必须将 Java 标注@ActivityRegistrationOptions和@活动应用于接口定义。 AWS Flow Framework 这些注解提供配置信息，并指 AWS Flow Framework 示 Java 注解处理器使用接口定义生成活动客户端类，稍后将对此进行讨论。

`@ActivityRegistrationOptions`有多个命名值，用于配置活动的行为。 HelloWorldWorkflow 指定了两个超时：

- defaultTaskScheduleToStartTimeoutSeconds 指定任务可以在活动任务列表中排队的时间长度，设置为 300 秒 (5 分钟)。
- defaultTaskStartToCloseTimeoutSeconds 指定活动执行任务可以使用的最长时间，设置为 10 秒。

这些超时确保活动在合理的时间范围内完成其任务。如果超过了任何超时，该框架会生成错误，工作流程工作线程必须决定如何处理问题。有关如何处理此类错误的讨论，请参阅错误处理。

@Activities 有多个值，不过通常仅指定活动的版本号，您可以通过该版本号来跟踪活动实施的不同代。如果您在将活动接口注册到 Amazon SWF 之后对其进行了更改，包括更改 @ActivityRegistrationOptions 值，那么您必须使用新版本号。

HelloWorldWorkflow 在中实现活动方法GreeterActivitiesImpl，如下所示：

```
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);
   }
}
```

请注意，该代码与 HelloWorld 实现相同。从本质上讲， AWS Flow Framework 活动只是一种执行某些代码并可能返回结果的方法。标准应用程序与 Amazon SWF 工作流应用程序的区别在于工作流如何执行活动、在哪里执行活动以及如何将结果返回到工作流工作线程。

HelloWorldWorkflow 工作流程工作者

Amazon SWF 工作流工作线程包含三个基本组件。

- 工作流程实施，这是执行与工作流程相关任务的类。
- 活动客户端 类，这基本上活动类的代理，由工作流程实施用于异步执行活动方法。

- 一个WorkflowWorker类，用于管理工作流程与 Amazon SWF 之间的交互。

此部分讨论工作流程实施和活动客户端；WorkflowWorker 类将在后文讨论。

HelloWorldWorkflow 在中定义了工作流界面GreeterWorkflow，如下所示：

```
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();
}
```

对于适用于 Java 的应用程序来说，这个接口也不是必需的， HelloWorld AWS Flow Framework 但却是必不可少的。必须将 Java AWS Flow Framework 标注@工作流和@WorkflowRegistrationOptions应用于工作流界面定义。这些注解提供了配置信息，还指示 AWS Flow Framework Java 注解处理器根据接口生成工作流客户端类，如后面所述。

@Workflow有一个可选参数 DataConvert er，该参数通常与其默认值一起使用 NullDataConverter，表示应使用默认值。 JsonDataConverter

@WorkflowRegistrationOptions 还有多个可选的参数，这些参数可用于配置工作流程工作线程。在这里，我们将 defaultExecutionStartToCloseTimeoutSeconds（指定工作流的运行时间）设置为 3600 秒（1 小时）。

GreeterWorkflow接口定义与@Execute注释 HelloWorld 的不同之处在于一个重要方面。工作流程接口指定可由应用程序调用的方法，例如工作流程启动程序，并且限制为几个方法，每种方法具有特定角色。该框架不指定工作流接口方法的名称或参数列表；您可以使用适合您工作流的名称和参数列表，并应用适用于 Java 的 AWS Flow Framework 注释来标识方法的角色。

@Execute 有两个用途：

- 它确定 greet 作为工作流程的入口点，也就是工作流程启动程序调用以启动工作流程的方法。通常，入口点可以接受一个或多个参数，这使得启动程序可以初始化工作流程，但本示例中无需初始化。

- 它指定工作流程的版本号，通过该版本号可以跟踪工作流程实施的不同代。要在将工作流接口注册到 Amazon SWF 之后对其进行更改（包括更改超时值），您必须使用新版本号。

有关可包括在工作流程接口中的其他方法的信息，请参阅工作流和活动合同。

HelloWorldWorkflow 在中实现工作流程GreeterWorkflowImpl，如下所示：

```
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);
   }
}
```

该代码与类似 HelloWorld，但有两个重要区别。

- GreeterWorkflowImpl 创建活动客户端 GreeterActivitiesClientImpl 的实例，而不是 GreeterActivitiesImpl，并通过在客户端对象上调用方法来执行活动。
- 命名和问候活动返回 Promise<String> 对象而不是 String 对象。

HelloWorld 是一个作为单个进程在本地运行的标准 Java 应用程序，因此GreeterWorkflowImpl只需创建的实例GreeterActivitiesImpl、按顺序调用方法并将返回值从一个活动传递到下一个活动即可实现工作流拓扑。使用 Amazon SWF 工作流，活动的任务仍由来自 GreeterActivitiesImpl 的活动方法执行。但是，该方法不一定要在与工作流程相同的进程中运行，甚至可以运行在不同系统上，工作流程需要异步执行活动。这些要求引发了下列问题：

- 如何执行可能运行在不同进程中 (也有可能运行在不同系统上) 的活动方法。
- 如何异步执行活动方法。
- 如何管理活动的输入和返回值。例如，如果活动 A 返回的值是活动 B 的输入，您必须确保活动 B 不会在活动 A 完成之前执行。

您可以使用熟悉的 Java 流程控制并与活动客户端和 Promise<T> 相结合，通过应用程序的控制流程实施各种工作流程拓扑。

活动客户端

GreeterActivitiesClientImpl 基本上是 GreeterActivitiesImpl 的代理，允许工作流程实施异步执行 GreeterActivitiesImpl 方法。

使用应用到您的 GreeterActivities 类的注释中提供的信息，自动为您生成 GreeterActivitiesClient 和 GreeterActivitiesClientImpl 类。您无需自行实施这些内容。

> **Note**
>
> Eclipse 会在您保存项目时生成这些类。您可在项目目录的 .apt_generated 子目录中查看生成的代码。
> 为避免 GreeterWorkflowImpl 类中出现编译错误，最好将 .apt_generated 目录移至 Java 生成路径对话框的排序和导出选项卡顶部。

工作流程工作线程通过调用对应的客户端方法来执行活动。该方法异步执行，并立即返回 Promise<T> 对象，其中 T 是活动的返回类型。返回的 Promise<T> 对象基本上是活动方法最终将返回的值的占位符。

- 当活动客户端方法返回时，Promise<T> 对象最初处于未就绪状态，这表示对象尚未提供有效返回值。
- 当对应的活动方法完成其任务并返回时，该框架会将返回值分配给 Promise<T> 对象并将其置于就绪状态。

承诺 <T> 类型

Promise<T> 对象的主要用途是管理异步组件之间的数据流并控制什么时候执行。它使得应用程序无需明确管理同步或依赖于计时器等机制来确保异步组件不会过早执行。当您调用活动客户端方法时，它会立即返回，但框架会延迟对应活动方法的执行，直至输入 Promise<T> 对象已就绪并提供了有效值。

从 GreeterWorkflowImpl 的角度，全部三个活动客户端方法立即返回。从 GreeterActivitiesImpl 的角度，框架在 name 完成前不会调用 getGreeting，并且在 getGreeting 完成前不会调用 say。

通过使用 Promise<T> 将数据从一个活动传递到另一个，HelloWorldWorkflow 不仅确保活动方法不会尝试使用无效数据，还会控制执行活动的时间并隐式定义工作流程拓扑。将每个活动

的 Promise<T> 返回值传递到下一个活动时，需要按顺序执行活动，后文中将讨论定义线性拓扑。使用 f AWS Flow Framework or Java，您无需使用任何特殊的建模代码来定义甚至复杂的拓扑，只需使用标准的 Java 流量控制和Promise<T>。有关如何实施简单并行拓扑的示例，请参阅HelloWorldWorkflowParallel 活动工作线程。

> **Note**
>
> 当 say 等活动方法未返回值时，对应的客户端方法会返回 Promise<Void> 对象。对象不提供数据，不过最初它处于未就绪状态，并在活动完成后处于就绪状态。因此，您可以将 Promise<Void> 对象传递到其他活动客户端方法，确保它们延迟执行直至原始活动完成。

Promise<T> 允许工作流程实施使用活动客户端方法及其返回值，这类似于同步方法。但是，对于访问 Promise<T> 对象的值，您务必保持谨慎。与 Java Future<T> 类型不同，该框架处理 Promise<T> 而非应用程序的同步。如果您调用 Promise<T>.get，但对象未准备就绪，则 get 会引发异常。请注意，HelloWorldWorkflow 从不直接访问 Promise<T> 对象；它仅将对象从一个活动传递到下一个。在对象就绪之后，框架提取值并将其作为标准类型传递到活动方法。

Promise<T> 对象仅应由异步代码访问，其中框架确保对象已就绪并提供了有效值。HelloWorldWorkflow 通过仅将 Promise<T> 对象传递到活动客户端方法来处理此问题。您可以在工作流实现中访问 对象的值，方法是将该对象传递给异步工作流方法，该方法的行为与活动非常相似。有关示例，请参阅HelloWorldWorkflowAsync 应用程序。

HelloWorldWorkflow 工作流程和活动实施

工作流程和活动实现具有关联的工作人员类ActivityWorker和WorkflowWorker。它们通过轮询相应 Amazon SWF 任务列表中以获取任务、为每个任务执行相应的方法以及管理数据流，来处理 Amazon SWF 与活动和工作流实现之间的通信。有关详细信息，请参阅 AWS Flow Framework 基本概念：应用程序结构

要将活动和工作流程实施与对应的工作线程对象关联，您需要实施一个或多个工作线程应用程序，以便：

- 将工作流或活动注册到 Amazon SWF。
- 创建工作线程对象并将这些对象与工作流程或活动工作线程实施关联。
- 指示工作线程对象开始与 Amazon SWF 通信。

如果您希望将工作流程和活动作为单独的进程运行，则必须实施单独的工作流程和活动工作线程宿主。有关示例，请参阅HelloWorldWorkflowDistributed 应用程序。为简单起见， HelloWorldWorkflow实现了在同一进程中运行活动和工作流工作人员的单个工作主机，如下所示：

```
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();
   }
}
```

GreeterWorker没有 HelloWorld 对应的类，因此必须GreeterWorker向项目中添加一个名为的 Java 类，然后将示例代码复制到该文件中。

第一步是创建和配置一个AmazonSimpleWorkflowClient对象，该对象调用底层 Amazon SWF 服务方法。为此，GreeterWorker 将会：

1. 创建一个ClientConfiguration对象并将套接字超时指定为 70 秒。此值指定在已建立的开放连接上等待数据传输的时间长度，超过该时间将关闭套接字。
2. 创建一个 B asic AWSCredentials 对象来标识 AWS 账户，并将账户密钥传递给构造函数。为方便起见以及避免在代码中以纯文本格式暴露密钥，密钥作为环境变量存储。
3. 创建一个AmazonSimpleWorkflowClient对象来表示工作流程，并将`BasicAWSCredentials`和`ClientConfiguration`对象传递给构造函数。
4. 设置客户端对象的服务端点 URL。Amazon SWF 目前已在所有 AWS 地区上市。

为方便起见，`GreeterWorker` 定义两个字符串常量。

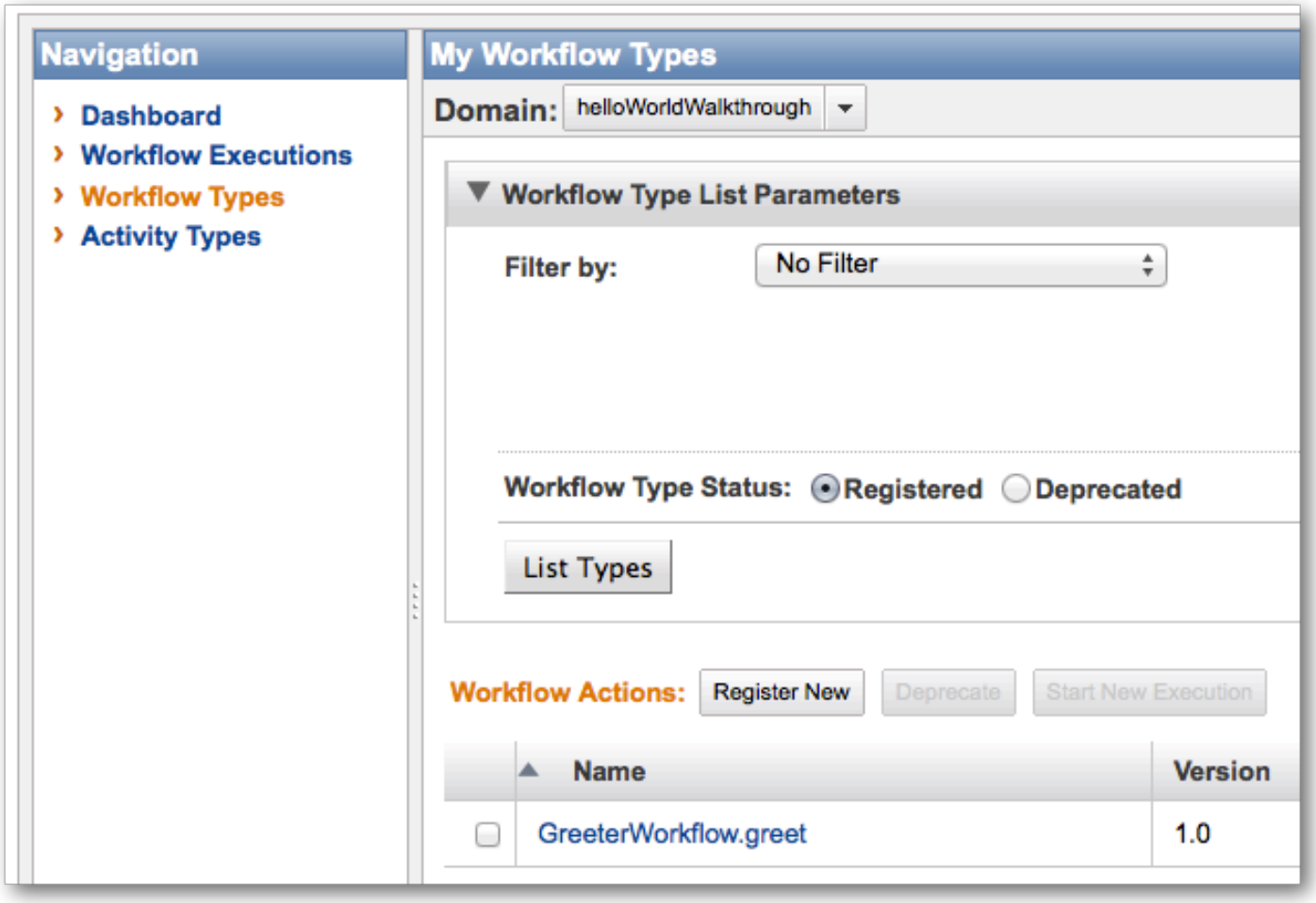
- `domain`是工作流程的 Amazon SWF 域名，它是您在设置亚马逊 SWF 账户时创建的。`HelloWorldWorkflow`假设您正在 “helloWorldWalkthrough” 域中运行工作流程。
- `taskListToPoll` 是 Amazon SWF 用来管理工作流与活动工作线程之间的通信的任务列表的名称。您可以将名称设置为任意方便的字符串。 HelloWorldWorkflow 在工作流和活动任务列表中都使用 HelloWorldList “”。在后台，这些名称对应着不同的命名空间，因此两个任务列表是不同的。

`GreeterWorker`使用字符串常量和对象创建工作线程AmazonSimpleWorkflowClient对象，这些对象管理活动和工作程序实现与 Amazon SWF 之间的交互。具体而言，工作线程对象处理轮询相应任务列表以查找任务的工作。

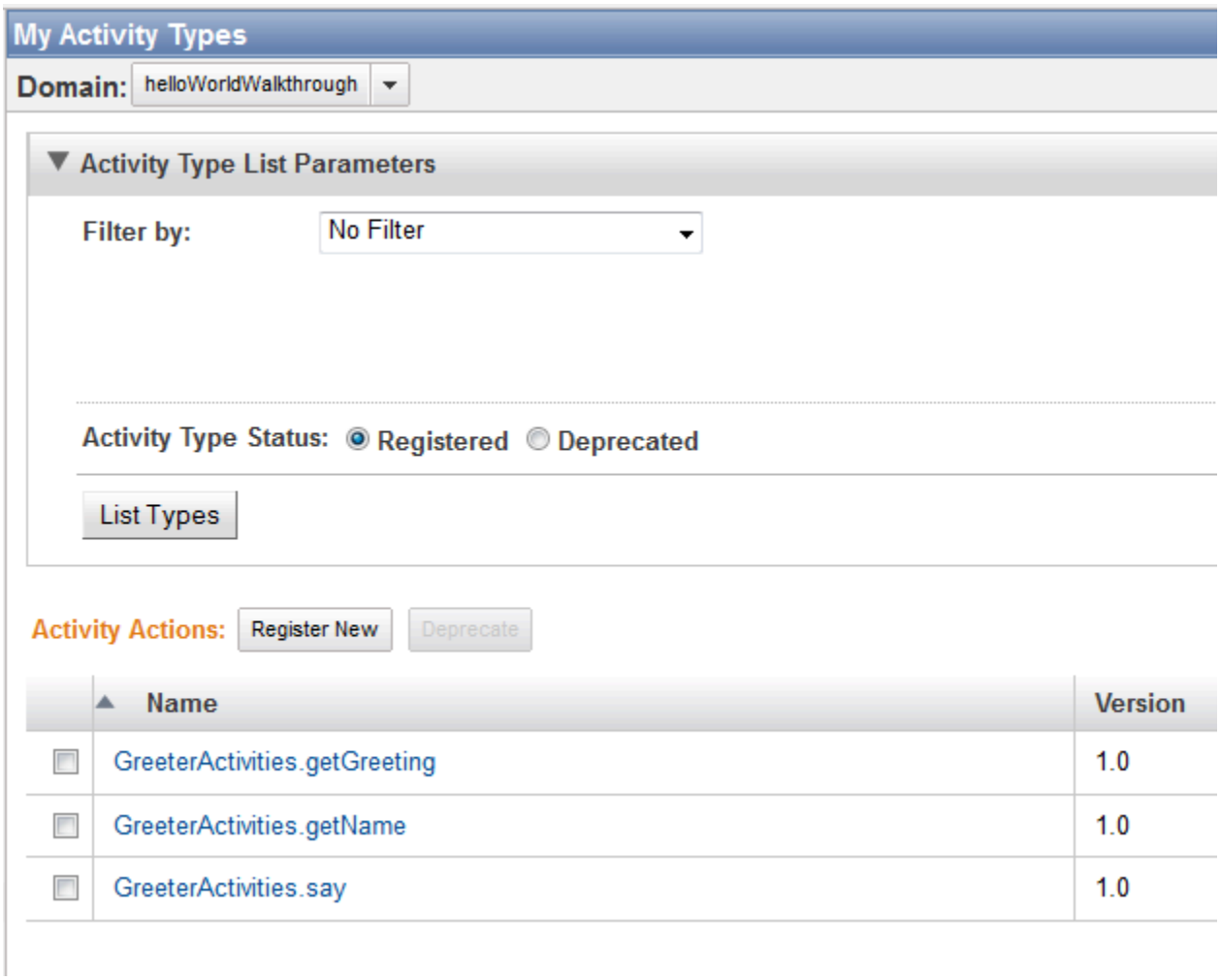
`GreeterWorker` 创建 `ActivityWorker` 对象，并通过添加新的类实例将其配置为处理 `GreeterActivitiesImpl`。`GreeterWorker` 然后调用 `ActivityWorker` 对象的 `start` 方法，该方法会指导对象开始轮询指定的活动任务列表。

`GreeterWorker` 创建 `WorkflowWorker` 对象，并通过添加类文件名 `GreeterWorkflowImpl.class` 将其配置为处理 `GreeterWorkflowImpl`。然后，它会调用 `WorkflowWorker` 对象的 `start` 方法，这会指导对象开始轮询指定的工作流程任务列表。

此时您可以成功运行 `GreeterWorker`。它将工作流和活动注册到 Amazon SWF，并启动工作线程对象来轮询其相应的任务列表。要验证这一点，请运行 `GreeterWorker` 并转到 Amazon SWF 控制台，然后从域列表中选择 `helloWorldWalkthrough`。如果在导航窗格中选择工作流类型，您应该会看到 `GreeterWorkflow.greet`：



如果选择 Activity Types (活动类型)，则将显示 `GreeterActivities` 方法：



不过，如果您选择 Workflow Executions (工作流程执行)，则不会看到任何处于活动状态的执行。虽然工作流程和活动工作线程正在轮询任务，但我们尚未启动工作流程执行。

HelloWorldWorkflow 入门

最后一道难题是实施工作流程启动程序，这是启动工作流程执行的应用程序。执行状态由 Amazon SWF 存储，因此您可以查看其历史记录和执行状态。 HelloWorldWorkflow 通过修改`GreeterMain`类来实现工作流程启动器，如下所示：

```
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 使用与 GreeterWorker 相同的代码创建 AmazonSimpleWorkflowClient 对象。然后创建 GreeterWorkflowClientExternal 对象，该对象用作工作流程的代理，其方式与在 GreeterWorkflowClientImpl 中创建的活动客户端用作活动方法的代理大致相同。您无需使用 new 创建工作流程客户端对象，而是需要：

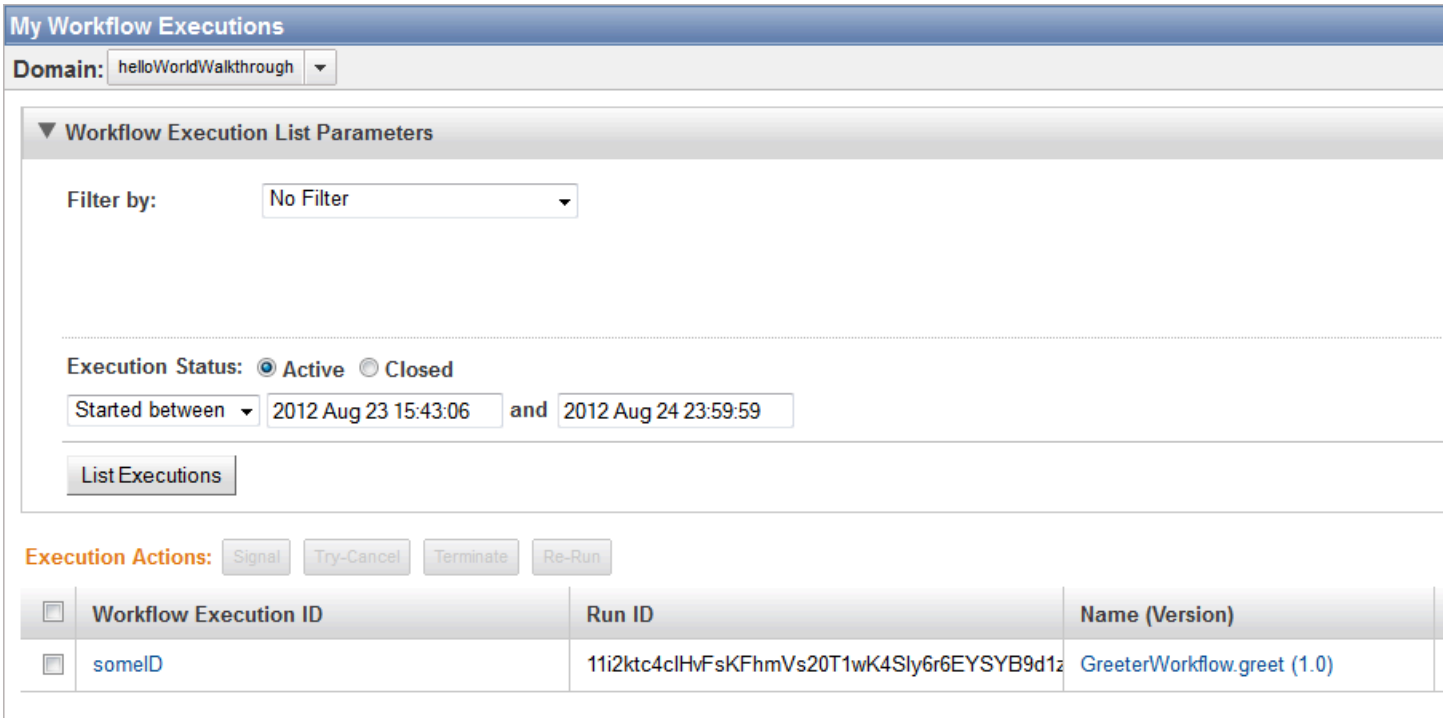
1. 创建外部客户端工厂对象，并将 AmazonSimpleWorkflowClient 对象和 Amazon SWF 域名传递到构造函数。客户端工厂对象由框架的注释处理器创建，它只需在工作流程接口名称后附加 “ClientExternalFactoryImpl” 即可创建对象名称。
2. 通过调用工厂对象的getClient方法来创建外部客户端对象，该方法通过在工作流程接口名称后附加 ClientExternal “” 来创建对象名称。您可以选择向 getClient 传递一个字符串，Amazon SWF 将使用该字符串来标识此工作流实例。否则，Amazon SWF 将使用生成的 GUID 来表示工作流实例。

从工厂返回的客户端将仅创建以传入 getClient 方法的字符串命名的工作流（从工厂返回的客户端在 Amazon SWF 中已拥有状态）。要运行具有不同 ID 的工作流程，您需要返回到工厂并使用指定的不同 ID 创建新客户端。

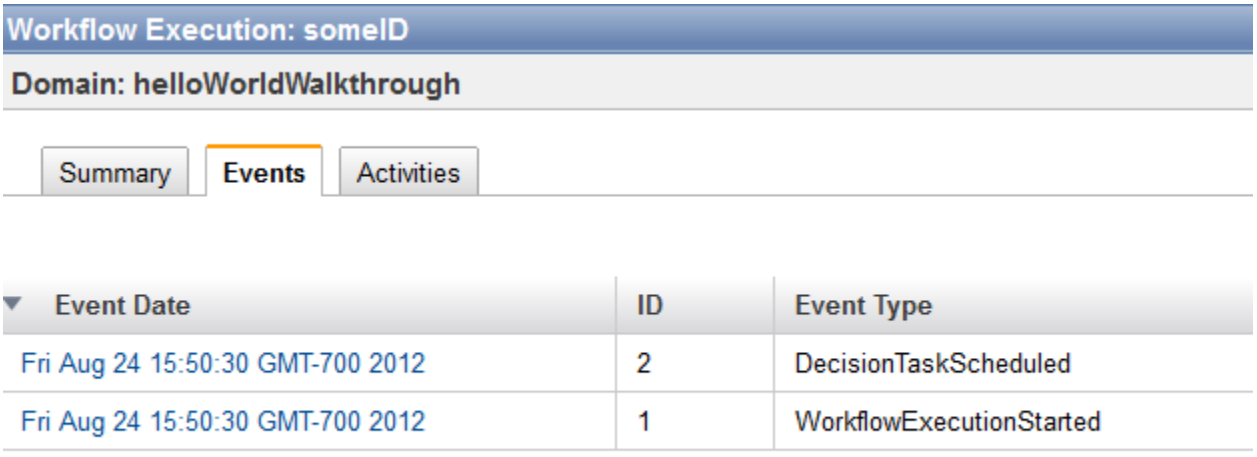
工作流程客户端公开 greet 方法，GreeterMain 调用该方法来启动工作流程，因为 greet() 是使用 @Execute 注释指定的方法。

> **Note**
>
> 注释处理器还会创建用于创建子工作流程的内部客户端工厂对象。有关更多信息，请参阅 子工作流程执行。

如果当前 `GreeterWorker` 仍在运行，则关闭它，并运行 `GreeterMain`。现在，您应该可以在 Amazon SWF 控制台的活动工作流执行列表中看到 someID：



如果您选择 `someID` 并选择 Events (事件) 选项卡，则将显示事件：

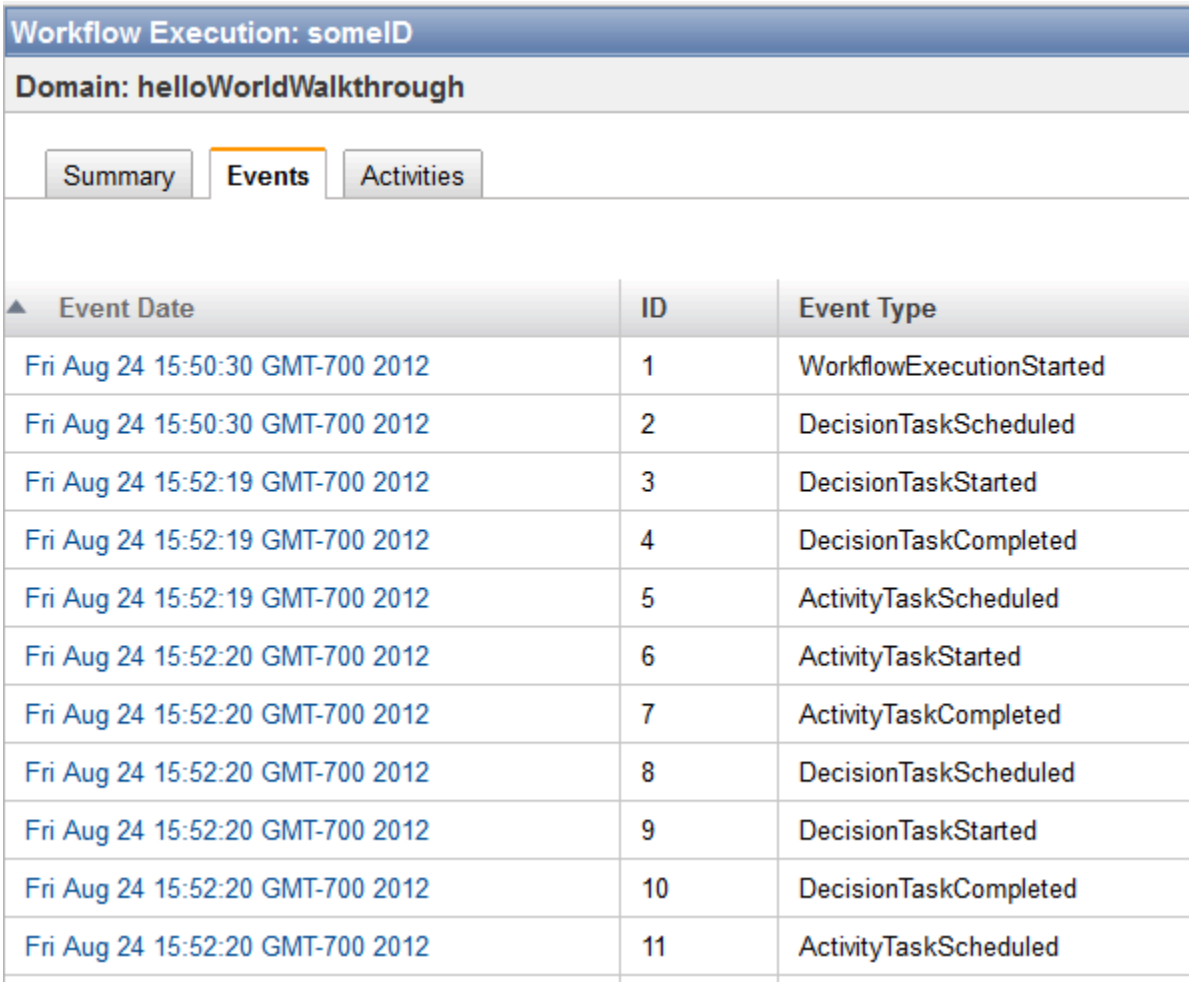


> **Note**
>
> 如果您之前启动了 GreeterWorker 并且它仍在运行，则会看到较长的事件列表，其原因刚刚讨论过。停止 GreeterWorker 并尝试重新运行 GreaterMain。

Events (事件) 选项卡仅显示两个事件：

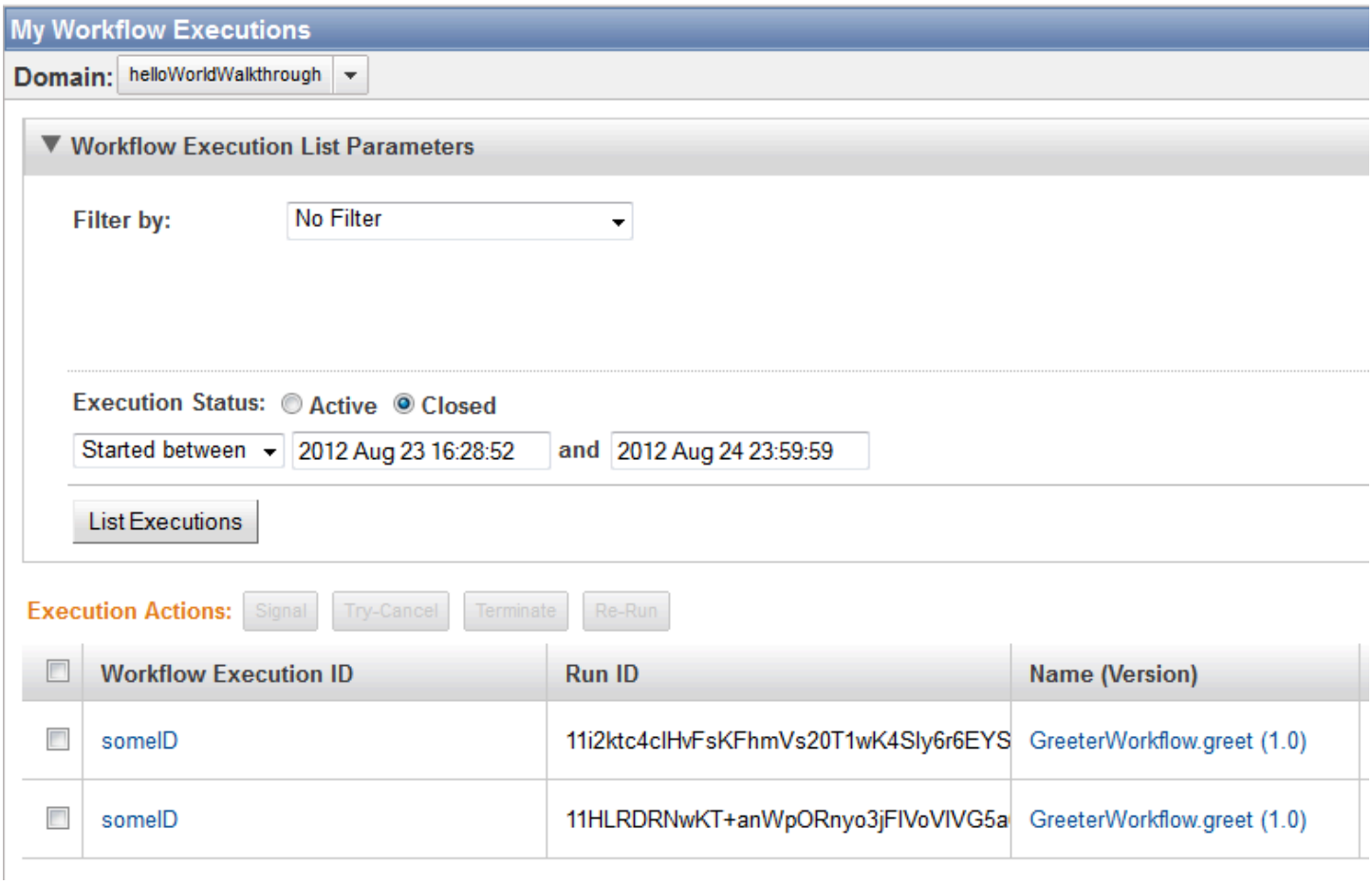
- WorkflowExecutionStarted 指示工作流程已开始执行。
- DecisionTaskScheduled 表示 Amazon SWF 已将第一个决策任务排队。

在执行第一个决策任务时阻止工作流程的原因是，工作流程分布在两个应用程序（GreeterMain 和 GreeterWorker）上。GreeterMain 启动了工作流程执行，但 GreeterWorker 未运行，因此工作线程不会轮询列表和执行任务。您可以独立运行任何一个应用程序，不过工作流程执行需要两个应用程序，才能在第一个决策任务之后继续。如果您现在运行 GreeterWorker，则工作流程和活动工作线程将开始轮询，各个任务会快速完成。如果您现在查看 Events (事件) 选项卡，将显示第一批事件。



您可以选择单独的事件来获取更多信息。在您查看完之后，工作流应该已将“Hello World!” 输出到您的控制台。

当工作流程完成后，不再显示在活动执行的列表上。不过，如果您希望查看它，请选择 Closed (已关闭) 执行状态，然后选择 List Executions (列出执行)。这将显示指定域 (`helloWorldWalkthrough`) 中未超过其保留时间的所有已完成工作流程实例，该保留时间是在您创建域时指定的。



请注意，每个工作流程实例具有一个唯一的 Run ID (运行 ID) 值。您可以为不同的工作流程实例使用相同的执行 ID，但一次只能为一个活动执行指定。

HelloWorldWorkflowAsync 应用程序

有时，最好让工作流程在本地执行某些任务，而不是使用活动。不过，工作流程任务通常涉及处理 `Promise<T>` 对象表示的值。如果将 `Promise<T>` 对象传递给一个同步工作流程方法，该方法将立即执行，但无法访问 `Promise<T>` 对象的值，直到该对象准备就绪。您可以轮询 `Promise<T>.isReady`，直到它返回 `true`，但这样做的效率非常低，并且该方法可能会阻止很长时间。更好的方法是使用异步方法。

异步方法的实现与标准方法类似，通常是作为工作流实现类的成员，并在工作流实现的上下文中运行。您可以应用 `@Asynchronous` 注释将其指定为异步方法，这会指示框架像对待活动一样处理它。

- 在工作流程实现调用异步方法时，它将立即返回。异步方法通常返回一个 `Promise<T>` 对象，在该方法完成时，该对象将变为就绪状态。

- 如果为异步方法传递一个或多个 `Promise<T>` 对象，它将推迟执行，直到所有输入对象准备就绪。因此，异步方法可以访问其输入 `Promise<T>` 值，而不会出现引发异常的风险。

> **Note**
>
> 鉴于适用于 Java 的 AWS Flow Framework 执行工作流的方式，异步方法通常会执行多次，因此，您只能将其用于快速的低开销任务。您应该使用活动执行时间较长的任务，如规模较大的计算。有关详细信息，请参阅AWS Flow Framework 基本概念：分布式执行。

本主题是对 HelloWorldWorkflowAsync 的一次演练，它是 HelloWorldWorkflow 的改版，将其中一个活动替换为异步方法。要实现该应用程序，请在您的项目目录中创建 helloWorld.HelloWorldWorkflow 程序包的副本，并将其命名为 helloWorld.HelloWorldWorkflowAsync。

> **Note**
>
> 本主题以 HelloWorld 应用程序和 HelloWorldWorkflow 应用程序主题中提供的概念和文件为基础。熟悉这些文件和中介绍的概念，这些主题，然后再继续。

以下几节介绍了如何修改原始 HelloWorldWorkflow 代码以使用异步方法。

HelloWorldWorkflowAsync 活动实现

HelloWorldWorkflowAsync 在 `GreeterActivities` 中实现其活动工作线程接口，如下所示：

```
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);
}
```

该接口类似于 HelloWorldWorkflow 使用的接口，但存在以下差异：

- 它忽略 getGreeting 活动；该任务现在由异步方法进行处理。
- 版本号设置为 2.0。在 Amazon SWF 注册活动接口后，您无法对其进行修改，除非更改版本号。

其余活动方法实现与 HelloWorldWorkflow 完全相同。只需从 GreeterActivitiesImpl 中删除 getGreeting。

HelloWorldWorkflowAsync 工作流程实现

HelloWorldWorkflowAsync 按如下方式定义工作流程接口：

```
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();
}
```

该接口与 HelloWorldWorkflow 完全相同，但具有新的版本号。与活动一样，如果要更改注册的工作流程，您必须更改其版本。

HelloWorldWorkflowAsync 按如下方式实现工作流程：

```
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 将 getGreeting 活动替换为 getGreeting 异步方法，但 greet 方法的工作方式基本相同：

1. 执行 getName 活动，这会立即返回 Promise<String> 对象 name，它表示名称。
2. 调用 getGreeting 异步方法并为其传递 name 对象。getGreeting 立即返回 Promise<String> 对象 greeting，它表示问候语。
3. 执行 say 活动并为其传递 greeting 对象。
4. 在 getName 完成时，name 会变为就绪状态，getGreeting 将使用它的值构建问候语。
5. 在 getGreeting 完成时，greeting 会变为就绪状态，say 会将字符串输出到控制台。

不同之处在于，greet 调用异步 getGreeting 方法，而不是调用活动客户端来执行 getGreeting 活动。实际结果是相同的，但 getGreeting 方法的工作方式与 getGreeting 活动略有不同。

- 工作流程工作线程使用标准函数调用语义来执行 getGreeting。不过，活动异步执行是由 Amazon SWF 协调的。
- getGreeting 在工作流程实现的进程中运行。
- getGreeting 返回一个 Promise<String> 对象，而不是 String 对象。要获取 Promise 保留的字符串值，您需要调用其 get() 方法。不过，由于活动是异步运行的，其返回值可能不会立即可用；get() 将提出异常，直到异步方法的返回值可用。

  有关 Promise 如何工作的更多信息，请参阅 AWS Flow Framework 基本概念：活动与工作流之间的数据交换。

getGreeting 将问候语字符串传递给静态 Promise.asPromise 方法以创建返回值。该方法创建一个具有相应类型的 Promise<T> 对象，设置对象值，然后将其置于就绪状态。

HelloWorldWorkflowAsync 工作流程及活动主机和启动程序

HelloWorldWorkflowAsync 实现 GreeterWorker 以作为工作流程和活动实现的主机类。它与 HelloWorldWorkflow 实现完全相同，唯一的区别是 taskListToPoll 名称，它设置为“HelloWorldAsyncList”。

```
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 在 `GreeterMain` 中实现工作流程启动程序；它与 HelloWorldWorkflow 实现完全相同。

要执行工作流程，请运行 `GreeterWorker` 和 `GreeterMain`，就像 HelloWorldWorkflow 一样。

HelloWorldWorkflowDistributed 应用程序

使用 HelloWorldWorkflow 和 HelloWorldWorkflowAsync，Amazon SWF 可以协调工作流和活动实现之间的交互，但它们作为单个进程在本地运行。`GreeterMain` 位于单独的进程中，但仍在同一系统上运行。

Amazon SWF 的一个重要功能是支持分布式应用程序。例如，您可以在 Amazon EC2 实例上运行工作流工作线程，在数据中心计算机上运行工作流启动程序，在客户端台式计算机上运行活动。您甚至可以在不同的系统上运行不同的活动。

HelloWorldWorkflowDistributed 应用程序扩展了 HelloWorldWorkflowAsync 以在两个系统和三个进程之间分配应用程序。

- 工作流程和工作流程启动程序作为单独的进程在一个系统上运行。
- 活动在不同的系统上运行。

要实现该应用程序，请在您的项目目录中创建 helloWorld.HelloWorldWorkflowAsync 程序包的副本，并将其命名为 helloWorld.HelloWorldWorkflowDistributed。以下几节介绍了如何修改原始 HelloWorldWorkflowAsync 代码以在两个系统和三个进程之间分配应用程序。

您不需要更改工作流程或活动实现以在不同的系统上运行，甚至不需要更改版本号。您也不需要修改 `GreeterMain`。您只需要更改活动和工作流程主机。

对于 HelloWorldWorkflowAsync，单个应用程序用作工作流程和活动主机。要在不同的系统上运行工作流程和活动实现，您必须实现单独的应用程序。从项目中删除 GreeterWorker 并添加两个新类文件 (GreeterWorkflowWorker 和 GreeterActivitiesWorker)。

HelloWorldWorkflowDistributed 在 GreeterActivitiesWorker 中实现其活动主机，如下所示：

```
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 在 `GreeterWorkflowWorker` 中实现其工作流程主机，如下所示：

```
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();
    }
}
```

请注意，GreeterActivitiesWorker 就是没有 WorkflowWorker 代码的 GreeterWorker，GreeterWorkflowWorker 就是没有 ActivityWorker 代码的 GreeterWorker。

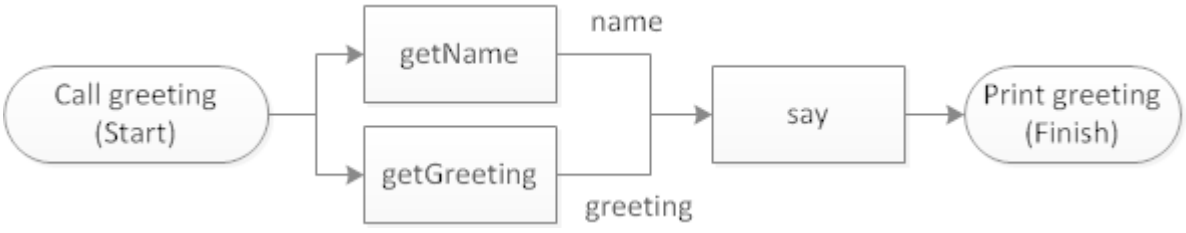
要运行工作流程，请执行以下操作：

1. 创建一个可运行的 JAR 文件并将 GreeterActivitiesWorker 作为入口点。
2. 将步骤 1 中的 JAR 文件复制到另一个系统，该系统可以运行支持 Java 的任何操作系统。
3. 确保在其他系统上提供可访问相同 AWS 域的 AWS 凭证。
4. 运行 JAR 文件。
5. 在您的开发系统上，使用 Eclipse 运行 GreeterMain 和 GreeterWorkflowWorker。

除了在与工作流程工作线程和工作流程启动程序不同的系统上运行活动以外，工作流程的工作方式与 HelloWorldAsync 完全相同。但是，输出到控制台的“Hello World!”的 println 调用 是在 say 活动中进行的，因此，输出将显示在运行活动工作线程的系统上。

HelloWorldWorkflowParallel 应用程序

以前版本的 Hello World! 均使用线性工作流程拓扑。不过，Amazon SWF 并不仅限于线性拓扑。HelloWorldWorkflowParallel 应用程序是修改的 HelloWorldWorkflow 版本，它使用并行拓扑，如下图所示。



对于 HelloWorldWorkflowParallel，getName 和 getGreeting 并行运行并分别返回问候语的一部分。然后，say 将两个字符串合并为问候语，并输出到控制台。

要实现该应用程序，请在您的项目目录中创建 helloWorld.HelloWorldWorkflow 程序包的副本，并将其命名为 helloWorld.HelloWorldWorkflowParallel。以下几节介绍了如何修改原始 HelloWorldWorkflow 代码以并行运行 `getName` 和 `getGreeting`。

HelloWorldWorkflowParallel 活动工作线程

HelloWorldWorkflowParallel 活动接口是在 `GreeterActivities` 中实现的，如以下示例所示。

```
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);
}
```

该接口类似于 HelloWorldWorkflow，但存在以下差异：

- `getGreeting` 不使用任何输入；它仅返回问候语字符串。
- `say` 使用两种输入字符串：问候语和名称。
- 该接口具有新的版本号，在任何时候更改注册的接口时，需要使用该版本号。

HelloWorldWorkflowParallel 在 `GreeterActivitiesImpl` 中实现活动，如下所示：

```
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 和 getGreeting 现在仅返回问候语字符串的一半。say 连接两个部分以生成完整的短语，并将其输出到控制台。

HelloWorldWorkflowParallel 工作流程工作线程

HelloWorldWorkflowParallel 工作流程接口是在 GreeterWorkflow 中实现的，如下所示：

```
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();
}
```

该类与 HelloWorldWorkflow 版本完全相同，但已更改版本号以与活动工作线程匹配。

工作流程是在 GreeterWorkflowImpl 中实现的，如下所示：

```
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);
   }
}
```

乍看上去，该实现与 HelloWorldWorkflow 非常相似；三个活动客户端方法是按顺序执行的。但活动不是按顺序执行的。

- HelloWorldWorkflow 将 name 传递到 getGreeting。由于 name 是一个 Promise<T> 对象，getGreeting 推迟执行活动，直到 getName 完成，因此，两个活动按顺序执行。
- HelloWorldWorkflowParallel 不会将任何输入传递到 getName 或 getGreeting。两种方法都不会推迟执行，并立即并行执行关联的活动方法。

say 活动将 greeting 和 name 作为输入参数。由于它们是 Promise<T> 对象，say 推迟执行，直到两个活动完成，然后构建并输出问候语。

请注意，HelloWorldWorkflowParallel 不使用任何特殊建模代码以定义工作流程拓扑。它使用标准 Java 流量控制并利用 Promise<T> 对象的属性来隐式执行该操作。适用于 Java 的 AWS Flow Framework 应用程序只需将 Promise<T> 对象与传统 Java 流程结合使用，即可实现复杂的拓扑结构。

HelloWorldWorkflowParallel 工作流程和活动主机和启动程序

HelloWorldWorkflowParallel 实现 GreeterWorker 以作为工作流程和活动实现的主机类。它与 HelloWorldWorkflow 实现完全相同，但 taskListToPoll 名称除外，它设置为“HelloWorldParallelList”。

HelloWorldWorkflowParallel 在 GreeterMain 中实现工作流程启动程序，它与 HelloWorldWorkflow 实现完全相同。

要执行工作流程，请运行 GreeterWorker 和 GreeterMain，就像 HelloWorldWorkflow 一样。

适用于 Java 的 AWS Flow Framework 的工作原理

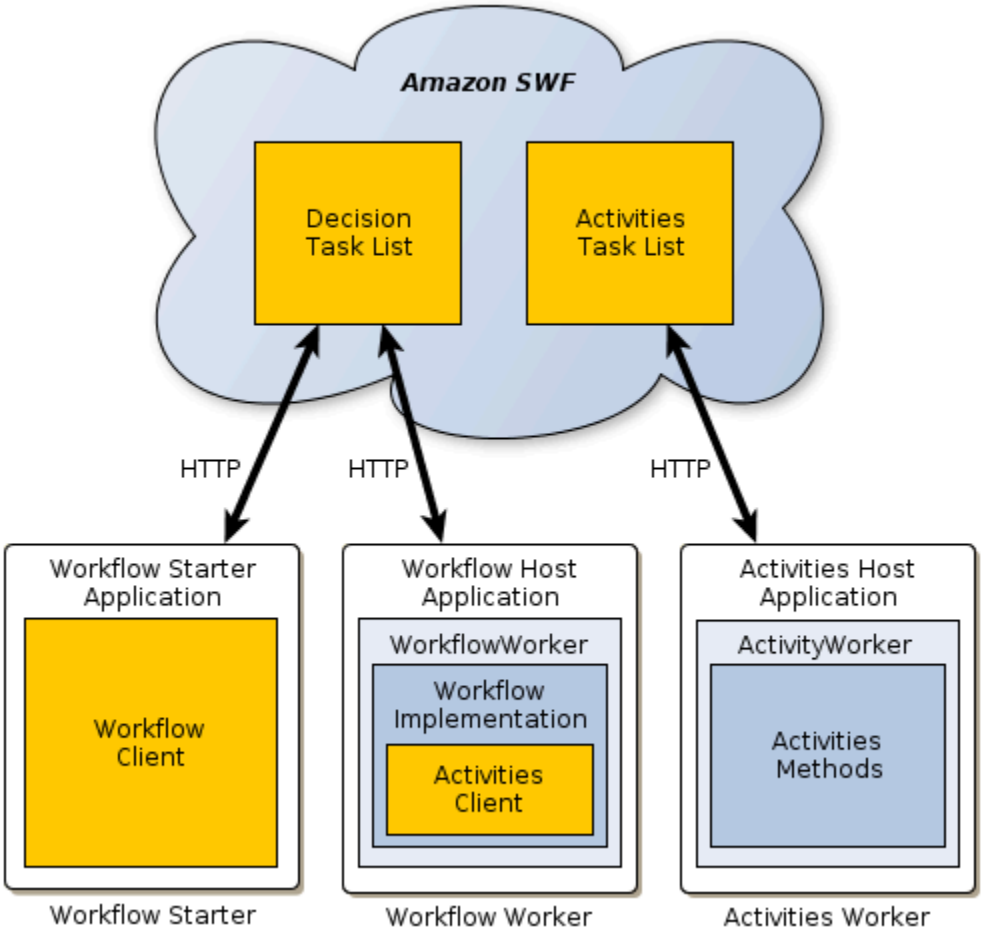
结合使用适用于 Java 的 AWS Flow Framework 与 Amazon SWF 可以轻松创建可扩展且容错的应用程序，以执行长时间运行和/或远程运行的异步任务。“Hello World!”示例（位于 Java 的 AWS Flow Framework 用法是什么？ 中）介绍了有关如何使用 AWS Flow Framework 实现基本工作流应用程序的基础知识。本节提供了有关 AWS Flow Framework 应用程序如何工作的概念信息。第一部分简要说明了 AWS Flow Framework 应用程序的基本结构，其余部分提供了 AWS Flow Framework 应用程序如何工作的详细信息。

主题

AWS Flow Framework 基本概念：应用程序结构

从概念上讲，AWS Flow Framework 应用程序包含三个基本组件：工作流程启动程序、工作流程工作线程 和活动工作线程。一个或多个主机应用程序负责向 Amazon SWF 注册工作线程（工作流和活动）、启动工作线程和处理清理。工作线程处理执行工作流程的机制，可以在几个主机上实现工作线程。

该图表示基本 AWS Flow Framework 应用程序：



> **Note**
>
> 从概念上讲，在三个单独的应用程序中实现这些组件是非常方便的，但您可以通过多种方式创建应用程序以实现该功能。例如，您可以在活动和工作流程工作线程中使用单个主机应用程序，或者使用单独的活动和工作流程主机。也可以使用多个活动工作线程，每个工作线程在单独的主机上处理一组不同的的活动，等等。

三个 AWS Flow Framework 组件通过向 Amazon SWF 发送 HTTP 请求进行间接交互，Amazon SWF 负责管理这些请求。Amazon SWF 将执行以下操作：

- 保留一个或多个决策任务列表，这些列表确定工作流程工作线程要执行的下一步骤。
- 保留一个或多个活动任务列表，这些列表确定活动工作线程将执行的任务。
- 保留工作流程执行的详细分步历史记录。

在使用 AWS Flow Framework 时，您的应用程序代码不需要直接处理图中显示的很多细节，例如，将 HTTP 请求发送到 Amazon SWF。您可以直接调用 AWS Flow Framework 方法，该框架将在后台处理这些细节。

活动工作线程角色

活动工作线程执行工作流程必须完成的各种任务。它包含以下内容：

- 活动实现，它包含一组为工作流程执行特定任务的活动方法。
- 一个 ActivityWorker 对象，使用 HTTP 长轮询请求来轮询 Amazon SWF 以获取要执行的活动任务。需要执行任务时，Amazon SWF 通过发送执行任务所需的信息来响应请求。然后，ActivityWorker 对象会调用相应的活动方法，并将结果返回到 Amazon SWF。

工作流程工作线程角色

工作流程工作线程协调各种活动的执行情况，管理数据流以及处理失败的活动。它包含以下内容：

- 工作流程实现，它包含活动协调逻辑，处理失败的活动，等等。
- 活动客户端，它作为活动工作线程的代理，并允许工作流程工作线程计划要异步执行的活动。
- 一个 WorkflowWorker 对象，使用 HTTP 长轮询请求轮询 Amazon SWF 以获取决策任务。如果工作流任务列表上具有任务，Amazon SWF 将通过返回执行任务所需的信息来响应请求。然后，该框架会执行工作流以执行任务，并将结果返回到 Amazon SWF。

工作流程启动程序角色

工作流程启动程序启动工作流程实例 (也称为工作流程执行)，并且可以在执行期间与实例交互，以便将额外数据传递给工作流程工作线程或获取当前工作流程状态。

工作流程启动程序使用工作流程客户端启动工作流程执行，在执行期间根据需要与工作流程交互以及处理清理。工作流启动程序可以是本地运行的应用程序、Web 应用程序、AWS CLI 甚至 AWS Management Console。

Amazon SWF 如何与您的应用程序交互

Amazon SWF 负责协调工作流组件之间的交互，并维护详细的工作流历史记录。Amazon SWF 不会启动与组件的通信，它会等待来自组件的 HTTP 请求，并根据需要管理这些请求。例如：

- 如果请求来自工作线程，目的是轮询以获取可用的任务，则 Amazon SWF 会在任务可用时直接响应工作线程。有关长轮询的更多信息，请参阅 Amazon Simple Workflow Service Developer Guide 中的 Long Polling。
- 如果请求是活动工作线程发出的说明任务已完成的通知，Amazon SWF 会将该信息记录在执行历史记录中，并在决策任务列表中添加一个任务，以通知工作流工作线程任务已完成，可以继续执行下一步骤。
- 如果请求是工作流工作线程发出的，指示执行活动，Amazon SWF 会将该信息记录在执行历史记录中，并在活动任务列表中添加一个任务，以指示活动工作线程执行相应的活动方法。

该方法允许工作线程在具有互联网连接的任何系统上运行，包括 Amazon EC2 实例、企业数据中心、客户端计算机等。它们甚至不必运行相同的操作系统。由于 HTTP 请求来自于工作线程，因此，不需要使用外部可见的端口；工作线程可以在防火墙后面运行。

了解更多信息

有关对 Amazon SWF 工作原理的更深入讨论，请参阅 Amazon Simple Workflow Service Developer Guide。

AWS Flow Framework 基本概念：可靠执行

异步分布式应用程序必须处理常规应用程序不会遇到的可靠性问题，包括：

- 如何在异步分布式组件之间提供可靠通信，如在远程系统上长时间运行的组件。
- 如何在组件故障或断开连接时确保结果不会丢失，特别是长时间运行的应用程序。
- 如何处理故障的分布式组件。

应用程序可以依靠 AWS Flow Framework 和 Amazon SWF 来管理这些问题。我们将探讨 Amazon SWF 如何提供一些机制来确保工作流以可靠且可预测的方式运行，即使这些工作流长时间运行并依赖通过计算和人机交互执行的异步任务。

提供可靠的通信

AWS Flow Framework 通过使用 Amazon SWF 将任务分配给分布式活动工作线程并将结果返回到工作流工作线程，从而在工作流工作线程及其活动工作线程之间提供可靠通信。Amazon SWF 使用以下方法来确保工作线程及其活动之间的可靠通信：

- Amazon SWF 可持久存储安排的活动和工作流任务，并确保它们最多执行一次。
- Amazon SWF 可确保活动任务要么成功完成并返回有效结果，要么通知工作流工作线程任务失败。
- Amazon SWF 可持久存储每个已完成的活动的结果，对于失败的活动，它还将存储相关的错误信息。

然后，AWS Flow Framework 会使用 Amazon SWF 中的活动结果确定如何继续执行工作流。

确保结果不会丢失

保留工作流程历史记录

对 PB 级数据执行数据挖掘操作的活动可能需要几小时才能完成；指示工作人员执行复杂任务的活动可能需要几天甚至几周才能完成！

要满足此类方案的要求，AWS Flow Framework 工作流程和活动可能需要任意长的时间才能完成：工作流程执行的时限长达一年。要可靠地执行长时间运行的进程，需要采用某种机制以持续永久存储工作流程的执行历史记录。

AWS Flow Framework 依靠 Amazon SWF 来处理这个问题，Amazon SWF 会维护每个工作流实例的运行历史记录。工作流程的历史记录提供了工作流程进度的完整且权威的记录，包括已计划并完成的所有工作流程和活动任务以及完成或失败的活动返回的信息。

AWS Flow Framework 应用程序通常不需要直接与工作流程历史记录交互，但它们可以在需要时访问该历史记录。对于大多数用途，应用程序可以直接让该框架在后台与工作流程历史记录进行交互。有关工作流历史记录的全面讨论，请参阅 Amazon Simple Workflow Service Developer Guide 中的 Workflow History。

无状态执行

执行历史记录允许工作流程工作线程是无状态的。如果具有多个工作流程或活动工作线程实例，则任何工作线程可以执行任何任务。工作线程会从 Amazon SWF 接收执行任务所需的所有状态信息。

该方法使工作流程变得更可靠。例如，如果某个活动工作线程失败，您不必重新启动工作流程。只需重新启动工作线程，它将开始轮询任务列表并处理列表上的任何任务，不受失败何时发生的影响。您可以使用两个或更多工作流程和活动工作线程 (可能在不同的系统上)，以使整个工作流程具有容错功能。然后，如果一个工作线程失败，其他工作线程继续处理计划任务，工作流程进度不会发生任何中断。

处理故障的分布式组件

活动通常由于临时原因而失败 (如短暂断开连接)，因此，处理失败活动的常见策略是重试活动。应用程序可以依赖 AWS Flow Framework，而不是通过实施复杂的消息传送策略来处理重试过程。它提供了几种机制以重试失败的活动，并提供内置异常处理机制以处理工作流程中的任务的异步分布式执行。

AWS Flow Framework 基本概念：分布式执行

工作流实例本质上是一个虚拟执行线程，可以跨越多台远程计算机上运行的活动和协调逻辑。Amazon SWF 和 AWS Flow Framework 函数可充当操作系统，通过以下方式管理虚拟 CPU 上的工作流实例：

- 保留每个实例的执行状态。
- 在实例之间切换。
- 从切换位置恢复执行实例。

重播工作流程

由于活动可能长时间运行，最好不要在完成之前直接阻止工作流程。相反，AWS Flow Framework 通过使用重放机制来管理工作流执行，该机制依靠 Amazon SWF 维护的工作流历史记录来分阶段执行工作流。

每个阶段以某种方式重播工作流程逻辑，以便仅执行一次每个活动，并确保在其 Promise 对象准备就绪后才执行活动和异步方法。

在启动工作流程执行时，工作流程启动程序启动第一个重播阶段。该框架调用工作流程的入口点方法，并且：

1. 执行所有不依赖于活动完成的工作流程任务，包括调用所有活动客户端方法。
2. 为 Amazon SWF 提供一个安排执行的活动任务列表。对于第一个阶段，该列表仅包含不依赖于 Promise 并且可立即执行的活动。
3. 通知 Amazon SWF 该阶段已完成。

Amazon SWF 会在工作流历史记录中存储活动任务，并通过将其放在活动任务列表中来安排执行。活动工作线程轮询任务列表并执行这些任务。

活动工作线程完成任务后，会将结果返回给 Amazon SWF，Amazon SWF 在工作流执行历史记录中记录该结果，并将其放在工作流任务列表中，以便为工作流工作线程安排新的工作流任务。工作流程工作线程轮询任务列表，并在收到该任务时运行下一个重播阶段，如下所示：

1. 该框架再次运行工作流程的入口点方法，并且：
   - 执行所有不依赖于活动完成的工作流程任务，包括调用所有活动客户端方法。不过，该框架检查执行历史记录，并且不会计划重复的活动任务。
   - 检查历史记录以查看哪些活动任务已完成，并执行依赖于这些活动的任何异步工作流程方法。
2. 在所有可执行的工作流任务都已完成后，该框架会向 Amazon SWF 报告：
   - 它会向 Amazon SWF 提供一份清单，列出自上一阶段以来输入 `Promise<T>` 对象已准备就绪并可安排执行的所有活动。
   - 如果该阶段没有生成任何额外的活动任务，但仍有未完成的活动，则框架会通知 Amazon SWF 该阶段已完成。然后，它等待另一个活动完成，从而启动下一个重播阶段。
   - 如果该阶段没有生成任何额外的活动任务，且所有活动都已完成，则框架会通知 Amazon SWF 工作流执行已完成。

有关重播行为的示例，请参阅适用于 Java 的 AWS Flow Framework 重播行为。

重播和异步工作流程方法

异步工作流程方法的使用方式通常与活动非常相似，因为方法推迟执行，直到所有输入 `Promise<T>` 对象准备就绪。不过，重播机制处理异步方法的方式与活动不同。

- 重播不保证异步方法仅执行一次。它推迟异步方法执行，直到其输入 Promise 对象准备就绪，但它为所有后续阶段执行该方法。
- 在异步方法完成时，它不会启动新的阶段。

在适用于 Java 的 AWS Flow Framework 重播行为中提供了重播异步工作流程的示例。

重播和工作流程实现

在很大程度上，您不需要关注重播机制的细节。它基本上是在后台执行的操作。不过，重播对您的工作流程实现有两个重大影响。

- 不要使用工作流程方法执行长时间运行的任务，因为重播将重复执行多次该任务。甚至异步工作流程方法通常也会运行多次。应使用活动执行长时间运行的任务；重播仅执行一次活动。
- 您的工作流程逻辑必须是完全确定性的；每个阶段必须采用相同的控制流程路径。例如，控制流程路径不应依赖于当前时间。有关重播和确定性要求的详细说明，请参阅不确定性。

AWS Flow Framework 基本概念：任务列表和任务执行

Amazon SWF 通过将工作流和活动任务发布到指定列表来管理它们。Amazon SWF 维护至少两个任务列表，一个用于工作流工作线程，另一个用于活动工作线程。

> **Note**
>
> 您可以根据需要指定任意数量的任务列表，将为每个列表分配不同的工作线程。对任务列表的数量没有任何限制。您通常在创建工作线程对象时在工作线程主机应用程序中指定工作线程的任务列表。

以下来自 `HelloWorldWorkflow` 主机应用程序的摘录创建新的活动工作线程并将其分配给 `HelloWorldList` 活动任务列表。

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

默认情况下，Amazon SWF 在 `HelloWorldList` 列表中安排工作线程的任务。然后工作线程轮询该列表中的任务。您可以向任务列表分配任何名称。您甚至可以对工作流和活动列表使用相同的名称。在内部，Amazon SWF 将工作流和活动任务列表的名称放在不同的命名空间中，因此这两个列表是不同的。

如果您未指定任务列表，则会 AWS Flow Framework 指定工作人员向 Amazon SWF 注册该类型时的默认列表。有关更多信息，请参阅工作流程和活动类型注册。

有时让一个特定工作线程或一组工作线程执行某些任务非常有用。例如，图像处理工作流可能使用一个活动来下载图像并使用另一个活动来处理该图像。在同一个系统上执行这两个任务会更高效，并可避免通过网络传输大型文件的开销。

要支持此类情况，您可以在调用活动客户端方法时使用包括 schedulingOptions 参数的重载显式指定任务列表。您可以通过向方法传递一个经过适当配置的ActivitySchedulingOptions对象来指定任务列表。

例如，假设 HelloWorldWorkflow 应用程序的 say 活动由不同于 getName 和 getGreeting 的活动工作线程托管。以下示例演示如何确保 say 使用与 getName 和 getGreeting 相同的任务列表，即使它们最初分配到了不同的列表。

```
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);
  }
}
```

异步 runSay 方法从其客户端对象获取 getGreeting 任务列表。然后它创建并配置 ActivitySchedulingOptions 对象，该对象确保 say 轮询与 getGreeting 相同的任务列表。

> **Note**
>
> 当您将 schedulingOptions 参数传递给活动客户端方法时，它会仅针对该活动执行覆盖原始任务列表。如果您在未指定任务列表的情况下再次调用活动客户端方法，Amazon SWF 会将任务分配给原始列表，且活动工作线程将轮询该列表。

AWS Flow Framework 基本概念：可扩展应用程序

Amazon SWF 具有两项重要功能，可以轻松扩展工作流应用程序以处理当前负载：

- 完整的工作流程执行历史记录；您可以实现无状态的应用程序。
- 松散耦合到任务执行的任务计划；可以轻松扩展应用程序以满足当前需求。

Amazon SWF 通过将任务发布到动态分配的任务列表（而不是直接与工作流和活动工作线程通信）来安排任务。相反，工作线程使用 HTTP 请求轮询相应的列表以查找任务。该方法将任务安排与任务执行进行松耦合，工作线程在任何适用的系统上运行，包括 Amazon EC2 实例、公司数据中心、客户端计算机等。由于 HTTP 请求来自于工作线程，因此，不需要使用外部可见的端口，这甚至允许工作线程在防火墙后面运行。

工作线程用于轮询任务的长轮询机制确保工作线程不会过载。即使在计划任务中出现峰值，工作线程也会按自己的频率提取任务。不过，由于工作线程是无状态的，您可以启动额外的工作线程实例，以便动态扩展应用程序以满足增加的负载要求。即使它们在不同的系统上运行，每个实例也会轮询相同的任务列表，并且第一个可用的工作线程实例执行每个任务，而无论工作线程位于何处或何时启动。在负载下降时，您可以相应地降低工作线程数。

AWS Flow Framework 基本概念：活动与工作流之间的数据交换

在您调用异步活动客户端方法时，它会立即返回 Promise (又称为 Future) 对象，这表示活动方法的返回值。最初，Promise 处于未就绪状态，并且未定义返回值。在活动方法完成其任务并返回后，框架跨网络将返回值封送给工作流工作线程，后者为 Promise 分配一个值并将对象置于就绪状态。

即使活动方法没有返回值，您仍可以使用 Promise 来管理工作流执行。如果您将返回的 Promise 传递给活动客户端方法或异步工作流方法，则会推迟执行，直至对象准备就绪。

如果您将一个或多个 Promise 传递给活动客户端方法，则框架会使任务排队，但推迟安排它，直到所有对象都准备就绪。然后，它从每个 Promise 提取数据并跨 Internet 将其封送给活动工作线程，后者将其作为标准类型传递给活动方法。

> **Note**
>
> 如果您需要在工作流与活动工作线程之间传输大量数据，首选方法是将数据存储在方便的位置，然后仅传递检索信息。例如，您可以将数据存储在 Amazon S3 存储桶中并传递关联的 URL。

承诺 <T> 类型

Promise<T> 类型在一些方面与 Java Future<T> 类型非常类似。这两种类型都表示异步方法返回的值，并且最初都未定义。您通过调用它的 get 方法来访问对象的值。除此之外，这两种类型的行为截然不同。

- Future<T> 是一个异步构造，它允许应用程序等待异步方法完成。如果您调用 get，但对象未准备就绪，则它会阻止，直到对象已准备就绪。
- 对于 Promise<T>，同步由框架处理。如果您调用 get，但对象未准备就绪，则 get 会引发异常。

Promise<T> 的主要用途是管理从一个活动到另一个活动的数据流。它确保直到输入数据都有效后才执行活动。在许多情况下，工作流工作线程无需直接访问 Promise<T> 对象；它们只需将对象从一个活动传递给另一个活动，然后让框架和活动工作线程处理详细信息。要在工作流工作线程中访问 Promise<T> 对象的值，您必须在调用其 get 方法之前确定对象已准备就绪。

- 首选方法是将 Promise<T> 对象传递给异步工作流方法并在那里处理值。异步方法将推迟执行，直到其所有输入 Promise<T> 对象都已准备就绪，这保证您可以安全地访问它们的值。
- Promise<T> 公开 isReady 方法，如果对象已准备就绪，则该方法会返回 true。不建议使用 isReady 来轮询 Promise<T> 对象，但 isReady 在某些情况下非常有用。有关示例，请参阅 AWS Flow Framework Recipes。

适用于 Java 的 AWS Flow Framework 还包含 Settable<T> 类型，该类型派生自 Promise<T> 并具有类似的行为。不同之处在于，框架通常负责设置 Promise<T> 对象的值，而工作流工作线程负责设置 Settable<T> 的值。有关示例，请参阅 AWS Flow Framework Recipes

在一些情况下，工作流工作线程需要创建 Promise<T> 对象并设置其值。例如，返回 Promise<T> 对象的异步方法需要创建返回值。

- 要创建表示类型化值的对象，请调用静态 Promise.asPromise 方法，该方法创建适当类型的 Promise<T> 对象，设置其值，然后将其置于就绪状态。
- 要创建 Promise<Void> 对象，请调用静态 Promise.Void 方法。

> **Note**
>
> Promise<T> 可以表示任何有效类型。但是，如果必须跨 Internet 封送数据，则类型必须与数据转换器兼容。有关详细信息，请参阅下一节。

数据转换器和封送

AWS Flow Framework 使用数据转换器跨 Internet 封送数据。默认情况下，框架使用基于 Jackson JSON 处理器的数据转换器。但是，此转换器具有一些限制。例如，它无法封送不使用字符串作为键的映射。如果默认转换器对您的应用程序来说不合适，您可以实现自定义数据转换器。有关详细信息，请参阅DataConverters。

AWS Flow Framework 基本概念：应用程序和工作流执行之间的数据交换

工作流程入口点方法可能具有一个或多个参数，这允许工作流程启动程序将初始数据传递给工作流程。要在执行期间为工作流程提供额外数据，这也是非常有用的。例如，如果客户更改其送货地址，您可以通知订单处理工作流程，以便它可以进行相应的更改。

Amazon SWF 允许工作流实现信号方法，这允许应用程序（如工作流启动程序）随时将数据传递给工作流。信号方法可能具有任何方便的名称和参数。您可以将其包含在工作流程接口定义中，并将 `@Signal` 注释应用于方法声明以将其指定为信号方法。

以下示例显示一个声明信号方法 `changeOrder` 的订单处理工作流程接口，这允许工作流程启动程序在工作流程启动后更改原始订单。

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

该框架的注释处理器创建一个具有与信号方法相同的名称的工作流程客户端方法，并且工作流程启动程序调用该客户端方法以将数据传递给工作流程。有关示例，请参阅 AWS Flow Framework Recipes

Amazon SWF 超时类型

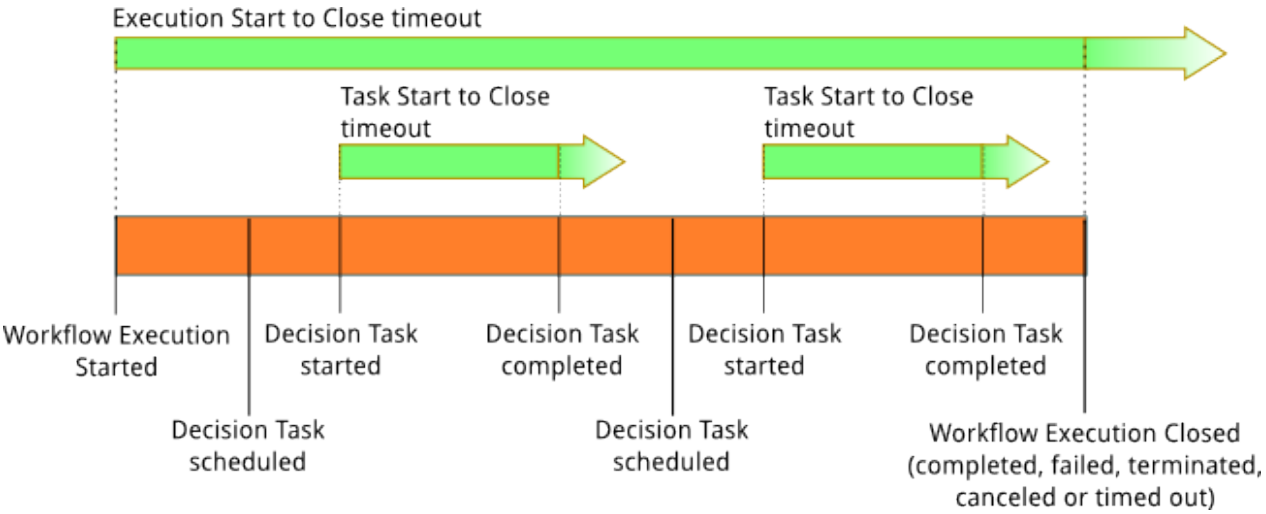
为确保工作流执行正确运行，您可以通过 Amazon SWF 设置不同类型的超时。一些超时指定工作流程的总运行时长。其它超时指定活动任务在被分配给工作程序之前所花费的时间以及从排定到完成所花费的时间。Amazon SWF API 中的所有超时都以秒为单位。Amazon SWF 还支持将字符串 `NONE` 作为超时值，表示没有超时。

对于与决策任务和活动任务相关的超时，Amazon SWF 会在工作流执行历史中添加一个事件。事件的属性提供了所发生超时类型的信息以及以及受影响的决策任务或活动任务。Amazon SWF 还会安排决策任务。决策程序接收新决策任务时将会在历史中看到超时事件，并会通过调用 RespondDecisionTaskCompleted 操作来采取适当操作。

任务被视为从排定时开始到其关闭为止都处于开启状态。因此，当工作程序处理任务时，任务被报告为开启状态。当工作程序将任务状态报告为已完成、已取消或失败时，任务关闭。Amazon SWF 也可能会因超时而关闭任务。

工作流程和决策任务中的超时

下图显示工作流程和决策超时如何与工作流程的生命周期相关：



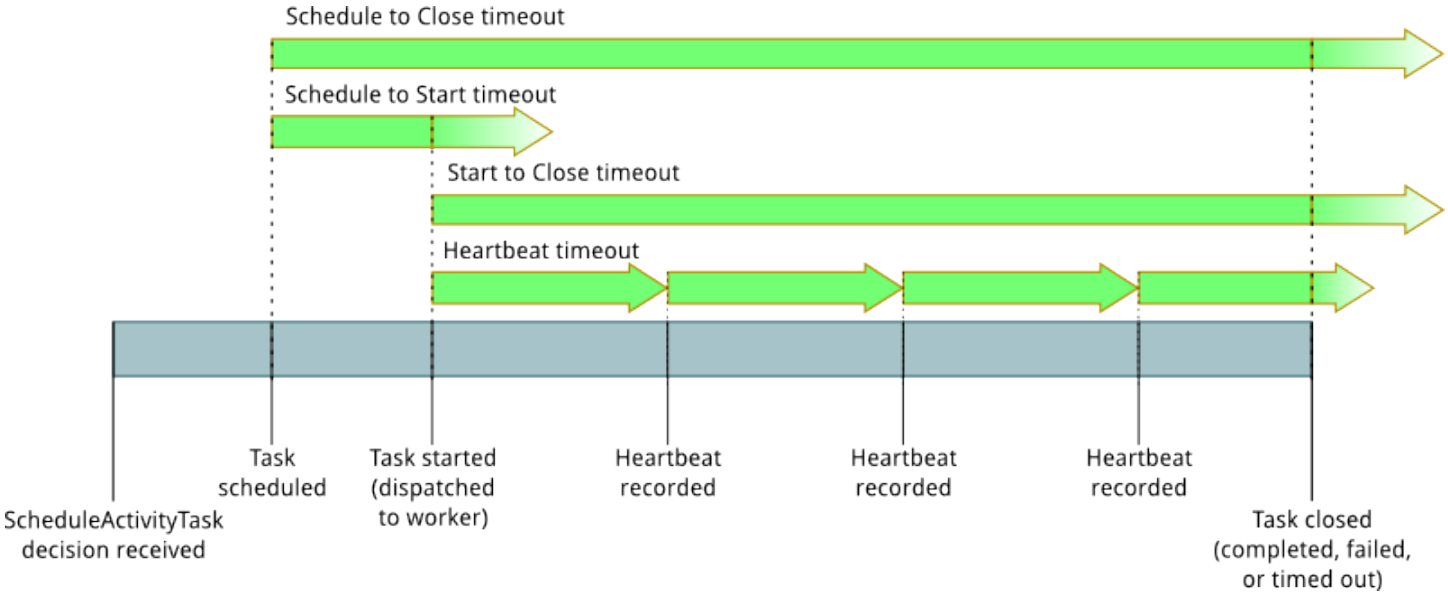
与工作流程和决策任务相关的超时类型有两个：

- 工作流启动到关闭 (**`timeoutType: START_TO_CLOSE`**) - 该超时指定了完成工作流执行所需的最长时间。工作流程注册期间，这一超时被设置为默认值，但当工作流程启动时，可用其它值覆盖该默认值。如果超出了此超时，Amazon SWF 将关闭工作流执行，并将 WorkflowExecutionTimedOut 类型的事件添加到工作流执行历史记录中。除了 `timeoutType` 之外，事件属性还会指定对此工作流程执行有效的 `childPolicy`。子策略指定上级工作流程执行超时或终止时子工作流程执行的处理方法。例如，如果 `childPolicy` 被设置为 TERMINATE，则子工作流程执行将被终止。一旦工作流程执行超时，则不能对其执行可视性调用之外的其他任何操作。
- 决策任务启动到关闭 (**`timeoutType: START_TO_CLOSE`**) – 该超时指定了相应决策程序完成决策任务所需的最长时间。该超时在工作流程类型注册期间设置。如果超出了此超时，任务将在工作流执行历史记录中被标记为超时，且 Amazon SWF 会在工作流历史记录中添加一个 DecisionTaskTimedOut 类型的事件。当此决策任务被排定 (`scheduledEventId`) 且启动 (`startedEventId`) 时，事件属性中将包含对应事件的 ID。除了添加事件外，Amazon

SWF 还会安排新决策任务以警告决策程序此决策任务已超时。此超时发生后，使用 `RespondDecisionTaskCompleted` 完成超时决策任务的尝试将失败。

活动任务中的超时

下图显示超时如何与活动任务的生命周期相关：



与活动任务相关的超时类型有四个：

- 活动任务启动到关闭 (**`timeoutType: START_TO_CLOSE`**) – 该超时指定了活动工作线在接收到任务后处理任务所需的最长时间。使用 RespondActivityTaskCanceled、RespondActivityTaskCompleted 和 RespondActivityTaskFailed 尝试关闭超时的活动任务将会失败。
- 活动任务检测信号 (**`timeoutType: HEARTBEAT`**) – 该超时指定了任务在通过 操作提供其进程前可以运行的最长时间。
- 活动任务安排到开始 (**`timeoutType: SCHEDULE_TO_START`**) – 该超时指定了在没有工作线执行活动任务时 Amazon SWF 在活动任务超时前等待的时间。一旦超时，过期的任务就不能分配给另一工作程序。
- 活动任务安排到关闭 (**`timeoutType: SCHEDULE_TO_CLOSE`**) – 该超时指定了任务从安排到完成所需的时间。最好的做法是，这个值不应该大于任务排定到启动超时与任务启动到关闭超时的总和。

> **Note**
>
> 每一个超时类型都有默认值，一般设置为 NONE (无限)。但是任何活动执行的最长时间均被限制为一年。

您在活动类型注册期间设置这些活动的默认值，但当您排定活动任务时您可以用新值覆盖默认值。上述超时中有任何一个发生时，Amazon SWF 会向工作流历史记录添加一个 ActivityTaskTimedOut 类型的事件。此事件的 `timeoutType` 值属性将指定发生了何种超时。对于其中每一个超时，`timeoutType` 的值都显示在括号中。当活动任务被排定 (`scheduledEventId`) 且启动 (`startedEventId`) 时，事件属性中还将包含对应事件的 ID。除了添加事件外，Amazon SWF 还会安排新决策任务以警告决策程序已发生超时。

最佳实践

通过这些最佳实践充分利用适用于 Java 的 AWS Flow Framework。

主题

对决策程序代码进行更改：版本控制和功能标志

本节介绍了如何使用以下两种方法避免对决策程序进行不向后兼容的更改：

- 版本控制提供了一个基本解决方案。
- 具有功能标志的版本控制以版本控制解决方案为基础：未引入新的工作流程版本，并且不需要推送新代码以更新版本。

在尝试这些解决方案之前，请参阅示例方案一节，其中说明了不向后兼容的决策程序更改的原因和影响。

重播过程和代码更改

当适用于 Java 的 AWS Flow Framework 决策程序工作线程执行决策任务时，它必须先重建执行的当前状态，然后才能在其中添加步骤。决策程序使用称为重播 的过程执行该操作。

重播过程从头重新开始执行决策程序代码，同时浏览已发生的事件的历史记录。通过浏览事件历史记录，该框架可以对信号或任务完成作出反应，并在代码中取消阻止 `Promise` 对象。

在执行决策程序代码时，该框架会增加计数器，以便为每个安排的任务（活动、Lambda 函数、计时器、子工作流或传出信号）分配一个 ID。该框架向 Amazon SWF 通报该 ID，并将该 ID 添加到历史事件中，如 `ActivityTaskCompleted`。

要成功完成重播过程，请务必使决策程序代码是确定性的，并按相同顺序为每个工作流程执行中的每个决策计划相同的任务。例如，如果不符合该要求，该框架可能无法将 `ActivityTaskCompleted` 事件中的 ID 与现有 `Promise` 对象相匹配。

示例方案

可以将某些类型的代码更改视为不向后兼容。这些更改包括修改计划任务数量、类型或顺序的更新。考虑以下示例：

您编写决策程序代码以计划两个计时器任务。您启动一个执行并运行一个决策。结果，计划了两个计时器任务，分别具有 ID 1 和 2。

如果您更新决策程序代码以仅在要执行的下一个决策之前计划一个计时器，在下一个决策任务期间，该框架无法重播第二个 `TimerFired` 事件，因为 ID 2 与代码已生成的任何计时器任务不匹配。

方案概述

以下概述显示了该方案的步骤。该方案的最终目标是迁移到仅计划一个计时器的系统，但不会导致在迁移之前启动的执行失败。

1. 初始决策程序版本
   a. 编写决策程序。
   b. 启动决策程序。
   c. 决策程序计划两个计时器。
   d. 决策程序启动 5 个执行。
   e. 停止决策程序。
2. 不向后兼容的决策程序更改
   a. 修改决策程序。
   b. 启动决策程序。
   c. 决策程序计划一个计时器。
   d. 决策程序启动 5 个执行。

以下几节包含说明如何实现该方案的 Java 代码示例。Solutions一节中的代码示例说明了几种修复不向后兼容的更改的方法。

> **Note**
>
> 您可以使用最新版本的AWS SDK for Java运行该代码。

通用代码

不会在该方案中的示例之间更改以下 Java 代码。

`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;
    }

}
```

编写初始决策程序代码

下面是决策程序的初始 Java 代码。它注册为版本 1，并计划两个 5 秒计时器任务。

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);
        }

    }
}
```

模拟不向后兼容的更改

以下修改的决策程序 Java 代码是一个很好的不向后兼容更改例子。该代码仍注册为版本 1，但仅计划一个计时器。

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);
        }

    }
}
```

通过使用以下 Java 代码，您可以运行修改的决策程序以模拟进行不向后兼容更改的问题。

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();
    }

}
```

在运行该程序时，三个失败的执行是在初始决策程序版本中启动的，并在迁移后继续运行。

Solutions

您可以使用以下解决方案以避免不向后兼容的更改。有关更多信息，请参阅对决策程序代码进行更改和示例方案。

使用版本控制

在该解决方案中，您将决策程序复制到一个新类，修改决策程序，然后在新工作流程版本中注册决策程序。

`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);
        }

    }

}
```

在更新的 Java 代码中，第二个决策程序工作线程运行两个版本的工作流程以允许动态执行继续运行，而与版本 2 中的更改无关。

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();
    }

}
```

在运行该程序时，将成功完成所有执行。

使用功能标志

向后兼容性问题的另一个解决方案是，创建代码分支以在同一类中支持两个实现，以根据输入数据执行分支，而不是根据工作流程版本。

在使用该方法时，每次引入敏感的更改时，您在输入对象中添加一些字段 (或修改现有的字段)。对于在迁移之前启动的执行，输入对象不包含字段 (或具有不同的值)。因此，您不必增加版本号。

> **Note**
>
> 如果添加新的字段，请确保 JSON 反序列化过程向后兼容。在迁移后，在引入字段之前序列化的对象仍会成功反序列化。由于在每次缺少字段时 JSON 都会设置 null 值，请始终使用装箱类型 (Boolean 而不是 boolean) 并处理值为 null 的情况。

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);
            }
        }

    }
}
```

在更新的 Java 代码中，仍会为版本 1 注册两个版本的工作流程的代码。不过，在迁移后，新执行启动并将输入数据的 skipSecondTimer 字段设置为 true。

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();
    }

}
```

在运行该程序时，将成功完成所有执行。

适用于 Java 的 AWS Flow Framework 编程指南

本部分提供了有关如何使用适用于 Java 的 AWS Flow Framework 的功能实现工作流应用程序的详细信息。

主题

使用 AWS Flow Framework 实施工作流程应用程序

使用 AWS Flow Framework 开发工作流程所涉及的典型步骤包括：

1. 定义活动与工作流程合同。分析您的应用程序要求，然后确定所需的活动和工作流程拓扑。活动处理所需的处理任务，工作流程拓扑 定义工作流程的基本结构和业务逻辑。

例如，媒体处理应用程序可能需要下载一个文件，处理该文件，然后将处理后的文件上传到 Amazon Simple Storage Service (S3) 存储桶。此过程可分为四个活动任务：

1. 从服务器下载文件
2. 处理文件 (例如，将文件转码为其他媒体格式)
3. 将文件上传到 S3 存储桶
4. 通过删除本地文件来执行清除操作

此工作流程将有一个入口点方法，并将实施一个按顺序运行活动的简单线性拓扑，与HelloWorldWorkflow 应用程序很相似。

2. 实施活动和工作流程接口。工作流程和活动合同由 Java 接口 定义，使其调用惯例可由 SWF 预测，并在您实施工作流程逻辑和活动任务时为您提供灵活性。您的程序的各个部分可充当彼此数据的使用者，但不需要知道其他部分的许多实施详细信息。

   例如，您可以定义一个 `FileProcessingWorkflow` 接口，并提供针对视频编码、压缩、缩略图等的不同的工作流程实施。这些工作流程中的每个工作流程均具有不同的控制流程，并且可以调用不同的方法；工作流程启动程序不需要知道。通过使用接口，也可以使用稍后将替换为工作代码的模拟实施来测试工作流程。

3. 生成活动和工作流程客户端。AWS Flow Framework 使您无需实施管理异步执行、发送 HTTP 请求、封送数据等操作的详细信息。相反，工作流程启动程序通过对工作流程客户端调用方法来执行工作流程实例，而工作流程实施通过对活动客户端调用方法来执行活动。框架在后台处理这些交互的细节。

   如果您使用的是 Eclipse 并且已配置项目（就像设置适用于 Java 的 AWS Flow Framework 中一样），则 AWS Flow Framework 注释处理器将使用接口定义来自动生成工作流程和活动客户端，以将一组相同的方法公开为相应的接口。

4. 实施活动和工作流程宿主应用程序。您的工作流和活动实现必须嵌套在主机应用程序中，这些应用程序会轮询 Amazon SWF 以获取任务、收集任何数据并调用相应的实现方法。适用于 Java 的 AWS Flow Framework 包括 WorkflowWorker 和 ActivityWorker 类，这两个类使得主机应用程序变得直接而简单。

5. 测试您的工作流。适用于 Java 的 AWS Flow Framework 提供JUnit 集成，用于内联和本地测试您的工作流。

6. 部署工作线程。您可以相应地部署工作线程，例如，您可以将工作线程部署到 Amazon EC2 实例或数据中心内的计算机。在部署并启动后，工作线程将开始轮询 Amazon SWF 以获取任务并按需处理任务。

7. 启动执行。应用程序通过使用工作流程客户端调用工作流程的入口点来启动工作流程实例。您也可以使用 Amazon SWF 控制台启动工作流。无论您通过什么方式启动工作流实例，都可以使用 Amazon SWF 控制台监控正在运行的工作流实例，并检查正在运行、已完成和已失败实例的工作流历史记录。

AWS SDK for Java包含一组适用于 Java 的 AWS Flow Framework 示例，您可以浏览这些示例并按照根目录下 readme.html 文件中的说明操作来运行这些示例。此外，还有一组简单的应用程序演示了如何处理各种特定的编程问题，您可以从 AWS Flow Framework Recipes 获取这些应用程序。

工作流和活动合同

Java 接口用于声明工作流和活动的签名。此接口在工作流 (或活动) 的实现与该工作流 (或活动) 的客户端之间形成合同。例如，通过使用 `@Workflow` 注释进行注释的接口来定义工作流类型 `MyWorkflow`：

```
@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();
}
```

合同没有特定于实现的设置。使用实现中立的合同使客户端与实现分离，从而可以灵活地更改实现详细信息而无需中断客户端。相反地，您也可以更改客户端，而无需更改所使用的工作流或活动。例如，可能使用 Promise (`Promise<T>`) 修改客户端以异步调用活动，而无需更改活动实现。同样，您也可以通过更改活动实现使其异步完成（如通过人员发送电子邮件），而无需更改活动的客户端。

在上面的示例中，工作流接口 `MyWorkflow` 包含方法 `startMyWF` 以启动新执行。使用 `@Execute` 注释对此方法进行了注释，并且此方法必须具有返回类型 `void` 或 `Promise<>`。在给定工作流接口中，

最多可以使用此注释对一个方法进行注释。此方法是工作流逻辑的入口点，并且框架在收到决策任务时会调用此方法来执行工作流逻辑。

工作流接口还定义可能发送给工作流的信号。在工作流执行收到具有匹配名称的信号时，会调用信号方法。例如，MyWorkflow 接口声明信号方法 signal1，使用 @Signal 注释对此方法进行了注释。

在信号方法上需要 @Signal 注释。信号方法的返回类型必须是 void。工作流接口中可以定义零个或多个信号方法。您可以声明没有 @Execute 方法但有一些 @Signal 方法的工作流接口，以生成无法启动其执行但可以向正在运行的执行发送信号的客户端。

使用 @Execute 和 @Signal 注释进行注释的方法可以具有除 Promise<T> 或其衍生物之外的任何类型的任何数量的参数。这允许您在工作流执行启动时和正在运行时向其传递强类型输入。@Execute 方法的返回类型必须是 void 或 Promise<>。

此外，您还可以在工作流接口中声明方法来报告工作流执行的最新状态，如上述示例中的 getState 方法。此状态不是工作流的整个应用程序状态。此功能的预期用途是允许您存储最多 32 KB 的数据来指示执行的最新状态。例如，在订单处理工作流中，您可以存储一个字符串来指示订单已收到、已处理或已取消。框架在每次完成一个决策任务时就会调用此方法来获取最新状态。状态存储在 Amazon Simple Workflow Service (Amazon SWF) 中，可使用生成的外部客户端进行检索。这允许您查看工作流执行的最新状态。使用 @GetState 进行注释的方法不得使用任何参数，并且不得具有 void 返回类型。您可以从此方法返回适合您的需求的任何类型。在上述示例中，此方法返回用于存储字符串状态和数值完成百分比的 MyWorkflowState 对象 (请参见下面的定义)。此方法应对工作流实现对象执行只读访问并同步调用，它不允许使用任何异步操作，如调用使用 @Asynchronous 进行注释的方法。在工作流接口中，最多可以使用 @GetState 注释对一个方法进行注释。

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

同样地，通过使用 @Activities 注释进行注释的接口来定义一组活动。接口中的每种方法都对应一个活动，例如：

```
@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);
}
```

接口允许您将一组相关活动分组在一起。您可以在活动接口中定义任意数量的活动，并可以根据需要定义任意数量的活动接口。与 @Execute 和 @Signal 方法类似，活动方法可以使用除 Promise<T> 或其衍生物之外的任何类型的任何数量的参数。活动的返回类型不得是 Promise<T> 或其衍生物。

工作流程和活动类型注册

Amazon SWF 要求先注册活动和工作流类型，然后才能使用它们。该框架自动在添加到工作线程的实现中注册工作流程类型和活动。该框架会查找实现工作流和活动的类型，并向 Amazon SWF 注册它们。默认情况下，该框架使用接口定义推断工作流程和活动类型的注册选项。所有工作流程接口需要具有 @WorkflowRegistrationOptions 注释或 @SkipRegistration 注释。工作流程工作线程注册为其配置的具有 @WorkflowRegistrationOptions 注释的所有工作流程类型。同样，需要使用 @ActivityRegistrationOptions 注释或 @SkipRegistration 注释对每个活动方法进行注释，或者必须在 @Activities 接口上提供其中的一个注释。活动工作线程注册为其配置的应用了 @ActivityRegistrationOptions 注释的所有活动类型。在启动其中的一个工作线程时，将自动执行注册。工作流程和活动类型中的 @SkipRegistration 注释未注册。@ActivityRegistrationOptions 和 @SkipRegistration 注释具有覆盖语义，最具体的语义将应用于活动类型。

请注意，在类型注册后，Amazon SWF 便不允许您进行重新注册或修改。该框架将尝试注册所有类型，但如果某个类型已注册，则不会重新注册该类型，并且不会报告错误。

如果您需要修改注册的设置，必须注册新的类型版本。您也可以在启动新的执行或调用使用所生成客户端的活动时覆盖注册的设置。

注册要求提供类型名称以及其他几个注册选项。默认实现按如下方式确定这些内容：

工作流程类型名称和版本

该框架从工作流程接口中确定工作流程类型的名称。默认工作流程类型名称的格式为 {*prefix*} {*name*}。{*prefix*} 设置为 @Workflow 接口的名称，后跟“.”，{*name*} 设置为 @Execute 方法的名称。上例中工作流程类型的默认名称为 MyWorkflow.startMyWF。您可以使用 @Execute 方法的 name 参数覆盖默认名称。该示例中的工作流程类型的默认名称为 startMyWF。名称不能是空字符

串。请注意，在使用 @Execute 覆盖名称时，框架不会自动在名称前添加前缀。您可以使用任意的命名方案。

工作流程版本是使用 @Execute 注释的 version 参数指定的。version 没有默认值，必须显式指定该值；version 是一个自由格式字符串，您可以使用任意的版本控制方案。

信号名称

可以使用 @Signal 注释的 name 参数指定信号的名称。如果未指定，则默认为信号方法的名称。

活动类型名称和版本

该框架从活动接口中确定活动类型的名称。默认活动类型名称的格式为 {*prefix*}{*name*}。{*prefix*} 设置为 @Activities 接口的名称，后跟“.”，{*name*} 设置为方法名称。您可以在活动界面的 @Activities 注释中覆盖默认的 {*prefix*}。您也可以在活动方法上使用 @Activity 注释指定活动类型名称。请注意，在使用 @Activity 覆盖名称时，该框架不会自动在名称前添加前缀。您可以使用任意的命名方案。

活动版本是使用 @Activities 注释的 version 参数指定的。该版本用作接口中定义的所有活动的默认值，可以使用 @Activity 注释覆盖每个活动的版本。

默认任务列表

可以使用 @WorkflowRegistrationOptions 和 @ActivityRegistrationOptions 注释并设置 defaultTaskList 参数以配置默认任务列表。默认情况下，将它设置为 USE_WORKER_TASK_LIST。这是一个特殊值，它指示该框架使用在用于注册活动或工作流程类型的工作线程对象上配置的任务列表。您也可以选择使用这些注释将默认任务列表设置为 NO_DEFAULT_TASK_LIST，以便不注册默认任务列表。如果想要求在运行时指定任务列表，则可以使用这种方法。如果未注册任何默认任务列表，在启动工作流程或在生成的客户端的相应方法重载上使用 StartWorkflowOptions 和 ActivitySchedulingOptions 参数调用活动方法时，必须指定任务列表。

其他注册选项

Amazon SWF API 允许的所有工作流和活动类型注册选项都可以通过该框架指定。

有关工作流程注册选项的完整列表，请参阅：

- @工作流

- @Execute
- @WorkflowRegistrationOptions
- @Signal

有关活动注册选项的完整列表，请参阅：

- @活动
- @活动
- @ActivityRegistrationOptions

如果要对类型注册进行完全控制，请参阅工作线程扩展性。

活动和工作流程客户端

工作流程和活动客户端由框架基于 `@Workflow` 和 `@Activities` 接口生成。所生成的单独客户端接口包含仅对该客户端有意义的方法和设置。如果您使用 Eclipse 开发，则每次保存包含相应接口的文件时，Amazon SWF Eclipse 插件都会执行此操作。所生成的代码放在项目中生成的源目录中，位于与接口相同的程序包内。

> **Note**
>
> 请注意，Eclipse 使用的默认目录名称为 .apt_generated。Eclipse 不显示目录名称以“.”在 Package Explorer 中。如果您希望在 Project Explorer 中查看生成的文件，请使用其他目录名称。在 Eclipse 中，右键单击 Package Explorer 中的程序包，然后依次选择 Properties (属性)、Java Compiler (Java 编译器)、Annotation processing (注释处理)，并修改 Generate source directory (生成源目录) 设置。

工作流程客户端

为工作流程生成的构件包含三个客户端一侧的接口以及实施这些接口的类。生成的客户端包括：

- 异步客户端，应在工作流程实施内部使用，提供异步方法来启动工作流程执行和发送信号
- 外部客户端，可用于在工作流程实施范围之外启动执行和发送信号，以及检索工作流程状态
- 自助客户端，可用于创建连续工作流程

例如，为示例 MyWorkflow 接口生成的客户端接口为：

```
//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);
```

接口具有与所声明 `@Workflow` 接口中的各个方法对应的重载方法。

外部客户端通过接受 `StartWorkflowOptions` 的 `@Execute` 方法的一个额外重载，来镜像 `@Workflow` 接口上的方法。启动新工作流程执行时，您可以使用此重载方法传递额外的选项。通过这些选项，您可以覆盖默认任务列表、超时设置以及将标签与工作流程执行关联。

另一方面，异步客户端具有方法，这些方法允许异步调用 @Execute 方法。在客户端接口中，为工作流程接口中的 @Execute 方法生成以下方法重载：

1. 按原样接受原始参数的重载。如果原始方法返回了 void，此重载的返回类型将为 Promise<Void>；否则将为原始方法上声明的 Promise<>。例如：

   原始方法：

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

   生成的方法：

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

   在工作流的所有参数均可用并且不需要等待的时候，应使用此重载。

2. 按原样接受原始参数以及类型为 Promise<?> 的额外变量参数的重载。如果原始方法返回了 void，此重载的返回类型将为 Promise<Void>；否则将为原始方法上声明的 Promise<>。例如：

   原始方法：

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

   生成的方法：

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

   在工作流程的所有参数均可用并且不需要等待，但您希望等待一些其他 promise 就绪时，应使用此重载。变量参数可用于传递此类未声明为参数的 Promise<?> 对象，不过在执行调用之前您可能需要等待。

3. 按原样接受原始参数、一个类型为 StartWorkflowOptions 的额外参数以及类型为 Promise<?> 的额外变量参数的重载。如果原始方法返回了 void，此重载的返回类型将为 Promise<Void>；否则将为原始方法上声明的 Promise<>。例如：

   原始方法：

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

生成的方法：

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

在工作流程的所有参数均可用并且不需要等待的时候，当您希望覆盖用于启动工作流程执行的默认设置时，或者当您希望等待一些其他 promise 就绪时，应使用此重载。变量参数可用于传递此类未声明为参数的 Promise<?> 对象，不过在执行调用之前您可能需要等待。

4. 将原始方法中的各个参数使用 Promise<> 包装器替换的重载。如果原始方法返回了 void，此重载的返回类型将为 Promise<Void>；否则将为原始方法上声明的 Promise<>。例如：

   原始方法：

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

   生成的方法：

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

   要对传递到工作流程执行的参数进行异步求值时，应使用此重载。在传递到此方法重载的所有参数就绪之前，不会执行对该重载的调用。

   如果一些参数已经就绪，则通过 Promise.asPromise(*value*) 方法将这些参数转换为已处于就绪状态的 Promise。例如：

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

5. 将原始方法中的各个参数使用 Promise<> 包装器替换的重载。重载还具有类型为 Promise<?> 的额外变量参数。如果原始方法返回了 void，此重载的返回类型将为 Promise<Void>；否则将为原始方法上声明的 Promise<>。例如：

原始方法：

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

生成的方法：

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

要对传递到工作流程执行的参数进行异步求值，并且您还希望等待一些其他 promise 就绪时，应使用此重载。在传递到此方法重载的所有参数就绪之前，不会执行对该重载的调用。

6. 将原始方法中的各个参数使用 Promise<?> 包装器替换的重载。重载还具有一个类型为 StartWorkflowOptions 的额外参数以及类型为 Promise<?> 的变量参数。如果原始方法返回了 void，此重载的返回类型将为 Promise<Void>；否则将为原始方法上声明的 Promise<>。例如：

   原始方法：

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

   生成的方法：

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

   要对传递到工作流程执行的参数进行异步求值，并且您希望覆盖用于启动工作流程执行的默认设置时，使用此重载。在传递到此方法重载的所有参数就绪之前，不会执行对该重载的调用。

例如，工作流界面中的每个信号也会生成相应的方法：

原始方法：

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

生成的方法：

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

异步客户端不包含与原始接口中使用 @GetState 注释的方法相对应的方法。由于检索状态需要 Web 服务调用，它不适合在工作流程中使用。因此，它仅通过外部客户端提供。

自助客户端应在工作流程内部执行，用于在当前执行完成后启动新执行。此客户端上的方法类似于异步客户端上的方法，不过返回 void。此客户端没有与使用 @Signal 和 @GetState 注释的方法相对应的方法。有关详细信息，请参阅连续工作流程。

生成的客户端分别派生自基本接口：WorkflowClient 和 WorkflowClientExternal，其中提供了可用于取消或终止工作流程执行的方法。有关这些接口的详细信息，请参阅AWS SDK for Java文档。

生成的客户端允许您以强类型方式与工作流程执行交互。在创建后，所生成客户端的一个实例将绑定到特定工作流程执行，并且只能用于该执行。此外，该框架还提供非特定于工作流程类型或执行的动态客户端。在涵盖的范围内，生成的客户端依靠此客户端。您还可以直接使用这些客户端。请参阅动态客户端中的章节。

该框架还生成用于创建强类型客户端的工厂。为示例 MyWorkflow 接口生成的客户端工厂为：

```
//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` 基本接口为：

```
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);
}
```

您应使用这些工厂创建客户端的实例。使用工厂可以配置常规客户端 (常规客户端应该用于提供自定义客户端实施) 和由客户端用于编集数据的 `DataConverter`，以及用于启动工作流程执行 的选项。有关更多信息，请参阅DataConverters和子工作流程执行部分。`StartWorkflowOptions` 包含可用于覆盖在注册时指定的默认值（例如，超时值）的设置。有关 `StartWorkflowOptions` 类的详细信息，请参阅AWS SDK for Java文档。

外部客户端可用于从工作流程范围之外启动工作流程执行，而异步客户端可用于从工作流程中的代码启动工作流程执行。要启动执行，您只需使用生成的客户端调用与工作流程接口中使用 `@Execute` 注释的方法对应的方法。

该框架还为客户端接口生成实施类。这些客户端创建请求并发送给 Amazon SWF，以执行相应的操作。`@Execute` 方法的客户端版本要么启动新的工作流执行，要么使用 Amazon SWF API 创建子工作流执行。同样，`@Signal` 方法的客户端版本使用 Amazon SWF API 发送信号。

> **Note**
>
> 外部工作流客户端必须配置 Amazon SWF 客户端和域。您可以使用将这些配置作为参数的客户端工厂构造函数，或者传入已经配置好 Amazon SWF 客户端和域的通用客户端实施。
> 该框架遍历工作流程接口的类型层次，还会为父工作流程接口生成客户端接口并从它们进行派生。

活动客户端

与工作流程客户端类似，为使用 `@Activities` 注释的每个接口生成一个客户端。生成的构件包括客户端接口以及客户端类。为以上示例 `@Activities` 接口 (`MyActivities`) 生成的接口如下所示：

```
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);
}
```

接口包含一组重载方法，这些方法对应于 `@Activities` 接口中的各个活动方法。提供这些重载是为了方便起见，并允许异步调用活动。对于 `@Activities` 接口中的各个活动方法，在客户端接口中生成以下方法重载：

1. 按原样接受原始参数的重载。此重载的返回类型为 `Promise<`*T*`>`，其中 *T* 是原始方法的返回类型。例如：

   原始方法：

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

生成的方法：

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

在工作流的所有参数均可用并且不需要等待的时候，应使用此重载。

2. 按原样接受原始参数、一个类型为 ActivitySchedulingOptions 的参数以及类型为 Promise<?> 的额外变量参数的重载。此重载的返回类型为 Promise<*T*>，其中 *T* 是原始方法的返回类型。例如：

   原始方法：

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

   生成的方法：

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

   在工作流程的所有参数均可用并且不需要等待的时候，在您希望覆盖默认设置时，或者当您希望等待额外的 Promise 就绪时，应使用此重载。变量参数可用于传递此类未声明为参数的额外 Promise<?> 对象，不过在执行调用之前您可能需要等待。

3. 将原始方法中的各个参数使用 Promise<> 包装器替换的重载。此重载的返回类型为 Promise<*T*>，其中 *T* 是原始方法的返回类型。例如：

   原始方法：

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

   生成的方法：

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

要对传递到活动的参数进行异步求值时，应使用此重载。在传递到此方法重载的所有参数就绪之前，不会执行对该重载的调用。

4. 将原始方法中的各个参数使用 Promise<> 包装器替换的重载。重载还具有一个类型为 ActivitySchedulingOptions 的额外参数以及类型为 Promise<?> 的变量参数。此重载的返回类型为 Promise<*T*>，其中 *T* 是原始方法的返回类型。例如：

   原始方法：

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

   生成的方法：

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

   要对传递到活动的参数进行异步求值时，当您希望覆盖随类型注册的默认设置时，或者当您希望等待其他 Promise 就绪时，应使用此重载。在传递到此方法重载的所有参数就绪之前，不会执行对该重载的调用。生成的客户端类实施此接口。每个接口方法的实施都会创建请求并将其发送给 Amazon SWF，以便使用 Amazon SWF API 来安排相应类型的活动任务。

5. 按原样接受原始参数以及类型为 Promise<?> 的额外变量参数的重载。此重载的返回类型为 Promise<*T*>，其中 *T* 是原始方法的返回类型。例如：

   原始方法：

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

   生成的方法：

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

   在所有活动参数均可用并且无需等待，但您希望等待另一个 Promise 对象就绪时，应使用此重载。

6. 将原始方法中各个参数替换为 Promise 包装器并具有类型为 Promise<?> 的附加变量参数的重载。此重载的返回类型为 Promise<*T*>，其中 *T* 是原始方法的返回类型。例如：

原始方法：

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

生成的方法：

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

在活动的所有参数将等待异步处理并且您还希望等待某些其他的 Promise 就绪时，应使用此重载。在传递的所有 Promise 对象都就绪后，对此方法重载的调用将异步执行。

生成的活动客户端还具有与各个活动方法对应的受保护方法，方法的名称为 {*activity method name*}Impl()，所有活动重载都将调用该方法。您可以覆盖此方法以创建模拟客户端实施。此方法接受以下对象作为参数：传递到 Promise<> 包装器中原始方法的所有参数、ActivitySchedulingOptions，以及类型为 Promise<?> 的变量参数。例如：

原始方法：

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

生成的方法：

```
Promise<Void> activity2Impl(
   Promise<Integer> foo,
   ActivitySchedulingOptions optionsOverride,
   Promise<?>...waitFor);
```

计划选项

生成的活动客户端允许您传入 ActivitySchedulingOptions 作为参数。ActivitySchedulingOptions 结构包含的设置可决定框架在 Amazon SWF 中安排的活动任务配置。这些设置覆盖指定为注册选项的默认值。要动态指定计划选项，请创建 ActivitySchedulingOptions 对象，根据需要进行配置，并将该对象传递到活动方法。在以下示例中，我们指定了应该用于活动任务的任务列表。这会覆盖活动的此调用的默认注册任务列表。

```
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);
    }
}
```

动态客户端

除了所生成的客户端之外，框架还会提供通用客户端 DynamicWorkflowClient 和 DynamicActivityClient，您可以使用这些客户端来动态启动工作流执行、发送信号和安排活动等。例如，您可能希望计划其类型在设计时未知的活动。您可以使用 DynamicActivityClient 来计划此类活动任务。与此类似，您可以使用 DynamicWorkflowClient 来动态计划子工作流程执行。在以下示例中，工作流程从数据库中查找活动并使用动态活动客户端来计划该活动：

```
//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);
}
```

有关详细信息，请参阅AWS SDK for Java文档。

发送信号和取消工作流程执行

所生成的工作流程客户端具有与可发送到工作流程的各个信号相对应的方法。您可以在工作流程中使用这些方法将信号发送到其他工作流程执行。这为发送信号提供了类型化机制。但有时您可能需要动态确定信号名称，例如，在消息中接收信号名称时。您可以使用动态工作流程客户端将信号动态地发送到任意工作流程执行。与此类似，您可以使用客户端来请求取消另一个工作流程执行。

在以下示例中，工作流程查找将信号从数据库发送到的执行，并使用动态工作流程客户端来动态发送信号。

```
//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);
}
```

工作流程实施

要实施工作流程，您可以编写一个实施所需 @Workflow 接口的类。例如，可按以下所示实施示例工作流程接口 (MyWorkflow)：

```
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);
  }
}
```

此类中的 @Execute 方法是工作流程逻辑的入口点。由于在要处理决策任务时，框架会使用回放来重新构造对象状态，因此为每个决策任务创建一个新对象。

禁止在 @Workflow 接口中的 @Execute 方法内使用 Promise<*T*> 作为参数。这样做是因为发出异步调用完全是调用方的决策。工作流程实施本身不依赖于调用是同步的还是异步的。因此，生成的客户端接口具有采用 Promise<*T*> 参数的重载，以便能够异步调用这些方法。

@Execute 方法的返回类型只能为 void 或 Promise<*T*>。请注意，相应的外部客户端的返回类型为 void 而不是 Promise<>。由于外部客户端不是通过异步代码使用的，因此，外部客户端不会返回 Promise 对象。要获得外部声明的工作流执行的结果，您可以设计工作流，通过活动来更新外部数据存储中的状态。Amazon SWF 的可见性 API 也可以用于检索工作流结果，以便进行诊断。一般情况下，建议不要使用可见性 API 检索工作流的执行结果，因为这些 API 调用可能会被 Amazon SWF 限制。可见性 API 需要您使用 WorkflowExecution 结构标识工作流程执行。您可以通过调用 getWorkflowExecution 方法，从生成的工作流程客户端获取此结构。此方法将返回与客户端绑定到的工作流程执行对应的 WorkflowExecution 结构。有关可见性 API 的更多详细信息，请参阅 Amazon Simple Workflow Service API Reference。

在从工作流程实施调用活动时，您应使用生成的活动客户端。同样，要发送信号，请使用生成的工作流程客户端。

决策上下文

当框架执行工作流程代码时，它会提供一个环境上下文。此上下文提供了您可在工作流程实施中访问的上下文特定的功能，例如创建计时器。有关更多信息，请参阅执行关联部分。

公开执行状态

Amazon SWF 允许您在工作流历史记录中添加自定义状态。工作流执行所报告的最新状态将通过对 Amazon SWF 服务的可见性调用和 Amazon SWF 控制台返回给您。例如，在订单处理工作流程中，您可以在不同的阶段 (如“已收到订单”、“订单已配送”等) 报告订单状态。在适用于 Java 的 AWS Flow Framework 中，这将通过在工作流接口上使用 @GetState 注释进行注释的方法实现。在决策程序处理完决策任务后，它将调用此方法，以便从工作流程实施中获取最新状态。除了可见性调用之外，还可使用生成的外部客户端 (它在内部使用可见性 API 调用) 来检索状态。

以下示例演示了如何设置执行上下文。

```
@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()
```

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

生成的外部客户端可随时用于检索工作流程执行的最新状态。

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

在上述示例中，在各个阶段报告执行状态。当工作流程实例启动时，periodicWorkflow 将初始状态报告为“刚刚启动”。之后，每次调用 callPeriodicActivity 都会更新工作流程状态。在调用 activity1 100 次后，此方法返回并且工作流程实例完成。

本地工作流程

有时，您可能需要在工作流程实施中使用静态变量。例如，您可能需要存储要从工作流程实施中的各个位置 (可能是不同的类) 访问的计数器。但是，您不能依赖工作流程中的静态变量，因为静态变量是跨线程共享的，这是有问题的，因为一个工作线程可能同时在不同的线程上处理不同的决策任务。或者，您可以将此类状态存储在工作流程实施的字段中，但您随后需要传递实施对象。为了满足此需求，框架提供了一个 WorkflowExecutionLocal<?> 类。任何需要静态变量 (如语义) 的状态都应通过 WorkflowExecutionLocal<?> 在本地保留为实例。您可以声明和使用此类型的静态变量。例如，在以下代码段中，WorkflowExecutionLocal<String> 用于存储用户名。

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

活动实现

通过提供 @Activities 接口的实现来实现活动。适用于 Java 的 AWS Flow Framework 使用在工作线程上配置的活动实现实例在运行时处理活动任务。工作线程会自动查找适当类型的活动实现。

您可以使用属性和字段来将资源传递给活动实例，如数据库连接。由于可能从多个线程访问活动实现对象，共享资源必须是线程安全的。

请注意，活动实现不使用 Promise<> 类型的参数或返回该类型的对象。这是因为活动的实现不应依赖它的调用方式 (同步或异步)。

之前显示的活动接口可按如下方式实现：

```
public class MyActivitiesImpl implements MyActivities {

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

   @Override
   public void activity2(int foo){
```

```
      //implementation
    }
}
```

可以向活动实现提供线程本地上下文，这可用于检索任务对象、所使用的数据转换器对象等。可通过 `ActivityExecutionContextProvider.getActivityExecutionContext()` 访问当前上下文。有关详细信息，请参阅 `ActivityExecutionContext` 的AWS SDK for Java文档和执行关联一节。

手动完成活动

以上示例中的 `@ManualActivityCompletion` 注释是可选注释。仅在实现活动的方法上允许它，用于将活动配置为不在活动方法返回时自动完成。如果您想异步完成活动，例如，在人工操作完成后手动完成活动，这会非常有用。

默认情况下，当您的活动方法返回时，框架认为活动已完成。这意味着活动工作线程会向 Amazon SWF 报告活动任务已完成，并向其提供结果（如果有）。但是，在一些使用案例中，您不希望在活动方法返回时将活动任务标记为已完成。在为人工任务建模时，这尤其有用。例如，活动方法可能将电子邮件发送给某人，该人必须完成某些工作，活动任务才能完成。在这种情况下，您可以使用 `@ManualActivityCompletion` 注释来注释此活动方法以告诉活动工作线程它不应自动完成活动。要手动完成活动，您可以使用框架中提供的 `ManualActivityCompletionClient` 或使用 Amazon SWF SDK 中提供的 Amazon SWF Java 客户端上的 `RespondActivityTaskCompleted` 方法。有关详细信息，请参阅AWS SDK for Java文档。

要完成活动任务，您需要提供任务令牌。Amazon SWF 会使用任务令牌来唯一标识任务。您可以从活动实现中的 `ActivityExecutionContext` 访问此令牌。您必须将此令牌传递给负责完成任务的一方。可以通过调用 `ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken()` 来从 `ActivityExecutionContext` 检索此令牌。

Hello World 示例的 `getName` 活动可实现为发送电子邮件来要求某人提供问候语：

```
@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";
}
```

以下代码段可用于提供问候并通过使用 ManualActivityCompletionClient 来关闭任务。或者，您也可以使任务失败：

```
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);
    }
}
```

实施 AWS Lambda 任务

主题

- 关于 AWS Lambda
- 使用 Lambda 任务的优势和限制
- 在适用于 Java 的 AWS Flow Framework 工作流中使用 Lambda 任务
- 查看 HelloLambda 示例

关于 AWS Lambda

AWS Lambda 是一种完全托管的计算服务，可运行您的代码以响应由自定义代码生成的事件或来自各种 AWS 服务（如 Amazon S3、DynamoDB、Amazon Kinesis、Amazon SNS 和 Amazon Cognito）的事件。有关 Lambda 的更多信息，请参阅 AWS Lambda 开发人员指南。

Amazon Simple Workflow Service 提供了一项 Lambda 任务，以便您可以运行 Lambda 函数来代替传统的 Amazon SWF 活动，或与此类活动一起运行。

> **Important**
>
> 您的 AWS 账户需要针对 Amazon SWF 代表您执行的 Lambda 执行（请求）付费。有关 Lambda 定价的详细信息，请参阅 https://aws.amazon.com/lambda/pricing/。

使用 Lambda 任务的优势和限制

使用 Lambda 任务替代传统 Amazon SWF 活动具有许多优势：

- Lambda 任务不需要像 Amazon SWF 活动类型一样注册或版本化。
- 您可以使用已在工作流中定义的任何现有 Lambda 函数。
- Lambda 函数由 Amazon SWF 直接调用，无需像传统活动那样，需要实现工作线程程序才能执行。
- Lambda 为您提供指标和日志，用于跟踪和分析函数的执行情况。

您还应了解 Lambda 任务有很多限制：

- Lambda 任务只能在支持 Lambda 的 AWS 区域运行。要详细了解当前支持 Lambda 的区域，请参阅 Amazon Web Services General Reference中的 Lambda Regions and Endpoints。
- 目前，只有基础 SWF HTTP API 和在适用于 Java 的 AWS Flow Framework 中才支持 Lambda 任务。适用于 Ruby 的 AWS Flow Framework 中当前不支持 Lambda 任务。

在适用于 Java 的 AWS Flow Framework 工作流中使用 Lambda 任务

在适用于 Java 的 AWS Flow Framework 工作流中使用 Lambda 任务有三个要求：

- 要执行的 Lambda 函数。您可以使用已定义的任何 Lambda 函数。有关创建 Lambda 函数的更多信息，请参阅 AWS Lambda Developer Guide。

- IAM 角色，用于提供访问权限以从 Amazon SWF 工作流中执行 Lambda 函数。
- 代码，用于在工作流中安排 Lambda 任务。

设置 IAM 角色

在从 Amazon SWF 调用 Lambda 函数之前，您必须先提供一个 IAM 角色，用于从 Amazon SWF 访问 Lambda。您可以：

- 选择预定义的角色 AWSLambdaRole，为工作流提供调用与您账户关联的任何 Lambda 函数的访问权限。
- 定义您自己的策略和关联角色，为工作流提供调用由其 Amazon 资源名称 (ARN) 指定的特定 Lambda 函数的访问权限。

限制 IAM 角色的访问权限

您可以使用资源信任策略中的 `SourceArn` 和 `SourceAccount` 上下文密钥来限制提供给 Amazon SWF 的 IAM 角色的访问权限。这些密钥会限制 IAM 策略的使用，使其只能在属于指定域 ARN 的 Amazon Simple Workflow Service 执行中使用。如果您同时使用两个全局条件上下文密钥，则在同一策略语句中使用 `aws:SourceAccount` 值和 `aws:SourceArn` 值中引用的账户时，必须使用相同的账户 ID。

在下面的信任策略示例中，我们使用 `SourceArn` 上下文密钥将 IAM 服务角色限制为只能在属于账户 `123456789012` 中的 `someDomain` 的 Amazon Simple Workflow Service 执行中使用。

```
{
  "Version": "2012-10-17",
  "Statement": [
     {
       "Sid": "",
       "Effect": "Allow",
       "Principal": {
         "Service": "swf.amazonaws.com"
       },
       "Action": "sts:AssumeRole",
       "Condition": {
         "ArnLike": {
           "aws:SourceArn": "arn:aws:swf:*:123456789012:/domain/someDomain"
       }
      }
```

```
    }
  ]
}
```

在下面的信任策略示例中，我们使用 SourceAccount 上下文密钥将 IAM 服务角色限制为只能在属于账户 123456789012 的 Amazon Simple Workflow Service 执行中使用。

```
{
  "Version": "2012-10-17",
  "Statement": [
     {
       "Sid": "",
       "Effect": "Allow",
       "Principal": {
         "Service": "swf.amazonaws.com"
       },
       "Action": "sts:AssumeRole",
       "Condition": {
         "StringLike": {
           "aws:SourceAccount": "123456789012"
       }
      }
    }
  ]
}
```

为 Amazon SWF 提供调用任何 Lambda 角色的访问权限

您可以使用预定义的角色 AWSLambdaRole，使您的 Amazon SWF 工作流能够调用与您的账户相关联的任何 Lambda 函数。

使用 AWSLambdaRole 为 Amazon SWF 提供调用 Lambda 函数的访问权限

1. 打开 Amazon IAM 控制台。
2. 选择 Roles，然后选择 Create New Role。
3. 提供角色名称 (如 swf-lambda)，然后选择 Next Step。
4. 在 AWS 服务角色下，选择 Amazon SWF，然后选择下一步。
5. 在 Attach Policy 屏幕上，从列表中选择 AWSLambdaRole。
6. 检查角色之后，选择 Next Step，然后选择 Create Role。

定义 IAM 角色以提供调用特定 Lambda 函数的访问权限

如果您要提供从工作流调用特定 Lambda 函数的访问权限，则需要定义自己的 IAM 策略。

创建 IAM 策略以提供对特定 Lambda 函数的访问权限

1. 打开 Amazon IAM 控制台。
2. 选择 Policies，然后选择 Create Policy。
3. 选择复制 AWS 托管式策略，然后从列表中选择 AWSLambdaRole。随即将生成策略。根据需要编辑策略的名称和描述。
4. 在策略文档的资源字段中，添加您的 Lambda 函数的 ARN。例如：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:us-east-1:111111000000:function:hello_lambda_function"
      ]
    }
  ]
}
```

> **Note**
>
> 有关如何在 IAM 角色中指定资源的完整说明，请参阅《Using IAM》中的 Overview of IAM Policies。

5. 选择 Create Policy 完成策略创建。

您随后可以在创建新的 IAM 角色时选择该策略，并使用该角色提供对 Amazon SWF 工作流的调用访问权限。此过程与使用 AWSLambdaRole 策略创建角色非常相似。不同之处是在创建角色时选择自己的策略。

使用 Lambda 策略创建 Amazon SWF 角色

1. 打开 Amazon IAM 控制台。
2. 选择 Roles，然后选择 Create New Role。
3. 提供角色名称 (如 `swf-lambda-function`)，然后选择 Next Step。
4. 在 AWS 服务角色下，选择 Amazon SWF，然后选择下一步。
5. 在附加策略屏幕上，从列表中选择特定于 Lambda 函数的策略。
6. 检查角色之后，选择 Next Step，然后选择 Create Role。

安排要执行的 Lambda 任务

在定义允许调用 Lambda 函数的 IAM 角色后，您可以安排在工作流中执行这些任务。

> **Note**
>
> AWS SDK for Java 中的 HelloLambda sample 完整演示了这一过程。

安排要执行的 Lambda 任务

1. 在您的工作流程实现中，在 `DecisionContext` 实例上调用 `getLambdaFunctionClient()` 以获取一个 `LambdaFunctionClient` 实例。

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

2. 使用 `LambdaFunctionClient` 上的 `scheduleLambdaFunction()` 方法安排任务，并向其传递创建的 Lambda 函数名称和 Lambda 任务的任何输入数据。

```
// 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. 在工作流执行启动程序中，使用 `StartWorkflowOptions.withLambdaRole()` 将 IAM Lambda 角色添加到默认工作流选项中，然后在启动工作流时传递这些选项。

```
// 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);
```

查看 HelloLambda 示例

AWS SDK for Java 中提供了使用 Lambda 任务实现工作流的示例。要查看和/或运行该示例，请下载源文件。

有关如何构建和运行 HelloLambda 示例的完整说明，请参阅随适用于 Java 的 AWS Flow Framework 示例提供的 README 文件。

运行使用适用于 Java 的 AWS Flow Framework 编写的程序

主题

- WorkflowWorker
- ActivityWorker
- 工作线程的线程模型
- 工作线程扩展性

该框架提供工作线程类，用于初始化适用于 Java 的 AWS Flow Framework 运行时并与 Amazon SWF 通信。为了实现工作流或活动工作线程，您必须创建并启动工作线程类的实例。这些工作线程类负责管理持续进行的异步操作、调用取消阻止的异步方法，以及与 Amazon SWF 通信。可通过工作流和活动实现、线程数量、要轮询的任务列表等对其进行配置。

该框架附带两个工作线程类，一个用于活动，一个用于工作流。要运行工作流逻辑，需使用 `WorkflowWorker` 类。同样，对活动使用 `ActivityWorker` 类。这些类会自动轮询 Amazon SWF 以获取活动任务，并在实现中调用相应的方法。

以下示例显示如何实例化 `WorkflowWorker` 和开始轮询任务：

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

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

创建 `ActivityWorker` 实例和开始轮询任务的基本步骤如下：

```
AmazonSimpleWorkflow swfClient
      = new AmazonSimpleWorkflowClient(awsCredentials);
ActivityWorker worker = new ActivityWorker(swfClient,
                                           "domain1",
                                           "tasklist1");
worker.addActivitiesImplementation(new MyActivitiesImpl());

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

当您要关闭活动或决策程序时，您的应用程序应关闭正在使用的工作线程类的实例以及 Amazon SWF Java 客户端实例。这将确保正确释放该工作线程类使用的所有资源。

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

要开始执行，只需创建生成的外部客户端的实例并调用 `@Execute` 方法。

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

WorkflowWorker

顾名思义，此工作线程类供工作流实现使用。它配置了任务列表和工作流实现类型。工作线程类运行循环，在指定的任务列表中轮询决策任务。当收到决策任务时，它会创建工作流实现的实例并调用 `@Execute` 方法来处理任务。

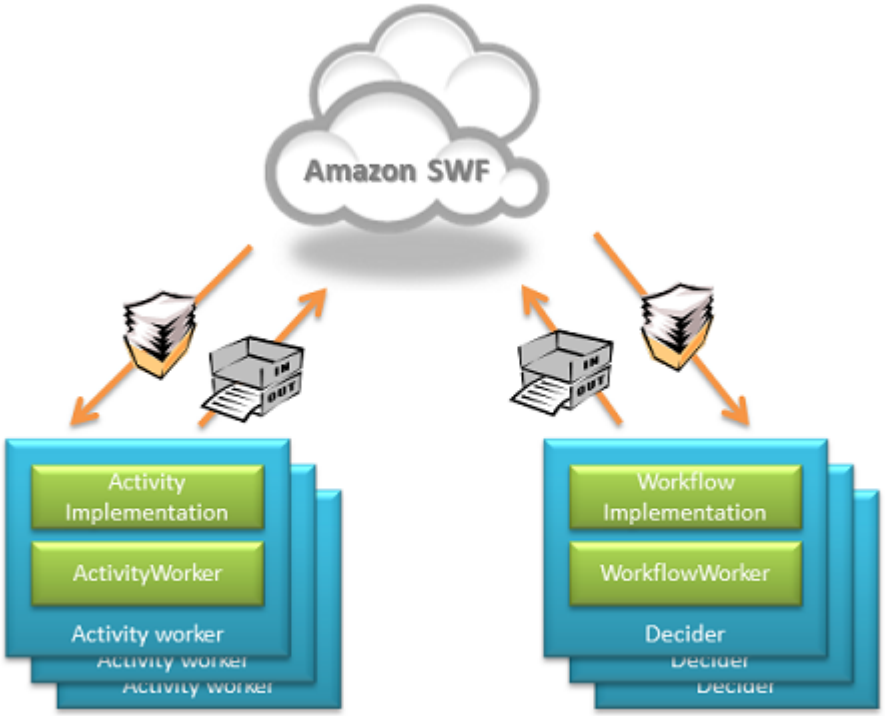
ActivityWorker

对于实现活动工作线程，您可以使用 `ActivityWorker` 类来方便地轮询任务列表，以查找活动任务。可为活动工作线程配置活动实现对象。此工作线程类运行循环，在指定的任务列表中轮询活动任务。收到活动任务时，它会查找您提供的相应实现并调用活动方法来处理任务。与调用工厂来为每个决策任务创建新实例的 `WorkflowWorker` 不同，`ActivityWorker` 直接使用您提供的对象。

`ActivityWorker` 类使用适用于 Java 的 AWS Flow Framework 注释来确定注册和执行选项。

工作线程的线程模型

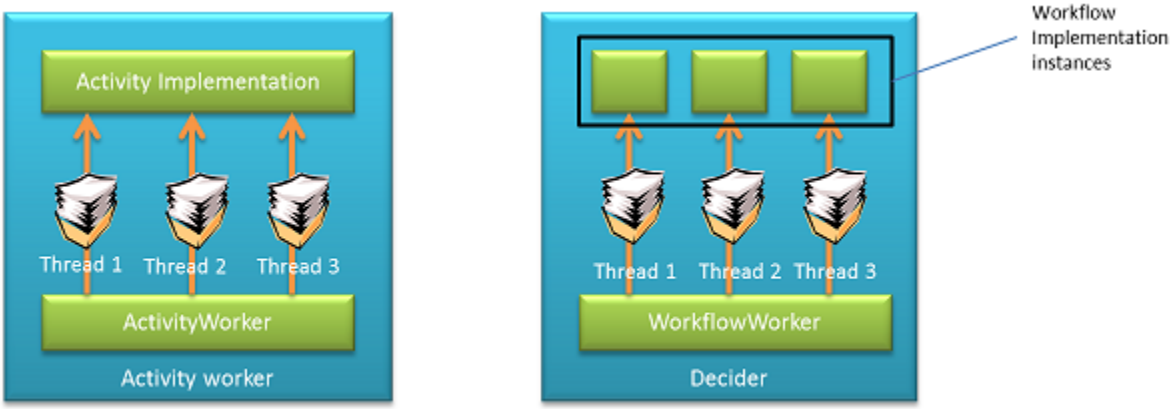
在适用于 Java 的 AWS Flow Framework 中，活动或决策程序都体现为工作线程类的实例。您的应用程序负责在每台机器和每个进程上配置和实例化应作为工作线程的工作线程对象。然后，工作线程对象将自动接收来自 Amazon SWF 的任务，将其分派给您的活动或工作流实现，并向 Amazon SWF 报告结果。单个工作流实例可能跨越许多工作线程。如果 Amazon SWF 有一个或多个待处理的活动任务，它会将任务分配给第一个可用的工作线程，然后再分派给下一个，以此类推。这样便可同时在不同工作线程中处理属于同一工作流实例的任务。



此外，还可以将每个工作线程配置为在多个线程上处理任务。这意味着，即使只有一个工作线程，工作流实例的活动任务也可以并行运行。

决策任务的行为类似，但例外情况是 Amazon SWF 保证在执行给定工作流时，一次仅执行一个决策。单个工作流执行通常需要多个决策任务；因此，这也可能导致在多个进程和线程上执行。决策程序配置了工作流实现的类型。当决策程序收到决策任务时，它会创建工作流实现的实例 (对象)。该框架提供了用于创建这些实例的可扩展工厂模式。默认工作流工厂每次创建一个新对象。您可以提供自定义工厂来覆盖此行为。

与配置了工作流实现类型的决策程序不同，活动工作线程配置了活动实现的实例 (对象)。当活动工作线程收到活动任务时，该任务将被分派至适当的活动实现对象。



工作流工作线程维护单个线程池，并在用于轮询 Amazon SWF 以获取任务的同一线程中执行工作流。由于活动要长时间运行（至少与工作流逻辑相比），活动工作线程类需要维护两个独立的线程池；一个用于轮询 Amazon SWF 以获取活动任务，另一个通过执行活动实现来处理任务。这样，您便可以独立于用于执行任务的线程数，来配置用于轮询任务的线程数。例如，您可以将少量线程用于轮询，而将大量线程用于执行任务。活动工作线程类仅在有可用的轮询线程和用于处理任务的可用线程时，才会轮询 Amazon SWF 以获取任务。

此线程和实例行为意味着：

1. 活动实现必须是无状态的。不应使用实例变量在活动对象中存储应用程序状态。但是，您可以使用字段来存储数据库连接等资源。
2. 活动实现必须是线程安全的。由于相同实例可同时用来处理来自不同线程的任务，因此活动代码对共享资源的访问必须同步。
3. 工作流实现可以是有状态的，实例变量可用于存储状态。即使创建工作流实现的新实例来处理每个决策任务，架构也将确保可正确地重新创建状态。但是，工作流实现必须是确定性的。有关更多详细信息，请参阅深入剖析 一节。
4. 在使用默认工厂时，工作流实现不需要是线程安全的。默认实现确保，一次只有一个线程使用工作流实现的一个实例。

工作线程扩展性

适用于 Java 的 AWS Flow Framework 还包含一些低级别的工作线程类，可为您提供精细控制和扩展性。使用它们，您可以完全自定义工作流和活动类型注册，并设置用于创建实现对象的工厂。这些工作线程为 `GenericWorkflowWorker` 和 `GenericActivityWorker`。

`GenericWorkflowWorker` 可以配置一个工厂，以创建工作流定义工厂。工作流定义工厂负责创建工作流实现的实例以及提供注册选项等配置设置。在正常情况下，您应直接使用 `WorkflowWorker` 类。它会自动创建和配置框架中所提供的工厂 `POJOWorkflowDefinitionFactoryFactory` 和 `POJOWorkflowDefinitionFactory` 的实现。该工厂要求工作流实现类必须具有无参数的构造函数。该构造函数用于在运行时创建工作流对象的实例。该工厂会查看您在工作流接口和实现中使用的注释，以创建相应的注册和执行选项。

您可以通过实现 `WorkflowDefinitionFactory`、`WorkflowDefinitionFactoryFactory` 和 `WorkflowDefinition` 来提供您自己的工厂实现。`WorkflowDefinition` 类供工作线程类用来分派决策任务和信号。通过实现这些基类，您可以完全自定义工厂并向工作流实现分派请求。例如，您可以使用这些扩展性点根据您自己的注释提供用于编写工作流的自定义编程模型，或从 WSDL 而不是框

架使用的代码优先方法生成该模型。要使用您的自定义工厂，您必须使用 GenericWorkflowWorker 类。有关这些类的详细信息，请参阅AWS SDK for Java 文档。

同样，GenericActivityWorker 允许您提供自定义活动实现工厂。通过实现 ActivityImplementationFactory 和 ActivityImplementation 类，您可以完全控制活动实例化以及自定义注册和执行选项。有关这些类的详细信息，请参阅AWS SDK for Java 文档。

执行关联

主题

该框架为工作流程和活动实现提供环境上下文。该上下文是处理的任务特有的，并提供一些可在您的实现中使用的实用工具。每次工作线程处理新任务时，将会创建一个上下文对象。

决策上下文

在执行决策任务时，该框架通过 DecisionContext 类为工作流程实现提供上下文。DecisionContext 提供上下文敏感信息，如工作流程执行运行 ID 以及时钟和计时器功能。

在工作流程实现中访问 DecisionContext

您可以使用 DecisionContextProviderImpl 类在您的工作流程实现中访问 DecisionContext。或者，您也可以使用 Spring 在您的工作流程实现的字段或属性中注入上下文，如“可测性和依赖关系注入”一节中所示。

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

创建时钟和计时器

DecisionContext 包含一个 WorkflowClock 类型的属性，它提供计时器和时钟功能。由于工作流程逻辑需要是确定性的，因此，不应在您的工作流程实现中直接使用系统时钟。WorkflowClock 上的 currentTimeMills 方法返回处理的决策的启动事件的时间。这可确保在重播期间获得相同的时间值，从而使您的工作流程逻辑具有确定性。

WorkflowClock 还具有一个返回 Promise 对象的 createTimer 方法，该对象在指定的间隔后变为就绪状态。您可以将该值作为其他异步方法的参数，以将其执行推迟指定的一段时间。这样，您实际上可以将异步方法或活动安排在以后的时间执行。

以下列表中的示例说明了如何定期调用活动。

```
@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() {
      ...
      }
}
```

在上述列表中，callPeriodicActivity 异步方法调用 activity1，然后使用当前 AsyncDecisionContext 创建一个计时器。它将返回的 Promise 作为参数传递到对其自身的递归调用。该递归调用等到计时器触发 (本示例中为 1 小时)，然后再执行。

活动执行上下文

就像 DecisionContext 在处理决策任务时提供上下文信息一样，ActivityExecutionContext 在处理活动任务时提供类似的上下文信息。将通过 ActivityExecutionContextProviderImpl 类为您的活动代码提供该上下文。

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

通过使用 ActivityExecutionContext，您可以执行以下操作：

为长时间运行的活动提供检测信号

如果活动长期运行，则必须定期向 Amazon SWF 报告进度，让它知道任务仍在进行中。如果没有此类检测信号，并且在注册活动类型或计划活动时设置了任务检测信号超时，则任务可能会超时。要发送检测信号，您可以在 recordActivityHeartbeat 上使用 ActivityExecutionContext 方法。检测信号还提供了一个机制以取消正在执行的活动。有关更多详细信息和示例，请参阅错误处理一节。

获取活动任务详细信息

如果需要，您可以获取 Amazon SWF 在执行程序获取任务时传递的所有活动任务详细信息。这包括有关任务输入、任务类型和任务令牌等的信息。如果要实施手动完成的活动（例如，通过人工操作完成），则必须使用 `ActivityExecutionContext` 来检索任务令牌，并将其传递给最终完成活动任务的流程。有关更多详细信息，请参阅有关手动完成活动的一节。

获取执行程序使用的 Amazon SWF 客户端对象

执行程序使用的 Amazon SWF 客户端对象可以通过调用 `ActivityExecutionContext` 上的 `getService` 方法来获取。如果您想直接调用 Amazon SWF 服务，这会非常有用。

子工作流程执行

在以前的示例中，我们直接从应用程序中启动工作流程执行。不过，可以在生成的客户端上调用工作流程入口点方法，以从工作流程中启动工作流程执行。在从其他工作流程执行的上下文中启动工作流程执行时，它称为子工作流程执行。这样，您就可以将复杂工作流程重构为较小的单元，并且可能会在不同的工作流程之间共享这些单元。例如，您可以创建一个支付处理工作流程，并从订单处理工作流程中调用该工作流程。

从语义上讲，子工作流程执行的行为与单独工作流程相同，但存在以下差异：

1. 如果父工作流因用户执行的显式操作而终止（例如，因调用 `TerminateWorkflowExecution` Amazon SWF API 或因超时而终止），那么子工作流的执行将由子策略决定。您可以设置该子策略以终止、取消或放弃 (保持运行) 子工作流程执行。
2. 就像异步方法返回的 `Promise<T>` 一样，父工作流程执行可以使用子工作流程的输出 (入口点方法的返回值)。这与单独执行不同，在单独执行中，应用程序必须使用 Amazon SWF API 才能获得输出。

在以下示例中，`OrderProcessor` 工作流程创建一个 `PaymentProcessor` 子工作流程：

```
@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);

}
```

连续工作流程

在某些使用案例中，您可能需要使用永远执行或长时间运行的工作流程，例如，监控服务器队列运行状况的工作流程。

> **Note**
>
> 由于 Amazon SWF 保留工作流程执行的完整历史记录，因此，历史记录将随着时间的推移不断增大。在执行重播时，该框架从 Amazon SWF 中检索该历史记录，如果历史记录太大，这会产生高昂的成本。在这种长时间运行或连续工作流程中，您应该定期关闭当前执行并启动新的执行以继续处理。

这是工作流程执行的逻辑延续。可以将生成的自客户端用于该目的。在您的工作流程实现中，直接在自客户端上调用 @Execute 方法。在当前执行完成后，该框架将使用相同的工作流程 ID 启动新的执行。

您也可以在可从当前 DecisionContext 中检索的 GenericWorkflowClient 上调用 continueAsNewOnCompletion 方法以继续执行。例如，以下工作流程实现设置一个计时器以在一天后触发，并调用自己的入口点以启动新的执行。

```
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();
    }
}
```

在工作流程递归调用自身时，该框架将在所有待办任务完成时关闭当前工作流程，并启动新的工作流程执行。请注意，只要具有待办任务，就不会关闭当前工作流程执行。新执行不会从原始执行中自动继承任何历史记录或数据；如果要将某种状态转移到新执行，您必须显式将其作为输入传递。

设置任务优先级

默认情况下，任务列表上的任务将基于其到达时间进行交付：首先安排的任务通常会尽可能首先运行。通过设置可选的任务优先级，您可以设定特定任务的优先级：Amazon SWF 会尝试先交付任务列表上优先级较高的任务，然后再交付优先级较低的任务。

您可以同时为工作流和活动设置任务优先级。工作流的任务优先级既不影响其安排的任何活动的任务优先级，也不影响其开始的任何子工作流。活动或工作流的默认优先级是由您或 Amazon SWF 在注册过程中设置的，除非在安排活动或启动工作流执行时覆盖了注册的任务优先级，否则应始终使用默认优先级。

任务优先级值的范围可为“-2147483648”到“2147483647”，数字越大优先级越高。如果您未为活动或工作流设置任务优先级，则将为其分配零（“0”）优先级。

主题

- 设置工作流的任务优先级
- 设置活动的任务优先级

设置工作流的任务优先级

在您注册或开始工作流时，可以设置其任务优先级。注册工作流类型时设置的任务优先级将用作执行该类型的任何工作流的默认优先级，除非该优先级在开始执行工作流时被覆盖。

要注册具有默认任务优先级的工作流类型，请在声明工作流类型时在 WorkflowRegistrationOptions 中设置 defaultTaskPriority 选项：

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

您还可以在启动工作流时为它设置 taskPriority，覆盖注册的 (默认) 任务优先级。

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

PriorityWorkflowClientExternalFactory cf
    = new PriorityWorkflowClientExternalFactoryImpl(swfService, domain);

priority_workflow_client = cf.getClient();

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

此外，在启动子工作流或将工作流作为新的工作流继续执行时，可以设置任务优先级。例如，您可以在 ContinueAsNewWorkflowExecutionParameters 或 StartChildWorkflowExecutionParameters 中设置 taskPriority 选项。

设置活动的任务优先级

您可以在注册或安排活动时设置该活动的任务优先级。注册活动类型时设置的任务优先级将用作运行活动时的默认优先级，除非该优先级在安排活动时被覆盖。

要注册具有默认任务优先级的活动类型，请在声明活动类型时在 ActivityRegistrationOptions 中设置 defaultTaskPriority 选项：

```
@Activities(version = "1.0")
@ActivityRegistrationOptions(
    defaultTaskPriority = 10,
    defaultTaskStartToCloseTimeoutSeconds = 120)
public interface ImportantActivities {
    int doSomethingImportant();
}
```

您还可以在安排活动时为它设置 taskPriority，覆盖注册的 (默认) 任务优先级。

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

ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();

activityClient.doSomethingImportant(activityOptions);
```

DataConverters

当您的工作流实现调用远程活动时，必须序列化传递给它的输入和执行该活动的结果，以便可以通过线路发送它们。框架使用 DataConverter 类来实现此目的。这是一个抽象类，您可以实现它来提供自己的序列化器。框架中提供了一个基于 Jackson 序列化器的默认实施 `JsonDataConverter`。有关更多详细信息，请参阅AWS SDK for Java 文档。有关 Jackson 如何执行序列化以及可用于影响它的 Jackson 注释的详细信息，请参阅 Jackson JSON 处理器文档。所使用的线路格式被视为合同的一部分。因此，您可以在活动和工作流接口上通过设置 `@Activities` 和 `@Workflow` 注释的 `DataConverter` 属性来指定 `DataConverter`。

框架将创建您在 `@Activities` 注释上指定的 `DataConverter` 类型的对象来序列化活动的输入并反序列化其结果。同样地，您在 `@Workflow` 注释上指定的 `DataConverter` 类型的对象将用于序列化您传递给工作流的参数，并为子工作流反序列化结果。除了输入之外，框架还向 Amazon SWF 传递其他数据，例如异常详细信息，工作流序列化器也将用于序列化这些数据。

您还可以提供 `DataConverter` 的实例 (如果您不希望框架自动创建它)。生成的客户端具有使用 `DataConverter` 的构造函数重载。

如果您没有指定 `DataConverter` 类型，并且没有传递 `DataConverter` 对象，则默认使用 `JsonDataConverter`。

将数据传递到异步方法

主题

在前几节中介绍了如何使用 Promise<*T*>。此处介绍 Promise<*T*> 的一些高级使用案例。

将集合和映射传递到异步方法

该框架支持将数组、集合和映射作为 Promise 类型传递到异步方法。例如，异步方法可以将 Promise<ArrayList<String>> 作为参数，如以下列表所示。

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

从语义地讲，它与任何其他 Promise 类型的参数类似，异步方法将等到集合变为可用，然后再执行。如果集合成员是 Promise 对象，您可以让该框架等待所有成员变为就绪状态，如以下代码段所示。这会使异步方法等待集合的每个成员变为可用。

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

请注意，必须在参数中使用 @Wait 注释以指示它包含 Promise 对象。

另请注意，活动 printActivity 使用 String 参数，但生成的客户端中的匹配方法使用 Promise<String>。我们将调用客户端上的方法，而不是直接调用活动方法。

可设置 <T>

Settable<*T*> 是 Promise<T> 的派生类型，它提供一个设置方法以允许手动设置 Promise 值。例如，以下工作流程等待 Settable<?> 以等待接收信号，后者是在信号方法中设置的：

```
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;
   }
}
```

每次还可以将一个 Settable<?> 链接到另一个 promise。您可以使用 AndPromise 和 OrPromise 对 promise 进行分组。您可以在链接的 Settable 上调用 unchain() 方法以将其取消链接。在链接后，在链接的 promise 变为就绪状态时，Settable<?> 自动变为就绪状态。如果要在程序的其他部分中使用从 doTry() 作用域中返回的 promise，链接是特别有用的。由于 TryCatchFinally 用作嵌套类，您无法在父类的作用域中声明 Promise<> 并在 doTry() 中设置它。这是因为，Java 要求在父类作用域中声明变量，并在嵌套类中使用变量以标记为 final。例如：

```
@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;
}
```

每次可以将一个 Settable 链接到一个 promise。您可以在链接的 Settable 上调用 unchain() 方法以将其取消链接。

@NoWait

在将 Promise 传递到一个异步方法时，默认情况下，该框架等待 Promise 变为就绪状态，然后再执行该方法 (集合类型除外)。您可以在异步方法声明中的参数上使用 @NoWait 注释以覆盖该行为。如果传入 Settable<T> (异步方法本身将对其进行设置)，这是非常有用的。

承诺 < 撤消 >

异步方法中的依赖关系是通过将一个方法返回的 Promise 作为参数传递到另一个方法实现的。但在某些情况下，您希望从一个方法中返回 void，但仍希望其他异步方法在该方法完成后执行。在这些情况下，您可以将 Promise<Void> 作为方法的返回类型。Promise 类提供了一个静态 Void 方法，可用于创建 Promise<Void> 对象。在异步方法完成执行时，该 Promise 将变为就绪状态。您可以将该 Promise 传递到另一个异步方法，就像任何其他 Promise 对象一样。如果您使用 Settable<Void>，则使用 null 在其上调用设置方法以使其变为就绪状态。

AndPromise 和 OrPromise

通过使用 AndPromise 和 OrPromise，您可以将多个 Promise<> 对象划分到单个逻辑 promise。在用于构建 AndPromise 的所有 promise 变为就绪状态时，它将变为就绪状态。在用于构建 OrPromise 的 promise 集合中的任何 promise 变为就绪状态时，它将变为就绪状态。您可以在 AndPromise 和 OrPromise 上调用 getValues() 以检索组成 promise 的值列表。

可测试性和依赖关系注入

主题

框架设计为适当地控制反转 (IoC)。可以使用像 Spring 这样的容器配置和实例化活动和工作流程实施以及框架提供的工作线程和上下文对象。框架提供了与 Spring 框架的现成集成。此外，已针对单元测试工作流程和活动实施提供与 JUnit 的集成。

Spring 集成

利用 com.amazonaws.services.simpleworkflow.flow.spring 程序包中包含的类，可以在您的应用程序中轻松地使用 Spring 框架。其中包括自定义范围和 Spring 感知的活动和工作流程工作线程：`WorkflowScope`、`SpringWorkflowWorker` 和 `SpringActivityWorker`。利用这些类，您可以通过 Spring 完整配置工作流程和活动实施以及工作线程。

WorkflowScope

`WorkflowScope` 是由框架提供的自定义 Spring 范围实施。此范围允许您在其生命周期被限定于决策任务范围的 Spring 容器中创建对象。每当工作线程收到新的决策任务时，将实例化此范围中的 bean。您应将此范围用于工作流程实施 bean 及其依赖的任何其他 bean。Spring 提供的单例和原型范围不应用于工作流程实施 bean，因为框架需要为每个决策任务创建一个新 bean。该操作失败将会导致意外行为。

以下示例显示 Spring 配置的代码段，它将注册 `WorkflowScope`，然后将其用于配置工作流程实施 bean 和活动客户端 bean。

```
<!-- 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>
```

配置行：需要 <aop:scoped-proxy proxy-target-class="false" /> (在 workflowImpl bean 的配置中使用)，因为 WorkflowScope 不支持使用 CGLIB 作为代理。应将此配置用于 WorkflowScope 中连接到其他范围中的另一个 bean 的任何 bean。在此情况下，workflowImpl bean 需要连接到单例范围中的工作流程工作线程 bean (参见以下完整示例)。

您可以参阅 Spring 框架文档以详细了解如何使用自定义范围。

Spring 感知的工作线程

在使用 Spring 时，您应使用框架提供的 Spring 感知的工作线程类：SpringWorkflowWorker 和 SpringActivityWorker。可使用 Spring 将这些工作线程注入应用程序中，如下一个示例所示。Spring 感知的工作线程可实施 Spring 的 SmartLifecycle 接口，并且 (默认情况下) 在初始化 Spring 上下文时自动开始轮询任务。您可以通过将工作线程的 disableAutoStartup 属性设置为 true 来禁用此功能。

以下示例说明如何配置决策程序。此示例使用 MyActivities 和 MyWorkflow 接口 (此处未显示) 以及相应的实施 (MyActivitiesImpl 和 MyWorkflowImpl)。生成的客户端接口和实施为 MyWorkflowClient/MyWorkflowClientImpl 和 MyActivitiesClient/MyActivitiesClientImpl (此处也未显示)。

使用 Spring 的自动连接功能将活动客户端注入工作流程实施中。

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

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

```
   }
}
```

决策程序的 Spring 配置如下所示：

```
<?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>
```

由于 SpringWorkflowWorker 在 Spring 中完全配置并在初始化 Spring 上下文时自动开始轮询，因此决策程序的宿主进程非常简单：

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

同样，可以配置活动工作线程，如下所示：

```
<?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>
```

活动工作线程宿主进程与决策程序的类似：

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

注入决策上下文

如果工作流程实施依赖于上下文对象，则也可通过 Spring 轻松注入它们。框架将在 Spring 容器中自动注册与上下文相关的 bean。例如，在以下代码段中，已自动连接各种上下文对象。不需要上下文对象的其他 Spring 配置。

```
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();
   }
}
```

若要通过 Spring XML 配置来配置工作流程实施中的上下文对象，可使用 com.amazonaws.services.simpleworkflow.flow.spring 程序包中的 `WorkflowScopeBeanNames` 类中声明的 bean 名称。例如：

```
<!-- 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>
```

或者，您可以在工作流程实施 bean 中注入 `DecisionContextProvider`，并使用它创建上下文。如果您要提供提供程序和上下文的自定义实施，这会很有用。

在活动中注入资源

您可以使用控制反转 (IoC) 容器实例化和配置活动实施，并通过将数据库连接等资源声明为活动实施类的属性来轻松注入这些资源。此类资源通常被限定为单例范围。请注意，活动实施由多个线程上的活动工作线程调用。因此，必须同步对共享资源的访问。

JUnit 集成

框架提供可用于通过 JUnit 编写和运行单元测试的上下文对象 (如测试时钟) 的 JUnit 扩展和测试实施。利用这些扩展，您可以本地内联测试工作流程实施。

编写简单单元测试

要编写工作流程的测试，请使用 com.amazonaws.services.simpleworkflow.flow.junit 程序包中的 WorkflowTest 类。该类是框架特定的 JUnit MethodRule 实现，它在本地运行工作流代码，并内联调用活动，而不是通过 Amazon SWF 执行此操作。这使您能够根据需要灵活地频繁运行测试，而不会产生任何费用。

要使用此类，只需声明类型 WorkflowTest 的字段并使用 @Rule 注释标记它。在运行您的测试之前，请创建新的 WorkflowTest 对象并向其添加活动和工作流程实施。之后，您可以使用生成的工作流程客户端工厂来创建客户端并启动工作流程执行。框架还提供了一个自定义 JUnit 运行程序 FlowBlockJUnit4ClassRunner，您必须将其用于工作流程测试。例如：

```
@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);
    }
}
```

您还可以为添加到 WorkflowTest 的每个活动实施指定一个单独的任务列表。例如，如果您有一个计划宿主特定的任务列表中的活动的工作流程实施，则可在每个宿主的任务列表中注册活动：

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

请注意，@Test 中的代码是异步的。因此，您应使用异步工作流程客户端来启动执行。为了验证您的测试结果，还提供了一个 AsyncAssert 帮助类。此类使您能够在验证结果前等待 Promise 就绪。在此示例中，我们等待工作流程执行的结果就绪，然后再验证测试输出。

如果您使用的是 Spring，则可以使用 SpringWorkflowTest 类而不是 WorkflowTest 类。SpringWorkflowTest 提供了配置属性，您可以使用它们通过 Spring 配置轻松地配置活动和工作流程实施。就像 Spring 感知的工作线程一样，您应使用 WorkflowScope 配置工作流程实施 bean。这将确保为每个决策任务创建一个新的工作流程实施 bean。确保配置这些 bean，并且 scoped-proxy proxy-target-class 设置设为 false。有关更多详细信息，请参阅“Spring 集成”部分。可以更改“Spring 集成”部分中显示的 Spring 配置示例来使用 SpringWorkflowTest 测试工作流程：

```
<?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>
```

模拟活动实施

您可以在测试期间使用真实活动实施，但如果您要仅对工作流程逻辑进行单元测试，则应模拟活动。可以通过向 WorkflowTest 类提供活动接口的模拟实施来做到这一点。例如：

```
@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);
    }
}
```

或者，您可以提供活动客户端的模拟实施并将其注入您的工作流程实施中。

测试上下文对象

如果您的工作流实现依赖于框架上下文对象（例如 DecisionContext），那么您无需执行任何特殊操作即可测试此类工作流。在通过 WorkflowTest 运行测试时，它会自动注入测试上下文对象。当您的工作流实现访问上下文对象时（例如，使用 DecisionContextProviderImpl），它将获得测试实现。您可以在测试代码 (@Test 方法) 中操作这些测试上下文对象，以创建有意义的测试用例。例如，如果您的工作流程创建了一个计时器，则可通过调用 WorkflowTest 类的 clockAdvanceSeconds 方法来使时钟及时向前移动，从而触发计时器。您也可以加快时钟，以使计时器的触发时间早于通常通过使用 WorkflowTest 的 ClockAccelerationCoefficient 属性实现的触发时间。例如，如果您的工作流程创建了一个 1 小时的计时器，则可将

ClockAccelerationCoefficient 设置为 60 来使计时器在 1 分钟内触发。默认情况下，将 ClockAccelerationCoefficient 设置为 1。

有关 com.amazonaws.services.simpleworkflow.flow.test 和 com.amazonaws.services.simpleworkflow.flow.junit 程序包的更多详细信息，请参阅 AWS SDK for Java 文档。

错误处理

主题

Java 中的 try/catch/finally 结构使得错误处理变得轻松，并且可普遍使用。这使您能够将错误处理程序关联到代码块。在内部，这是通过在调用堆栈上填充有关错误处理程序的附加元数据来实现的。当引发异常时，运行时会查看已关联错误处理程序的调用堆栈并调用它；如果未找到相应的错误处理程序，它会在调用链中向上传播异常。

这对于同步代码很有效，但处理异步和分布式程序中的错误会带来额外的挑战。由于异步调用立即返回，因此，当异步代码执行时，调用方不在调用堆栈上。这意味着，调用方无法通过常规方式处理异步代码中的未处理异常。通常，通过将错误状态传递给回调 (该回调将传递给异步方法) 来处理源自异步代码的异常。或者，如果正在使用 Future<?>，它会在您尝试访问它时报告错误。这并不理想，因为接收异常的代码 (使用 Future<?> 的回调或代码) 没有原始调用的上下文，并且可能无法充分处理异常。此外，在分布式异步系统中，当组件同时运行时，可能会同时出现多个错误。这些错误的类型和严重性可能不同，并且需要适当处理。

在异步调用之后清除资源也很困难。与同步代码不同，您不能在调用代码中使用 try/catch/finally 来清除资源，因为当 finally 块执行时，try 块中启动的工作可能仍在进行中。

该框架提供了一种机制，使得分布式异步代码中的错误处理与 Java 的 try/catch/finally 相似且非常简单。

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

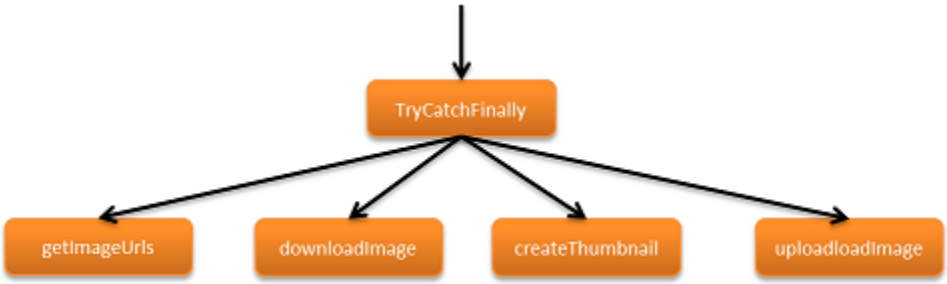
TryCatchFinally 类及其变体 (TryFinally 和 TryCatch) 与 Java 的 try/catch/finally 的工作方式类似。通过使用它，您可以将异常处理程序关联到可作为异步和远程任务执行的工作流程代码块。doTry() 方法在逻辑上等同于 try 块。此框架自动执行 doTry() 中的代码。Promise 对象列表可传递给 TryCatchFinally 的构造函数。当已传入构造函数的所有 Promise 对象就绪时，将执行 doTry 方法。如果已从 doTry() 中异步调用的代码引发了异常，则将取消 doTry() 中的任何待处理工作，并且将调用 doCatch() 以处理异常。例如，在上述列出内容中，如果 downloadImage 引发异常，则将取消 createThumbnail 和 uploadImage。最后，在完成所有异步工作 (完成、失败或取消) 时，将调用 doFinally()。它可用于清除资源。您也可以嵌套这些类来满足您的需求。

在 doCatch() 中报告异常时，框架将提供一个包含异步调用和远程调用的完整逻辑调用堆栈。这在调试时很有用，特别是在您具有调用其他异步方法的异步方法时。例如，来自 downloadImage 的异常将生成与以下内容类似的异常：

```
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 语义

适用于 Java 的 AWS Flow Framework 程序的执行可以直观地表示为由多个并发执行分支组成的树。调用异步方法、活动和 TryCatchFinally 本身将在此执行树中创建一个新的分支。例如，可以按照下图中显示的树的形式来查看图像处理工作流程。



一个执行分支中的错误将导致该分支展开，就像异常导致 Java 程序中的调用堆栈展开一样。展开会继续将执行分支上移，直到错误被处理或到达树的根，在这种情况下，工作流程执行将被终止。

框架将处理任务时发生的错误报告为异常。它将 TryCatchFinally 中定义的异常处理程序 (doCatch() 方法) 与相应的 doTry() 中的代码所创建的所有任务关联。如果任务失败（例如，由于超时或未处理异常），则会引发相应的异常，并调用相应的 doCatch() 进行处理。为此，框架与 Amazon SWF 协同工作，传播远程错误，并在调用方的上下文中将其恢复为异常。

取消

当同步代码中出现异常时，控制将直接跳至 catch 块，并跳过 try 块中的任何剩余代码。例如：

```
try {
    a();
    b();
    c();
}
catch (Exception e) {
    e.printStackTrace();
```

```
}
```

在此代码中，如果 b() 引发异常，则绝不会调用 c()。将其与工作流程进行比较：

```
new TryCatch() {

    @Override
    protected void doTry() throws Throwable {
        activityA();
        activityB();
        activityC();
    }

    @Override
    protected void doCatch(Throwable e) throws Throwable {
        e.printStackTrace();
    }
};
```

在此示例中，对 activityA、activityB 和 activityC 的调用全部成功返回，并导致创建将异步执行的三个任务。假定 activityB 的任务稍后将导致错误。Amazon SWF 会将此错误记录在历史记录中。要处理此错误，框架首先会尝试取消源自相同的 doTry() 范围的所有其他任务；在此示例中，为 activityA 和 activityC。当所有此类任务完成 (取消、失败或成功完成) 时，将调用相应的 doCatch() 方法以处理错误。

与同步示例不同，其中绝不会执行 c()，已调用 activityC，并且已计划执行一个任务；因此，框架将尝试取消它，但不能保证将其取消。不能保证取消，因为活动可能已完成、可能忽略取消请求或可能因错误而导致失败。不过，框架保证仅在完成从相应的 doTry() 启动的所有任务后，才调用 doCatch()。它还保证仅在完成从 doTry() 和 doCatch() 启动的所有任务后，才调用 doFinally()。例如，如果上述示例中的活动相互依赖，假设 activityB 依赖 activityA，activityC 依赖 activityB，那么 activityC 将被立即取消，因为在 activityB 完成之前，Amazon SWF 不会对其进行安排：

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

活动检测信号

适用于 Java 的 AWS Flow Framework 的协作取消机制允许正常取消进行中的活动任务。在触发取消时，将自动取消已被阻止或正在等待分配给工作线程的任务。不过，如果已将任务分配给工作线程，则框架将请求取消活动。您的活动实施必须明确处理此类取消请求。通过报告活动的检测信号来做到这一点。

报告检测信号将允许活动实施报告进行中的活动任务的进度，这对于监控很有用，并且它允许活动检查取消请求。如果已请求取消，则 `recordActivityHeartbeat` 方法将引发 `CancellationException`。活动实施可以捕获此异常并对取消请求进行操作，也可以通过承受异常来忽略请求。为了满足取消请求，活动应该执行所需的清除 (如果有)，然后重新引发 `CancellationException`。在从活动实施中引发此异常时，框架会将已完成的活动任务记录为取消状态。

以下示例显示了一个下载和处理映像的活动。在处理每个映像后，它会发出检测信号；如果请求取消，它会执行清除操作并重新引发异常以确认取消。

```
@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;
        }
    }
}
```

虽然不要求报告活动检测信号，但如果您的活动是长期运行的或是您希望在错误情形下取消的开销较大的操作，则建议这样做。您应定期从活动实施中调用 heartbeatActivityTask。

如果活动超时，则将引发 ActivityTaskTimedOutException，并且异常对象上的 getDetails 将返回已传递给上次对相应活动任务的 heartbeatActivityTask 的成功调用的数据。工作流程实施可以使用此信息来确定活动任务超时前的进度。

> **Note**
>
> 检测信号过于频繁不是好的做法，因为 Amazon SWF 可能会限制检测信号请求。有关 Amazon SWF 设置的限制，请参阅 Amazon Simple Workflow Service Developer Guide。

明确取消任务

除了错误情形之外，还存在其他您可能会明确取消任务的情况。例如，如果用户取消订单，则可能需要取消使用信用卡处理付款的活动。框架允许您明确取消在 TryCatchFinally 范围内创建的任务。在以下示例中，如果在处理付款时收到信号，则付款任务将被取消。

```
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);
        }
    }
}
```

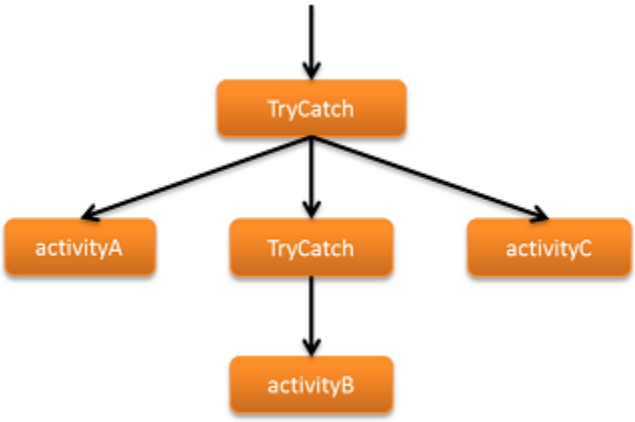
接收取消任务的通知

当任务在取消状态下完成时，框架通过引发 CancellationException 来将此情况告知工作流程逻辑。当活动在取消状态下完成时，将在历史记录中生成一条记录，框架将调用相应的 doCatch()，并引发 CancellationException。如上一个示例中所示，当取消付款处理任务时，工作流程将收到 CancellationException。

未处理的 CancellationException 会在执行分支中向上传播，就像任何其他异常一样。不过，doCatch() 方法仅在范围中没有任何其他异常时收到 CancellationException；其他异常的优先级高于取消异常的优先级。

嵌套的 TryCatchFinally

您可以嵌套 TryCatchFinally 以满足您的需求。由于每个 TryCatchFinally 均会在执行树中创建一个新的分支，因此您可以创建嵌套范围。父范围中的异常将导致尝试取消嵌套的 TryCatchFinally 在其中启动的所有任务。不过，嵌套的 TryCatchFinally 中的异常不会自动传播到父级。如果您希望将嵌套的 TryCatchFinally 中的异常传播到其包含的 TryCatchFinally，则应在 doCatch() 中重新引发该异常。换句话说，仅提供未处理的异常，就像 Java 的 try/catch 一样。如果您通过调用取消方法来取消嵌套的 TryCatchFinally，则将取消嵌套的 TryCatchFinally，但不会自动取消包含的 TryCatchFinally。




```
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);
    }
};
```

重试失败的活动

活动有时会由于临时原因而失败，例如，连接临时中断。在其他时间，活动可能会成功，因此，处理活动失败的相应方法通常是重试活动，可能会重试很多次。

可以使用几种不同的策略重试活动；最佳的策略取决于您的工作流程详细信息。这些策略分为三种基本类别：

- 重试直到成功策略就是不断重试活动，直到完成为止。
- 指数重试策略以指数方式增加重试尝试的时间间隔，直到活动完成或该过程达到指定的停止点，如最大尝试次数。
- 自定义重试策略确定在每个失败尝试后是否或如何重试活动。

以下几节介绍了如何实施这些策略。示例工作流程工作线程均使用单个活动 (unreliableActivity)，它随机执行以下操作之一：

- 立即完成
- 有意超过超时值而失败
- 有意引发 IllegalStateException 而失败

重试直到成功策略

最简单的重试策略是在每次失败时不断重试活动，直到最终成功。基本模式如下：

1. 在工作流程的入口点方法中实现施嵌套的 TryCatch 或 TryCatchFinally 类。
2. 在 doTry 中执行活动
3. 如果活动失败，该框架调用 doCatch，这会再次运行入口点方法。
4. 重复步骤 2-3，直到活动成功完成。

以下工作流程实施重试直到成功策略。工作流程接口是在 RetryActivityRecipeWorkflow 中实现的，并具有一个方法 (runUnreliableActivityTillSuccess)，它是工作流程的入口点。工作流程工作线程是在 RetryActivityRecipeWorkflowImpl 中实现的，如下所示：

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

工作流程的工作方式如下所示：

1. runUnreliableActivityTillSuccess 创建一个名为 retryActivity 的 Settable<Boolean> 对象，它用于指示活动是否失败并应进行重试。Settable<T> 派生自 Promise<T>，它的工作方式基本相同，但您手动设置 Settable<T> 对象的值。
2. runUnreliableActivityTillSuccess 实现一个匿名的嵌套 TryCatch 类以处理 unreliableActivity 活动引发的任何异常。有关如何处理异步代码引发的异常的详细讨论，请参阅错误处理。
3. doTry 执行 unreliableActivity 活动，它返回一个名为 activityRanSuccessfully 的 Promise<Void> 对象。
4. doTry 调用异步 setRetryActivityToFalse 方法，它使用两个参数：

- activityRanSuccessfully 使用 unreliableActivity 活动返回的 Promise<Void> 对象。
- retryActivity 使用 retryActivity 对象。

  如果 unreliableActivity 完成，activityRanSuccessfully 将变为就绪状态 setRetryActivityToFalse 并将 retryActivity 设置为 false。否则，activityRanSuccessfully 从不变为就绪状态并且 setRetryActivityToFalse 不执行。

5. 如果 unreliableActivity 引发异常，该框架将调用 doCatch 并为其传递异常对象。doCatch 将 retryActivity 设置为 true。
6. runUnreliableActivityTillSuccess 调用异步 restartRunUnreliableActivityTillSuccess 方法并为其传递 retryActivity 对象。由于 retryActivity 具有 Promise<T> 类型，restartRunUnreliableActivityTillSuccess 将推迟执行，直到 retryActivity 准备就绪，在 TryCatch 完成后将变为该状态。
7. 在 retryActivity 准备就绪时，restartRunUnreliableActivityTillSuccess 提取该值。
   - 如果值为 false，则重试成功。restartRunUnreliableActivityTillSuccess 不执行任何操作，并终止重试序列。
   - 如果值为 true，则重试失败。restartRunUnreliableActivityTillSuccess 调用 runUnreliableActivityTillSuccess 以再次执行活动。
8. 重复步骤 1-7，直到 unreliableActivity 完成。

> **Note**
>
> doCatch 不处理异常；它仅将 retryActivity 对象设置为 true 以指示活动失败。重试是由异步 restartRunUnreliableActivityTillSuccess 方法处理的，它推迟执行，直到 TryCatch 完成。使用这种方法的原因是，如果在 doCatch 中重试活动，则无法取消该活动。如果在 restartRunUnreliableActivityTillSuccess 中重试活动，则可以执行可取消的活动。

指数重试策略

在使用指数重试策略时，该框架在指定的时间段 (N 秒) 后再次执行失败的活动。如果该尝试失败，该框架在 2N 秒后再次执行该活动，然后在 4N 秒后再次执行该活动，依此类推。由于等待时间可能会变得很长，您通常会在某个时间点停止重试尝试，而不是无限期继续执行。

该框架提供三种方法以实施指数重试策略：

- `@ExponentialRetry` 注释是最简单的方法，但您必须在编译时设置重试配置选项。
- `RetryDecorator` 类允许您在运行时设置重试配置，并根据需要对其进行更改。
- `AsyncRetryingExecutor` 类允许您在运行时设置重试配置，并根据需要对其进行更改。此外，该框架调用用户实现的 `AsyncRunnable.run` 方法以运行每个重试尝试。

所有方法都支持以下配置选项，其中时间值以秒为单位：

- 初始重试等待时间。
- 退避系数，它用于计算重试间隔，如下所示：

  ```
  retryInterval = initialRetryIntervalSeconds * Math.pow(backoffCoefficient,
   numberOfTries - 2)
  ```

  默认值是 2.0。
- 最大重试次数。默认值为无限制。
- 最大重试间隔。默认值为无限制。
- 过期时间。在重试过程的总持续时间超过该值时，重试尝试将停止。默认值为无限制。
- 将触发重试过程的异常。默认情况下，每个异常都会触发重试过程。
- 不会触发重试尝试的异常。默认情况下，不会排除任何异常。

以下几节介绍了可实施指数重试策略的各种方法。

使用 @ExponentialRetry 的指数重试

为活动实施指数重试策略的最简单方法是，在接口定义中将 `@ExponentialRetry` 注释应用于活动。如果活动失败，该框架根据指定的选项值自动处理重试过程。基本模式如下：

1. 将 `@ExponentialRetry` 应用于相应的活动并指定重试配置。
2. 如果注释的活动失败，该框架根据注释的参数指定的配置自动重试活动。

ExponentialRetryAnnotationWorkflow 工作流程工作线程使用 @ExponentialRetry 注释实施指数重试策略。它使用在 ExponentialRetryAnnotationActivities 中实现接口定义的 unreliableActivity 活动，如下所示：

```
@Activities(version = "1.0")
@ActivityRegistrationOptions(
    defaultTaskScheduleToStartTimeoutSeconds = 30,
    defaultTaskStartToCloseTimeoutSeconds = 30)
public interface ExponentialRetryAnnotationActivities {
    @ExponentialRetry(
        initialRetryIntervalSeconds = 5,
        maximumAttempts = 5,
        exceptionsToRetry = IllegalStateException.class)
    public void unreliableActivity();
}
```

@ExponentialRetry 选项指定以下策略：

- 仅在活动引发 IllegalStateException 时重试。
- 使用 5 秒初始等待时间。
- 不超过 5 次重试尝试。

工作流程接口是在 RetryWorkflow 中实现的，并具有一个方法 (process)，它是工作流程的入口点。工作流程工作线程是在 ExponentialRetryAnnotationWorkflowImpl 中实现的，如下所示：

```
public class ExponentialRetryAnnotationWorkflowImpl implements RetryWorkflow {
    public void process() {
        handleUnreliableActivity();
    }

    public void handleUnreliableActivity() {
        client.unreliableActivity();
    }
}
```

工作流程的工作方式如下所示：

1. process 运行同步 handleUnreliableActivity 方法。
2. handleUnreliableActivity 执行 unreliableActivity 活动。

如果活动引发 IllegalStateException 而失败，该框架自动运行在 ExponentialRetryAnnotationActivities 中指定的重试策略。

使用 RetryDecorator 类的指数重试

@ExponentialRetry 易于使用。不过，配置是静态的，并且是在编译时设置的，因此，每次活动失败时，该框架使用相同的重试策略。您可以使用 RetryDecorator 类实施更灵活的指数重试策略，它允许在运行时指定配置并根据需要进行更改。基本模式如下：

1. 创建并配置一个 ExponentialRetryPolicy 对象以指定重试配置。
2. 创建一个 RetryDecorator 对象，并将步骤 1 中的 ExponentialRetryPolicy 对象传递给构造函数。
3. 将活动客户端的类名传递给 RetryDecorator 对象的 decorate 方法，以将装饰器对象应用于活动。
4. 执行活动。

如果活动失败，该框架根据 ExponentialRetryPolicy 对象的配置重试活动。您可以根据需要修改该对象以更改重试配置。

> **Note**
>
> @ExponentialRetry 注释和 RetryDecorator 类相互排斥。您无法使用 RetryDecorator 动态覆盖 @ExponentialRetry 注释指定的重试策略。

以下工作流程实现说明了如何使用 RetryDecorator 类实施指数重试策略。它使用一个没有 @ExponentialRetry 注释的 unreliableActivity 活动。工作流程接口是在 RetryWorkflow 中实现的，并具有一个方法 (process)，它是工作流程的入口点。工作流程工作线程是在 DecoratorRetryWorkflowImpl 中实现的，如下所示：

```
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();
    }
}
```

工作流程的工作方式如下所示：

1. process 使用以下方法创建并配置一个 ExponentialRetryPolicy 对象：
   - 将初始重试间隔传递给构造函数。
   - 调用对象的 withMaximumAttempts 方法以将最大尝试次数设置为 5。ExponentialRetryPolicy 公开了其他 with 对象，您可以使用这些对象指定其他配置选项。
2. process 创建一个名为 retryDecorator 的 RetryDecorator 对象，并将步骤 1 中的 ExponentialRetryPolicy 对象传递给构造函数。
3. process 调用 retryDecorator.decorate 方法并为其传递活动客户端的类名，以将装饰器应用于活动。
4. handleUnreliableActivity 执行活动。

如果活动失败，该框架根据步骤 1 中指定的配置重试活动。

> **Note**
>
> ExponentialRetryPolicy 类的几个 with 方法具有相应的 set 方法，您可以随时调用该方法以修改相应的配置选项：setBackoffCoefficient、setMaximumAttempts、setMaximumRetryIntervalSeconds 和 setMaximumRetryExpirationIntervalSeconds。

使用 AsyncRetryingExecutor 类的指数重试

RetryDecorator 类在配置重试过程方面提供比 @ExponentialRetry 更大的灵活性，但该框架仍会根据 ExponentialRetryPolicy 对象的当前配置自动运行重试尝试。更灵活的方法是使用 AsyncRetryingExecutor 类。除了允许您在运行时配置重试过程以外，该框架还会调用用户实现的 AsyncRunnable.run 方法以运行每个重试尝试，而不是直接执行活动。

基本模式如下：

1. 创建并配置一个 ExponentialRetryPolicy 对象以指定重试配置。
2. 创建一个 AsyncRetryingExecutor 对象，并为其传递 ExponentialRetryPolicy 对象以及一个工作流程时钟实例。
3. 实现一个匿名的嵌套 TryCatchFinally 或 TryCatch 类。
4. 实现一个匿名的 AsyncRunnable 类，并覆盖 run 方法以实现用于运行活动的自定义代码。
5. 覆盖 doTry 以调用 AsyncRetryingExecutor 对象的 execute 方法，并为其传递步骤 4 中的 AsyncRunnable 类。AsyncRetryingExecutor 对象调用 AsyncRunnable.run 以运行活动。
6. 如果活动失败，则 AsyncRetryingExecutor 对象根据步骤 1 中指定的重试策略再次调用 AsyncRunnable.run 方法。

以下工作流程说明了如何使用 AsyncRetryingExecutor 类实施指数重试策略。它使用与前面讨论的 DecoratorRetryWorkflow 工作流程相同的 unreliableActivity 活动。工作流程接口是在 RetryWorkflow 中实现的，并具有一个方法 (process)，它是工作流程的入口点。工作流程工作线程是在 AsyncExecutorRetryWorkflowImpl 中实现的，如下所示：

```
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 {
            }
        };
    }
}
```

工作流程的工作方式如下所示：

1. process 调用 handleUnreliableActivity 方法并为其传递配置设置。
2. handleUnreliableActivity 使用步骤 1 中的配置设置创建 ExponentialRetryPolicy 对象 retryPolicy。
3. handleUnreliableActivity 创建 AsyncRetryExecutor 对象 executor，并将步骤 2 中的 ExponentialRetryPolicy 对象以及一个工作流程时钟实例传递给构造函数。
4. handleUnreliableActivity 实现一个匿名的嵌套 TryCatch 类，并覆盖 doTry 和 doCatch 方法以运行重试尝试并处理任何异常。
5. doTry 创建一个匿名的 AsyncRunnable 类，并覆盖 run 方法以实现自定义代码以执行 unreliableActivity。为简单起见，run 仅执行活动，但您可以根据需要实现更复杂的方法。
6. doTry 调用 executor.execute 并将其传递到 AsyncRunnable 对象。execute 调用 AsyncRunnable 对象的 run 方法以运行活动。
7. 如果活动失败，执行程序根据 retryPolicy 对象配置再次调用 run。

有关如何使用 TryCatch 类处理错误的详细讨论，请参阅 适用于 Java 的 AWS Flow Framework 异常。

自定义重试策略

重试失败活动的最灵活方法是自定义策略，它递归调用一个异步方法以运行重试尝试，这与重试直到成功策略非常相似。不过，您实现自定义逻辑以确定是否以及如何运行每个连续重试尝试，而不是直接再次运行活动。基本模式如下：

1. 创建一个 Settable<T> 状态对象，它用于指示活动是否失败。
2. 实现一个嵌套 TryCatch 或 TryCatchFinally 类。
3. doTry 执行活动。
4. 如果活动失败，doCatch 设置该状态对象以指示活动失败。
5. 调用一个异步失败处理方法，并为其传递该状态对象。该方法推迟执行，直到 TryCatch 或 TryCatchFinally 完成。
6. 该失败处理方法确定是否重试活动，如果重试，确定何时重试活动。

以下工作流程说明如何实现自定义重试策略。它使用与 DecoratorRetryWorkflow 和 AsyncExecutorRetryWorkflow 工作流程相同的 unreliableActivity 活动。工作流程接口是在 RetryWorkflow 中实现的，并具有一个方法 (process)，它是工作流程的入口点。工作流程工作线程是在 CustomLogicRetryWorkflowImpl 中实现的，如下所示：

```
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;
    }
}
```

工作流程的工作方式如下所示：

1. process 调用异步 callActivityWithRetry 方法。
2. callActivityWithRetry 创建一个名为 failure 的 Settable<Throwable> 对象，此对象用于指示活动是否失败。Settable<T> 派生自 Promise<T>，它的工作方式基本相同，但您手动设置 Settable<T> 对象的值。
3. callActivityWithRetry 实现一个匿名的嵌套 TryCatchFinally 类以处理 unreliableActivity 引发的任何异常。有关如何处理异步代码引发的异常的详细讨论，请参阅适用于 Java 的 AWS Flow Framework 异常。
4. doTry 执行 unreliableActivity。
5. 如果 unreliableActivity 引发异常，则该框架将调用 doCatch 并将其传递给异常对象。doCatch 将 failure 设置为异常对象（指示活动失败）并使该对象处于就绪状态。
6. doFinally 检查 failure 是否准备就绪，只有在 doCatch 设置了 failure 时，它才为 true。
   - 如果 failure 准备就绪，则 doFinally 不执行任何操作。
   - 如果 failure 未准备就绪，则活动完成并且 doFinally 将 failure 设置为 null。
7. callActivityWithRetry 调用异步 retryOnFailure 方法并为其传递 failure。由于 failure 具有 Settable<T> 类型，callActivityWithRetry 将推迟执行，直到 failure 准备就绪，在 TryCatchFinally 完成后将变为该状态。
8. retryOnFailure 从 failure 中获取值。
   - 如果 failure 设置为 null，则重试尝试成功。retryOnFailure 不执行任何操作，这会终止重试过程。
   - 如果 failure 设置为一个异常对象并且 shouldRetry 返回 true，则 retryOnFailure 调用 callActivityWithRetry 以重试活动。

     shouldRetry 实现自定义逻辑以确定是否重试失败的活动。为简单起见，shouldRetry 始终返回 true 并且 retryOnFailure 立即执行活动，但您可以根据需要实现更复杂的逻辑。
9. 重复步骤 2 到 8，直至 unreliableActivity 完成 shouldRetry 决定停止该过程。

> **Note**
>
> doCatch 不处理重试过程；它仅设置 failure 以指示活动失败。重试过程是由异步 retryOnFailure 方法处理的，它推迟执行，直到 TryCatch 完成。使用这种方法的原因是，如果在 doCatch 中重试活动，则无法取消该活动。如果在 retryOnFailure 中重试活动，则可以执行可取消的活动。

守护程序任务

适用于 Java 的 AWS Flow Framework 允许将特定任务标记为 daemon。这将允许您创建执行某项后台工作的任务，在完成所有其他工作时将取消这些任务。例如，在完成工作流程的其他任务时，将取消运行状况监控任务。您可以通过在异步方法或 TryCatchFinally 实例上设置 daemon 标志来做到这一点。在以下示例中，异步方法 monitorHealth() 将标记为 daemon。

```
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();
  }
}
```

在上述示例中，在 doUsefulWorkActivity 完成后，将自动取消 monitoringHealth。这反过来将取消基于此异步方法的整个执行分支。取消的语义与 TryCatchFinally 中的相同。同样，您可以通过将布尔值标记传递到构造函数来将 TryCatchFinally 标记为 daemon。

```
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
            }
        };
    }
}
```

在 TryCatchFinally 中启动的守护程序任务的范围限于创建它的上下文，也就是其范围将限定于 doTry()、doCatch() 或 doFinally() 方法。例如，在以下示例中，startMonitoring 异步方法将标记为 daemon 并从 doTry() 进行调用。一旦在 doTry() 中启动的其他任务 (在此示例中，为 doUsefulWorkActivity) 完成，就将取消为它创建的任务。

```
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();
```

```
    }
```

适用于 Java 的 AWS Flow Framework 重播行为

本主题使用Java 的 AWS Flow Framework 用法是什么？一节中的示例讨论重播行为的示例。同时讨论了同步和异步情形。

示例 1：同步重播

对于重播在同步工作流中的工作原理的示例，修改 HelloWorldWorkflow 工作流和活动实现，方法是在各自实现中添加 `println` 调用，如下所示：

```
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);
   }
}
```

有关代码的详细信息，请参阅 HelloWorldWorkflow 应用程序。以下是输出的已编辑版本，使用注释来指示每个重播阶段的开始。

```
//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
```

此示例的重播过程的工作方式如下：

- 第一个阶段安排 getName 活动任务，它没有任何依赖项。
- 第二个阶段安排 getGreeting 活动任务，它依赖 getName。
- 第三个阶段安排 say 活动任务，它依赖 getGreeting。
- 最后一个阶段没有安排任何其他任务并且没有发现任何未完成的活动，它终止工作流执行。

> **Note**
>
> 为每个阶段调用一次这三个活动客户端方法。但是，只有其中一个调用活动任务中的结果，因此每个任务只执行一次。

示例 2：异步重播

与同步重播示例类似，您可以修改 HelloWorldWorkflowAsync 应用程序来了解异步重播的工作方式。它生成以下输出：

```
//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 使用三个重播阶段，因为只有两个活动。`getGreeting` 活动被替换为 getGreeting 异步工作流方法，该方法在完成时不启动重播阶段。

第一个阶段没有调用 `getGreeting`，因为它依赖 name 活动的完成。但是，在 getName 完成之后，重播将为每个后续阶段调用一次 getGreeting。

另请参阅

- AWS Flow Framework 基本概念：分布式执行

深入剖析

主题

任务

适用于 Java 的 AWS Flow Framework 用于管理异步代码执行的基础基元是 Task 类。Task 类型的对象表示必须异步执行的工作。在您调用异步方法时，框架会创建 Task 来执行该方法中的代码并将其放在列表中以稍后执行。同样地，在您调用 Activity 时，会为它创建 Task。方法调用在此之后返回，通常返回 Promise<T> 作为调用的未来结果。

Task 类是公有的，可以直接使用。例如，我们可以重写 Hello World 示例来使用 Task 代替异步方法。

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

框架在传递给 Task 的构造函数的所有 Promise 都已准备就绪后调用 doExecute() 方法。有关 Task 类的详细信息，请参阅AWS SDK for Java文档。

框架还包括一个名为 Functor 的类，它表示也是 Promise<T> 的 Task。Functor 对象在 Task 完成后变为就绪状态。在以下示例中，创建了 Functor 来获取问候语：

```
Promise<String> greeting = new Functor<String>() {
```

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

执行顺序

仅当传递给对应异步方法或活动的所有 Promise<T> 类型化参数都已准备就绪后，任务才可执行。已准备好执行的 Task 在逻辑上移动到就绪队列。换言之，将安排它执行。工作线程类执行任务的方法是调用您在异步方法的正文中编写的代码，或在 Amazon Simple Workflow Service AWS (AWS) 中安排活动任务（对于活动方法）。

在任务执行并生成结果后，它们引发其他任务变为就绪状态，从而使程序继续执行。框架执行任务的方式对了解异步代码的执行顺序非常重要。程序中按顺序显示的代码实际上可能不按该顺序执行。

```
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!");
}
```

以上列表中的代码将输出以下内容：

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

这可能与您的预期不同，但通过思考异步方法的任务的执行方式，可以轻松地进行解释：

1. 对 getUserName 的调用会创建 Task。我们将它称为 Task1。由于 getUserName 不使用任何参数，Task1 立即被放入就绪队列。
2. 接下来，对 printHelloName 的调用会创建需要等待 getUserName 的结果的 Task。我们将它称为 Task2。由于必需值尚未准备就绪，Task2 被放入等待列表。
3. 然后创建 printHelloWorld 的任务并将其添加到就绪队列。我们将它称为 Task3。
4. 之后 println 语句会将“Hello, Amazon!”输出到 控制台。
5. 此时，Task1 和 Task3 在就绪队列中，Task2 在等待列表中。
6. 工作程序执行 Task1，其结果使 Task2 就绪。Task2 在 Task3 之后添加到就绪队列。
7. 然后，Task3 和 Task2 按该顺序执行。

活动的执行采取相同的模式。在活动客户端上调用方法时会创建一个 Task，任务执行后会在 Amazon SWF 中安排一个活动。

框架依赖代码生成和动态代理等功能来在您的程序中注入将方法调用转换为活动调用和异步任务的逻辑。

工作流程执行

工作流实现的执行也由工作线程类管理。在工作流客户端上调用方法时，它会调用 Amazon SWF 来创建工作流实例。Amazon SWF 中的任务不应与框架中的任务混淆。Amazon SWF 中的任务是活动任务或决策任务。活动任务的执行很简单。活动工作线程类从 Amazon SWF 接收活动任务，调用您的实施中的相应活动方法，然后将结果返回给 Amazon SWF。

决策任务的执行较复杂。工作流工作线程从 Amazon SWF 接收决策任务。决策任务实际上是询问工作流逻辑接下来做什么的请求。在通过工作流客户端启动工作流实例时，为该实例生成了第一个决策任务。在收到此决策任务后，框架开始执行使用 @Execute 进行注释的工作流方法中的代码。此方法执行安排活动的协调逻辑。当工作流实例的状态更改时（例如，当活动完成时），就会安排进一步的决策任务。此时，工作流逻辑可以根据活动的结果决定采取一种操作；例如，它可能决定安排另一个活动。

框架通过无缝地将决策任务转换为工作流逻辑向开发人员隐藏了所有这些详细信息。从开发人员的角度来看，代码看上去像一个常规程序。框架会使用 Amazon SWF 维护的历史记录将其映射到对 Amazon SWF 和决策任务的调用。在决策任务到达时，框架会重播插入了迄今为止已完成活动的结果的程序执行。将取消阻止等待这些结果的异步方法和活动，程序将继续执行。

下表显示了示例图像处理工作流的执行和相应的历史记录。

缩略图工作流的执行

工作流的程序执行	由 Amazon SWF 维护的历史记录
初次执行	
1. 分派循环 2. getImageUrls 3. downloadImage 4. createThumbnail (等待队列中的任务) 5. uploadImage (等待队列中的任务) 6. <循环的下一个迭代>	1. 已启动工作流实例，id=“1” 2. 已安排 downloadImage
回放	
1. 分派循环 2. getImageUrls 3. downloadImage 图像 路径=“foo” 4. createThumbnail 5. uploadImage (等待队列中的任务) 6. <循环的下一个迭代>	1. 已启动工作流实例，id=“1” 2. 已安排 downloadImage 3. 已完成 downloadImage，返回=“foo” 4. 已安排 createThumbnail
回放	
1. 分派循环 2. getImageUrls 3. downloadImage 图像 路径=“foo” 4. createThumbnail 缩略图路径=“bar” 5. uploadImage 6. <循环的下一个迭代>	1. 已启动工作流实例，id=“1” 2. 已安排 downloadImage 3. 已完成 downloadImage，返回=“foo” 4. 已安排 createThumbnail 5. 已完成 createThumbnail，返回=“bar” 6. 已安排 uploadImage
回放	
1. 分派循环 2. getImageUrls	1. 已启动工作流实例，id=“1” 2. 已安排 downloadImage

工作流的程序执行	由 Amazon SWF 维护的历史记录
3. downloadImage 图像 路径=“foo”	3. 已完成 downloadImage，返回=“foo”
4. createThumbnail 缩略图路径=“bar”	4. 已安排 createThumbnail
5. uploadImage	5. 已完成 createThumbnail，返回=“bar”
6. <循环的下一个迭代>	6. 已安排 uploadImage
	7. 已完成 uploadImage
	...

在调用 `processImage` 时，框架会在 Amazon SWF 中创建新的工作流实例。这是要启动的工作流实例的持久记录。程序会一直执行到调用 `downloadImage` 活动，它要求 Amazon SWF 安排活动。工作流进一步执行并为后续活动创建任务，但这些任务要到 `downloadImage` 活动完成后才能执行；因此，此回放阶段结束。Amazon SWF 会分派要执行的 `downloadImage` 活动地任务，任务完成后，会在历史记录中记录并保存结果。工作流现在可以继续执行，且 Amazon SWF 生成了一个决策任务。框架收到决策任务并重播插入了历史记录中记录的已下载图像的结果的工作流。这可以取消阻止 `createThumbnail` 的任务，并通过在 Amazon SWF 中安排 `createThumbnail` 活动任务来继续执行程序。对 `uploadImage` 重复相同的过程。程序继续以这种方式执行，直到工作流已处理了所有图像，并且没有任何待处理的任务。由于执行状态不存储在本地，每个决策任务可能在不同的计算机上执行。这使您能够轻松编写容错且可轻松扩展的程序。

不确定性

由于框架依赖重播，协调代码 (除活动实现以外的所有工作流代码) 是确定的非常重要。例如，程序中的控制流不应依赖随机数或当前时间。由于这些东西在调用之间会更改，重播可能没有按照相同的路径通过协调逻辑。这将导致意外结果或错误。框架提供了可用于以确定方式获取当前时间的 `WorkflowClock`。有关更多详细信息，请参阅有关执行关联的一节。

> **Note**
>
> 工作流实现对象的 Spring 连接不正确也可能会导致不确定性。工作流实现 bean 及其依赖的 bean 必须在工作流范围 (`WorkflowScope`) 中。例如，将工作流实现 bean 连接到保存状态并在全局上下文中的 bean 将导致意外行为。有关更多详细信息，请参阅Spring 集成 一节。

故障排除和调试提示

主题

本节介绍您在开发用 AWS Flow Framework 于 Java 的工作流程时可能会遇到的一些常见陷阱。它还提供一些提示来帮助您诊断和调试问题。

编译错误

如果您使用 AspectJ 编译时编织选项，则可能会遇到编译器无法找到为您的工作流和活动生成的客户端类的编译时错误。此类编译错误的可能原因是 AspectJ 生成器忽略了在编译期间生成的客户端。解决此问题的方法是从项目中删除 AspectJ 功能，然后重新启用该功能。请注意，每次您的工作流或活动接口更改时，您都需要执行此操作。由于此问题，建议您改用加载时编织选项。有关更多详细信息，请参阅设置适用于 Java 的 AWS Flow Framework 一节。

未知资源错误

当您尝试对不可用的资源执行操作时，Amazon SWF 会返回未知资源错误。此错误的常见原因是：

- 您配置的工作线程的域不存在。要解决这个问题，请先使用 Amazon SWF 控制台或 Amazon SWF 服务 API 注册域。
- 您尝试创建的工作流执行或活动任务的类型尚未注册。如果您在工作线程运行之前尝试创建工作流执行，则会发生此情况。由于工作线程在首次运行时注册其类型，您必须在尝试开始执行之前至少运行它们一次 (或使用控制台或服务 API 手动注册类型)。请注意，在注册类型之后，您便可以创建执行，即使没有工作线程在运行也是如此。
- 工作线程尝试完成的任务已超时。例如，如果工作人员处理任务的时间太长并且超过了超时时间，那么当它尝试完成任务或任务失败时，它就会出 UnknownResource 错。 AWS Flow Framework 工作

人员将继续对 Amazon SWF 进行民意调查，并处理其他任务。但是，您应考虑调整超时。调整超时需要您注册新版本的活动类型。

对 Promise 调用 get() 时出现异常

与 Java Future 不同，`Promise` 是非阻塞构造，对尚未就绪的 `Promise` 调用 `get()` 会引发异常而非阻塞。使用 `Promise` 的正确方法是将其传递给异步方法（或任务），并在相应异步方法中访问其值。AWS Flow Framework 可确保仅在传递到异步方法的所有 `Promise` 参数都已就绪时才调用异步方法。如果您认为自己的代码是正确的，或者在运行其中一个 AWS Flow Framework 示例时遇到了这个问题，那么很可能是由于AspectJ的配置不正确。有关详细信息，请参阅设置适用于 Java 的 AWS Flow Framework 一节。

非确定性工作流

正如不确定性一节所述，工作流的实现必须是确定性的。可导致非确定性的一些常见错误是使用系统时钟、使用随机数和生成 GUID。由于这些构造可能在不同的时间返回不同的值，工作流的控制流可能在每次执行时使用不同的路径 (有关详细信息，请参阅 AWS Flow Framework 基本概念：分布式执行和深入剖析)。如果框架在执行工作流时检测到非确定性，则将引发异常。

因版本控制而出现问题

在您实现工作流或活动的新版本时（例如添加新功能时），应使用适当的注解来增加该类型的版本：@Workflow、@Activites 或 @Activity。在部署工作流的新版本时，您通常具有已在运行的现有版本的执行。因此，您需要确保使用工作流和活动的适当版本的工作线程获得任务。您可以对每个版本使用一组不同的任务列表来实现此目的。例如，您可以向任务列表名称附加版本号。这确保将属于工作流和活动的不同版本的任务分配给适当的工作线程。

对工作流执行进行故障排除和调试

对工作流执行进行故障排除的第一步是使用 Amazon SWF 控制台来查看工作流历史记录。工作流历史记录是更改工作流执行的执行状态的所有事件的完整的权威记录。此历史记录由 Amazon SWF 维护，对于诊断问题来说非常重要。利用 Amazon SWF 控制台，您可以搜索工作流执行并深入了解各个历史记录事件。

AWS Flow Framework 提供了一个`WorkflowReplayer`类，您可以使用该类在本地重播工作流程执行并对其进行调试。使用此课程，您可以调试已关闭和正在运行的工作流执行。`WorkflowReplayer` 依

赖存储在 Amazon SWF 中的历史记录来执行回放。您可以将其指向您 Amazon SWF 账户中的工作流执行，或向其提供历史记录事件（例如，您可以从 Amazon SWF 中检索历史记录，并在本地对其进行序列化以供以后使用）。在您使用 `WorkflowReplayer` 重播工作流执行时，这不会影响您的账户中正在运行的工作流执行。重播完全在客户端上进行。您可以像往常一样使用调试工具来调试工作流、创建断点和进入代码。如果您使用的是 Eclipse，请考虑添加步骤过滤器来筛选 AWS Flow Framework 软件包。

例如，以下代码段可用于重播工作流执行：

```
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 还允许您获取工作流程执行的异步线程转储。此线程转储可为您提供所有已打开的异步任务的调用堆栈。此信息可用于确定执行中的哪些任务正在等待处理并可能被卡住。例如：

```
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);
}
```

任务丢失

有时，您可能关闭了工作线程，然后紧接着启动了新工作线程，但是发现任务被传输给了旧工作线程。这可能因分布在多个进程中的系统中的争用条件而发生。在紧密循环中运行单元测试时，也可能会出现此问题。在 Eclipse 中停止测试有时也会引发此问题，因为可能没有调用关闭处理程序。

要确定此问题实际上是由于旧工作线程获得任务，您应查看工作流历史记录来确定哪个进程获得了应由新工作线程获得的任务。例如，历史记录中的 `DecisionTaskStarted` 事件包含获得任务的工作流工作线程的标识。Flow Framework 使用的 ID 的格式为：{*processId*}@{*host name*}。例如，以下是 Amazon SWF 控制台中示例执行的 `DecisionTaskStarted` 事件的详细信息：

事件时间戳	Mon Feb 20 11:52:40 GMT-800 2012
求同	2276@ip-0A6C1DF5
预定事件 ID	33

为了避免出现此情况，请对每个测试使用不同的任务列表。另外，请考虑在关闭旧工作线程和启动新工作线程之间添加延迟。

适用于 Java 的 AWS Flow Framework 参考

主题

适用于 Java 的 AWS Flow Framework 的注释

主题

@活动

此注释可在接口上用于声明一组活动类型。使用此注释进行注释的接口中的每个方法都表示活动类型。接口不能同时具有 `@Workflow` 和 `@Activities` 注释。

可对此注释指定以下参数：

activityNamePrefix

指定在接口中声明的活动类型的名称前缀。如果设置为空字符串（这是默认值），则后跟“.”的接口名称将用作前缀。

version

指定在接口中声明的活动类型的默认版本。默认值为 1.0。

dataConverter

指定在创建此活动类型的任务及其结果时用于序列化/反序列化数据的 DataConverter 的类型。默认设置为 NullDataConverter，这指示应使用 JsonDataConverter。

@活动

可在使用 @Activities 进行注释的接口中的方法上使用此注释。

可对此注释指定以下参数：

name

指定活动类型的名称。默认为空字符串，这指示应使用默认的前缀和活动方法名称来确定活动类型的名称（其格式为 {prefix}{name}）。请注意，在 @Activity 注释中指定名称时，框架不会自动在它之前加上前缀。您可以使用任意的命名方案。

version

指定活动类型的版本。这将覆盖在包含接口上的 @Activities 注释中指定的默认版本。默认值是空字符串。

@ActivityRegistrationOptions

指定活动类型的注册选项。可在使用 @Activities 进行注释的接口或其中的方法上使用此注释。如果同时在这两个地方指定，则在方法上使用的注释生效。

可对此注释指定以下参数：

defaultTasklist

指定此活动类型要向 Amazon SWF 注册的默认任务列表。在使用 ActivitySchedulingOptions 参数生成的客户端上调用活动方法时，会覆盖此默认值。默认设

置为 USE_WORKER_TASK_LIST。这是一个特殊值，指示应使用执行注册的工作线程所使用的任务列表。

defaultTaskScheduleToStartTimeoutSeconds

指定此+活动类型向 Amazon SWF 注册的 defaultTaskScheduleToStartTimeout。这是在将此活动类型的任务分配给工作线程之前，允许它等待的最长时间。有关更多详细信息，请参阅 Amazon Simple Workflow Service API 参考。

defaultTaskHeartbeatTimeoutSeconds

指定此活动类型向 Amazon SWF 注册的 defaultTaskHeartbeatTimeout。活动工作线程必须提供位于此持续时间内的检测信号；否则，任务将超时。默认设置为 -1，这是一个特殊值，指示应禁用此超时。有关更多详细信息，请参阅 Amazon Simple Workflow Service API 参考。

defaultTaskStartToCloseTimeoutSeconds

指定此活动类型向 Amazon SWF 注册的 defaultTaskStartToCloseTimeout。此超时确定工作线程可用于处理此类型的活动任务的最长时间。有关更多详细信息，请参阅 Amazon Simple Workflow Service API 参考。

defaultTaskScheduleToCloseTimeoutSeconds

指定此活动类型向 Amazon SWF 注册的 defaultScheduleToCloseTimeout。此超时确定任务可保持打开状态的总持续时间。默认设置为 -1，这是一个特殊值，指示应禁用此超时。有关更多详细信息，请参阅 Amazon Simple Workflow Service API 参考。

@异步

在工作流协调逻辑中的方法上使用时，指示该方法应异步执行。对该方法的调用将立即返回，但实际执行将在传递给该方法的所有 Promise<> 参数都已准备就绪时异步进行。使用 @Asynchronous 进行注释的方法必须具有返回类型 Promise<> 或 void。

daemon

指示为异步方法创建的任务是否应为守护程序任务。默认情况下为 False。

@Execute

在使用 @Workflow 注释进行注释的接口中的方法上使用时，标识工作流的入口点。

> **Important**
>
> 只有接口中的一个方法可以使用 @Execute 进行修饰。

可对此注释指定以下参数：

`name`

指定工作流类型的名称。如果未设置，则名称默认为 {prefix}{name}，其中 {prefix} 是后跟“.”的工作流程接口，{name} 是工作流程中 @Execute 装饰的方法。

`version`

指定工作流类型的版本。

@ExponentialRetry

在活动或异步方法上使用时，设置该方法引发未处理异常时的指数重试策略。在退避期之后进行重试尝试，退避期通过尝试次数的幂进行计算。

可对此注释指定以下参数：

`intialRetryIntervalSeconds`

指定在进行第一次重试尝试之前等待的持续时间。此值不应大于 `maximumRetryIntervalSeconds` 和 `retryExpirationSeconds`。

`maximumRetryIntervalSeconds`

指定重试尝试之间的最长持续时间。达到后，重试间隔的上限为此值。默认设置为 -1，这表示持续时间不受限制。

`retryExpirationSeconds`

指定在多长的持续时间之后，指数重试将停止。默认设置为 -1，这表示不会过期。

`backoffCoefficient`

指定用于计算重试间隔的系数。请参阅指数重试策略。

`maximumAttempts`

指定在多少次尝试之后，指数重试将停止。默认设置为 -1，这表示对重试尝试次数没有限制。

`exceptionsToRetry`

指定应触发重试的异常类型列表。这些类型的未处理异常不会进一步传播，将在计算的重试间隔后重试方法。默认情况下，此列表包含 `Throwable`。

`excludeExceptions`

指定不应触发重试的异常类型列表。将允许传播此类型的未处理异常。此列表默认为空。

@GetState

在使用 `@Workflow` 注释进行注释的接口中的方法上使用时，标识该方法用于检索最新工作流执行状态。在具有 `@Workflow` 注释的接口中最多可以有一个具有此注释的方法。具有此注释的方法不得使用任何参数，并且必须具有非 `void` 的返回类型。

@ManualActivityCompletion

此注释可在活动方法上用于指示在该方法返回时活动任务不应完成。活动任务不会自动完成，需要直接使用 Amazon SWF API 来手动完成。对于将活动任务委派给一些不是自动的或需要人为干预才能完成的外部系统的使用案例，这非常有用。

@Signal

在使用 `@Workflow` 注释进行注释的接口中的方法上使用时，标识可通过该接口声明的工作流类型的执行获得的信号。定义信号方法需要使用此注释。

可对此注释指定以下参数：

`name`

指定信号名称的名称部分。如果未设置，将使用方法的名称。

@SkipRegistration

在使用 `@Workflow` 注释进行注释的接口上使用时，可用于指示该工作流类型不应在 Amazon SWF 注册。必须在使用 `@Workflow` 注释的接口上使用 `@WorkflowRegistrationOptions` 和 `@SkipRegistrationOptions` 注释之一，但不能同时使用这两个注释。

@Wait 和 @NoWait

这些注释可在 Promise<> 类型的参数中使用，用于指示适用于 Java 的 AWS Flow Framework 是否应在执行方法之前等待其准备就绪。默认情况下，传递给 @Asynchronous 方法的 Promise<> 参数必须已准备就绪，然后方法才能执行。在某些情况下，有必要覆盖此默认行为。传递到 @Asynchronous 方法且使用 @NoWait 注释的 Promise<> 参数不等待。

必须使用 @Wait 注释来注释包含 Promise 的参数 (或其子类) 的集合，如 List<Promise<Int>>。默认情况下，框架不等待集合的成员。

@工作流

此注释在接口上用于声明工作流 类型。使用此注释修饰的接口应包含一个使用 @Execute 注释修饰的方法来声明您的工作流的入口点。

> **Note**
>
> 接口不能同时声明 @Workflow 和 @Activities 注释；它们相互排斥。

可对此注释指定以下参数：

`dataConverter`

指定在将请求发送给此工作流类型的工作流执行并从其接收结果时要使用哪个 DataConverter。

默认值为 NullDataConverter，这转而依赖 JsonDataConverter 以 JavaScript 对象表示法 (JSON) 来处理所有请求和响应数据。

示例

```
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

在使用 @Workflow 进行注释的接口上使用时，可用于提供 Amazon SWF 在注册工作流类型时使用的默认设置。

> **Note**
>
> 必须在使用 @Workflow 进行注释的接口上使用 @WorkflowRegistrationOptions 或 @SkipRegistrationOptions，但不能同时指定这两个。

可对此注释指定以下参数：

描述

工作流类型的可选文本描述。

defaultExecutionStartToCloseTimeoutSeconds

指定此工作流类型向 Amazon SWF 注册的 defaultExecutionStartToCloseTimeout。这是此类型的工作流执行完成所需的总时间。

有关工作流超时的更多信息，请参阅 Amazon SWF 超时类型 。

defaultTaskStartToCloseTimeoutSeconds

指定此工作流类型向 Amazon SWF 注册的 defaultTaskStartToCloseTimeout。这指定此类型的工作流执行的一个决策任务完成所需的时间。

如果您没有指定 defaultTaskStartToCloseTimeout，则默认为 30 秒。

有关工作流超时的更多信息，请参阅 Amazon SWF 超时类型 。

defaultTaskList

用于此工作流类型的执行的决策任务的默认任务列表。在启动工作流执行时，可以使用 StartWorkflowOptions 覆盖此处设置的默认值。

如果您没有指定 defaultTaskList，则默认设置为 USE_WORKER_TASK_LIST。这指示应使用执行工作流注册的工作线程所使用的任务列表。

defaultChildPolicy

指定在此类型的执行终止时用于子工作流的策略。默认值为 ABANDON。可能的值包括：

- ABANDON – 允许子工作流执行保持运行
- TERMINATE – 终止子工作流执行
- REQUEST_CANCEL – 请求取消子工作流执行

适用于 Java 的 AWS Flow Framework 异常

适用于 Java 的 AWS Flow Framework 使用以下异常。本章提供异常的概述。有关详细信息，请参阅各个异常的AWS SDK for Java文档。

主题

- ActivityFailureException
- ActivityTaskException
- ActivityTaskFailedException
- ActivityTaskTimedOutException
- ChildWorkflowException
- ChildWorkflowFailedException
- ChildWorkflowTerminatedException
- ChildWorkflowTimedOutException
- DataConverterException
- DecisionException
- ScheduleActivityTaskFailedException
- SignalExternalWorkflowException
- StartChildWorkflowFailedException
- StartTimerFailedException
- TimerException
- WorkflowException

ActivityFailureException

框架在内部使用此异常来沟通活动故障。当活动因未处理的异常而失败时，它将被封装在 ActivityFailureException 中并报告给 Amazon SWF。仅当您使用活动工作线程扩展性点时，才需要处理此异常。您的应用程序代码将从来不需要处理此异常。

ActivityTaskException

这是用于以下活动任务失败异常的基类：ScheduleActivityTaskFailedException、ActivityTaskFailedException、ActivityTask
它包含失败任务的任务 ID 和活动类型。您可以在工作流实现中捕获此异常，以常规方式处理活动故障。

ActivityTaskFailedException

在活动中未处理的异常将通过引发 ActivityTaskFailedException 报告回工作流实现。原始异常可从此异常的原因属性中检索。此异常还可提供对调试有用的其他信息，如历史记录中的唯一活动标识符。

架构可以通过从活动工作线程中序列化原始异常来提供远程异常。

ActivityTaskTimedOutException

如果活动超时，Amazon SWF 会引发此异常。如果活动任务无法在要求的时间段内分配给工作线程或在要求时间内无法由工作线程完成，则会出现此异常。您可以在调用活动方法时使用 @ActivityRegistrationOptions 注释或使用 ActivitySchedulingOptions 参数来为活动设置这些超时。

ChildWorkflowException

用于报告子工作流执行故障的异常的基类。该异常包含子工作流执行的 ID 及其工作流类型。您可以捕获此异常，以常规方式处理子工作流执行故障。

ChildWorkflowFailedException

子工作流中未处理的异常将通过引发 ChildWorkflowFailedException 报告回父工作流实现。原始异常可从此异常的 cause 属性中检索。该异常还可提供对调试有用的其他信息，如子执行的唯一标识符。

ChildWorkflowTerminatedException

此异常在父工作流执行中引发，以报告子工作流执行的终止。如果要处理子工作流的终止，例如执行清除或补偿，则应捕获此异常。

ChildWorkflowTimedOutException

此异常在父工作流执行中引发，用于报告子工作流执行超时并被由 Amazon SWF 关闭。如果要处理子工作流的强制关闭，例如执行清除或补偿，则应捕获此异常。

DataConverterException

架构使用 `DataConverter` 组件来封送和拆收通过线路发送的数据。如果 `DataConverter` 无法封送或拆收数据，将引发此异常。可能由于各种原因而出现这种情况，例如，因为用于封送和拆收数据的 `DataConverter` 组件中存在不匹配。

DecisionException

这一异常基类表示 Amazon SWF 执行决策失败。您可以捕获此异常，以常规方式处理此类异常。

ScheduleActivityTaskFailedException

如果 Amazon SWF 未能安排活动任务，将引发此异常。出现这种情况可能有多种原因，例如，活动被弃用，或您的账户已达到 Amazon SWF 的限制。异常中的 `failureCause` 属性指明了计划活动失败的确切原因。

SignalExternalWorkflowException

如果 Amazon SWF 无法处理某工作流执行向其他工作流执行发出的请求，将引发此异常。如果找不到目标工作流执行，也就是说，您指定的工作流执行不存在或处于关闭状态，将出现这种情况。

StartChildWorkflowFailedException

如果 Amazon SWF 未能启动子工作流执行，将引发此异常。出现这种情况可能有多种原因，例如，指定的子工作流类型被弃用或您的账户已达到 Amazon SWF 的限制。异常中的 `failureCause` 属性指明无法启动子工作流执行的确切原因。

StartTimerFailedException

如果 Amazon SWF 未能启动工作流执行请求的计时器，将引发此异常。如果指定计时器 ID 已在使用中或您的账户已达到 Amazon SWF 的限制，就可能会出现此情况。异常中的 `failureCause` 属性指明了故障的确切原因。

TimerException

这是与计时器相关的异常的基类。

WorkflowException

框架在内部使用此异常来报告工作流执行中的故障。仅当您使用工作流工作线程扩展性点时，才需要处理此异常。

适用于 Java 的 AWS Flow Framework 软件包

本节概述了适用于 Java 的 AWS Flow Framework 附带的软件包。有关每个软件包的更多信息，请参阅 AWS SDK for Java API Reference 中的 com.amazonaws.services.simpleworkflow.flow。

com.amazonaws.services.simpleworkflow.flow

包含与 Amazon SWF 集成的组件。

com.amazonaws.services.simpleworkflow.flow.annotations

包含适用于 Java 的 AWS Flow Framework 编程模型使用的注释。

com.amazonaws.services.simpleworkflow.flow.aspectj

包含功能所需的适用于 Java 的 AWS Flow Framework 组件，如 @异步 和 @ExponentialRetry。

com.amazonaws.services.simpleworkflow.flow.common

包含常见实用工具，例如框架定义的常量。

com.amazonaws.services.simpleworkflow.flow.core

包含核心功能，例如 `Task` 和 `Promise`。

com.amazonaws.services.simpleworkflow.flow.generic

包含作为其他功能的构建基础的核心组件，例如通用客户端。

com.amazonaws.services.simpleworkflow.flow.interceptors

包含框架提供的修饰器 (包括 `RetryDecorator`) 的实施。

com.amazonaws.services.simpleworkflow.flow.junit

包含提供 Junit 集成的组件。

com.amazonaws.services.simpleworkflow.flow.pojo

包含为基于注释的编程模型实施活动和工作流程定义的类。

com.amazonaws.services.simpleworkflow.flow.spring

包含提供 Spring 集成的组件。

com.amazonaws.services.simpleworkflow.flow.test

包含用于单元测试工作流程实施的帮助程序类，例如 `TestWorkflowClock`。

com.amazonaws.services.simpleworkflow.flow.worker

包含活动和工作流程工作线程的实施。

文档历史记录

下表描述了自《适用于 Java 的 AWS Flow Framework 开发人员指南》上次发布以来对该文档所做的重要更改。

- API 版本：2012-01-25
- 最近文档更新时间：2018 年 6 月 25 日

更改	描述	更改日期
更新	修复了 `backoffCoefficient` 的 `@ExponentialRetry` 描述中的错误。请参阅@ExponentialRetry。	2018 年 25 月 6 日
更新	清除了本指南中的代码示例。	2017 年 6 月 5 日
更新	简化并改进了本指南的组织结构和内容。	2017 年 5 月 19 日
更新	简化并改进了对决策程序代码进行更改：版本控制和功能标志一节。	2017 年 4 月 10 日
更新	增加了新的最佳实践一节，其中具有有关对决策程序代码进行更改的新指导。	2017 年 3 月 3 日
新特征	除了传统活动任务之外，您还可以在工作流中指定 Lambda 任务。有关更多信息，请参阅实施 AWS Lambda 任务。	2015 年 7 月 21 日
新特征	Amazon SWF 支持在任务列表中设置任务优先级，并尝试先交付优先级较高的任务，然后再交付优先级较低的任务。有关更多信息，请参阅设置任务优先级。	2014 年 12 月 17 日
更新	进行了更新和修复。	2013 年 8 月 1 日
更新	• 进行了更新和修正，包括更新 Eclipse 4.3 和 AWS SDK for Java 1.4.7 的设置说明。	2013 年 6 月 28 日

更改	描述	更改日期
	• 增加了一组新的有关构建初学者场景的教程	
新特征	适用于 Java 的 AWS Flow Framework 的初始版本。	2012 年 2 月 27 日

AWS 术语表

有关最新的 AWS 术语，请参阅《AWS 词汇表参考》中的 AWS 词汇表。

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