Guía para desarrolladores

AWS Flow Framework para Java

Versión de API 2021-04-28

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Flow Framework para Java: Guía para desarrolladores

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Las marcas comerciales y la imagen comercial de Amazon no se pueden utilizar en relación con ningún producto o servicio que no sea de Amazon de ninguna manera que pueda causar confusión entre los clientes y que menosprecie o desacredite a Amazon. Todas las demás marcas registradas que no son propiedad de Amazon son propiedad de sus respectivos propietarios, que pueden o no estar afiliados, conectados o patrocinados por Amazon.

Table of Contents

¿Qué es AWS Flow Framework para Java? ... 1
¿Qué puede encontrarse en esta guía? ... 1
Introducción ... 3
Configuración del marco de trabajo ... 3
Agregue el marco de flujo con Maven ... 4
HelloWorld Solicitud ... 4
HelloWorld Actividades: Implementación ... 5
HelloWorld Trabajador de Workflow ... 6
HelloWorld Inicio del flujo de trabajo ... 7
HelloWorldWorkflow Solicitud ... 8
HelloWorldWorkflow Trabajador de actividades ... 10
HelloWorldWorkflow Trabajador de Workflow ... 12
HelloWorldWorkflow Implementación de flujos de trabajo y actividades ... 17
HelloWorldWorkflow Motor de arranque ... 21
HelloWorldWorkflowAsync Solicitud ... 26
HelloWorldWorkflowAsync Implementación de actividades ... 27
HelloWorldWorkflowAsync Aplicación del flujo de trabajo ... 28
HelloWorldWorkflowAsync Organizador e iniciador del flujo de trabajo y las actividades ... 30
HelloWorldWorkflowDistributed Solicitud ... 31
HelloWorldWorkflowParallel Solicitud ... 34
HelloWorldWorkflowParallel Actividades: Trabajador ... 35
HelloWorldWorkflowParallel Workflow Worker ... 36
HelloWorldWorkflowParallel Flujo de trabajo y actividades: anfitrión e iniciador ... 38
Comprensión AWS Flow Framework ... 39
Estructura de la aplicación ... 39
Función del proceso de trabajo de actividad ... 41
Función del proceso de trabajo de flujo de trabajo ... 41
Función del iniciador del flujo de trabajo ... 42
Cómo interactúa Amazon SWF con la aplicación ... 42
Para obtener más información ... 43
Ejecución de confianza ... 43
Proporcionar comunicación de confianza ... 43
Garantizar que no se pierden los resultados ... 44
Tratamiento de componentes distribuidos en los que se ha producido un error ... 45

- Ejecución distribuida ... 45
  - Reproducción de flujos de trabajo ... 46
  - Métodos de flujo de trabajo asíncronos y de reproducción ... 47
  - Reproducción e implementación de flujos de trabajo ... 47
- Listas de tareas y ejecución de tareas ... 48
- Aplicaciones escalables ... 50
- Intercambio de datos entre actividades y flujos de trabajo ... 51
  - El Promise <T> Type ... 52
  - Conversores de datos y serialización ... 53
- Intercambio de datos entre aplicaciones y ejecuciones de flujo de trabajo ... 53
- Tipos de tiempo de espera ... 54
  - Tiempos de espera de las tareas de decisión y flujo de trabajo ... 55
  - Tiempos de espera de las tareas de actividad ... 56

Comprensión de las tareas ... 58
- Tarea ... 58
- Orden de ejecución ... 59
- Ejecución del flujo de trabajo ... 61
- No determinismo ... 63

Guía de programación ... 65
- Implementación de aplicaciones de flujo de trabajo ... 65
- Contratos de flujo de trabajo y de actividad ... 67
- Registro de tipos de flujos de trabajo y de actividades ... 70
  - Nombre del tipo de flujo de trabajo y versión ... 71
  - Nombre de la señal ... 71
  - Nombre del tipo de actividad y versión ... 71
  - Default Task List ... 72
  - Otras opciones de registro ... 72
- Clientes de actividad y flujo de trabajo ... 73
  - Clientes de flujo de trabajo ... 73
  - Clientes de actividad ... 82
  - Opciones de programación ... 86
  - Clientes dinámicos ... 87
- Implementación de flujos de trabajo ... 89
  - Contexto de la decisión ... 90
  - Exposición del estado de ejecución ... 90
  - Variables locales del flujo de trabajo ... 93

Implementación de actividades ... 94
Finalización manual de actividades ... 95
Implementación de tareas de Lambda ... 97
Acerca de AWS Lambda ... 97
Beneficios y limitaciones de la utilización de tareas de Lambda ... 97
Uso de tareas Lambda en sus flujos de trabajo AWS Flow Framework para Java ... 98
Ver la HelloLambda muestra ... 103
Ejecución de programas escritos con AWS Flow Framework para Java ... 103
WorkflowWorker ... 105
ActivityWorker ... 105
Modelo de subprocesos de proceso de trabajo ... 105
Extensibilidad de proceso de trabajo ... 108
Contexto de ejecución ... 109
Contexto de la decisión ... 109
Contexto de ejecución de actividad ... 111
Ejecuciones de flujo de trabajo secundario ... 112
Flujos de trabajo continuos ... 115
Establecer la prioridad de las tareas ... 116
Establecimiento de prioridad de las tareas para flujos de trabajo ... 117
Establecimiento de prioridad de las tareas para actividades ... 118
DataConverters ... 118
Paso de datos a los métodos asíncronos ... 119
Paso de colecciones y mapas a métodos asíncronos ... 119
Settable <T> ... 120
@NoWait ... 122
Promise <Void> ... 122
AndPromise y OrPromise ... 122
Capacidad de realización de pruebas e inserción de dependencias ... 123
Integración con Spring ... 123
JUnit Integration ... 130
Gestión de errores ... 136
TryCatchFinally Semántica ... 138
Cancelación ... 139
Anidado TryCatchFinally ... 144
Reintento de actividades con errores ... 145
Retry-Until-Success Estrategia ... 146

Estrategia de reintento exponencial 149
Estrategia de reintento personalizada 156
Tareas del demonio 159
Comportamiento de reproducción 161
Ejemplo 1: reproducción síncrona 161
Ejemplo 2: reproducción asíncrona 163
Véase también 164
Prácticas recomendadas 165
Realización de cambios en el código del decisor 165
El proceso de reproducción y cambios de códigos 165
Escenario de ejemplo 166
Soluciones 173
Resolución de problemas 179
Errores de compilación 179
Fallo de recurso desconocido 179
Excepciones al llamar a get () con una promesa 180
Flujos de trabajo no deterministas 180
Problemas debidos al control de versiones 181
Solución de problemas y depuración de la ejecución de un flujo de trabajo 181
Tareas perdidas 183
Fallo de validación debido a restricciones de longitud de los parámetros de la API 183
Referencia 185
Anotaciones 185
@Tareas 185
@Actividad 186
@ActivityRegistrationOptions 187
@Asynchronous 188
@Execute 188
@ExponentialRetry 189
@GetState 190
@ManualActivityCompletion 190
@Signal 190
@SkipRegistration 190
@Wait y @ NoWait 191
@Flujo de trabajo 191
@WorkflowRegistrationOptions 192

Excepciones ... 193
ActivityFailureException ... 194
ActivityTaskException ... 194
ActivityTaskFailedException ... 194
ActivityTaskTimedOutException ... 195
ChildWorkflowException ... 195
ChildWorkflowFailedException ... 195
ChildWorkflowTerminatedException ... 195
ChildWorkflowTimedOutException ... 195
DataConverterException ... 196
DecisionException ... 196
ScheduleActivityTaskFailedException ... 196
SignalExternalWorkflowException ... 196
StartChildWorkflowFailedException ... 196
StartTimerFailedException ... 197
TimerException ... 197
WorkflowException ... 197
Paquetes ... 197
Historial de documentos ... 199
... cci

¿Qué es AWS Flow Framework para Java?

Con él AWS Flow Framework, puede centrarse en implementar la lógica de su flujo de trabajo. Entre bastidores, el marco utiliza las capacidades de programación, enrutamiento y administración del estado de Amazon SWF para gestionar la ejecución del flujo de trabajo y hacerlo escalable, fiable y auditable. AWS Flow Framework los flujos de trabajo basados en datos son muy simultáneos. Los flujos de trabajo se pueden distribuir en varios componentes, que pueden ejecutarse como procesos independientes en equipos distintos y escalarse de forma independiente. La aplicación puede seguir progresando si alguno de sus componentes está en ejecución, lo que la hace muy tolerante a los errores.

¿Qué puede encontrarse en esta guía?

Esta guía contiene información sobre cómo instalar, configurar y utilizar AWS Flow Framework para crear aplicaciones de Amazon SWF.

Primeros pasos con el AWS Flow Framework para Java

Si acaba de empezar con el software AWS Flow Framework para Java, lea la Primeros pasos con el AWS Flow Framework para Java sección. Le guiará a través de la descarga e instalación del programa AWS Flow Framework para Java, cómo configurar su entorno de desarrollo y le mostrará un ejemplo sencillo de cómo crear un flujo de trabajo.

Comprensión AWS Flow Framework de Java

Presenta los AWS Flow Framework conceptos y Amazon SWF básicos, y describe la estructura básica de una AWS Flow Framework aplicación y el modo en que se intercambian los datos entre las partes de un flujo de trabajo distribuido.

AWS Flow Framework Guía de programación para Java

Este capítulo proporciona una guía de programación básica para el desarrollo de aplicaciones de flujo de trabajo con AWS Flow Framework para Java, que incluye cómo registrar tipos de actividades y de flujos de trabajo, cómo implementar clientes de flujo de trabajo, cómo crear flujos de trabajo secundarios, cómo gestionar errores, etc.

Descripción de una tarea en AWS Flow Framework for Java

En este capítulo se ofrece un análisis más detallado del funcionamiento de Java y se proporciona información adicional sobre el AWS Flow Framework orden de ejecución de los flujos de trabajo asíncronos y un paso lógico para la ejecución de un flujo de trabajo estándar.

[Consejos de solución de problemas y depuración AWS Flow Framework para Java]

Este capítulo proporciona información sobre errores habituales que puede utilizar para solucionar los problemas de sus flujos de trabajo o que puede utilizar para aprender cómo evitar errores habituales.

[AWS Flow Framework para Java Reference]

Este capítulo es una referencia a las anotaciones, excepciones y paquetes que el AWS Flow Framework para Java añade al SDK para Java.

Primeros pasos con el AWS Flow Framework para Java

En esta sección, AWS Flow Framework se presenta una serie de ejemplos de aplicaciones sencillas que presentan el modelo de programación básico y la API. Las aplicaciones de ejemplo están basadas en la aplicación Hello World estándar que se utiliza para realizar introducciones a C y los lenguajes de programación relacionados. Aquí tiene una implementación típica en Java de Hello World:

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

A continuación, se ofrece una breve descripción de las aplicaciones de ejemplo. Se incluye el código fuente completo para que pueda implementar y ejecutar las aplicaciones usted mismo. Antes de empezar, primero debe configurar su entorno de desarrollo y crear un proyecto AWS Flow Framework para Java, como enConfiguración del AWS Flow Framework para Java.

- En HelloWorld Solicitud, se ofrece una introducción a las aplicaciones de flujo de trabajo. Para ello, se implementa Hello World como aplicación de Java estándar, pero estructurándola como una aplicación de flujo de trabajo.
- HelloWorldWorkflow Solicitudusa AWS Flow Framework for Java para HelloWorld convertirlo en un flujo de trabajo de Amazon SWF.
- HelloWorldWorkflowAsync Solicitud modifica `HelloWorldWorkflow` para utilizar un método de flujo de trabajo asíncrono.
- HelloWorldWorkflowDistributed Solicitud modifica `HelloWorldWorkflowAsync` para que los procesos de trabajo de flujo de trabajo y actividad puedan ejecutarse en distintos sistemas.
- HelloWorldWorkflowParallel Solicitud modifica `HelloWorldWorkflow` para ejecutar dos actividades en paralelo.

Configuración del AWS Flow Framework para Java

El AWS Flow Framework para Java se incluye con. AWS SDK para Java Si aún no lo ha configurado AWS SDK para Java, consulte Introducción en la Guía para AWS SDK para Java desarrolladores para obtener información sobre la instalación y la configuración del propio SDK.

Agregue el marco de flujo con Maven

Las herramientas de compilación de Amazon SWF son de código abierto; para ver o descargar el código o para crear las herramientas usted mismo, visite el repositorio en. https://github.com/aws/aws-swf-build-tools

Amazon proporciona herramientas de compilación de Amazon SWF en el repositorio central de Maven.

Para configurar el marco de trabajo del flujo para Maven, añada la siguiente dependencia al archivo `pom.xml` de sus proyectos:

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

HelloWorld Solicitud

Para presentar de qué manera se estructuran las aplicaciones de Amazon SW, crearemos una aplicación de Java que se comporta como flujo de trabajo, pero que se ejecuta localmente en un proceso individual. No será necesario conectarse a Amazon Web Services.

Note

El HelloWorldWorkflowejemplo se basa en este ejemplo y se conecta a Amazon SWF para gestionar el flujo de trabajo.

Una aplicación de flujo de trabajo consta de tres componentes básicos:

- Un proceso de trabajo de actividades admite un conjunto de actividades, cada uno de ellos es un método que se ejecuta de manera independiente para realizar una tarea en particular.
- Un proceso de trabajo de flujo de trabajo organiza la ejecución de las actividades y administra el flujo de datos. Se trata de una realización programática de una topología de flujo de trabajo, que es básicamente un diagrama que define cuándo se ejecutan las diferentes actividades, si se ejecutan secuencial o simultáneamente, etc.

- Un iniciador de flujo de trabajo comienza una instancia de flujo de trabajo, que se denomina ejecución, y puede interactuar con ella durante la ejecución.

HelloWorld se implementa como tres clases y dos interfaces relacionadas, que se describen en las siguientes secciones. Antes de empezar, debe configurar su entorno de desarrollo y crear un nuevo proyecto AWS Java, tal y como se describe en[Configuración del AWS Flow Framework para Java](). Los paquetes utilizados para los siguientes tutoriales se denominan todos `helloWorld.`*`XYZ`*. Para utilizar esos nombre, configure el atributo `within` en aop.xml tal y como se indica a continuación:

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

Para implementarlo HelloWorld, cree un nuevo paquete Java en su proyecto de AWS SDK con el nombre `helloWorld.HelloWorld` y añada los siguientes archivos:

- Un archivo de interfaz denominado `GreeterActivities.java`
- Un archivo de clase denominado `GreeterActivitiesImpl.java`, que implementa el proceso de trabajo de actividades.
- Un archivo de interfaz denominado `GreeterWorkflow.java`.
- Un archivo de clase denominado `GreeterWorkflowImpl.java`, que implementa el proceso de trabajo de flujo de trabajo.
- Un archivo de clase denominado `GreeterMain.java`, que implementa el iniciador de flujo de trabajo.

Los detalles se discuten en las siguientes secciones e incluyen el código completo para cada componente, que puede añadir al archivo apropiado.

HelloWorld Actividades: Implementación

HelloWorld divide la tarea general de imprimir un `"Hello World!"` saludo en la consola en tres tareas, cada una de las cuales se realiza mediante un método de actividad. Los métodos de la actividad se definen en la interfaz `GreeterActivities` de la siguiente manera.

```
public interface GreeterActivities {
```

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

HelloWorld tiene una implementación de actividad`GreeterActivitiesImpl`, que proporciona los `GreeterActivities` métodos que se muestran a continuación:

```
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);
   }
}
```

Las actividades son independientes entre sí y con frecuencia pueden ser utilizadas por diferentes flujos de trabajo. Por ejemplo, cualquier flujo de trabajo puede usar la actividad `say` para imprimir una cadena en la consola. Los flujos de trabajo también pueden tener implementaciones de múltiples actividades, cada una de las cuales realiza un conjunto diferentes de tareas.

HelloWorld Trabajador de Workflow

Cómo imprimir “Hello World!” en la consola, las tareas de actividades se deben ejecutar en secuencia en el orden correcto y con los datos correctos. El trabajador del HelloWorld flujo de trabajo organiza la ejecución de las actividades basándose en una topología de flujo de trabajo lineal simple, como se muestra en la siguiente figura.

Las tres actividades se ejecutan en secuencia y los datos fluyen de una actividad a la siguiente.

El trabajador del HelloWorld flujo de trabajo tiene un único método, el punto de entrada del flujo de trabajo, que se define en la `GreeterWorkflow` interfaz de la siguiente manera:

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

La clase `GreeterWorkflowImpl` implementa esta interfaz de la siguiente manera:

```
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);
   }
}
```

El `greet` método implementa la HelloWorld topología creando una instancia de`GreeterActivitiesImpl`, llamando a cada método de actividad en el orden correcto y pasando los datos correspondientes a cada método.

HelloWorld Inicio del flujo de trabajo

Un iniciador del flujo de trabajo es una aplicación que inicia la ejecución de un flujo de trabajo y que también podría comunicarse con el flujo de trabajo mientras se está ejecutando. La `GreeterMain` clase implementa el iniciador del HelloWorld flujo de trabajo de la siguiente manera:

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

`GreeterMain` crea una instancia de `GreeterWorkflowImpl` y llama a `greet` para ejecutar el proceso de trabajo del flujo de trabajo. Ejecute `GreeterMain` como una aplicación de Java; al hacerlo, debería poder ver "Hello World!" en la consola.

HelloWorldWorkflow Solicitud

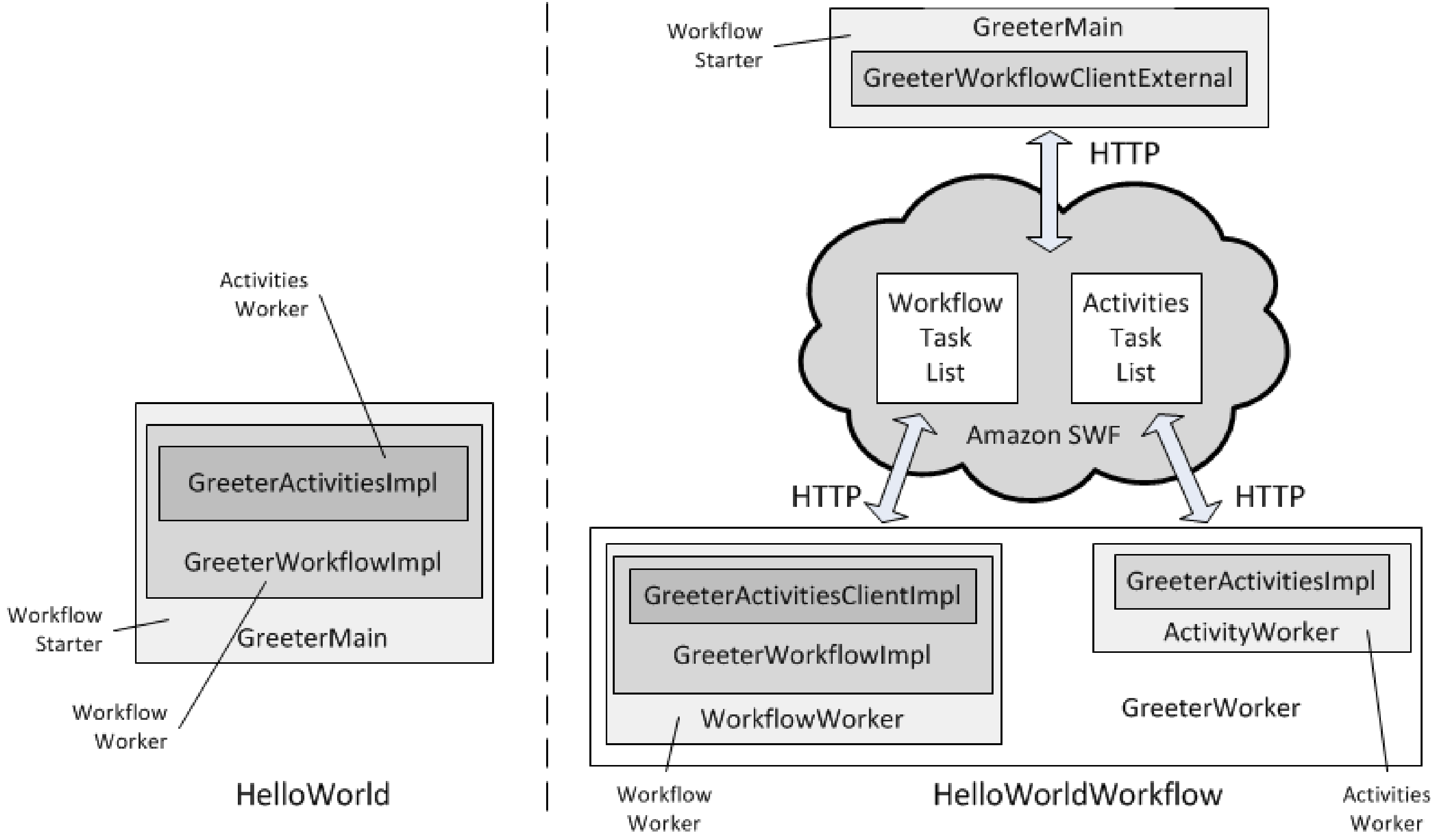
Aunque el HelloWorldejemplo básico está estructurado como un flujo de trabajo, se diferencia de un flujo de trabajo de Amazon SWF en varios aspectos clave:

Aplicaciones de flujo de trabajo convencionales y de Amazon SWF

HelloWorld	Flujo de trabajo de Amazon SWF
Se ejecuta de manera local como un solo proceso.	Se ejecuta como varios procesos que se pueden distribuir en varios sistemas, incluidas EC2 las instancias de Amazon, los centros de datos privados, los ordenador es cliente, etc. Ni siquiera tienen que ejecutar el mismo sistema operativo.
Las actividades son métodos síncronos, que bloquean hasta que terminan.	Las actividades están representadas mediante métodos asíncronos, que se devuelven inmediatamente y permiten que el flujo de trabajo realice otras tareas mientras esperan a que finalice la actividad.
El proceso de trabajo del flujo de trabajo interactúa con el proceso de trabajo de las actividades llamando al método adecuado.	Los procesos de trabajo del flujo de trabajo interactú an con los procesos de trabajo de las actividades mediante solicitudes HTTP, donde Amazon SWF actúa de intermediario.
El iniciador del flujo de trabajo interactúa con el proceso de trabajo del flujo de trabajo llamando al método adecuado.	Los iniciadores de flujos de trabajo interactúan con los procesos de trabajo de los flujos de trabajo mediante solicitudes HTTP, donde Amazon SWF actúa de intermediario.

Podría implementar una aplicación de flujo de trabajo asíncrono distribuida a partir de cero haciendo, por ejemplo, que el proceso de trabajo del flujo de trabajo interactúe con el proceso de trabajo de actividad directamente a través de llamadas a servicios web. Sin embargo, luego tendrá que implementar todo el código tan complicado que es necesario para administrar la ejecución asíncrona de varias actividades, controlar el flujo de datos, etc. Los SWF AWS Flow Framework para Java y Amazon se ocupan de todos esos detalles, lo que le permite centrarse en la implementación de la lógica empresarial.

HelloWorldWorkflow es una versión modificada HelloWorld que se ejecuta como un flujo de trabajo de Amazon SWF. La siguiente figura resume cómo funcionan las dos aplicaciones.



HelloWorld se ejecuta como un proceso único y el iniciador, el trabajador del flujo de trabajo y el trabajador de actividades interactúan mediante llamadas a métodos convencionales. Con `HelloWorldWorkflow`, el iniciador, el proceso de trabajo del flujo de trabajo y el proceso de trabajo de las actividades son componentes distribuidos que interactúan a través de Amazon SWF mediante solicitudes HTTP. Para gestionar la interacción, Amazon SWF mantiene listas de tareas de actividad y de flujos de trabajo, que envía a los componentes correspondientes. En esta sección se describe cómo funciona el marco para HelloWorldWorkflow.

HelloWorldWorkflow se implementa mediante la API AWS Flow Framework para Java, que gestiona los detalles, a veces complicados, de la interacción con Amazon SWF en segundo plano y simplifica considerablemente el proceso de desarrollo. Puede utilizar el mismo proyecto para el que utilizó HelloWorld, que ya está configurado AWS Flow Framework para las aplicaciones Java. Sin embargo, para ejecutar la aplicación, debe configurar una cuenta de Amazon SWF de la siguiente forma:

- Crea una AWS cuenta, si aún no la tienes, en Amazon Web Services.

- Asigna el ID de acceso y el ID secreto de tu cuenta a las variables de AWS_SECRET_KEY entorno AWS_ACCESS_KEY_ID y a las variables de entorno, respectivamente. Es recomendable no exponer los valores de las claves literales en el código. Para solucionar este problema, lo más práctico es guardarlas en variables de entorno.
- Cree una cuenta de Amazon SWF en Amazon Simple Workflow Service.
- Inicie sesión Consola de administración de AWS y seleccione el servicio Amazon SWF.
- Elija Administrar dominios en la esquina superior derecha y registre un dominio nuevo de Amazon SWF. Un dominio es un contenedor lógico para los recursos de la aplicación, como tipos de flujo de trabajo y actividad, además de para ejecuciones de flujo de trabajo. Puede usar cualquier nombre de dominio conveniente, pero los tutoriales usan "». helloWorldWalkthrough

Para implementar el HelloWorldWorkflow, cree una copia de HelloWorld. HelloWorld empaquete en el directorio de su proyecto y asígnele el nombre HelloWorld. HelloWorldWorkflow. En las siguientes secciones se describe cómo modificar el HelloWorld código original para usarlo AWS Flow Framework para Java y ejecutarlo como una aplicación de flujo de trabajo de Amazon SWF.

HelloWorldWorkflow Trabajador de actividades

HelloWorld implementó sus actividades como trabajador como una sola clase. Un trabajador de actividades AWS Flow Framework para Java tiene tres componentes básicos:

- Los métodos de actividad, que realizan las tareas propiamente dichas, se definen en una interfaz y se implementan en una clase relacionada.
- Una ActivityWorkerclase gestiona la interacción entre los métodos de actividad y Amazon SWF.
- Una aplicación host de actividades registra e inicia el proceso de trabajo de actividades y se encarga de la limpieza.

En esta sección, se explican los métodos de actividades; las otras dos clases se explicarán más adelante.

HelloWorldWorkflow define la interfaz de actividades de `GreeterActivities` la siguiente manera:

```
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);
}
```

Esta interfaz no era estrictamente necesaria para una aplicación Java HelloWorld, pero sí lo es AWS Flow Framework para una aplicación Java. Fíjese en que la definición de la interfaz en sí no ha cambiado. Sin embargo, debe aplicar dos AWS Flow Framework para las anotaciones de Java @ActivityRegistrationOptions y@Tareas, a la definición de la interfaz. Las anotaciones proporcionan información de configuración e indican al AWS Flow Framework procesador de anotaciones de Java que utilice la definición de la interfaz para generar una clase de cliente de actividades, como se explica más adelante.

`@ActivityRegistrationOptions`tiene varios valores con nombre que se utilizan para configurar el comportamiento de las actividades. HelloWorldWorkflow especifica dos tiempos de espera:

- `defaultTaskScheduleToStartTimeoutSeconds` especifica durante cuánto tiempo pueden estar en cola las tareas en la lista de tareas de actividad y se establece en 300 segundos (5 minutos).
- `defaultTaskStartToCloseTimeoutSeconds` especifica el tiempo máximo que puede tardar la actividad en realizar la tarea y se establece en 10 segundos.

Estos tiempos de espera garantizan que la actividad complete su tarea en un tiempo razonable. Si se supera cualquiera de estos tiempos de espera, el marco de trabajo genera un error y el proceso de trabajo de flujo de trabajo debe decidir cómo tratar el problema. Para leer un debate sobre cómo tratar estos errores, consulte Gestión de errores.

`@Activities` tiene varios valores, pero normalmente tan solo especifica el número de versión de las actividades, lo que le permite realizar un seguimiento de diferentes generaciones de implementaciones de las actividades. Si cambia una interfaz de actividad después de registrarla con Amazon SWF, lo que incluye cambiar los valores de `@ActivityRegistrationOptions`, debe utilizar un número de versión nuevo.

HelloWorldWorkflow implementa los métodos de actividad de `GreeterActivitiesImpl` la siguiente manera:

```
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);
   }
}
```

Observe que el código es idéntico a la HelloWorld implementación. En esencia, una AWS Flow Framework actividad es solo un método que ejecuta algún código y, tal vez, devuelve un resultado. La diferencia entre una aplicación estándar y una aplicación de flujo de trabajo de Amazon SWF radica en la forma en que el flujo de trabajo ejecuta las actividades, dónde se ejecutan estas y cómo se devuelven los resultados al proceso de trabajo de flujo de trabajo.

HelloWorldWorkflow Trabajador de Workflow

Un proceso de trabajo de flujo de trabajo de Amazon SWF posee tres componentes básicos:

- Una implementación de flujo de trabajo, que es una clase que realiza las tareas relacionadas con el flujo de trabajo.
- Una clase de cliente de actividades, que básicamente es un proxy para la clase de actividades y se utiliza en una implementación de flujo de trabajo para ejecutar métodos de actividades de manera asíncrona.
- Una WorkflowWorkerclase que gestiona la interacción entre el flujo de trabajo y Amazon SWF.

En esta sección, se explica la implementación del flujo de trabajo y el cliente de actividades; la clase `WorkflowWorker` se explicará más adelante.

HelloWorldWorkflow define la interfaz del flujo de trabajo de `GreeterWorkflow` la siguiente manera:

```
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();
}
```

Esta interfaz tampoco es estrictamente necesaria HelloWorld , pero es esencial para una aplicación AWS Flow Framework para Java. Debe aplicar dos AWS Flow Framework para las anotaciones de Java @Flujo de trabajo y@WorkflowRegistrationOptions, a la definición de la interfaz de flujo de trabajo. Las anotaciones proporcionan información de configuración y también indican al procesador de anotaciones AWS Flow Framework para Java que genere una clase de cliente de flujo de trabajo basada en la interfaz, como se explica más adelante.

`@Workflow`tiene un parámetro opcional, DataConverter, que suele utilizarse con su valor predeterminado, NullDataConverter, que indica que JsonDataConverter debe utilizarse.

`@WorkflowRegistrationOptions` también posee una serie de parámetros opcionales que se pueden utilizar para configurar el proceso de trabajo de flujo de trabajo. Aquí, hay que establecer `defaultExecutionStartToCloseTimeoutSeconds`, que especifica el tiempo durante el que puede ejecutarse el flujo de trabajo: hasta 3600 segundos (1 hora).

La definición de la `GreeterWorkflow` interfaz difiere de HelloWorld la @Execute anotación en un aspecto importante. Las interfaces de flujo de trabajo especifican los métodos a los que pueden llamar las aplicaciones como el iniciador de flujo de trabajo y se limitan a un puñado de métodos, cada uno con un rol particular. El marco de trabajo no especifica una lista de parámetros o de nombres para los métodos de la interfaz del flujo de trabajo; se utiliza una lista de parámetros y nombres que sea adecuada para el flujo de trabajo y se aplica una anotación de AWS Flow Framework para Java con el fin de identificar el rol del método.

`@Execute` tiene dos fines:

- Identifica `greet` como el punto de entrada del flujo de trabajo; es decir, el método al que llama el iniciador del flujo de trabajo para iniciar el flujo de trabajo. En general, un punto de entrada puede tomar uno o más parámetros, lo que permite al iniciador inicializar el flujo de trabajo, pero en este ejemplo no es necesario realizar la inicialización.

- Especifica el número de versión del flujo de trabajo, lo que le permite realizar un seguimiento de diferentes generaciones de implementaciones del flujo de trabajo. Para cambiar una interfaz de flujo de trabajo después de registrarla con Amazon SWF, lo que incluye cambiar los valores de tiempo de espera, debe utilizar un número de versión nuevo.

Para obtener información acerca de otros métodos que se pueden incluir en una interfaz de flujo de trabajo, consulte Contratos de flujo de trabajo y de actividad.

HelloWorldWorkflow implementa el flujo de trabajo de `GreeterWorkflowImpl` la siguiente manera:

```
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);
   }
}
```

El código es similar a HelloWorld, pero con dos diferencias importantes.

- `GreeterWorkflowImpl` crea una instancia de `GreeterActivitiesClientImpl`, el cliente de actividades, en lugar de `GreeterActivitiesImpl`, y ejecuta las actividades llamando a los métodos en el objeto de cliente.
- Las actividades name y greeting devuelven objetos `Promise<String>` en lugar de objetos `String`.

HelloWorld es una aplicación Java estándar que se ejecuta localmente como un proceso único, por lo que `GreeterWorkflowImpl` puede implementar la topología del flujo de trabajo simplemente creando una instancia de`GreeterActivitiesImpl`, llamando a los métodos en orden y pasando los valores devueltos de una actividad a la siguiente. Con un flujo de trabajo de Amazon SWF, un método de actividad sigue realizando una tarea de actividad desde `GreeterActivitiesImpl`. Sin embargo, el método no se ejecuta necesariamente en el mismo proceso que el flujo de trabajo (es posible que ni siquiera se ejecute en el mismo sistema) y el flujo de trabajo tiene que ejecutar la actividad de manera asíncrona. Estos requisitos plantean los siguientes problemas:

- Cómo ejecutar un método de actividad que podría estar ejecutándose en un proceso diferente, quizá en otro sistema.
- Cómo ejecutar un método de actividad de manera asíncrona.
- Cómo administrar los valores de entrada y retorno de las actividades. Por ejemplo, si el valor de retorno de la Actividad A es una entrada en la Actividad B, debe asegurarse de que la Actividad B no se ejecute hasta que la Actividad A haya terminado.

Puede implementar una gran variedad de topologías de flujos de trabajo a través del flujo de control de la aplicación utilizando un control de flujo de Java conocido combinado con el cliente de actividades y `Promise<T>`.

Cliente de actividades

`GreeterActivitiesClientImpl` básicamente es un proxy para `GreeterActivitiesImpl` que permite que una implementación de flujo de trabajo ejecute los métodos `GreeterActivitiesImpl` de manera asíncrona.

Las clases `GreeterActivitiesClient` y `GreeterActivitiesClientImpl` se generan automáticamente utilizando la información que se proporciona en las anotaciones que se aplican a la clase `GreeterActivities`. No es necesario que las implemente personalmente.

Note

Eclipse genera estas clases al guardar el proyecto. Puede ver el código generado en el subdirectorio `.apt_generated` del directorio del proyecto.
Para evitar errores de compilación en la clase `GreeterWorkflowImpl`, es recomendable mover el directorio `.apt_generated` a la parte superior de la pestaña Order and Export (Pedido y exportación) del cuadro de diálogo Java Build Path (Ruta de compilación Java).

El proceso de trabajo del flujo de trabajo ejecuta una actividad llamando al método de cliente correspondiente. El método es asíncrono y devuelve inmediatamente un objeto `Promise<T>`, donde T es el tipo de retorno de la actividad. El objeto `Promise<T>` devuelto es básicamente un marcador de posición del valor que devolverá al final el método de la actividad.

- Cuando se devuelve el método del cliente de las actividades, el objeto `Promise<T>` se encuentra inicialmente en estado no preparado, lo que indica que el objeto no representa aún un valor de retorno válido.

- Cuando el método de actividad correspondiente termina su tarea y se devuelve, el marco de trabajo asigna el valor de retorno al objeto `Promise<T>` y lo establece en estado preparado.

Promise <T> Type

La principal finalidad de los objetos `Promise<T>` es administrar el flujo de datos entre los componentes asíncronos y el control cuando se ejecutan. De esta forma, la aplicación ya no tiene que administrar la sincronización de forma explícita ni depender de mecanismos, como temporizadores, para asegurarse de que los componentes asíncronos no se ejecutan de forma prematura. Cuando se llama a un método del cliente de actividades, este se devuelve inmediatamente, pero el marco de trabajo aplaza la ejecución del método de actividad correspondiente hasta que haya preparado algún objeto `Promise<T>` de entrada que represente datos válidos.

Desde la perspectiva de `GreeterWorkflowImpl`, los tres métodos de cliente de las actividades se devuelven inmediatamente. Desde la perspectiva de `GreeterActivitiesImpl`, el marco de trabajo no llama a `getGreeting` hasta que `name` termina y no llama a `say` hasta que `getGreeting` termina.

Si utilizamos `Promise<T>` para pasar datos de una actividad a la siguiente, `HelloWorldWorkflow` no solo se asegura de que los métodos de actividad no intentan utilizar datos no válidos, sino que también controla cuándo se ejecutan las actividades y define de manera implícita la topología del flujo de trabajo. Para pasar el valor de retorno de `Promise<T>` de cada actividad a la siguiente, es necesario que las actividades se ejecuten en secuencia, definiendo la topología lineal que se ha explicado anteriormente. En el caso AWS Flow Framework de Java, no es necesario utilizar ningún código de modelado especial para definir incluso topologías complejas, solo el control de flujo estándar de Java y. `Promise<T>` Para ver un ejemplo detallado de cómo implementar una topología en paralelo sencilla, consulte HelloWorldWorkflowParallel Actividades: Trabajador.

> **Note**
>
> Cuando un método de actividad como `say` no devuelve un valor, el método de cliente correspondiente devuelve un objeto `Promise<Void>`. El objeto no representa datos, pero al principio no está preparado y lo está cuando termina la actividad. Por lo tanto, se puede pasar un objeto `Promise<Void>` a otros métodos de cliente de actividades para asegurarse de que aplazan su ejecución hasta que termina la actividad original.

`Promise<T>` permite que en una implementación de flujo de trabajo se utilicen métodos de cliente de actividades y sus valores de retorno de una forma muy parecida a los métodos síncronos. Sin embargo, debe tener cuidado al obtener acceso al valor de un objeto `Promise<T>`. A diferencia del tipo Future<T> de Java, el marco de trabajo se encarga de la sincronización para `Promise<T>`, no la aplicación. Si llama a `Promise<T>.get` y el objeto no está preparado, `get` genera una excepción. Fíjese en que `HelloWorldWorkflow` nunca obtiene acceso directamente al objeto `Promise<T>`; simplemente pasa los objetos de una actividad a la siguiente. Cuando un objeto está preparado, el marco de trabajo extrae el valor y lo pasa al método de actividad como un tipo estándar.

Solo se debería obtener acceso a los objetos `Promise<T>` por medio de código asíncrono, donde el marco de trabajo garantiza que el objeto está preparado y representa un valor válido. Para solucionar este problema, `HelloWorldWorkflow` pasa objetos `Promise<T>` solo a métodos de cliente de actividades. Para obtener acceso al valor de un objeto `Promise<T>` en la implementación del flujo de trabajo, pase el objeto a un método de flujo de trabajo asíncrono, que se comporta de una forma muy parecida a una actividad. Para ver un ejemplo, consulta HelloWorldWorkflowAsync Solicitud.

HelloWorldWorkflow Implementación de flujos de trabajo y actividades

Las implementaciones del flujo de trabajo y las actividades tienen clases de trabajadores asociadas, ActivityWorkery WorkflowWorker. Estas clases se encargan de la comunicación entre Amazon SWF y las implementaciones de actividades y flujos de trabajo al sondear la lista de tareas de Amazon SWF apropiada para buscar tareas, a la vez que ejecutan el método adecuado para cada tarea y administran el flujo de datos. Para obtener más información, consulte AWS Flow Framework Conceptos básicos: estructura de la aplicación

Para asociar las implementaciones de actividades y flujos de trabajo con los objetos de proceso de trabajo correspondientes, tiene que implementar una o varias aplicaciones de proceso de trabajo que:

- Registren flujos de trabajo o actividades en Amazon SWF.
- Creen objetos de procesos de trabajo y los asocien con las implementaciones del proceso de trabajo del flujo de trabajo o la actividad.
- Indiquen a los objetos del proceso de trabajo que comiencen a comunicarse con Amazon SWF.

Si desea ejecutar el flujo de trabajo y las actividades como procesos independientes, debe implementar distintos hosts de procesos de trabajo de flujos de trabajo y actividades. Para ver un ejemplo, consulta HelloWorldWorkflowDistributed Solicitud. Para simplificar, HelloWorldWorkflow

implementa un único servidor de trabajo que ejecuta las actividades y los trabajadores del flujo de trabajo en el mismo proceso, de la siguiente manera:

```
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`no tiene HelloWorld contrapartida, por lo que debe añadir una clase Java denominada `GreeterWorker` al proyecto y copiar el código de ejemplo en ese archivo.

El primer paso consiste en crear y configurar un AmazonSimpleWorkflowClientobjeto que invoque los métodos de servicio de Amazon SWF subyacentes. Para ello, `GreeterWorker`:

1. Crea un ClientConfigurationobjeto y especifica un tiempo de espera del socket de 70 segundos. Este valor especifica cuánto hay que esperar a que se transfieran los datos en una conexión abierta establecida antes de cerrar el conector.
2. Crea un AWSCredentials objeto básico para identificar la AWS cuenta y pasa las claves de la cuenta al constructor. Por comodidad, y para evitar que queden expuestas en texto sin formato en el código, las claves se almacenan en variables de entorno.
3. Crea un AmazonSimpleWorkflowClientobjeto para representar el flujo de trabajo y pasa los `ClientConfiguration` objetos `BasicAWSCredentials` y al constructor.
4. Establece la dirección URL del punto de conexión del servicio del objeto del cliente. Amazon SWF está disponible actualmente en todas las AWS regiones.

Por comodidad, `GreeterWorker` define dos constantes de cadena.

- `domain`es el nombre de dominio Amazon SWF del flujo de trabajo, que creó al configurar su cuenta de Amazon SWF. `HelloWorldWorkflow`asume que está ejecutando el flujo de trabajo en el dominio helloWorldWalkthrough "».
- `taskListToPoll` es el nombre de las listas de tareas que utiliza Amazon SWF para administrar la comunicación entre los procesos de trabajo de flujos de trabajo y de actividades. Puede establecer el nombre en cualquier cadena que le resulte conveniente. HelloWorldWorkflow usa "HelloWorldList" tanto para las listas de tareas del flujo de trabajo como para las de actividades. Internamente, los nombres terminan en diferentes espacios de nombres, por lo que las dos listas de tareas están diferenciadas.

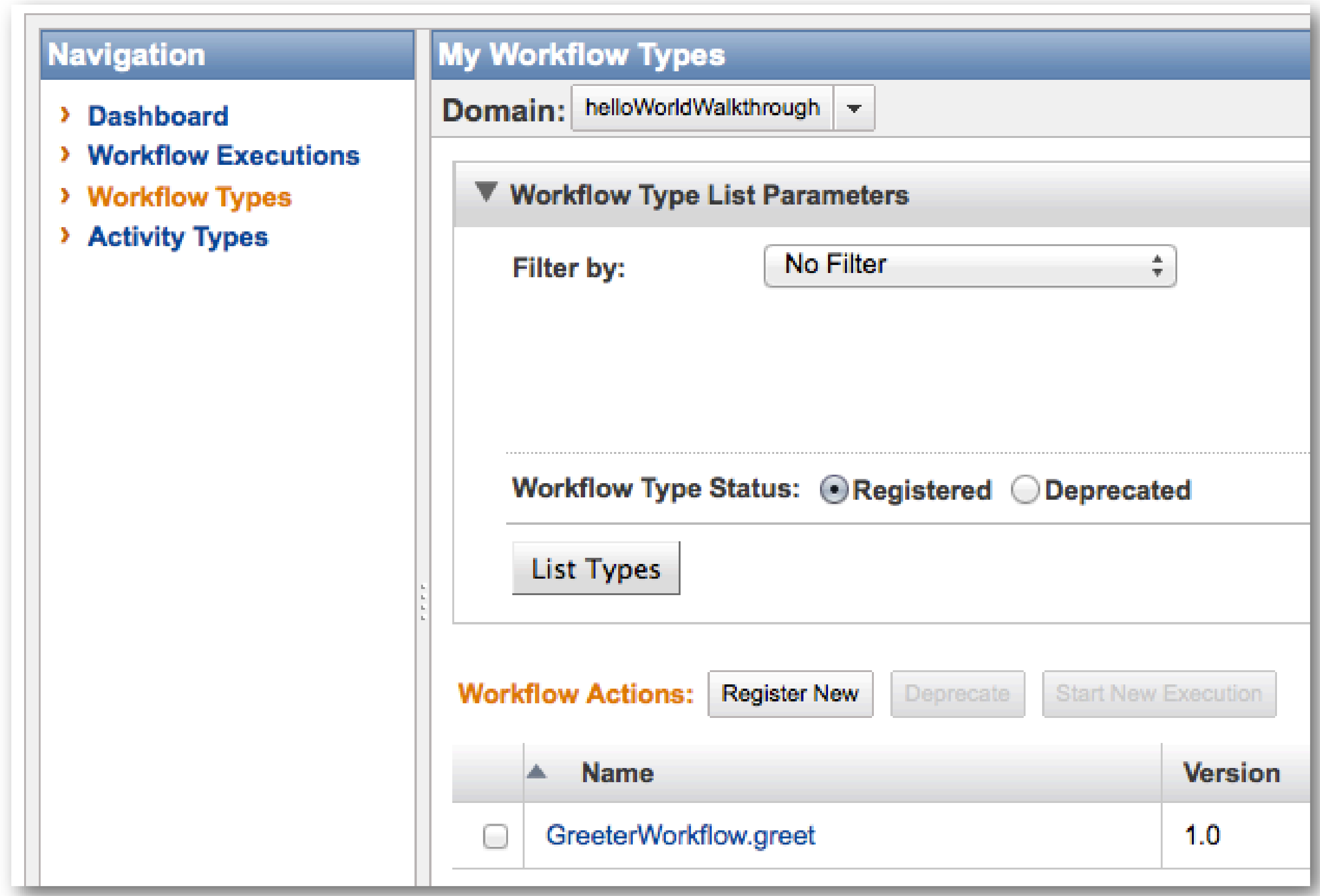
`GreeterWorker`utiliza las constantes de cadena y el AmazonSimpleWorkflowClientobjeto para crear objetos de trabajo, que gestionan la interacción entre las actividades y las implementaciones de los trabajadores y Amazon SWF. En particular, los objetos del proceso de trabajo se encargan de la tarea de sondeo de la lista de tareas correspondiente para buscar tareas.

`GreeterWorker` crea un objeto `ActivityWorker` y lo configura para que se encargue de `GreeterActivitiesImpl` añadiendo una nueva instancia de clase. Luego, `GreeterWorker` llama al método `start` del objeto `ActivityWorker`, que le indica al objeto que comience a sondear la lista de tareas de actividad especificada.

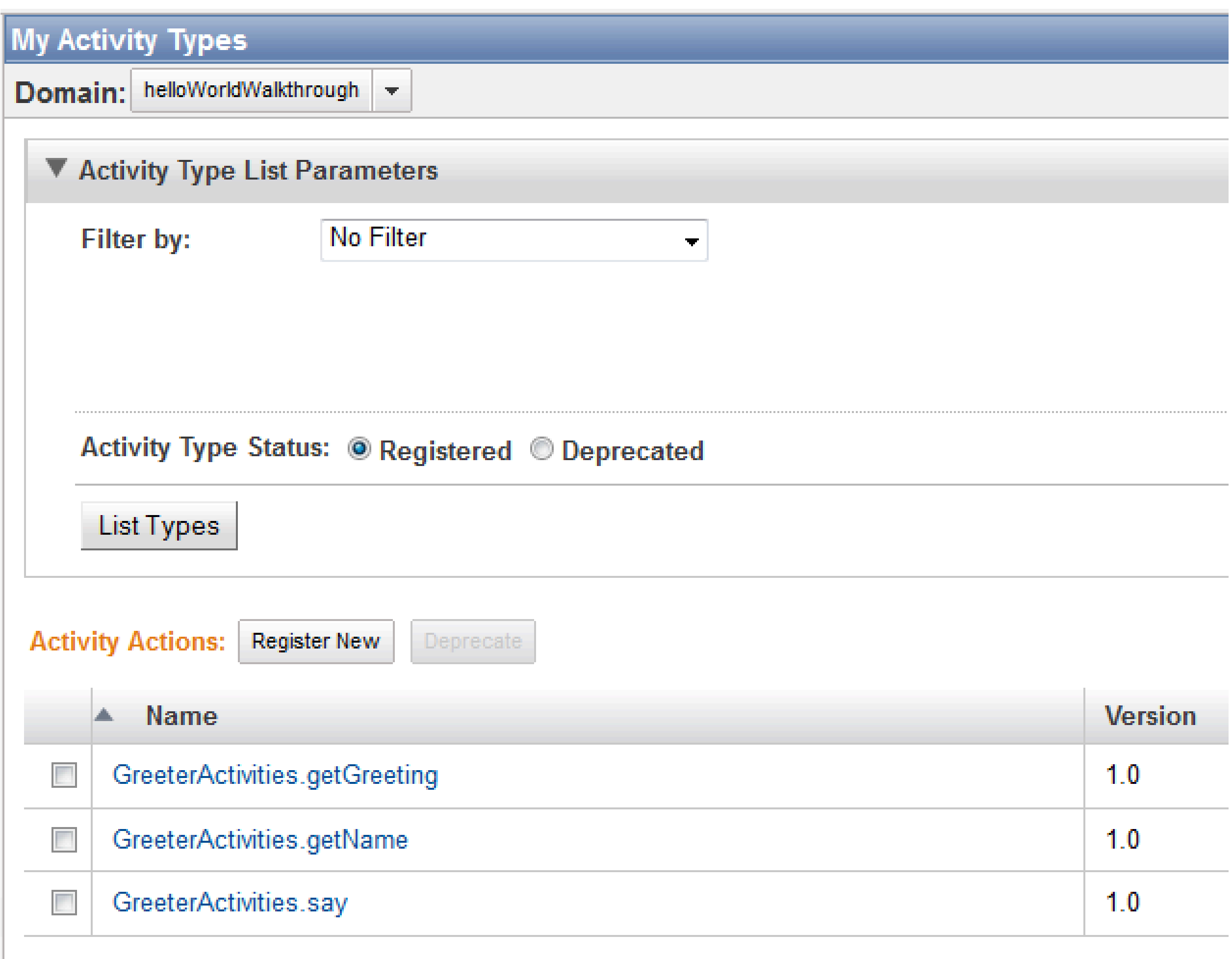
`GreeterWorker` crea un objeto `WorkflowWorker` y lo configura para que se encargue de `GreeterWorkflowImpl` añadiendo el nombre de archivo de clase,

`GreeterWorkflowImpl.class`. Luego, llama al método `start` del objeto `WorkflowWorker`, que le indica al objeto que comience a sondear la lista de tareas del flujo de trabajo especificada.

En este momento, puede ejecutar `GreeterWorker` correctamente. Registra el flujo de trabajo y las actividades con Amazon SWF y hace que los objetos del proceso de trabajo comiencen a sondear sus correspondientes listas de tareas. Para verificar esto, ejecute `GreeterWorker`, vaya a la consola de Amazon SWF y seleccione `helloWorldWalkthrough` en la lista de dominios. Si elige Workflow Types (Tipos de flujo de trabajo) en el panel Navigation (Navegación), debería ver `GreeterWorkflow.greet`:



Si elige Activity Types (Tipos de actividad), se muestran los métodos `GreeterActivities`:



Sin embargo, si elige Workflow Executions (Ejecuciones de flujo de trabajo), verá las ejecuciones activas. Aunque los procesos de trabajo de flujos de trabajo y actividades sondean tareas, aún no hemos iniciado una ejecución de flujo de trabajo.

HelloWorldWorkflow Motor de arranque

La última pieza del puzle es la implementación de un iniciador de flujo de trabajo, que es una aplicación que inicia la ejecución del flujo de trabajo. Amazon SWF almacena el estado de ejecución para que pueda ver su historial y su estado de ejecución. HelloWorldWorkflow implementa un iniciador de flujo de trabajo modificando la `GreeterMain` clase de la siguiente manera:

```
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` crea un objeto `AmazonSimpleWorkflowClient` utilizando el mismo código que `GreeterWorker`. Luego, crea un objeto `GreeterWorkflowClientExternal`, que sirve de proxy para el flujo de trabajo, de una forma muy parecida al cliente de actividades creado en `GreeterWorkflowClientImpl`, que sirve de proxy para los métodos de actividades. En lugar de crear un objeto de cliente de flujo de trabajo mediante `new`, debe:

1. Crear un objeto de fábrica de clientes externo y pasar el objeto `AmazonSimpleWorkflowClient` y el nombre de dominio de Amazon SWF al constructor. El objeto de fábrica del cliente lo crea el procesador de anotaciones del marco, que crea el nombre del objeto simplemente añadiendo «ClientExternalFactoryImpl» al nombre de la interfaz de flujo de trabajo.
2. Cree un objeto cliente externo llamando al `getClient` método del objeto de fábrica, que crea el nombre del objeto añadiendo "ClientExternal" al nombre de la interfaz de flujo de trabajo. Opcionalmente, puede pasar a `getClient` una cadena que utilizará Amazon SWF para identificar esta instancia del flujo de trabajo. De lo contrario, Amazon SWF utiliza un GUID generado para representar una instancia de flujo de trabajo.

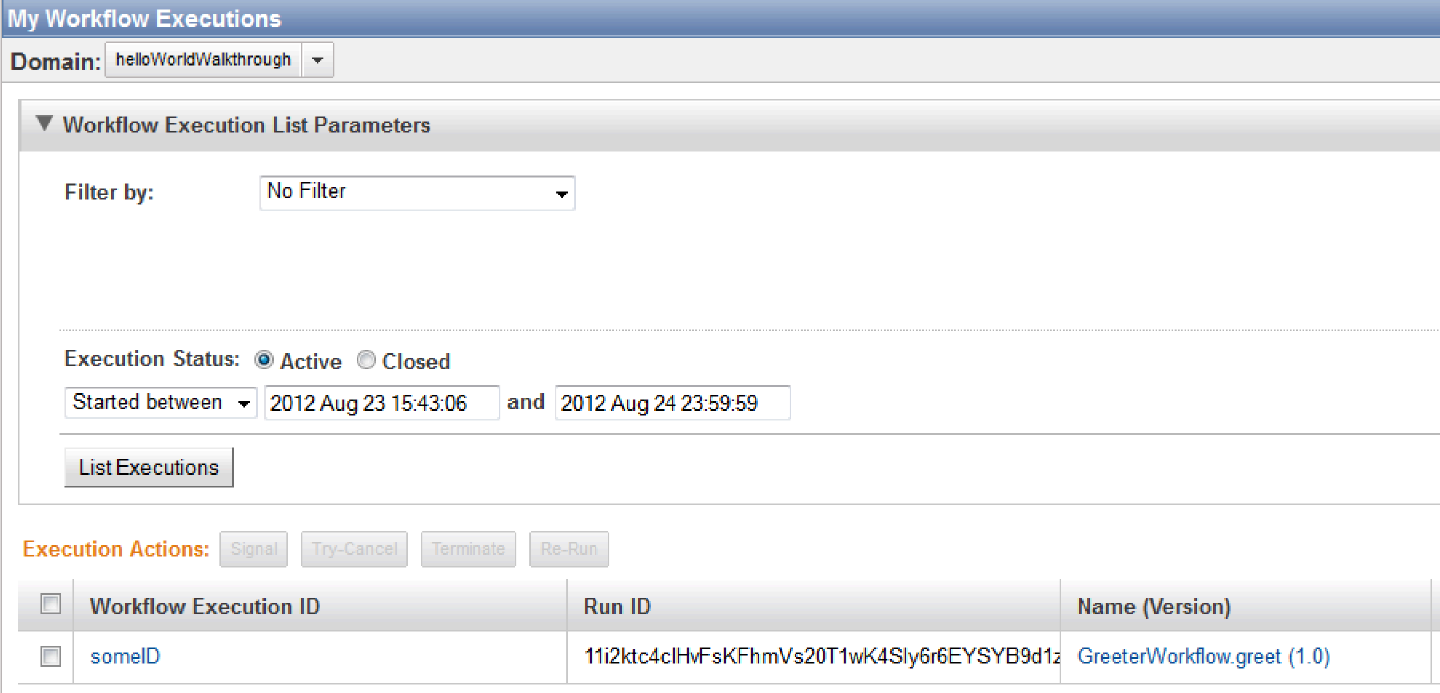
El cliente que se devuelve de la fábrica solo crea flujos de trabajo con el nombre de la cadena que se haya pasado al método getClient (el cliente que se ha devuelto de la fábrica ya tiene un estado en Amazon SWF). Para ejecutar un flujo de trabajo con un identificador diferente, hay que volver a la fábrica y crear un cliente nuevo especificando el identificador diferente.

El cliente de flujo de trabajo expone un método `greet` al que llama `GreeterMain` para comenzar el flujo de trabajo, ya que `greet()` es el método que se ha especificado con la anotación `@Execute`.

Note

El procesador de anotaciones también crea un objeto de fábrica de clientes interno que se utiliza para crear flujos de trabajo secundarios. Para obtener más información, consulte Ejecuciones de flujo de trabajo secundario.

Cierre `GreeterWorker` por ahora, si aún se está ejecutando, y ejecute `GreeterMain`. Ahora debería ver someID en la lista de ejecuciones de flujos de trabajo activas de la consola de Amazon SWF:



Si elige `someID` y luego la pestaña Events (Eventos), se muestran los eventos:

Workflow Execution: someID

Domain: helloWorldWalkthrough

Summary | **Events** | Activities

Event Date	ID	Event Type
Fri Aug 24 15:50:30 GMT-700 2012	2	DecisionTaskScheduled
Fri Aug 24 15:50:30 GMT-700 2012	1	WorkflowExecutionStarted

> **Note**
>
> Si ha iniciado `GreeterWorker` anteriormente y sigue ejecutándose, verá una lista de eventos más larga por los motivos que vamos a explicar en breve. Detenga `GreeterWorker` e intente ejecutar de nuevo `GreaterMain`.

En la pestaña Events (Eventos) solo aparecen dos eventos:

- `WorkflowExecutionStarted` indica que el flujo de trabajo ha comenzado a ejecutarse.
- `DecisionTaskScheduled` indica que Amazon SWF ha puesto en cola la primera tarea de decisión.

El motivo de que el flujo de trabajo se bloquee en la primera tarea de decisión es porque está distribuido en dos aplicaciones: `GreeterMain` y `GreeterWorker`. `GreeterMain` ha comenzado la ejecución del flujo de trabajo, pero `GreeterWorker` no se está ejecutando, por lo que los procesos de trabajo no sondean las listas ni ejecutan tareas. Puede ejecutar cualquiera de las aplicaciones de forma independiente, pero necesita ambas para que la ejecución del flujo de trabajo continúe después de la primera tarea de decisión. Si ahora ejecuta `GreeterWorker`, los procesos de trabajo de flujo de trabajo y actividad comenzarán a realizar el sondeo y las diversas tareas se realizarán rápidamente. Si comprueba ahora la pestaña `Events` (Eventos), verá el primer lote de eventos.

Workflow Execution: someID

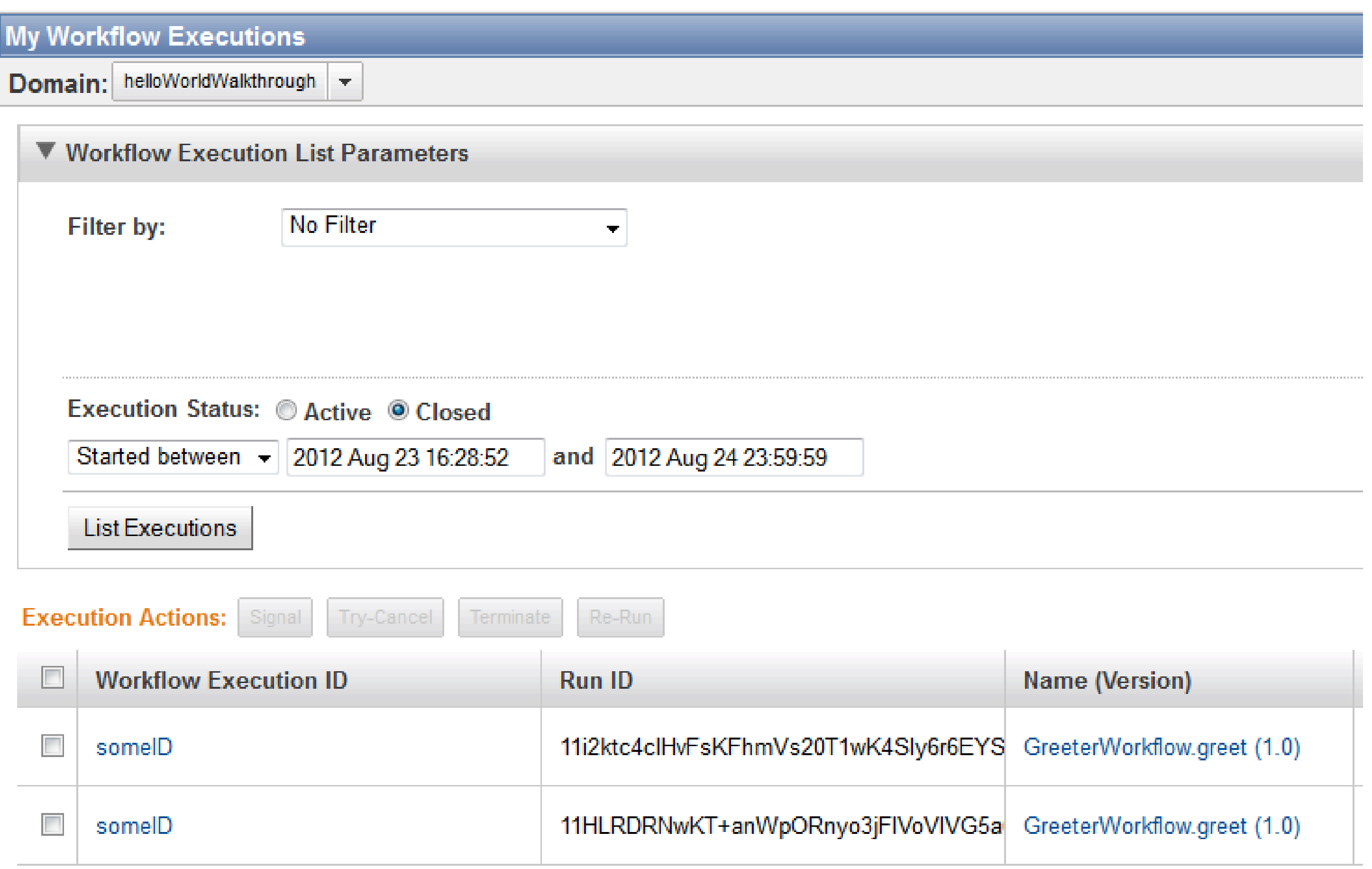
Domain: helloWorldWalkthrough

Summary | Events | Activities

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

Puede elegir eventos individuales para obtener más información. Cuando haya terminado de consultarlos, el flujo de trabajo debería haber imprimido las palabras “Hello World!” en la consola.

Una vez completado el flujo de trabajo, desaparece de la lista de ejecuciones activas. Sin embargo, si desea revisarlo, elija el botón de estado de ejecución Closed (Cerrado) y luego elija List Executions (Lista de ejecuciones). Aquí se muestran todas las instancias de flujo de trabajo completadas en el dominio especificado (`helloWorldWalkthrough`) que no han superado su tiempo de retención, que especificó en el momento de crear el dominio.



Fíjese en que cada instancia de flujo de trabajo tiene un valor de Run ID (Identificador de ejecución) único. Puede usar el mismo ID de flujo de trabajo para distintas instancias de flujo de trabajo, pero solo para una ejecución activa a la vez.

HelloWorldWorkflowAsync Solicitud

En ocasiones es preferible que un flujo de trabajo realice determinadas tareas localmente en lugar de usar una actividad. No obstante, las tareas de flujo de trabajo con frecuencia conllevan procesar valores representados por objetos `Promise<T>`. Si pasa un objeto `Promise<T>` a un método de flujo de trabajo asíncrono, el método se ejecuta inmediatamente pero no puede obtener acceso al valor del objeto `Promise<T>` hasta que el objeto esté listo. Podría sondear `Promise<T>.isReady` hasta que devuelva `true`, pero eso, además de no ser eficaz, haría que el método pudiera quedar bloqueado durante un largo periodo. Es mejor utilizar un método asíncrono.

Un método asíncrono se implementa de forma muy parecida a un método estándar, con frecuencia como miembro de la clase de implementación de flujo de trabajo, y se ejecuta en el contexto de la implementación de flujo de trabajo. Usted lo designa como método asíncrono aplicando una

anotación `@Asynchronous`, la cual indica al marco de trabajo que debe tratarlo de forma muy parecida a una actividad.

- Cuando una implementación de flujo de trabajo llama a un método asíncrono, se devuelve de inmediato. Los métodos asíncronos habitualmente devuelven un objeto `Promise<T>`, el cual queda listo una vez que el método termina.
- Si pasa uno o más objetos `Promise<T>` a un método asíncrono, difiere la ejecución hasta que todos los objetos de entrada estén listos. Un método asíncrono puede por tanto obtener acceso a su valores `Promise<T>` de entrada sin arriesgar a que se produzca una excepción.

> **Note**
>
> Debido a la forma en que Java ejecuta el flujo AWS Flow Framework de trabajo, los métodos asíncronos suelen ejecutarse varias veces, por lo que solo debe usarlos para tareas rápidas y de bajo coste. Debería utilizar actividades para realizar tareas prolongadas, como grandes cálculos. Para obtener más información, consulte AWS Flow Framework Conceptos básicos: ejecución distribuida.

Este tema es un recorrido por una versión modificada HelloWorldWorkflow que reemplaza una de las actividades por un método asíncrono. HelloWorldWorkflowAsync Para implementar la aplicación, cree una copia de HelloWorld. HelloWorldWorkflow empaquete en el directorio de su proyecto y asígnele el nombre HelloWorld. HelloWorldWorkflowAsync.

> **Note**
>
> En este tema se basa en los conceptos y los archivos que se presentan en los temas HelloWorld Solicitud y HelloWorldWorkflow Solicitud. Familiarizarse con los archivos y conceptos que se presentan en estos temas antes de continuar.

En las siguientes secciones se describe cómo modificar el HelloWorldWorkflow código original para utilizar un método asíncrono.

HelloWorldWorkflowAsync Implementación de actividades

HelloWorldWorkflowAsync implementa su interfaz de trabajo de actividades de `GreeterActivities` la siguiente manera:

```
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);
}
```

Esta interfaz es similar a la utilizada por HelloWorldWorkflow, con las siguientes excepciones:

- Omite la actividad `getGreeting`; esa tarea la realiza ahora un método asíncrono.
- El número de versión se establece en 2.0. Una vez que haya registrado una interfaz de actividades con Amazon SWF, no podrá modificarla a menos que cambie el número de versión.

El resto de las implementaciones del método de actividad son idénticas a HelloWorldWorkflow. Simplemente elimine `getGreeting` de `GreeterActivitiesImpl`.

HelloWorldWorkflowAsync Aplicación del flujo de trabajo

HelloWorldWorkflowAsync define la interfaz de flujo de trabajo de la siguiente manera:

```
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();
}
```

La interfaz es idéntica a, HelloWorldWorkflow excepto por un nuevo número de versión. Al igual que ocurre con las actividades, si desea cambiar un flujo de trabajo registrado, tiene que cambiar su versión.

HelloWorldWorkflowAsync implementa el flujo de trabajo de la siguiente manera:

```
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 reemplaza la `getGreeting` actividad por un método `getGreeting` asíncrono, pero el `greet` método funciona prácticamente de la misma manera:

1. Ejecute la actividad `getName`, lo cual devuelve inmediatamente un objeto `Promise<String>`, `name`, que representa el nombre.
2. Llame al método asíncrono `getGreeting` y pásele el objeto `name`. `getGreeting` devuelve inmediatamente un objeto `Promise<String>`, `greeting`, que representa el saludo.
3. Ejecute la actividad `say` y pásele el objeto `greeting`.
4. Una vez que se completa `getName`, `name` pasa a estar listo y `getGreeting` utiliza su valor para construir el saludo.
5. Una vez que `getGreeting` finaliza, `greeting` está listo y `say` imprime la cadena en la consola.

La diferencia está en que, en lugar de llamar al cliente de actividades para ejecutar una actividad `getGreeting`, el saludo llama al método asíncrono `getGreeting`. El resultado final es el mismo, pero el método `getGreeting` funciona de manera algo diferente a la actividad `getGreeting`.

- El proceso de trabajo de flujo de trabajo utiliza la semántica de llamada de función estándar para ejecutar `getGreeting`. No obstante, Amazon SWF actúa como intermediario en la ejecución asíncrona de la actividad.
- `getGreeting` ejecuta el proceso de implementación del flujo de trabajo.
- `getGreeting` devuelve un objeto `Promise<String>` en lugar de un objeto `String`. Para obtener el valor de la cadena contenido en `Promise`, usted tendrá que llamar a su método `get()`. Sin embargo, dado que la actividad se ejecuta de forma asíncrona, es posible que su valor devuelto no esté listo inmediatamente; se `get()` generará una excepción hasta que el valor devuelto por el método asincrónico esté disponible.

  Para obtener más información sobre cómo funciona `Promise`, consulte AWS Flow Framework Conceptos básicos: Data Exchange entre actividades y flujos de trabajo.

`getGreeting` crea un valor de retorno pasando la cadena del saludo al método estático `Promise.asPromise`. Este método crea un objeto `Promise<T>` del tipo apropiado, establece el valor y lo pone en el estado listo.

HelloWorldWorkflowAsync Organizador e iniciador del flujo de trabajo y las actividades

HelloWorldWorkflowAsync implementa `GreeterWorker` como clase anfitriona para las implementaciones de flujos de trabajo y actividades. Es idéntica a la HelloWorldWorkflow implementación excepto por el `taskListToPoll` nombre, que se establece en "`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 implementa el flujo de trabajo inicial en`GreeterMain`; es idéntico a la HelloWorldWorkflow implementación.

Para ejecutar el flujo de trabajo, ejecute `GreeterWorker` y`GreeterMain`, igual que con HelloWorldWorkflow.

HelloWorldWorkflowDistributed Solicitud

Con HelloWorldWorkflow y HelloWorldWorkflowAsync, Amazon SWF media la interacción entre el flujo de trabajo y las implementaciones de las actividades, pero se ejecutan localmente como un solo proceso. `GreeterMain`se encuentra en un proceso independiente, pero sigue ejecutándose en el mismo sistema.

Una característica clave de Amazon SWF es que admite aplicaciones distribuidas. Por ejemplo, puedes ejecutar el Workflow Worker en una EC2 instancia de Amazon, el iniciador del flujo de trabajo

en un ordenador de un centro de datos y las actividades en un ordenador de escritorio cliente. Es posible incluso ejecutar diferentes actividades en diferentes sistemas.

La HelloWorldWorkflowDistributed aplicación se extiende HelloWorldWorkflowAsync para distribuirla en dos sistemas y tres procesos.

- El flujo de trabajo y el iniciador del flujo de trabajo se ejecutan como procesos independientes en un sistema.
- Las actividades se ejecutan en un sistema independiente.

Para implementar la aplicación, cree una copia de HelloWorld. HelloWorldWorkflowAsync empaquete en el directorio de su proyecto y asígnele el nombre HelloWorld. HelloWorldWorkflowDistributed. En las siguientes secciones se describe cómo modificar el HelloWorldWorkflowAsync código original para distribuir la aplicación en dos sistemas y tres procesos.

No es necesario que cambie el flujo de trabajo o las implementaciones de actividades para ejecutarlos en sistemas independientes, ni tan siquiera los números de versión. Tampoco es necesario que modifique `GreeterMain`. Solo tiene que cambiar las actividades y el host del flujo de trabajo.

Con HelloWorldWorkflowAsync, una sola aplicación sirve como anfitrión del flujo de trabajo y de la actividad. Para ejecutar las implementaciones de flujo de trabajo y actividad en sistemas independientes, tiene que implementar aplicaciones independientes. Elimine GreeterWorker del proyecto y añada dos nuevos archivos de clases, GreeterWorkflowWorker y GreeterActivitiesWorker.

HelloWorldWorkflowDistributed implementa sus actividades alojadas en GreeterActivitiesWorker, de la siguiente manera:

```
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 implementa su host de flujo de trabajo `GreeterWorkflowWorker` de la siguiente manera:

```
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();
    }
}
```

Tenga en cuenta que `GreeterActivitiesWorker` es simplemente `GreeterWorker` sin el código `WorkflowWorker` y `GreeterWorkflowWorker` es simplemente `GreeterWorker` sin el código `ActivityWorker`.

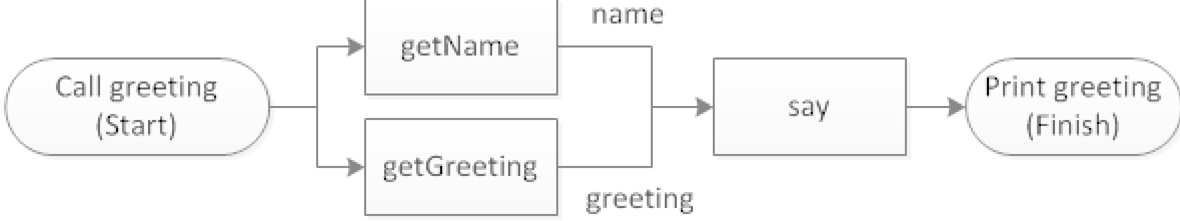
Para ejecutar el flujo de trabajo:

1. Cree un archivo JAR ejecutable con `GreeterActivitiesWorker` como punto de entrada.
2. Copie el archivo JAR del Paso 1 en otro sistema, que puede estar ejecutando cualquier sistema operativo que admita Java.
3. Asegúrese de que AWS las credenciales con acceso al mismo dominio de Amazon SWF estén disponibles en el otro sistema.
4. Ejecute el archivo JAR.
5. Utilice Eclipse para ejecutar `GreeterWorkflowWorker` y `GreeterMain` en su sistema de desarrollo.

Además del hecho de que las actividades se ejecutan en un sistema diferente al de Workflow Worker y al que inician el flujo de trabajo, el flujo de trabajo funciona exactamente de la misma manera que HelloWorldAsync. Sin embargo, como la llamada `println` que imprime “Hello World!” en la consola se encuentra en la actividad `say`, el resultado aparecerá en el sistema que ejecuta el proceso de trabajo de las actividades.

HelloWorldWorkflowParallel Solicitud

Las versiones anteriores de Hello World! todas utilizar una topología de flujo de trabajo lineal. No obstante, Amazon SWF no se limita a las topologías lineales. La HelloWorldWorkflowParallel aplicación es una versión modificada HelloWorldWorkflow que utiliza una topología paralela, como se muestra en la siguiente figura.



Con HelloWorldWorkflowParallel, `getName` y `getGreeting` corre en paralelo y cada uno devuelve parte del saludo. `say`luego fusiona las dos cadenas en un saludo y lo imprime en la consola.

Para implementar la aplicación, cree una copia de HelloWorld. HelloWorldWorkflow empaquete en el directorio de su proyecto y asígnele el nombre HelloWorld. HelloWorldWorkflowParallel. En las siguientes secciones se describe cómo modificar el HelloWorldWorkflow código original para que se ejecute `getName` y `getGreeting` en paralelo.

HelloWorldWorkflowParallel Actividades: Trabajador

La interfaz de HelloWorldWorkflowParallel actividades está implementada en`GreeterActivities`, como se muestra en el siguiente ejemplo.

```
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);
}
```

La interfaz es similar a HelloWorldWorkflow, con las siguientes excepciones:

- `getGreeting` no toma ninguna entrada; simplemente devuelve una cadena de saludos.
- `say` toma dos cadenas de entrada, el saludo y el nombre.
- La interfaz tiene un número de versión nuevo, que se necesita siempre que cambia una interfaz registrada.

HelloWorldWorkflowParallel implementa las actividades de `GreeterActivitiesImpl` la siguiente manera:

```
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` y `getGreeting` ahora simplemente devuelven la mitad de la cadena del saludo. `say` concatena las dos piezas para producir la frase completa y la imprime en la consola.

HelloWorldWorkflowParallel Workflow Worker

La interfaz HelloWorldWorkflowParallel de flujo de trabajo se implementa `GreeterWorkflow` de la siguiente manera:

```
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();
}
```

La clase es idéntica a la HelloWorldWorkflow versión, excepto que el número de versión se ha cambiado para que coincida con la actividad del trabajador.

El flujo de trabajo se implementa en `GreeterWorkflowImpl` de la siguiente manera:

```
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);
   }
}
```

A simple vista, esta implementación es muy similar a HelloWorldWorkflow las tres actividades que los métodos del cliente ejecutan en secuencia. No obstante, no es así para las actividades.

- HelloWorldWorkflow pasado `name` a`getGreeting`. Dado que `name` era un objeto `Promise<T>`, `getGreeting` aplazó la ejecución de la actividad hasta que se completó `getName`, por lo que las dos actividades se ejecutaron en secuencia.
- HelloWorldWorkflowParallel no pasa ninguna entrada `getName` o`getGreeting`. Ninguno de los métodos difiere la ejecución y los métodos de actividad asociados se ejecutan inmediatamente y en paralelo.

La actividad `say` toma `greeting` y `name` como parámetros de entrada. Dado que se trata de objetos `Promise<T>`, `say` difiere la ejecución hasta que se completan ambas actividades y, a continuación, construye e imprime el saludo.

Observe que HelloWorldWorkflowParallel no utiliza ningún código de modelado especial para definir la topología del flujo de trabajo. Lo hace de forma implícita mediante el uso del control de flujo estándar de Java y aprovechando las propiedades de `Promise<T>` los objetos. AWS Flow Framework ya que las aplicaciones de Java pueden implementar incluso topologías complejas simplemente utilizando `Promise<T>` objetos junto con las construcciones de flujo de control de Java convencionales.

HelloWorldWorkflowParallel Flujo de trabajo y actividades: anfitrión e iniciador

HelloWorldWorkflowParallel implementa `GreeterWorker` como clase anfitriona para las implementaciones de flujos de trabajo y actividades. Es idéntica a la HelloWorldWorkflow implementación excepto por el `taskListToPoll` nombre, que se establece en "HelloWorldParallelList».

`HelloWorldWorkflowParallel`implementa el flujo de trabajo inicial en `GreeterMain` y es idéntico a la HelloWorldWorkflow implementación.

Para ejecutar el flujo de trabajo, ejecute `GreeterWorker` y `GreeterMain`, de la misma manera que con `HelloWorldWorkflow`.

Comprensión AWS Flow Framework de Java

The AWS Flow Framework for Java funciona con Amazon SWF para facilitar la creación de aplicaciones escalables y tolerantes a errores para realizar tareas asíncronas que pueden ser de ejecución prolongada, remotas o ambas. Los ejemplos de "Hello World!" Los ejemplos en ¿Qué es AWS Flow Framework para Java? introdujeron los conceptos básicos de cómo utilizarlos para implementar aplicaciones básicas de flujo de trabajo AWS Flow Framework . En esta sección se proporciona información conceptual sobre el funcionamiento de AWS Flow Framework las aplicaciones. La primera sección resume la estructura básica de una AWS Flow Framework aplicación y las secciones restantes proporcionan más detalles sobre el funcionamiento de AWS Flow Framework las aplicaciones.

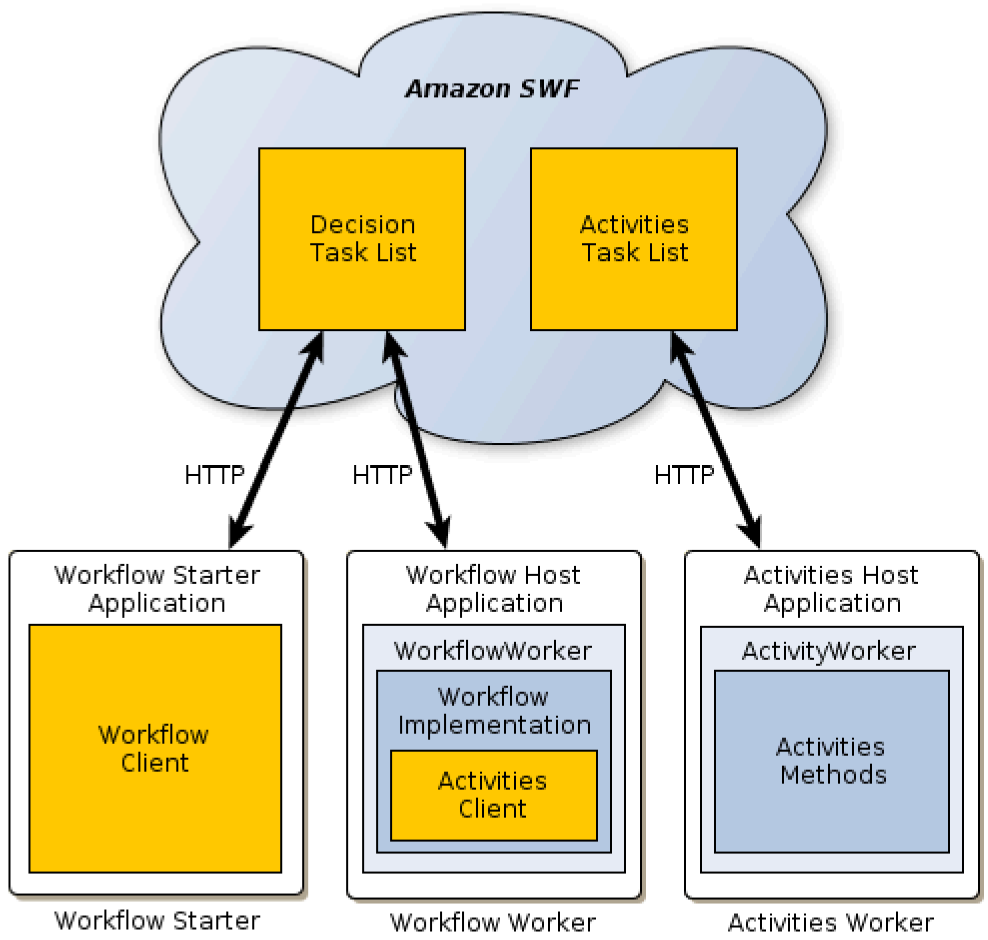
Temas

- AWS Flow Framework Conceptos básicos: estructura de la aplicación
- AWS Flow Framework Conceptos básicos: ejecución confiable
- AWS Flow Framework Conceptos básicos: ejecución distribuida
- AWS Flow Framework Conceptos básicos: listas de tareas y ejecución de tareas
- AWS Flow Framework Conceptos básicos: aplicaciones escalables
- AWS Flow Framework Conceptos básicos: Data Exchange entre actividades y flujos de trabajo
- AWS Flow Framework Conceptos básicos: Data Exchange entre aplicaciones y ejecuciones de flujos de trabajo
- Tipos de tiempo de espera de Amazon SWF

AWS Flow Framework Conceptos básicos: estructura de la aplicación

Conceptualmente, una AWS Flow Framework aplicación consta de tres componentes básicos: iniciadores del flujo de trabajo, trabajadores del flujo de trabajo y trabajadores activos. Una o más aplicaciones de host son responsables del registro de los procesos de trabajo (flujo de trabajo y actividad) en Amazon SWF, de comenzar los procesos de trabajo y de encargarse de la limpieza. Los procesos de trabajo se encargan de la mecánica de la ejecución del flujo de trabajo y pueden implementarse en varios hosts.

Este diagrama representa una aplicación básica AWS Flow Framework :



> **Note**
>
> La implementación de estos componentes en tres aplicaciones separadas es cómodo desde el punto de vista conceptual pero puede crear aplicaciones para implementar esta funcionalidad de diferentes formas. Por ejemplo, puede utilizar una aplicación de host individual para los procesos de trabajo de flujo de trabajo y de actividad, o utilizar hosts de flujo de trabajo y actividad separados. También puede tener múltiples procesos de trabajo de actividad, cada uno de ellos controla un conjunto diferente de actividades en hosts separados, etc.

Los tres AWS Flow Framework componentes interactúan indirectamente mediante el envío de solicitudes HTTP a Amazon SWF, que las gestiona. Amazon SWF hace lo siguiente:

- Mantiene una o más listas de tareas de decisiones que determinan el siguiente paso que tiene que realizar un proceso de trabajo de flujo de trabajo.
- Mantiene una o más listas de tareas de actividades que determinan qué tareas realizará un proceso de trabajo de actividad.
- Mantiene un step-by-step historial detallado de la ejecución del flujo de trabajo.

Con el AWS Flow Framework, el código de su aplicación no necesita tratar directamente muchos de los detalles que se muestran en la figura, como el envío de solicitudes HTTP a Amazon SWF. Basta con llamar a AWS Flow Framework los métodos y el marco se ocupará de los detalles entre bastidores.

Función del proceso de trabajo de actividad

Un proceso de trabajo de actividad realiza varias tareas que el flujo de trabajo debe llevar a cabo. Consisten en lo siguiente:

- La implementación de actividades, que incluye un conjunto de métodos de actividad que realizan tareas particulares para el flujo de trabajo.
- Un ActivityWorkerobjeto que utiliza solicitudes de sondeo HTTP largas para sondear Amazon SWF para determinar las tareas de actividad que se van a realizar. Cuando se necesita una tarea, Amazon SWF responde a la solicitud enviando la información necesaria para realizar la tarea. A continuación, el ActivityWorkerobjeto llama al método de actividad correspondiente y devuelve los resultados a Amazon SWF.

Función del proceso de trabajo de flujo de trabajo

El proceso de trabajo de flujo de trabajo organiza la ejecución de las diferentes actividades, administra el flujo de datos y administra las actividades que han producido un error. Consisten en lo siguiente:

- La implementación de flujo de trabajo, que incluye la lógica de la organización de la actividad, gestiona las actividades que han producido un error, etc.
- Un cliente de actividades, que actúa como proxy para el proceso de trabajo de actividad y permite que el proceso de trabajo de flujo de trabajo programe actividades que se ejecutarán de manera asíncrona.

- Un WorkflowWorkerobjeto que utiliza solicitudes de sondeo HTTP largas para sondear Amazon SWF con el fin de realizar tareas de toma de decisiones. Si hay tareas en la lista de tareas de flujo de trabajo, Amazon SWF responde a la solicitud devolviendo la información necesaria para realizar la tarea. A continuación, el marco de trabajo ejecuta el flujo de trabajo para realizar la tarea y devuelve los resultados a Amazon SWF.

Función del iniciador del flujo de trabajo

El iniciador de flujo de trabajo inicia una instancia de flujo de trabajo, conocida también como ejecución de flujo de trabajo y puede interactuar con una instancia durante la ejecución para pasar datos adicionales al proceso de trabajo de flujo de trabajo u obtener el actual estado de flujo de trabajo.

El iniciador del flujo de trabajo utiliza un cliente de flujo de trabajo para comenzar la ejecución del flujo de trabajo, interactúa con el flujo de trabajo según sea necesario durante la ejecución y se encarga de la limpieza. El iniciador del flujo de trabajo podría ser una aplicación ejecutada localmente, una aplicación web AWS CLI o incluso una. Consola de administración de AWS

Cómo interactúa Amazon SWF con la aplicación

Amazon SWF se encarga de la interacción entre los componentes del flujo de trabajo y mantiene un historial detallado del flujo de trabajo. Amazon SWF no inicia la comunicación con los componentes, sino que espera las solicitudes HTTP de los componentes y las gestiona según sea necesario. Por ejemplo:

- Si la solicitud viene de un proceso de trabajo, que sondea para detectar tareas disponibles, Amazon SWF responderá directamente al proceso de trabajo si hay alguna tarea disponible. Para obtener más información sobre cómo funciona el sondeo, consulte Sondeo de tareas en la guía para desarrolladores de Amazon Simple Workflow Service.
- Si la solicitud es una notificación de un proceso de trabajo de actividad indicando que una tarea está completa, Amazon SWF registra la información en el historial de ejecución y añade una tarea a la lista de tareas de decisión para informar al proceso de trabajo de flujo de trabajo que la tarea está completa, lo que permite continuar con el siguiente paso.
- Si la solicitud procede de un proceso de trabajo de flujo de trabajo para la ejecución de una actividad, Amazon SWF registra la información en el historial de ejecución y añade una tarea a la lista de tareas de actividad para ordenar a un proceso de trabajo de actividad que ejecute el método de actividad apropiado.

Este enfoque permite a los trabajadores trabajar en cualquier sistema con conexión a Internet, incluidas las EC2 instancias de Amazon, los centros de datos corporativos, los ordenadores de los clientes, etc. Ni siquiera tienen que ejecutar el mismo sistema operativo. Dado que las solicitudes de HTTP tienen su origen en los procesos de trabajo, no se necesitan puertos visibles externamente, los procesos de trabajo pueden ejecutarse detrás de un firewall.

Para obtener más información

Para obtener información más detallada sobre el funcionamiento de Amazon SWF, consulte la Guía para desarrolladores de Amazon Simple Workflow Service.

AWS Flow Framework Conceptos básicos: ejecución confiable

Las aplicaciones distribuidas asíncronas deben tratar problemas de fiabilidad a los que no se enfrentan las aplicaciones convencionales, entre los que se incluyen los siguientes:

- Cómo proporcionar comunicación de confianza entre componentes distribuidos de manera asíncrona como por ejemplo componentes que se ejecutan desde hace tiempo en sistemas remotos.
- Cómo asegurarse de que los resultados no se pierden si se produce un error en un componente o se desconecta, en especial en aplicaciones que se ejecutan desde hace tiempo.
- Cómo tratar componentes en los que se ha producido un error.

Las aplicaciones pueden confiar en Amazon SWF AWS Flow Framework y en Amazon para gestionar estos problemas. Exploraremos de qué manera proporciona Amazon SWF mecanismos para garantizar que los flujos de trabajo funcionan de manera fiable y previsible, incluso cuando son de ejecución prolongada y dependen de tareas asíncronas realizadas informáticamente y con interacción humana.

Proporcionar comunicación de confianza

AWS Flow Framework proporciona una comunicación fiable entre un trabajador del flujo de trabajo y sus trabajadores de actividades mediante Amazon SWF para enviar tareas a los trabajadores de actividades distribuidas y devolver los resultados al trabajador del flujo de trabajo. Amazon SWF utiliza los siguientes métodos para garantizar una comunicación fiable entre un proceso de trabajo y sus actividades:

- Amazon SWF almacena de forma duradera las tareas programadas de flujo de trabajo y de actividades, y garantiza que se realizarán como máximo una vez.
- Amazon SWF garantiza que una tarea de actividad se realizará correctamente y devolverá un resultado válido o notificará al proceso de trabajo de flujo de trabajo que se ha producido un error en la tarea.
- Amazon SWF almacena de manera duradera el resultado de cada actividad realizada o, en el caso de las actividades en las que se ha producido un error, almacena información relevante sobre el error.

AWS Flow Framework A continuación, utiliza los resultados de la actividad de Amazon SWF para determinar cómo proceder con la ejecución del flujo de trabajo.

Garantizar que no se pierden los resultados

Mantener el historial de flujo de trabajo

Una actividad que realiza una operación de minado de datos en un petabyte de datos podría tardar horas en completarse y una actividad que indica a un proceso de trabajo humano que realice una tarea compleja podría tardar días o incluso semanas en completarse.

Para adaptarse a escenarios como estos, los AWS Flow Framework flujos de trabajo y las actividades pueden tardar arbitrariamente en completarse: hasta un máximo de un año para la ejecución de un flujo de trabajo. La ejecución de manera fiable de procesos de ejecución prolongada exige un mecanismo para almacenar de manera duradera el historial de ejecución del flujo de trabajo de forma continua.

Para AWS Flow Framework ello, depende de Amazon SWF, que mantiene un historial de ejecución de cada instancia de flujo de trabajo. El historial de flujo de trabajo proporciona un registro completo y autorizado del progreso del flujo de trabajo, incluidas todas las tareas de flujo de trabajo y actividad programadas y completadas, y la información devuelta por las actividades completadas o en las que se ha producido un error.

AWS Flow Framework por lo general, las aplicaciones no necesitan interactuar directamente con el historial del flujo de trabajo, aunque pueden acceder a él si es necesario. En la mayoría de las ocasiones, las aplicaciones pueden simplemente dejar que el marco de trabajo interactúe con el historial de flujo de trabajo en segundo plano. Para obtener un análisis completo del historial del flujo de trabajo, consulte el historial del flujo de trabajo en la Guía para desarrolladores de Amazon Simple Workflow Service.

Ejecución sin estado

El historial de ejecución permite que los procesos de trabajo de flujo de trabajo no tengan estado. Si tiene varias instancias de un proceso de trabajo de flujo de trabajo o actividad, cualquier proceso de trabajo puede realizar cualquier tarea. El proceso de trabajo recibe de Amazon SWF toda la información de estado que necesita para realizar la tarea.

Este enfoque hace que los flujos de trabajo sean de mayor confianza. Por ejemplo, si un proceso de trabajo de actividad falla, no tiene que reiniciar el flujo de trabajo. Simplemente reinicie el proceso de trabajo y comenzará a sondear la lista de tareas y a procesar las tareas de la lista, independientemente de cuándo se produjo el error. Puede hacer que el flujo de trabajo general tolere los errores utilizando dos o más procesos de trabajo de flujo de trabajo y de actividad, quizás en diferentes sistemas. Entonces, si uno de los procesos de trabajo falla, los otros seguirán realizando las tareas programadas sin ninguna interrupción en el progreso del flujo de trabajo.

Tratamiento de componentes distribuidos en los que se ha producido un error

Con frecuencia se producen errores en las actividades debido a razones efímeras, como una breve desconexión, por lo que una estrategia habitual para abordar actividades en las que se ha producido un error es volver a intentar realizar la actividad. En lugar de abordar el proceso de reintento implementando estrategias complejas de transferencia de mensajes, las aplicaciones pueden depender del AWS Flow Framework. Ofrece varios mecanismos para volver a intentar realizar actividades en las que se ha producido un error y proporciona un mecanismo incorporado para la gestión de excepciones que funciona con la ejecución asíncrona y distribuida de tareas en un flujo de trabajo.

AWS Flow Framework Conceptos básicos: ejecución distribuida

Una instancia de flujo de trabajo es básicamente un subproceso virtual de ejecución que puede abarcar las actividades y la lógica de orquestación que se ejecuta en varios equipos remotos. Amazon SWF y su AWS Flow Framework función como sistema operativo que gestiona las instancias de flujo de trabajo en una CPU virtual mediante:

- Mantienen el estado de ejecución de cada instancia.
- Cambian de una instancia a otra.
- Reanudan la ejecución de una instancia en el punto en el que se desactivó.

Reproducción de flujos de trabajo

Dado que las actividades pueden ser de larga duración, no es deseable que el flujo de trabajo simplemente se bloquee hasta completarse. En su lugar, AWS Flow Framework gestiona la ejecución del flujo de trabajo mediante un mecanismo de reproducción, que se basa en el historial del flujo de trabajo mantenido por Amazon SWF para ejecutar el flujo de trabajo en episodios.

Cada episodio reproduce la lógica del flujo de trabajo de manera que ejecuta cada actividad solo una vez, y se asegura de que las actividades y los métodos asíncronos no se ejecutan hasta que sus objetos Promise estén listos.

El iniciador de flujo de trabajo inicia el primer episodio de reproducción cuando comienza la ejecución del flujo de trabajo. El marco de trabajo llama al método de punto de entrada del flujo de trabajo y:

1. Ejecuta todas las tareas de flujo de trabajo que no dependen de la finalización de la actividad, lo que incluye llamar a todos los métodos del cliente de la actividad.
2. Proporciona a Amazon SWF una lista de las tareas de actividad que deben programarse para la ejecución. Para el primer episodio, esta lista consta tan solo de aquellas actividades que no dependen de una Promise y que pueden ejecutarse inmediatamente.
3. Notifica a Amazon SWF que el episodio se ha completado.

Amazon SWF almacena las tareas de la actividad en el historial de flujo de trabajo y las programa para su ejecución colocándolas en la lista de tareas de actividad. Los procesos de trabajo de actividad sondean la lista de tareas y ejecutan las tareas.

Cuando un proceso de trabajo de actividad completa una tarea, devuelve el resultado a Amazon SWF, que lo registra en el historial de ejecución del flujo de trabajo y programa una nueva tarea de flujo de trabajo para el proceso de trabajo de flujo de trabajo colocándola en la lista de tareas de flujo de trabajo. El proceso de trabajo del flujo de trabajo sondea la lista de tareas y cuando recibe la tarea, ejecuta el siguiente episodio de reproducción de la siguiente manera:

1. El marco de trabajo ejecuta de nuevo el método de punto de entrada del flujo de trabajo y:
    - Ejecuta todas las tareas de flujo de trabajo que no dependen de la finalización de la actividad, lo que incluye llamar a todos los métodos del cliente de la actividad. No obstante, el marco de trabajo comprueba el historial de ejecución y no programa tareas de actividad duplicadas.
    - Comprueba el historial para ver qué tareas de actividad se han completado y ejecuta cualquier método de flujo de trabajo asíncrono que dependa de esas actividades.

2. Cuando todas las tareas de flujo de trabajo que pueden ejecutarse se ha completado, el marco de trabajo se lo notifica a Amazon SWF:
   - Proporciona a Amazon SWF una lista de las actividades cuyos objetos `Promise<T>` de entrada están listos desde el último episodio y pueden programarse para la ejecución.
   - Si el episodio no generó ninguna tarea de actividad adicional pero sigue habiendo actividades sin completar, el marco de trabajo notifica a Amazon SWF que el episodio está completo. A continuación espera a que se complete otra actividad, iniciando el siguiente episodio de reproducción.
   - Si el episodio no generó ninguna tarea de actividad adicional y se han completado todas las actividades, el marco de trabajo notifica a Amazon SWF que la ejecución del flujo de trabajo está completa.

Para ver ejemplos de comportamiento de reproducción, consulte AWS Flow Framework para Java Replay Behavior.

Métodos de flujo de trabajo asíncronos y de reproducción

Los métodos de flujo de trabajo asíncrono se utilizan con frecuencia de manera parecida a actividades, porque el método difiere la ejecución hasta que todos los objetos `Promise<T>` de entrada están listos. No obstante, el mecanismo de reproducción gestiona los métodos asíncrono de manera diferente a las actividades.

- La reproducción no garantiza que un método asíncrono solo se ejecutará una vez. Difiere la ejecución en un método asíncrono hasta que sus objetos Promise de entrada están listo, pero a continuación ejecuta ese método para todos los episodios subsiguientes.
- Cuando un método asíncrono se completa, no comienza un episodio nuevo.

Se proporciona un ejemplo de reproducción de un flujo de trabajo asíncrono en AWS Flow Framework para Java Replay Behavior.

Reproducción e implementación de flujos de trabajo

La mayoría de las veces, no tiene que preocuparse de los detalles del mecanismo de reproducción. Se trata básicamente de algo que ocurre en segundo plano. No obstante, la reproducción tiene dos implicaciones importantes para la implementación de su flujo de trabajo.

- No utilice métodos de flujo de trabajo para realizar tareas de ejecución prolongada porque la reproducción repetirá esas tareas numerosas veces. Incluso los métodos de flujo de trabajo asíncronos se ejecutan habitualmente más de una vez. Utilice, en cambio, actividades para tareas de ejecución prolongada; la reproducción ejecuta actividades solo una vez.
- Su lógica de flujo de trabajo tiene que ser completamente determinista; cada episodio debe tomar la misma ruta de flujo de control. Por ejemplo, la ruta de flujo de control no debería depender de la hora actual. Para obtener una descripción detallada de los requisitos de reproducción y determinación, consulte No determinismo.

AWS Flow Framework Conceptos básicos: listas de tareas y ejecución de tareas

Para gestionar las tareas de flujo de trabajo y de actividad, Amazon SWF las publica en listas con nombres nominados. Amazon SWF mantiene al menos dos listas de tareas, una para los procesos de trabajo de los flujos de trabajo y otra para los procesos de trabajo de las actividades.

Note

Puede especificar tantas listas de tareas como necesite, con diferentes procesos de trabajo asignados a cada lista. No existe ningún límite en el número de listas de tareas. Típicamente especificará la lista de tareas de un proceso de trabajo en la aplicación host del proceso de trabajo al crear el objeto del proceso de trabajo.

El siguiente extracto de la aplicación host `HelloWorldWorkflow` crea un nuevo proceso de trabajo de actividad y lo asigna a la lista de tareas de actividades `HelloWorldList`.

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

```
}
```

De manera predeterminada, Amazon SWF programa las tareas del proceso de trabajo en la lista `HelloWorldList`. A continuación, el proceso de trabajo sondea la lista en busca de tareas. Puede asignar cualquier nombre a una lista de tareas. Incluso puede usar el mismo nombre para las listas de flujo de trabajo y de actividades. Internamente, Amazon SWF pone los nombres de las listas de tareas de flujo de trabajo y de actividad en diferentes espacios de nombres, por lo que las dos listas estarán diferenciadas.

Si no especificas una lista de tareas, AWS Flow Framework especifica una lista predeterminada cuando el trabajador registra el tipo en Amazon SWF. Para obtener más información, consulte Registro de tipos de flujos de trabajo y de actividades.

En ocasiones resulta útil que un proceso de trabajo específico o un grupo de procesos de trabajo realice determinadas tareas. Por ejemplo, un flujo de trabajo de procesamiento de imágenes podría utilizar una actividad para descargar una imagen y otra actividad para procesar la imagen. Resulta más eficaz realizar ambas tareas en el mismo sistema y evitar los costos de transferencia de grandes archivos a través de la red.

Para respaldar dichos escenarios, puede especificar explícitamente una lista de tareas al llamar a un método de cliente de actividad utilizando una sobrecarga que incluye un parámetro `schedulingOptions`. Para especificar la lista de tareas, pase al método un `ActivitySchedulingOptions` objeto configurado adecuadamente.

Por ejemplo, imagine que la actividad `say` de la aplicación `HelloWorldWorkflow` es hospedada por un proceso de trabajo de actividad distinto de `getName` y `getGreeting`. El siguiente ejemplo muestra cómo garantizar que `say` utiliza la misma lista de tareas que `getName` y `getGreeting`, incluso si se habían asignado originalmente a listas diferentes.

```
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);
  }
}
```

El método asíncrono `runSay` obtiene la lista de tareas `getGreeting` de su objeto de cliente. A continuación crea y configura un objeto `ActivitySchedulingOptions` que garantiza que `say` sondea la misma lista de tareas que `getGreeting`.

Note

Cuando pasa un parámetro `schedulingOptions` a un método de cliente de actividad, anula la lista de tareas original solo para la ejecución de esa actividad. Si llama de nuevo al método del cliente de actividades sin especificar una lista de tareas, Amazon SWF asignará la tarea a la lista original y el proceso de trabajo de actividad sondeará esa lista.

AWS Flow Framework Conceptos básicos: aplicaciones escalables

Amazon SWF cuenta con dos características clave que permiten escalar fácilmente una aplicación de flujo de trabajo para controlar la carga actual:

- Un historial de ejecución de flujo de trabajo completo que le permite implementar una aplicación sin estado.
- Programación de tareas que se acopla de manera flexible a la ejecución de tareas, lo que facilita escalar su aplicación para responder a las demandas actuales.

Para programar las tareas, Amazon SWF las publica en listas de tareas asignadas dinámicamente en lugar de comunicarse directamente con los procesos de trabajo de flujo de trabajo y de actividad. En su lugar, los procesos de trabajo utilizan solicitudes HTTP para sondear sus listas respectivas para obtener tareas. Este enfoque combina de manera flexible la programación de tareas con la ejecución de las tareas y permite a los trabajadores ejecutar en cualquier sistema adecuado, incluidas EC2 las instancias de Amazon, los centros de datos corporativos, las computadoras de los clientes, etc. Como las solicitudes HTTP se originan en los trabajadores, no se necesitan puertos visibles desde el exterior, lo que permite a los trabajadores trabajar incluso detrás de un firewall.

El mecanismo de sondeo largo que los procesos de trabajo utilizan para sondear para obtener tareas garantizan que los procesos de trabajo no se sobrecarguen. Incluso si se produce un pico en tareas programadas, los procesos de trabajo extraen tareas siguiendo su propio ritmo. No obstante, debido a que los procesos de trabajo no tienen estado, es posible escalar dinámicamente una aplicación para satisfacer el aumento de carga iniciando instancias de proceso de trabajo adicionales. Incluso si se están ejecutando en sistemas diferentes, cada instancia sondea la misma lista de tareas y la primera instancia de proceso de trabajo disponible ejecuta cada tarea, independientemente de dónde esté ubicado el proceso de trabajo o cuándo comenzó. Cuando la carga disminuye, es posible reducir el número de procesos de trabajo en consecuencia.

AWS Flow Framework Conceptos básicos: Data Exchange entre actividades y flujos de trabajo

Cuando se llama a un método del cliente de actividades asíncronas, este devuelve inmediatamente un objeto de Promesa (también denominado Futuro), que representa el valor de retorno del método de la actividad. Al principio, la promesa se encuentra en un estado no preparado y el valor de retorno no está definido. Cuando el método de la actividad realiza la tarea y se devuelve, el marco de trabajo serializa el valor de retorno a través de la red hasta el proceso de trabajo del flujo de trabajo, que asigna un valor a la promesa y pone el objeto en estado preparado.

Puede utilizar la promesa para administrar la ejecución del flujo de trabajo incluso si un método de actividad no tiene un valor de retorno. Si pasa una promesa devuelta a un método del cliente de actividad o un método de flujo de trabajo asíncrono, la ejecución se aplaza hasta que el objeto está preparado.

Si pasa una o más promesas a un método de cliente de actividad, el marco de trabajo pone en cola la tarea, pero aplaza su programación hasta que todos los objetos están preparados. Luego, extrae los datos de cada promesa y los serializa a través de Internet hasta el proceso de trabajo de actividad, que lo pasa al método de actividad como un tipo estándar.

Note

Si tiene que transferir grandes cantidades de datos entre los procesos de trabajo de flujos de trabajo y actividades, lo mejor es almacenar los datos en un lugar práctico y pasar solamente la información de recuperación. Por ejemplo, puede almacenar los datos en un bucket de Amazon S3 y pasar la URL asociada.

El Promise <T> Type

El tipo `Promise<T>` es similar en cierto modo al tipo Java `Future<T>`. Ambos tipos representan valores que devuelven los métodos asíncronos y que inicialmente no están definidos. Para obtener acceso al valor de un objeto, hay que llamar a su método `get`. Aparte de eso, los dos tipos se comportan de forma muy diferente.

- `Future<T>` es una construcción de sincronización que permite que una aplicación espere a que finalice el método asíncrono. Si llama a `get` y el objeto no está preparado, se bloquea hasta que lo está.
- Con `Promise<T>`, el marco de trabajo se encarga de la sincronización. Si llama a `get` y el objeto no está preparado, `get` genera una excepción.

La principal finalidad de `Promise<T>` es administrar el flujo de datos de una actividad a otra. Garantiza que una actividad no se ejecute hasta que los datos de entrada sean válidos. En muchos casos, los procesos de trabajo no tienen que obtener acceso a los objetos `Promise<T>` directamente; simplemente pasan los objetos de una actividad a otra y dejan que el marco de trabajo y los procesos de trabajo de la actividad se encarguen de los detalles. Para obtener acceso al valor de un objeto `Promise<T>` en un proceso de trabajo de flujo de trabajo, debe estar seguro de que el objeto está preparado antes de llamar a su método `get`.

- Lo más recomendable es pasar el objeto `Promise<T>` a un método de flujo de trabajo asíncrono y procesar allí los valores. Un método asíncrono aplaza la ejecución hasta que todos sus objetos `Promise<T>` de entrada estén preparados, lo que garantiza que pueda obtener acceso a sus valores de manera segura.
- `Promise<T>` expone un método `isReady` que devuelve `true` si el objeto está preparado. No se recomienda utilizar `isReady` para sondear un objeto `Promise<T>`, pero `isReady` resulta útil en determinadas circunstancias.

AWS Flow Framework Para Java también incluye un `Settable<T>` tipo, que se deriva de `Promise<T>` y tiene un comportamiento similar. La diferencia es que el marco suele establecer el valor de un `Promise<T>` objeto y el trabajador del flujo de trabajo es responsable de establecer el valor de a`Settable<T>`.

Hay algunas circunstancias en las que un proceso de trabajo de flujo de trabajo tiene que crear un objeto `Promise<T>` y establecer su valor. Por ejemplo, un método asíncrono que devuelve un objeto `Promise<T>` tiene que crear un valor de retorno.

- Para crear un objeto que represente un valor con tipo, llame al método `Promise.asPromise` estático, que crea un objeto `Promise<T>` del tipo apropiado, establece su valor y lo pone en estado preparado.
- Para crear un objeto `Promise<Void>`, llame al método `Promise.Void` estático.

Note

`Promise<T>` puede representar cualquier tipo válido. Sin embargo, si hay que serializar los datos por Internet, el tipo debe ser compatible con el conversor de datos. Para obtener más información, consulte la siguiente sección.

Conversores de datos y serialización

AWS Flow Framework Organiza los datos en Internet mediante un convertidor de datos. De manera predeterminada, el marco de trabajo utiliza un conversor de datos que está basado en el procesador JSON de Jackson. Sin embargo, este conversor tiene algunas limitaciones. Por ejemplo, puede serializar mapas que no utilizan cadenas como claves. Si el conversor predeterminado no es suficiente para su aplicación, puede implementar un conversor de datos personalizado. Para obtener información, consulte DataConverters.

AWS Flow Framework Conceptos básicos: Data Exchange entre aplicaciones y ejecuciones de flujos de trabajo

Un método de punto de entrada de flujo de trabajo puede tener uno o más parámetros, lo que permite que el iniciador de flujo de trabajo pase datos iniciales al flujo de trabajo. También puede ser útil proporcionar datos adicionales al flujo de trabajo durante la ejecución. Por ejemplo, si un cliente cambia la dirección de envío, podría notificar al flujo de trabajo de procesamiento de pedidos para que pueda hacer los cambios correspondientes.

Amazon SWF permite que los flujos de trabajo implementen un método de señal, que permite que aplicaciones como el iniciador de flujo de trabajo pasen datos al flujo de trabajo en cualquier momento. Un método de señal puede tener parámetros y un nombre útiles. Usted lo designa como método de señal incluyéndolo en su definición de interfaz de flujo de trabajo y aplicando una anotación `@Signal` a la declaración del método.

El siguiente método muestra una interfaz de flujo de procesamiento de pedidos que declara un método de señal, `changeOrder`, que permite que el iniciador del flujo de trabajo cambie el pedido original una vez que el flujo de trabajo ha comenzado.

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

El procesador de anotaciones del marco de trabajo crea un método del cliente de flujo de trabajo con el mismo nombre que el método de la señal y el iniciador del flujo de trabajo llama al método del cliente para pasar datos al flujo de trabajo. Para ver un ejemplo, consulte Recetas de AWS Flow Framework.

Tipos de tiempo de espera de Amazon SWF

Para garantizar que las ejecuciones del flujo de trabajo se ejecuten correctamente, puede establecer distintos tipos de tiempos de espera con Amazon SWF. Algunos tiempos de espera especifican cuánto puede tardar el flujo de trabajo en ejecutarse en su totalidad. Otros tiempos de espera especifican cuánto pueden tardar las tareas de actividad en asignarse a un proceso de trabajo y cuánto pueden tardar en completarse desde el momento de su programación. Todos los tiempos de espera de la API de Amazon SWF se especifican en segundos. Amazon SWF también admite la cadena `NONE` como valor de tiempo de espera, lo que indica que no se establecerá ningún tiempo de espera.

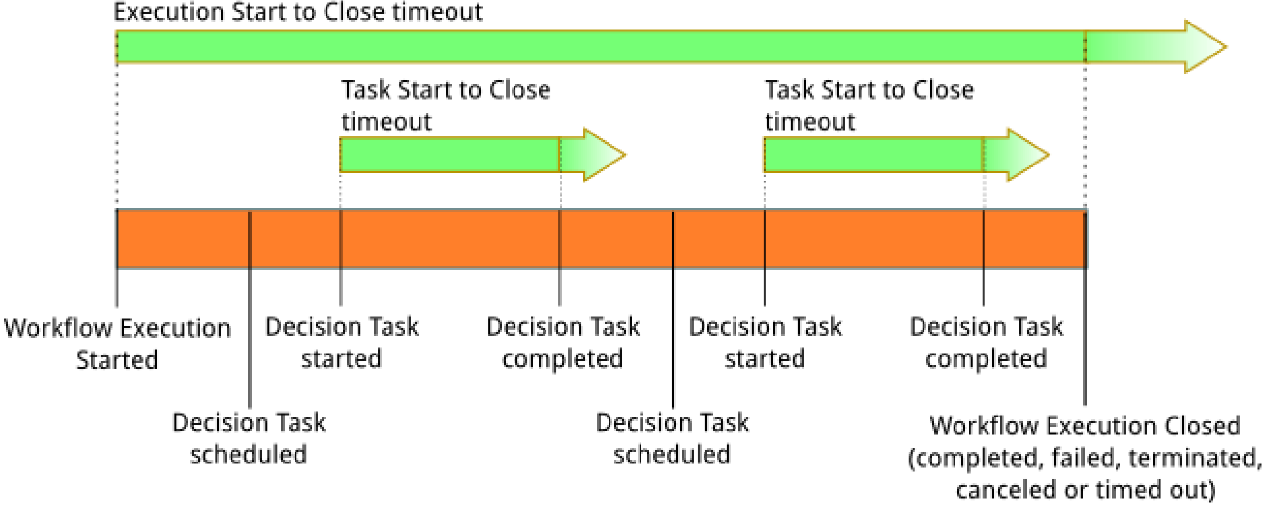
Para los tiempos de espera relacionados con las tareas de decisión y actividad, Amazon SWF añade un evento al historial de ejecución del flujo de trabajo. Los atributos del evento proporcionan información acerca del tipo de tiempo de espera que tuvo lugar y la tarea de decisión o actividad que se vio afectada. Amazon SWF también programa una tarea de decisión. Cuando el decisor reciba la nueva tarea de decisión, verá el evento de tiempo de espera en el historial y tomará la acción adecuada al solicitarla. RespondDecisionTaskCompleted

Una tarea se considera abierta desde el momento en que se programa hasta que se cierra. Por tanto, una tarea se registra como abierta mientras un proceso de trabajo la procesa. Una tarea se

cierra cuando un proceso de trabajo la registra como Completed, Canceled o Failed. Amazon SWF también puede cerrar una tarea como resultado de un tiempo de espera.

Tiempos de espera de las tareas de decisión y flujo de trabajo

En el siguiente diagrama se muestra cómo los tiempos de espera de decisión y flujo de trabajo están relacionados con la vida útil de un flujo de trabajo:



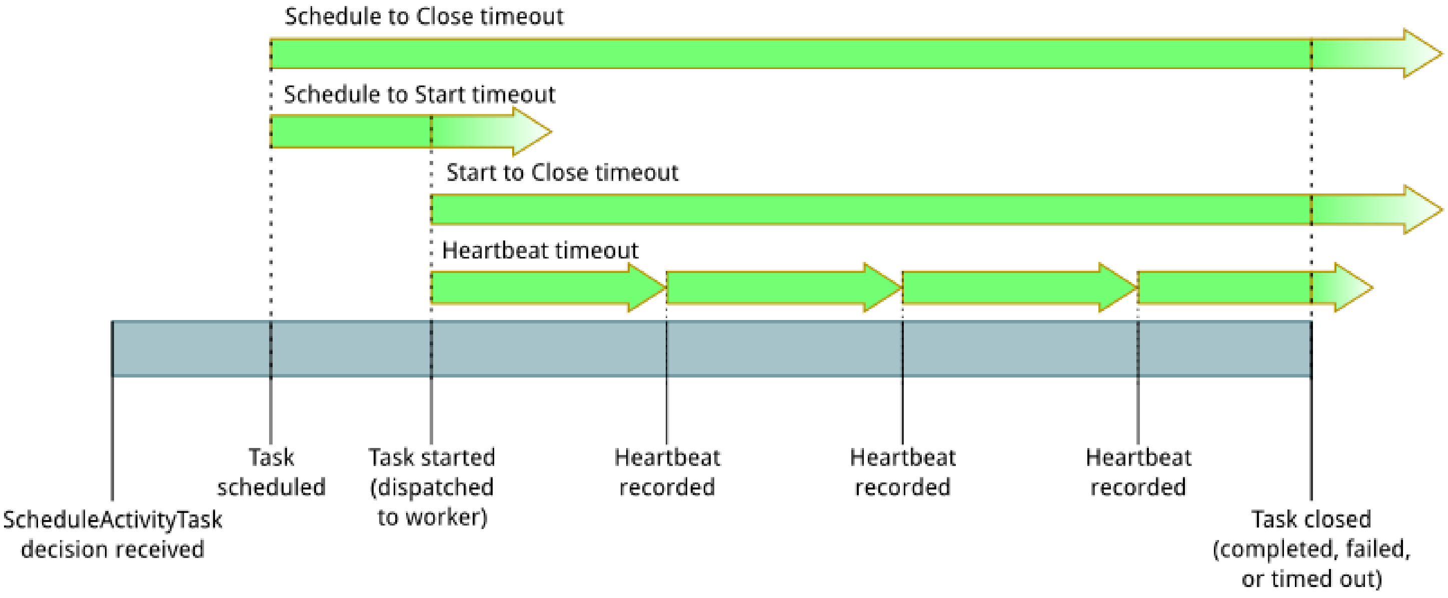
Hay dos tipos de tiempo de espera que son pertinentes para las tareas de decisión y flujo de trabajo:

- Tiempo de espera de inicio a cierre del flujo de trabajo (**`timeoutType: START_TO_CLOSE`**): este tiempo de espera especifica el tiempo máximo que puede tardar en completarse la ejecución de un flujo de trabajo. Se establece como valor predeterminado durante el registro del flujo de trabajo, pero se puede anular con un valor diferente al iniciarse el flujo de trabajo. Si se supera este tiempo de espera, Amazon SWF cierra la ejecución del flujo de trabajo y añade un WorkflowExecutionTimedOuttipo de evento al historial de ejecución del flujo de trabajo. Además de `timeoutType`, los atributos del evento especifican la `childPolicy` que se encuentra en vigor para esta ejecución de flujo de trabajo. La política secundaria especifica cómo se controlan las ejecuciones de los flujos de trabajo secundarios si se agota el tiempo de espera de la ejecución de flujo de trabajo principal o termina por otro motivo. Por ejemplo, si `childPolicy` se establece en TERMINATE, las ejecuciones de flujos de trabajo secundarios se terminarán. Una vez que se haya agotado el tiempo de espera de una ejecución de flujo de trabajo, la única medida que podrá tomar al respecto son las llamadas de visibilidad.
- Tiempo de espera de inicio a cierre de la tarea de decisión (**`timeoutType: START_TO_CLOSE`**): este tiempo de espera especifica el tiempo máximo que el decisor correspondiente puede tardar en completar una tarea de decisión. Se establece durante el registro del tipo de flujo de trabajo. Si se supera este tiempo de espera, la tarea se marca como agotada en el historial de ejecución

del flujo de trabajo y Amazon SWF añade un DecisionTaskTimedOuttipo de evento al historial del flujo de trabajo. Los atributos del evento incluirán los eventos IDs correspondientes al momento en que se programó la tarea de decisión (`scheduledEventId`) y al momento en que se inició (). `startedEventId` Además de añadir el evento, Amazon SWF también programa una tarea de decisión nueva para alertar al decisor de que se ha agotado el tiempo de espera de dicha tarea de decisión. Tras agotarse este tiempo de espera, un intento de completar la tarea de decisión con tiempo de espera agotado mediante `RespondDecisionTaskCompleted` producirá un error.

Tiempos de espera de las tareas de actividad

En el siguiente diagrama se muestra cómo los tiempos de espera están relacionados con la vida útil de una tarea de actividad:



Hay cuatro tipos de tiempo de espera que son pertinentes para las tareas de actividad:

- Tiempo de espera de inicio a cierre de la tarea de actividad (**`timeoutType: START_TO_CLOSE`**): este tiempo de espera especifica el tiempo máximo que el proceso de trabajo de una actividad puede tardar en procesar una tarea después de haberla recibido. Los intentos de cerrar una tarea de actividad agotada utilizando RespondActivityTaskCanceledRespondActivityTaskCompleted, y RespondActivityTaskFailedfallarán.
- Latido de la tarea de actividad (**`timeoutType: HEARTBEAT`**): este tiempo de espera especifica el tiempo máximo que se puede ejecutar una tarea antes de indicar su progreso en la acción `RecordActivityTaskHeartbeat`.

- Programación de tareas de actividad hasta su inicio (**`timeoutType: SCHEDULE_TO_START`**): este tiempo de espera especifica cuánto tiempo espera Amazon SWF antes de agotar el tiempo de espera de la tarea de actividad si no hay procesos de trabajo disponibles para realizarla. Una vez que se agota el tiempo de espera, la tarea caducada no se asignará a otro proceso de trabajo.
- Programación de tareas de actividad hasta su cierre (**`timeoutType: SCHEDULE_TO_CLOSE`**): este tiempo de espera especifica cuánto tiempo puede tardar la tarea desde el momento en que se programe hasta el momento en que se complete. Como práctica recomendada, este valor no debe ser mayor que la suma del tiempo de espera de la tarea y el schedule-to-start tiempo de espera de la tarea start-to-close.

> **Note**
>
> Cada uno de los tipos de tiempo de espera tiene un valor predeterminado, que generalmente se establece en `NONE` (infinito). El tiempo máximo de cualquier ejecución de actividad se limita a un año, sin embargo.

Puede establecer valores predeterminados para estos durante el registro del tipo de actividad, pero puede anularlos con nuevos valores al programar la tarea de actividad. Cuando se agote uno de estos tiempos de espera, Amazon SWF añadirá un ActivityTaskTimedOuttipo de evento al historial del flujo de trabajo. El atributo de valor `timeoutType` de este evento especificará cuál de estos tiempos de espera ha tenido lugar. Para cada uno de los tiempos de espera, el valor de `timeoutType` se muestra entre paréntesis. Los atributos del evento también incluirán los eventos IDs correspondientes al momento en que se programó la tarea de la actividad (`scheduledEventId`) y al momento en que se inició (`startedEventId`). Además de añadir el evento, Amazon SWF también programa una tarea de decisión nueva para alertar al decisor de que se ha agotado el tiempo de espera.

Descripción de una tarea en AWS Flow Framework for Java

Temas

- Tarea
- Orden de ejecución
- Ejecución del flujo de trabajo
- No determinismo

Tarea

La primitiva subyacente que utiliza Java AWS Flow Framework para gestionar la ejecución del código asíncrono es la clase. `Task` Un objeto tipo `Task` representa trabajo que hay que realizar de manera asíncrona. Cuando llama a un método asíncrono, el marco de trabajo crea una `Task` para ejecutar el código en ese método y lo pone en una lista para su ejecución más adelante. De manera parecida, al invocar una `Activity`, se crea una `Task` para dicha actividad. La llamada al método regresa después de esto, devolviendo normalmente una `Promise<T>` como resultado futuro de la llamada.

La clase `Task` es pública y puede usarse directamente. Por ejemplo, podemos volver a escribir el ejemplo de Hello World para que use una `Task` en lugar de un método asíncrono.

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

El marco de trabajo llama al método `doExecute()` cuando todas las `Promises` que se han pasado al constructor de las `Task` están listas. Para obtener más información sobre la `Task` clase, consulte la documentación. AWS SDK para Java

El marco de trabajo también incluye una clase llamada `Functor` que representa una `Task` que es también una `Promise<T>`. El objeto `Functor` está listo cuando se completa la `Task`. En el siguiente ejemplo, se crea un `Functor` para obtener el mensaje de saludo:

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

Orden de ejecución

Las tareas se vuelven elegibles para la ejecución solo cuando todos los parámetros escritos `Promise<T>`, que se han pasado a la actividad o al método asíncrono correspondientes, están listos. Una `Task` que está lista para la ejecución se mueve de manera lógica a una cola lista. En otras palabras, se programa para la ejecución. Para ejecutar la tarea, la clase de proceso de trabajo invoca el código que se escribió en el cuerpo del método asíncrono o programa una tarea de actividad en Amazon Simple Workflow Service (AWS) en el caso de un método de actividad.

A medida que las tareas ejecutan y producen resultados, hacen que otras tareas estén preparadas y la ejecución del programa continúa avanzando. La manera en que el marco de trabajo ejecuta tareas es importante para comprender el orden en el que se ejecuta su código asíncrono. El código que aparece secuencialmente en su programa podría no ejecutarse en ese orden.

```
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!");
}
```

El código en la lista de arriba imprimirá lo siguiente:

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

Quizás no sea lo que esperaba pero puede explicarse fácilmente analizando cómo se ejecutaron las tareas para los métodos asíncronos:

1. La llamada a `getUserName` crea una `Task`. La llamaremos `Task1`. Como `getUserName` no toma ningún parámetro, `Task1` se pone inmediatamente en la cola de preparación.
2. A continuación, la llamada a `printHelloName` crea una `Task` que tiene que esperar el resultado de `getUserName`. La llamaremos `Task2`. Como el valor requerido aún no está listo, `Task2` se coloca en la lista de espera.
3. A continuación se crea una tarea para `printHelloWorld` y se añade a la cola lista. La llamaremos `Task3`.
4. A continuación, la instrucción `println` imprime "Hello, Amazon!" en la consola.
5. En este punto, `Task1` y `Task3` están en la cola lista y `Task2` está en la lista de espera.
6. El proceso de trabajo ejecuta `Task1` y su resultado hace que `Task2` esté listo. `Task2` se añade a la cola lista por detrás de `Task3`.
7. `Task3` y `Task2` se ejecutan, a continuación, en ese orden.

La ejecución de actividades sigue el mismo patrón. Cuando se llama a un método en el cliente de actividad, este crea una `Task` que, cuando se ejecuta, programa una actividad en Amazon SWF.

El marco de trabajo confía en características como la generación de códigos y proxies dinámicos para inyectar la lógica para la conversión de llamadas de método en invocaciones de actividad y tareas asíncronas en su programa.

Ejecución del flujo de trabajo

La clase de proceso de trabajo también administra la ejecución de la implementación de flujo de trabajo. Cuando se llama a un método en el cliente de flujo de trabajo, llama a Amazon SWF para crear una instancia de flujo de trabajo. Las tareas de Amazon SWF no deben confundirse con las tareas del marco de trabajo. Una tarea en Amazon SWF es una tarea de actividad o una tarea de decisión. La ejecución de las tareas de actividad es sencilla. La clase de proceso de trabajo de actividad recibe tareas de actividad de Amazon SWF, invoca el método de actividad apropiado de la implementación y devuelve el resultado a Amazon SWF.

La ejecución de las tareas de decisión requiere más pasos. El proceso de trabajo de flujo de trabajo recibe las tareas de decisión de Amazon SWF. Una tarea de decisión es en realidad una solicitud que pregunta a la lógica de flujo de trabajo qué debe hacer a continuación. La primera tarea de decisión se genera para una instancia de flujo de trabajo cuando se inicia a través del cliente de flujo de trabajo. Al recibir esta tarea de decisión, el marco de trabajo comienza a ejecutar el código en el método de flujo de trabajo anotado con `@Execute`. Este método ejecuta la lógica de coordinación que programa actividades. Cuando el estado de la instancia de flujo de trabajo cambia, por ejemplo, cuando se completa una actividad, se programan tareas de decisión adicionales. En este punto, la lógica de flujo de trabajo puede decidir actuar en función del resultado de la actividad; por ejemplo, podría decidir programar otra actividad.

El marco de trabajo oculta todos estos detalles al desarrollador traduciendo a la perfección tareas de decisión a la lógica del flujo de trabajo. Desde el punto de vista de un desarrollador, el código tiene el mismo aspecto que un programa normal. Internamente, el marco de trabajo lo asigna a llamadas a Amazon SWF y a tareas de decisión mediante el historial que mantiene Amazon SWF. Cuando llega una tarea de decisión, el marco de trabajo reproduce la ejecución del programa incorporando los resultados de las actividades completadas hasta el momento. Las actividades y métodos asíncronos que estaban esperando estos resultados se desbloquean y la ejecución del programa avanza.

En la siguiente tabla se muestra la ejecución del flujo de trabajo de procesamiento de imágenes de ejemplo y el historial correspondiente.

La ejecución del flujo de trabajo de miniaturas

La ejecución del programa de flujo de trabajo	El historial que mantiene Amazon SWF
Ejecución inicial	
1. Bucle de envío	1. Instancia de flujo de trabajo iniciada, id="1"

La ejecución del programa de flujo de trabajo	El historial que mantiene Amazon SWF
2. getImageUrls 3. downloadImage 4. createThumbnail (tarea en la cola de espera) 5. uploadImage (tarea en la cola de espera) 6. <siguiente iteración del bucle>	2. downloadImage programada
Reproducción	
1. Bucle de envío 2. getImageUrls 3. ruta de imagen downloadImage="foo" 4. createThumbnail 5. uploadImage (tarea en la cola de espera) 6. <siguiente iteración del bucle>	1. Instancia de flujo de trabajo iniciada, id="1" 2. downloadImage programada 3. downloadImage completada, devuelve="foo" 4. createThumbnail programada
Reproducción	
1. Bucle de envío 2. getImageUrls 3. ruta de imagen downloadImage="foo" 4. ruta de miniatura createThumbnail="bar" 5. uploadImage 6. <siguiente iteración del bucle>	1. Instancia de flujo de trabajo iniciada, id="1" 2. downloadImage programada 3. downloadImage completada, devuelve="foo" 4. createThumbnail programada 5. createThumbnail completada, devuelve= "bar" 6. uploadImage programada
Reproducción	

La ejecución del programa de flujo de trabajo	El historial que mantiene Amazon SWF
1. Bucle de envío 2. getImageUrls 3. ruta de imagen downloadImage="foo" 4. ruta de miniatura createThumbnail="bar" 5. uploadImage 6. <siguiente iteración del bucle>	1. Instancia de flujo de trabajo iniciada, id="1" 2. downloadImage programada 3. downloadImage completada, devuelve="foo" 4. createThumbnail programada 5. createThumbnail completada, devuelve= "bar" 6. uploadImage programada 7. uploadImage completada ...

Cuando se realiza una llamada a `processImage`, el marco de trabajo crea una nueva instancia de flujo de trabajo en Amazon SWF. Se trata de un registro duradero de la instancia de flujo de trabajo que se está iniciando. El programa se ejecuta hasta la llamada a la actividad `downloadImage` que pide a Amazon SWF que programe una actividad. El flujo de trabajo se sigue ejecutando y crea tareas para las actividades posteriores, pero no se pueden ejecutar hasta que la actividad `downloadImage` se complete; por lo tanto, este episodio de reproducción finaliza. Amazon SWF envía la tarea de la actividad `downloadImage` para que se ejecute y, una vez finalizada, se registra en el historial junto con el resultado. El flujo de trabajo está ahora listo para avanzar y Amazon SWF genera una tarea de decisión. El marco de trabajo recibe la tarea de decisión y reproduce el flujo de trabajo incorporando el resultado de las imágenes descargadas tal y como está registrado en el historial. Esto desbloquea la tarea de `createThumbnail`, y la ejecución del programa continúa avanzando con la programación de la tarea de actividad `createThumbnail` en Amazon SWF. Se repite el mismo proceso para `uploadImage`. La ejecución del programa sigue su curso hasta que el flujo de trabajo ha procesado todas las imágenes y no hay tareas pendientes. Como ningún estado de ejecución se almacena localmente, es posible que cada tarea de decisión se ejecute en una máquina diferente. Esto le permite escribir fácilmente programas que sean tolerantes a errores y fácilmente escalables.

No determinismo

Como el marco se basa en la reproducción, es importante que el código de orquestación (todo el código del flujo de trabajo, con la excepción de las implementaciones de actividades) sea

determinista. Por ejemplo, el flujo de control en su programa no debería depender de un número aleatorio o de la hora actual. Como estas cosas cambiarán entre las invocaciones, es posible que la reproducción no siga el mismo camino a través de la lógica de orquestación. Esto llevará a errores o resultados imprevistos. El marco de trabajo proporciona un `WorkflowClock` que puede utilizar para obtener la hora actual de manera determinista. Consulte la sección en Contexto de ejecución para obtener más información.

> **Note**
>
> Una conexión de Spring incorrecta de objetos de implementación de flujo de trabajo también puede llevar al no determinismo. Los beans de implementación de flujo de trabajo así como los bean de los que dependen tienen que estar dentro del alcance del flujo de trabajo (`WorkflowScope`). Por ejemplo, la conexión de un bean de implementación de flujo de trabajo a un bean que mantiene estado y está dentro del contexto global dará como resultado un comportamiento inesperado. Consulte la sección Integración con Spring para obtener más información.

AWS Flow Framework Guía de programación para Java

En esta sección se proporcionan detalles sobre cómo utilizar las funciones de Java AWS Flow Framework para implementar aplicaciones de flujo de trabajo.

Temas

- Implementación de aplicaciones de flujo de trabajo con AWS Flow Framework
- Contratos de flujo de trabajo y de actividad
- Registro de tipos de flujos de trabajo y de actividades
- Clientes de actividad y flujo de trabajo
- Implementación de flujos de trabajo
- Implementación de actividades
- Implementación de AWS Lambda tareas
- Ejecución de programas escritos con AWS Flow Framework para Java
- Contexto de ejecución
- Ejecuciones de flujo de trabajo secundario
- Flujos de trabajo continuos
- Configuración de la prioridad de las tareas en Amazon SWF
- DataConverters
- Paso de datos a los métodos asíncronos
- Capacidad de realización de pruebas e inserción de dependencias
- Gestión de errores
- Reintento de actividades con errores
- Tareas del demonio
- AWS Flow Framework para Java Replay Behavior

Implementación de aplicaciones de flujo de trabajo con AWS Flow Framework

Los pasos típicos necesarios para desarrollar un flujo de trabajo con ellos AWS Flow Framework son:

1. Definición de contratos de flujo de trabajo y de actividad. Análisis de los requisitos de su aplicación y determinación, a continuación, de la topología del flujo de trabajo y de las actividades necesarias. Las actividades gestionan las tareas de procesamiento necesarias, mientras que la topología de flujo de trabajo define la estructura básica del flujo de trabajo y la lógica de negocio.

   Por ejemplo, es posible que una aplicación de procesamiento de medios tenga que descargar un archivo, procesarlo y, a continuación, cargar el archivo procesado a un bucket de Amazon Simple Storage Service (S3). Esto puede desglosarse en cuatro tareas de actividad:

   1. descarga del archivo de un servidor
   2. procesamiento del archivo (por ejemplo, mediante la transcodificación a un formato multimedia diferente)
   3. carga del archivo en el bucket de S3
   4. realización de una limpieza eliminando los archivos locales

   Este flujo de trabajo tendría un método de punto de entrada e implementaría una topología lineal sencilla que ejecuta las actividades en secuencia, de forma parecida a HelloWorldWorkflow Solicitud.

2. Implementación de interfaces de actividad y de flujo de trabajo. Los contratos de flujo de trabajo y de actividad se definen mediante interfaces de Java, haciendo sus convenciones de llamadas previsibles con SWF, y ofreciéndoles flexibilidad al implementar su lógica de flujo de trabajo y las tareas de actividad. Las diferentes partes de su programa pueden actuar como consumidores de los datos del otro, si bien no tiene que saber mucho de los detalles de implementación de ninguna de las otras partes.

   Por ejemplo, puede definir una interfaz `FileProcessingWorkflow` y proporcionar diferentes implementaciones de flujo de trabajo para codificación de vídeo, compresión, miniaturas, etc. Cada uno de esos flujos de trabajo puede tener diferentes flujos de control y puede llamar a diferentes métodos de actividad; su iniciador de flujo de trabajo no tiene que saberlo. Mediante el uso de interfaces, también es fácil probar sus flujos de trabajo utilizando implementaciones simuladas que pueden sustituirse más adelante con código funcional.

3. Generación de clientes de actividad y de flujo de trabajo. AWS Flow Framework Esto elimina la necesidad de implementar los detalles de la gestión de la ejecución asíncrona, el envío de solicitudes HTTP, la clasificación de los datos, etc. En su lugar, el iniciador de flujo de trabajo ejecuta una instancia de flujo de trabajo llamando a un método en un cliente de flujo de trabajo

y la implementación de flujo de trabajo ejecuta actividades llamando a métodos en el cliente de actividades. El marco de trabajo gestiona los detalles de estas interacciones en segundo plano.

Si utiliza Eclipse y ha configurado su proyecto, por ejemplo, enConfiguración del AWS Flow Framework para Java, el procesador de AWS Flow Framework anotaciones utiliza las definiciones de la interfaz para generar automáticamente clientes de flujos de trabajo y actividades que utilizan el mismo conjunto de métodos que la interfaz correspondiente.

4. Implementación de aplicaciones host de actividad y flujo de trabajo. Las implementaciones de sus flujos de trabajo y actividades deben estar integradas en aplicaciones host que consulten Amazon SWF en busca de tareas, recopilen los datos y utilicen los métodos de implementación adecuados. AWS Flow Framework para Java incluye ActivityWorkerclases que facilitan WorkflowWorkery simplifican la implementación de aplicaciones hospedadoras.
5. Pon a prueba tu flujo de trabajo. AWS Flow Framework para Java proporciona una JUnit integración que puede utilizar para probar sus flujos de trabajo en línea y de forma local.
6. Implementación de procesos de trabajo. Puede desplegar a sus trabajadores según convenga; por ejemplo, puede desplegarlos en las EC2 instancias de Amazon o en los ordenadores de su centro de datos. Una vez implementados e iniciados, los procesos de trabajo comienzan a sondear Amazon SWF para obtener tareas y gestionarlas según sea necesario.
7. Comience las ejecuciones. Una aplicación inicia una instancia de flujo de trabajo a través del cliente de flujo de trabajo para llamar al punto de entrada del flujo de trabajo. También puede comenzar los flujos de trabajo mediante la consola de Amazon SWF. Independientemente de cómo se inicie una instancia de flujo de trabajo, puede utilizar la consola de Amazon SWF para monitorear la instancia de flujo de trabajo en ejecución y examinar el historial de flujo de trabajo para las instancias en ejecución, las finalizadas y en las que se haya producido algún error.

AWS SDK para JavaIncluye un conjunto de ejemplos AWS Flow Framework de Java que puede explorar y ejecutar siguiendo las instrucciones del archivo readme.html del directorio raíz. También hay un conjunto de recetas, aplicaciones sencillas, que muestran cómo gestionar diferentes problemas específicos de programación, que están disponibles a través de Recetas de AWS Flow Framework.

Contratos de flujo de trabajo y de actividad

Las interfaces de Java se usan para declarar las firmas de flujos de trabajo y de actividades. La interfaz forma el contrato entre la implementación de flujo de trabajo (o actividad) y el cliente de ese

flujo de trabajo (o actividad). Por ejemplo, un tipo de flujo de trabajo `MyWorkflow` se define mediante el uso de una interfaz con la anotación `@Workflow`:

```
@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();
}
```

El contrato no tiene ajustes específicos de la implementación. Este uso de contratos neutros de implementación permite desacoplar a los clientes de la implementación y proporciona, por tanto, la flexibilidad para cambiar los detalles de la implementación sin que afecte al cliente. Y viceversa, también puede cambiar el cliente si tener que hacer cambios en el flujo de trabajo o en la actividad que se está consumiendo. Por ejemplo, el cliente puede modificarse para llamar a una actividad de manera asíncrona utilizando promesas (`Promise<T>`) sin que haya que cambiar la implementación de la actividad. Del mismo modo, la implementación de la actividad puede cambiarse para que la complete de manera asíncrona, por ejemplo, una persona que envía un correo, sin que haya que cambiar los clientes de la actividad.

En el ejemplo de arriba, la interfaz de flujo de trabajo `MyWorkflow` contiene un método, `startMyWF`, para comenzar una ejecución nueva. Este método se anota con la anotación `@Execute` y tiene que tener un tipo de retorno de `void` o `Promise<>`. En una interfaz de flujo de trabajo dada, es posible anotar como mucho un método con esta anotación. Este método es el punto de entrada de la lógica del flujo de trabajo y el marco de trabajo llama a este método para ejecutar la lógica del flujo de trabajo cuando se recibe una tarea de decisión.

La interfaz de flujo de trabajo también define las señales que pueden enviarse al flujo de trabajo. El método de la señal se invoca cuando la ejecución del flujo de trabajo recibe una señal con un nombre que coincide. Por ejemplo, la interfaz `MyWorkflow` declara un método de señal, `signal1`, anotado con la anotación `@Signal`.

La anotación `@Signal` es obligatoria en los métodos de señal. El tipo de retorno de un método de señal debe ser `void`. Una interfaz de flujo de trabajo puede tener cero o más métodos de señal definidos en ella. Puede declarar una interfaz de flujo de trabajo si un método `@Execute` y algunos métodos `@Signal` para generar clientes que no pueden comenzar su ejecución pero pueden enviar señales a ejecuciones en curso.

Los métodos con anotaciones `@Execute` y `@Signal` pueden tener cualquier número de parámetros de cualquier tipo distintos de `Promise<T>` o sus derivados. Esto le permite pasar entradas con establecimiento inflexible a una ejecución de flujo de trabajo al principio y durante su ejecución. El tipo de retorno del método `@Execute` debe ser `void` o `Promise<>`.

Además, también puede declarar un método en la interfaz de flujo de trabajo para informar el último estado de una ejecución de flujo de trabajo, por ejemplo, el método `getState` en el ejemplo anterior. Este estado no es el estado completo de la aplicación del flujo de trabajo. El uso previsto de esta característica es permitirle que almacene hasta 32 KB de datos para indicar el último estado de la ejecución. Por ejemplo, en un flujo de trabajo de procesamiento de pedidos, puede almacenar una cadena que indica que se ha recibido, procesado o cancelado el pedido. El marco de trabajo llama a este método siempre que se completa una tarea de decisión para obtener el último estado. El estado se almacena en Amazon Simple Workflow Service (Amazon SWF) y puede recuperarse mediante el cliente externo generado. Esto le permite comprobar el último estado de una ejecución de flujo de trabajo. Los métodos anotados con `@GetState` no deben usar ningún argumento y no deben tener un tipo de retorno de `void`. Puede devolver cualquier tipo, que se ajuste a sus necesidades, de este método. En el ejemplo anterior, un objeto de `MyWorkflowState` (consulte la definición a continuación) es devuelto por el método utilizado para almacenar un estado de cadena y un porcentaje numérico completo. Se espera que este método realice acceso de solo lectura del objeto de implementación de flujo de trabajo y se invoque de manera síncrona, lo que no permite el uso de cualquier operación asíncrona como la llamada de métodos anotados con `@Asynchronous`. Como máximo, puede anotarse un método en una interfaz de flujo de trabajo con la anotación `@GetState`.

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

Del mismo modo, se define un conjunto de actividades mediante una interfaz con la anotación `@Activities`. Cada método de la interfaz se corresponde con una actividad, por ejemplo:

```
@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);
}
```

La interfaz le permite agrupar un conjunto de actividades relacionadas. Puede definir cualquier número de actividades dentro de una interfaz de actividades y puede definir tantas interfaces de actividades como quiera. De manera parecida a los métodos `@Execute` y `@Signal`, los métodos de actividad pueden aceptar cualquier número de argumentos de cualquier tipo distintos de `Promise<T>` o sus derivados. El tipo de retorno de una actividad no debe ser `Promise<T>` o sus derivados.

Registro de tipos de flujos de trabajo y de actividades

Amazon SWF requiere que los tipos de actividades y de flujos de trabajo estén registrados para poder utilizarlos. El marco de trabajo registra automáticamente los flujos de trabajo y las actividades en las implementaciones que añada al proceso de trabajo. El marco de trabajo busca tipos que implementan los flujos de trabajo y las actividades, y los registra en Amazon SWF. De manera predeterminada, el marco de trabajo utiliza las definiciones de interfaz para inferir opciones de registro de tipos de flujos de trabajo y de actividades. Todas las interfaces de flujo de trabajo tienen que tener la anotación `@WorkflowRegistrationOptions` o la anotación `@SkipRegistration`. El proceso de trabajo de flujo de trabajo registra todos los tipos de flujo de trabajo con los que se ha configurado que tienen la anotación `@WorkflowRegistrationOptions`. De manera parecida, cada método de actividad deberá comentarse o bien mediante el comentario `@ActivityRegistrationOptions` o el comentario `@SkipRegistration`, o uno de estos comentarios tiene que estar presente en la interfaz `@Activities`. El proceso de trabajo de actividad registra todos los tipos de actividad con los que se ha configurado a los que se les aplica la anotación `@ActivityRegistrationOptions`. El registro se realiza automáticamente cuando inicia uno de los procesos de trabajo. Los tipos de flujos de trabajo y de actividades con la anotación `@SkipRegistration` no se registran. Las anotaciones `@ActivityRegistrationOptions` y

`@SkipRegistration` tienen semántica de anulación y se aplica la más específica a un tipo de actividad.

Tenga en cuenta que Amazon SWF no le permite volver a registrar ni modificar el tipo una vez que lo ha registrado. El marco de trabajo tratará de registrar todos los tipos pero si el tipo ya se ha registrado no volverá a registrarse y no se informará de ningún error.

Si tiene que modificar una configuración registrada, tiene que registrar una versión nueva del tipo. También puede anular configuraciones registradas cuando comienza una nueva ejecución o al llamar una actividad que utiliza los clientes generados.

El registro exige un nombre de tipo y otras opciones de registro. La implementación predeterminada los determina de la siguiente manera:

Nombre del tipo de flujo de trabajo y versión

El marco de trabajo determina el nombre del tipo de flujo de trabajo a partir de la interfaz del grupo de trabajo. La forma del nombre del tipo de flujo de trabajo predeterminado es {*prefix*}{*name*}. El {*prefix*} se establece con el nombre de la `@Workflow` interfaz seguido de un '.' y el {*name*} se establece con el nombre del `@Execute` método. El nombre predeterminado del tipo de flujo de trabajo en el ejemplo anterior es `MyWorkflow.startMyWF`. Puede anular el nombre predeterminado usando el parámetro del nombre del método `@Execute`. El nombre predeterminado del tipo de flujo de trabajo en el ejemplo es `startMyWF`. El nombre no debe ser una cadena vacía. Tenga en cuenta que cuando anula el nombre mediante `@Execute`, el marco de trabajo no le agrega automáticamente un prefijo. Puede utilizar su propio esquema de nomenclatura.

La versión del flujo de trabajo se especifica mediante el parámetro `version` de la anotación `@Execute`. No hay valor predeterminado para `version` y debe especificarse explícitamente; `version` es una cadena de formato libre y puede utilizar su propio esquema de control de versiones.

Nombre de la señal

El nombre de la señal puede especificarse utilizando el parámetro del nombre de la anotación `@Signal`. Si no se especifica, adopta de manera predeterminada el nombre del método de la señal.

Nombre del tipo de actividad y versión

El marco de trabajo determina el nombre del tipo de actividad a partir de la interfaz de las actividades. La forma del nombre del tipo de actividad predeterminado es {*prefix*} {*name*}. El {*prefix*} se establece con el nombre de la `@Activities` interfaz seguido de un '.' y el {*name*}

se establece con el nombre del método. El valor predeterminado {*prefix*} se puede anular en la `@Activities` anotación de la interfaz de actividades. También puede especificar el nombre del tipo de actividad utilizando la anotación `@Activity` en el método de la actividad. Tenga en cuenta que cuando anula el nombre mediante `@Activity`, el marco de trabajo no le agregará automáticamente un prefijo. Puede utilizar su propio esquema de nomenclatura.

La versión de la actividad se especifica mediante el parámetro de la versión de la anotación `@Activities`. Esta versión se utiliza como la versión predeterminada para todas las actividades definidas en la interfaz y puede anularse en función de la actividad mediante la anotación `@Activity`.

Default Task List

La lista de tareas predeterminadas puede configurarse mediante las anotaciones `@WorkflowRegistrationOptions` y `@ActivityRegistrationOptions` y configurando el parámetro `defaultTaskList`. De forma predeterminada, está establecido en `USE_WORKER_TASK_LIST`. Se trata de un valor especial que indica al marco de trabajo que utilice la lista de tareas configurada en el objeto del proceso de trabajo que se utiliza para registrar el tipo de actividad o de flujo de trabajo. También puede optar por no registrar una lista de tareas predeterminadas configurando la lista de tareas predeterminadas en `NO_DEFAULT_TASK_LIST` mediante estas anotaciones. Puede usarse en casos en los que quiera exigir que se especifique la lista de tareas en el tiempo de ejecución. Si no se ha registrado ninguna lista de tareas predeterminadas, tiene que especificar la lista de tareas al comenzar el flujo de trabajo o mediante una llamada al método de la actividad mediante los parámetros `StartWorkflowOptions` y `ActivitySchedulingOptions` en la sobrecarga respectiva del método del cliente generado.

Otras opciones de registro

Todas las opciones de registro de tipos de flujos de trabajo y de actividades que permite la API de Amazon SWF se pueden especificar a través del marco de trabajo.

Para obtener una lista completa de opciones de registro de flujo de trabajo, consulte los siguientes temas:

- @Flujo de trabajo
- @Execute
- @WorkflowRegistrationOptions
- @Signal

Para obtener una lista completa de opciones de registro de actividad, consulte los siguientes temas:

- @Actividad
- @Tareas
- @ActivityRegistrationOptions

Si desea tener control absoluto sobre el registro de tipos, consulte Extensibilidad de proceso de trabajo.

Clientes de actividad y flujo de trabajo

El marco de trabajo genera los clientes de actividad y flujo de trabajo en función de las interfaces `@Workflow` y `@Activities`. Se generan interfaces de cliente diferentes que contienen métodos y configuraciones que solo tienen sentido para el cliente. Si realiza su desarrollo con Eclipse, para hacer esto debe utilizar el complemento de Eclipse de Amazon SWF cada vez que guarde el archivo que contenga la interfaz apropiada. El código generado se coloca en el directorio de origen generado en el proyecto, en el mismo paquete que la interfaz.

Note

Tenga en cuenta que el nombre de directorio predeterminado que utiliza Eclipse es .apt_generated. Eclipse no muestra directorios cuyos nombres comienzan con un "." en Package Explorer. Si desea ver los archivos generados en Project Explorer (Explorador de paquetes), utilice un nombre de directorio diferente. En Eclipse, haga clic con el botón derecho en Package Explorer y luego elija Properties (Propiedades), Java Compiler (Compilador de Java) y Annotation processing (Procesamiento de anotaciones). A continuación, modifique el valor de Generate source directory (Generar directorio de origen).

Clientes de flujo de trabajo

Los artefactos que se generan para el flujo de trabajo contienen tres interfaces del cliente y las clases que las implementan. Los clientes generados incluyen:

- Un cliente asíncrono cuyo fin es que se consuma dentro de una implementación del flujo de trabajo que proporciona métodos asíncronos para iniciar ejecuciones de flujos de trabajo y enviar señales

- Un cliente externo que se puede utilizar para iniciar ejecuciones y enviar señales, además de recuperar el estado del flujo de trabajo fuera del ámbito de una implementación del flujo de trabajo
- Un autocliente que se puede utilizar para crear flujos de trabajo continuos

Por ejemplo, estas son las interfaces de cliente generadas en la interfaz `MyWorkflow` de ejemplo:

```
//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);
```

Las interfaces tienen métodos sobrecargados que se corresponden con cada método de la interfaz `@Workflow` que se declara.

El cliente externo refleja los métodos de la interfaz `@Workflow` con una sobrecarga adicional del método `@Execute` que toma `StartWorkflowOptions`. Puede utilizar esta sobrecarga para pasar opciones adicionales al comenzar una nueva ejecución del flujo de trabajo. Estas opciones le permiten anular la lista de tareas predeterminada y la configuración de tiempo de espera y asociar etiquetas con la ejecución del flujo de trabajo.

Por otra parte, el cliente asíncrono posee métodos que permiten la invocación asíncrona del método `@Execute`. Estas son las sobrecargas de métodos que se generan en la interfaz del cliente para el método `@Execute` de la interfaz de flujo de trabajo:

1. Una sobrecarga que toma los argumentos originales tal y como están. El tipo de retorno de esta sobrecarga será `Promise<Void>` si el método original ha devuelto `void`; de lo contrario, será `Promise<>`, tal y como se ha declarado en el método original. Por ejemplo:

   Método original:

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

   Método generado:

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

   Esta sobrecarga se debería utilizar cuando todos los argumentos del flujo de trabajo estén disponibles y no haya que esperar a que lo estén.

2. Una sobrecarga que toma los argumentos originales tal y como están y argumentos de variables adicionales del tipo `Promise<?>`. El tipo de retorno de esta sobrecarga será `Promise<Void>` si el método original ha devuelto `void`; de lo contrario, será `Promise<>`, tal y como se ha declarado en el método original. Por ejemplo:

   Método original:

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

   Método generado:

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

Esta sobrecarga se debería utilizar cuando todos los argumentos del flujo de trabajo estén disponibles y no haya que esperar a que lo estén, pero desee esperar a que otras promesas estén disponibles. Se puede utilizar el argumento de la variable para pasar los objetos `Promise<?>` que no se hayan declarado como argumentos, pero a los que desea esperar antes de ejecutar la llamada.

3. Una sobrecarga que toma los argumentos originales tal y como están, un argumento adicional del tipo `StartWorkflowOptions` y argumentos de variables adicionales del tipo `Promise<?>`. El tipo de retorno de esta sobrecarga será `Promise<Void>` si el método original ha devuelto `void`; de lo contrario, será `Promise<>`, tal y como se ha declarado en el método original. Por ejemplo:

   Método original:

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

   Método generado:

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

   Esta sobrecarga se debería utilizar cuando todos los argumentos del flujo de trabajo estén disponibles y no haya que esperar a que lo estén, cuando desee anular la configuración predeterminada que se utiliza para iniciar la ejecución del flujo de trabajo o cuando desee esperar a que otras promesas estén disponibles. Se puede utilizar el argumento de la variable para pasar los objetos `Promise<?>` que no se hayan declarado como argumentos, pero a los que desea esperar antes de ejecutar la llamada.

4. Una sobrecarga en la que cada argumento del método original se ha sustituido por un contenedor de `Promise<>`. El tipo de retorno de esta sobrecarga será `Promise<Void>` si el método original ha devuelto `void`; de lo contrario, será `Promise<>`, tal y como se ha declarado en el método original. Por ejemplo:

   Método original:

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

Método generado:

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

Esta sobrecarga se debería utilizar cuando los argumentos que se van a pasar a la ejecución del flujo de trabajo se vayan a evaluar de manera asíncrona. No se ejecutará una llamada a esta sobrecarga de métodos hasta que todos los argumentos que se van a pasar estén preparados.

Si algunos de los argumentos ya están preparados, conviértalos en un `Promise` que ya esté preparado a través del método `Promise.asPromise(`*`value`*`)`. Por ejemplo:

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

5. Una sobrecarga en la que cada argumento del método original se ha sustituido por un contenedor de `Promise<>`. La sobrecarga también tiene argumentos de variables adicionales del tipo `Promise<?>`. El tipo de retorno de esta sobrecarga será `Promise<Void>` si el método original ha devuelto `void`; de lo contrario, será `Promise<>`, tal y como se ha declarado en el método original. Por ejemplo:

   Método original:

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

   Método generado:

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

   Esta sobrecarga se debería utilizar cuando todos los argumentos que se van a pasar a la ejecución del flujo de trabajo se vayan a evaluar de manera asíncrona y también desee esperar

a que otras promesas estén disponibles. No se ejecutará una llamada a esta sobrecarga de métodos hasta que todos los argumentos que se van a pasar estén preparados.

6. Una sobrecarga en la que cada argumento del método original se ha sustituido por un contenedor de `Promise<?>`. La sobrecarga también tiene un argumento adicional del tipo `StartWorkflowOptions` y argumentos de variables del tipo `Promise<?>`. El tipo de retorno de esta sobrecarga será `Promise<Void>` si el método original ha devuelto `void`; de lo contrario, será `Promise<>`, tal y como se ha declarado en el método original. Por ejemplo:

   Método original:

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

   Método generado:

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

   Utilice esta sobrecarga cuando los argumentos que se van a pasar a la ejecución del flujo de trabajo se vayan a evaluar de manera asíncrona y desee anular la configuración predeterminada que se utiliza para iniciar la ejecución del flujo de trabajo. No se ejecutará una llamada a esta sobrecarga de métodos hasta que todos los argumentos que se van a pasar estén preparados.

También se genera un método correspondiente a cada señal de la interfaz de flujo de trabajo. Por ejemplo:

Método original:

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

Método generado:

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

El cliente asíncrono no contiene un método correspondiente al método anotado con `@GetState` en la interfaz original. Como la recuperación del estado requiere una llamada al servicio web, no es adecuada para su uso en un flujo de trabajo. Por tanto, solo se proporciona a través del cliente externo.

El autocliente está diseñado para utilizarse desde un flujo de trabajo para iniciar una nueva ejecución cuando termina la ejecución actual. Los métodos de este cliente son similares a los del cliente asíncrono, pero devuelven `void`. Este cliente no contiene métodos correspondiente a los métodos anotados con `@Signal` y `@GetState`. Para obtener más información, consulte Flujos de trabajo continuos.

Los clientes generados se obtienen de las interfaces básicas, `WorkflowClient` y `WorkflowClientExternal`, respectivamente, que proporcionan métodos que se pueden utilizar para cancelar o terminar la ejecución del flujo de trabajo. Para obtener más información sobre estas interfaces, consulte la documentación de AWS SDK para Java .

Los clientes generados le permiten interactuar con las ejecuciones de flujos de trabajo con establecimiento inflexible de tipos. La instancia de un cliente generado, una vez creada, se vincula a una ejecución de flujo de trabajo específica y solo se puede utilizar para esa ejecución. Asimismo, el marco de trabajo también ofrece clientes dinámicos que no son específicos de un tipo o ejecución de flujo de trabajo. Los clientes generados confían en este cliente a nivel profundo. También puede utilizar estos clientes directamente. Consulte la sección acerca de Clientes dinámicos

El marco de trabajo también genera fábricas para crear los clientes con establecimiento inflexible de tipo. Estas son las fábricas de clientes generadas para la interfaz `MyWorkflow` del ejemplo:

```
//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);
}
```

La interfaz `WorkflowClientFactory` básica es:

```
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);
}
```

Debería utilizar estas fábricas para crear instancias del cliente. La fábrica le permite configurar el cliente genérico (el cliente genérico se debería utilizar para proporcionar una implementación del cliente personalizada) y el `DataConverter` que utiliza el cliente para serializar los datos, además de las opciones que se utilizan para iniciar la ejecución del flujo de trabajo. Para obtener más información, consulte las secciones DataConverters y Ejecuciones de flujo de trabajo secundario. `StartWorkflowOptions` contiene la configuración que puede utilizar para reemplazar a la configuración predeterminada (por ejemplo, los tiempos de espera) que se especificó en el momento de realizar el registro. Para obtener más información sobre la `StartWorkflowOptions` clase, consulte la AWS SDK para Java documentación.

El cliente externo se puede utilizar para iniciar ejecuciones de flujos de trabajo desde fuera del ámbito de un flujo de trabajo, mientras que el cliente asíncrono se puede utilizar para iniciar una

ejecución de flujo de trabajo desde el código del interior de un flujo de trabajo. Para iniciar una ejecución, solo hay que utilizar el cliente generado para llamar al método que se corresponde con el método anotado con `@Execute` en la interfaz del flujo de trabajo.

El marco de trabajo también genera clases de implementación para las interfaces del cliente. Estos clientes crean y envían solicitudes a Amazon SWF para realizar la acción adecuada. La versión de cliente del `@Execute` método inicia una nueva ejecución de flujo de trabajo o crea una ejecución de flujo de trabajo secundaria mediante Amazon SWF APIs. Del mismo modo, la versión de cliente del `@Signal` método utiliza Amazon SWF APIs para enviar una señal.

> **Note**
>
> El cliente del flujo de trabajo externo se debe configurar con el cliente y el dominio de Amazon SWF. Puede utilizar el constructor de fábrica de clientes que los toma como parámetros o pasar una implementación de cliente genérica que ya esté configurada con el cliente y el dominio de Amazon SWF.
> El marco de trabajo recorre la jerarquía de tipos de la interfaz del flujo de trabajo y también genera interfaces de cliente para las interfaces de flujo de trabajo principales y las obtiene de ellas.

Clientes de actividad

De manera similar al cliente de flujo de trabajo, se genera un cliente por cada interfaz anotada con `@Activities`. Los artefactos generados incluyen una interfaz de cliente y una clase de cliente. La interfaz generada para la interfaz `@Activities` del ejemplo anterior (`MyActivities`) es la siguiente:

```
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);
}
```

La interfaz contiene un conjunto de métodos sobrecargados que se corresponden con cada método de actividad de la interfaz `@Activities`. Estas sobrecargas se ofrecen por comodidad y permiten llamar a las actividades de forma asíncrona. Por cada método de actividad de la interfaz de `@Activities`, se generan las siguientes sobrecargas de métodos en la interfaz de cliente:

1. Una sobrecarga que toma los argumentos originales tal y como están. El tipo de retorno de esta sobrecarga es `Promise<`*T*`>`, donde *T* es el tipo de retorno del método original. Por ejemplo:

   Método original:

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

   Método generado:

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

   Esta sobrecarga se debería utilizar cuando todos los argumentos del flujo de trabajo estén disponibles y no haya que esperar a que lo estén.

2. Una sobrecarga que toma los argumentos originales tal y como están, un argumento del tipo `ActivitySchedulingOptions` y argumentos de variables adicionales del tipo `Promise<?>`. El tipo de retorno de esta sobrecarga es `Promise<`*T*`>`, donde *T* es el tipo de retorno del método original. Por ejemplo:

   Método original:

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

   Método generado:

   ```
   Promise<Void> activity2(
     int foo,
   ```

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

Esta sobrecarga se debería utilizar cuando todos los argumentos del flujo de trabajo estén disponibles y no haya que esperar a que lo estén, cuando desee anular la configuración predeterminada o cuando desee esperar a que `Promise` adicionales estén preparados. Se pueden utilizar argumentos de variables para pasar los objetos `Promise<?>` adicionales que no se hayan declarado como argumentos, pero a los que desea esperar antes de ejecutar la llamada.

3. Una sobrecarga en la que cada argumento del método original se ha sustituido por un contenedor de `Promise<>`. El tipo de retorno de esta sobrecarga es `Promise<T>`, donde *T* es el tipo de retorno del método original. Por ejemplo:

   Método original:

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

   Método generado:

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

   Esta sobrecarga se debería utilizar cuando los argumentos que se van a pasar a la actividad se vayan a evaluar de manera asíncrona. No se ejecutará una llamada a esta sobrecarga de métodos hasta que todos los argumentos que se van a pasar estén preparados.

4. Una sobrecarga en la que cada argumento del método original se ha sustituido por un contenedor de `Promise<>`. La sobrecarga también tiene un argumento adicional del tipo `ActivitySchedulingOptions` y argumentos de variables del tipo `Promise<?>`. El tipo de retorno de esta sobrecarga es `Promise<T>`, donde *T* es el tipo de retorno del método original. Por ejemplo:

   Método original:

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

   Método generado:

   ```
   Promise<Void> activity2(
      Promise<Integer> foo,
   ```

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

Esta sobrecarga se debería utilizar cuando los argumentos que se van a pasar a la actividad se vayan a evaluar de manera asíncrona, cuando desee anular la configuración predeterminada registrada con el tipo o cuando desee esperar a que haya `Promise` adicionales preparados. No se ejecutará una llamada a esta sobrecarga de métodos hasta que todos los argumentos que se van a pasar estén preparados. La clase de cliente generada implementa esta interfaz. La implementación de cada método de interfaz crea y envía una solicitud a Amazon SWF para programar una tarea de actividad del tipo adecuado mediante Amazon SWF. APIs

5. Una sobrecarga que toma los argumentos originales tal y como están y argumentos de variables adicionales del tipo `Promise<?>`. El tipo de retorno de esta sobrecarga es `Promise<T>`, donde *T* es el tipo de retorno del método original. Por ejemplo:

   Método original:

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

   Método generado:

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

   Esta sobrecarga se debería utilizar cuando todos los argumentos de la actividad estén disponibles y no haya que esperar a que lo estén, pero desee esperar a que otros objetos `Promise` estén preparados.

6. Una sobrecarga en la que cada argumento del método original se ha sustituido por un contenedor `Promise` y argumentos de variables adicionales del tipo `Promise<?>`. El tipo de retorno de esta sobrecarga es `Promise<T>`, donde *T* es el tipo de retorno del método original. Por ejemplo:

   Método original:

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

   Método generado:

   ```
   Promise<Void> activity2(
   ```

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

Esta sobrecarga se debería utilizar cuando se vaya a esperar por todos los argumentos de la actividad de manera asíncrona y también desee esperar a que otros objetos `Promise` estén preparados. La llamada a esta sobrecarga de métodos se ejecuta de manera asíncrona cuando todos los objetos `Promise` que se han pasado están preparados.

El cliente de actividad generado también posee un método protegido que se corresponde con cada método de la actividad, denominado `{`*`activity method name`*`}Impl()`al que llaman todas las sobrecargas de la actividad. Puede anular este método para crear implementaciones de cliente simuladas. Este método toma como argumentos todos los argumentos del método original en contenedores `Promise<>`, `ActivitySchedulingOptions` y argumentos de variables del tipo `Promise<?>`. Por ejemplo:

Método original:

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

Método generado:

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

Opciones de programación

El cliente de actividad generado le permite pasar `ActivitySchedulingOptions` como argumento. La estructura `ActivitySchedulingOptions` contiene valores que determinan la configuración de la tarea de la actividad que programa el marco de trabajo en Amazon SWF. Esta configuración sustituye a la predeterminada que se especificó como opciones de registro. Para especificar opciones de programación de forma dinámica, cree un objeto `ActivitySchedulingOptions`, configúrelo como desee y páselo al método de la actividad. En el siguiente ejemplo, hemos especificado la lista de tareas que se debería utilizar para la tarea de actividad. De esta forma, se anula la lista de tareas registrada predeterminada para esta invocación de la actividad.

```
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);
    }
}
```

Clientes dinámicos

Además de los clientes generados, el marco de trabajo también dispone de clientes para fines generales, como `DynamicWorkflowClient` y `DynamicActivityClient`, que se pueden utilizar para iniciar dinámicamente ejecuciones de flujos de trabajo, enviar señales, programar actividades, etc. Por ejemplo, es posible que desee programar una actividad cuyo tipo no se conoce en el momento del diseño. Puede utilizar `DynamicActivityClient` para programar una tarea de actividad de ese tipo. De igual modo, puede programar dinámicamente una ejecución de flujo de trabajo secundaria utilizando `DynamicWorkflowClient`. En el siguiente ejemplo, el flujo de trabajo busca la actividad en una base de datos y utiliza el cliente de la actividad dinámica para programarlo:

```
//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);
}
```

Para obtener más información, consulte la AWS SDK para Java documentación.

Señalización y cancelación de ejecuciones de flujo de trabajo

El cliente del flujo de trabajo generado tiene métodos correspondientes a cada señal que se pueden enviar al flujo de trabajo. Puede utilizarlos desde el interior de un flujo de trabajo para enviar señales a otras ejecuciones de flujos de trabajo. Esto proporciona un mecanismo con tipo para enviar señales. Sin embargo, es posible que haya ocasiones en las que tenga que determinar el nombre de la señal de forma dinámica; por ejemplo, cuando el nombre de la señal se recibe en un mensaje. Puede utilizar el cliente de flujo de trabajo dinámico para enviar señales de forma dinámica a cualquier ejecución de flujo de trabajo. De igual modo, puede utilizar el cliente para solicitar la cancelación de otra ejecución de flujo de trabajo.

En el siguiente ejemplo, el flujo de trabajo busca la ejecución para enviar una señal desde una base de datos y envía la señal de forma dinámica utilizando el cliente de flujo de trabajo dinámico.

```
//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);
}
```

Implementación de flujos de trabajo

Para implementar un flujo de trabajo, deberá escribir una clase que implemente la interfaz `@Workflow` deseada. Por ejemplo, la interfaz de flujo de ejemplo (`MyWorkflow`) puede implementarse de la siguiente manera:

```
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);
  }
}
```

El método `@Execute` en esta clase es el punto de entrada de la lógica del flujo de trabajo. Como el marco utiliza la reproducción para reconstruir el estado del objeto cuando se va a procesar una tarea de decisión, se crea un objeto nuevo para cada tarea de decisión.

El uso de `Promise<T>` como parámetro no está permitido en el método `@Execute` dentro de una interfaz `@Workflow`. Esto se hace porque la realización de una llamada asíncrona es únicamente decisión del intermediario. La implementación de flujo de trabajo en sí no depende de si la invocación fue síncrona o asíncrona. Por tanto, la interfaz generada del cliente presenta sobrecargas que adoptan los parámetros `Promise<T>` para poder llamar a estos métodos de manera asíncrona.

El tipo de retorno de un método `@Execute` solo puede ser `void` o `Promise<T>`. Tenga en cuenta que un tipo de retorno del cliente externo correspondiente es `void` y no `Promise<>`. Como el cliente externo no está diseñado para usarse desde el código asíncrono, no devuelve objetos. `Promise` Para obtener resultados de ejecuciones de flujo de trabajo indicadas externamente, puede diseñar el flujo de trabajo para actualizar el estado en un almacén de datos externo a través de una actividad. La visibilidad de Amazon SWF también se APIs puede utilizar para recuperar el resultado de un flujo de trabajo con fines de diagnóstico. No se recomienda utilizar la visibilidad APIs para recuperar los resultados de las ejecuciones de flujos de trabajo como práctica general, ya que Amazon SWF puede limitar estas llamadas a la API. La visibilidad APIs requiere que identifique la ejecución del flujo de trabajo mediante una estructura. `WorkflowExecution` Puede obtener esta estructura del cliente de flujo de trabajo generado llamando al método `getWorkflowExecution`. Este método devolverá la estructura `WorkflowExecution` correspondiente a la ejecución del flujo de trabajo al que el cliente está vinculado. Consulte la referencia de la API de Amazon Simple Workflow Service para obtener más información sobre la visibilidad APIs.

Al llamar a actividades de su implementación de flujo de trabajo, debería utilizar el cliente de actividades generado. De manera parecida, para enviar señales, utilice los clientes de flujo de trabajo generados.

Contexto de la decisión

El marco de trabajo proporciona un contexto de ambiente siempre que el marco de trabajo ejecuta el código de flujo de trabajo. Este contexto proporciona funcionalidad específica para el contexto a la que puede obtener acceso en su implementación de flujo de trabajo, como crear un temporizador. Consulte la sección sobre Contexto de ejecución para obtener más información.

Exposición del estado de ejecución

Amazon SWF le permite añadir un estado personalizado en el historial de flujo de trabajo. El último estado notificado por la ejecución del flujo de trabajo se devuelve a través de llamadas de visibilidad al servicio de Amazon SWF y en la consola de Amazon SWF. Por ejemplo, en un flujo de trabajo de procesamiento de pedidos, puede informar sobre el estado del pedido en diferentes etapas, como por ejemplo "pedido recibido", "pedido enviado", etc. En AWS Flow Framework para Java, esto se hace mediante un método en la interfaz de flujo de trabajo anotado con la anotación `@GetState`. Cuando el decisor ha terminado de procesar una tarea de decisión, llama a este método para obtener el último estado de la implementación de flujo de trabajo. Además de las llamadas de visibilidad, el estado también puede recuperarse utilizando el cliente externo generado (que utiliza las llamadas a la API de visibilidad a nivel interno).

El siguiente ejemplo muestra cómo establecer el contexto de ejecución.

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

El cliente externo generado se puede utilizar para recuperar el último estado de la ejecución del flujo de trabajo en cualquier momento.

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

En el ejemplo de arriba, se informa del estado de ejecución en diferentes etapas. Cuando se inicia la instancia de flujo de trabajo, `periodicWorkflow` informa el estado inicial como "Just Started" (Recién iniciado). Cada llamada a `callPeriodicActivity` actualiza, a continuación, el

estado del flujo de trabajo. Una vez que se ha llamado a `activity1` 100 veces, el método devuelve resultados y se completa la instancia de flujo de trabajo.

Variables locales del flujo de trabajo

En ocasiones, es posible que tenga que usar variables estáticas en su implementación de flujo de trabajo. Por ejemplo, es posible que desee almacenar un contador al que se pueda obtener acceso desde diferentes lugares (posiblemente de diferentes clases) en la implementación del flujo de trabaje. No obstante, no puede confiar en variables estáticas en sus flujos de trabajo porque las variables estáticas se comparten entre subprocesos, lo cual es problemático porque un proceso de trabajo podría procesar diferentes tareas de decisión en diferentes subprocesos al mismo tiempo. De manera alternativa, puede almacenar dicho estado en un campo en la implementación de flujo de trabajo, pero luego tendrá que distribuir el objeto de implementación. Para abordar esta necesidad, el marco de trabajo proporciona una clase `WorkflowExecutionLocal<?>`. Cualquier estado que tenga que tener una semántica estática de tipo variables deberá mantenerse como variable local de instancias mediante `WorkflowExecutionLocal<?>`. Puede declarar y utilizar una variable estática de este tipo. Por ejemplo, en el siguiente fragmento, se utiliza una `WorkflowExecutionLocal<String>` para almacenar un nombre de usuario.

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

Implementación de actividades

Las actividades se implementan proporcionando una implementación de la interfaz `@Activities`. AWS Flow Framework Para Java, utiliza las instancias de implementación de actividades configuradas en el trabajador para procesar las tareas de la actividad en tiempo de ejecución. El proceso de trabajo busca automáticamente la implementación de la actividad del tipo apropiado.

Puede utilizar propiedades y campos para pasar recursos a instancias de actividad, como por ejemplo conexiones de bases de datos. Como se puede acceder al objeto de implementación de la actividad desde varios subprocesos, los recursos compartidos deben estar protegidos por subprocesos.

Tenga en cuenta que la implementación de la actividad no toma parámetros del tipo `Promise<>` ni devuelve objetos de ese tipo. Esto es así porque la implementación de la actividad no debería depender de cómo se invocó (de manera síncrona o asíncrona).

La interfaz de actividades mostrada con anterioridad puede implementarse de esta manera:

```
public class MyActivitiesImpl implements MyActivities {

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

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

```
}
```

Hay un contexto local de subproceso disponible para la implementación de la actividad que puede usarse para recuperar el objeto de la tarea, el objeto del convertidor de datos que se está usando, etc. Se puede obtener acceso al contexto actual a través de `ActivityExecutionContextProvider.getActivityExecutionContext()`. Para obtener más información, consulte la AWS SDK para Java documentación `ActivityExecutionContext` y la secciónContexto de ejecución.

Finalización manual de actividades

La anotación `@ManualActivityCompletion` en el ejemplo de arriba es una anotación opcional. Se permite solamente en métodos que implementan una actividad y se utiliza para configurar la actividad para que no se complete automáticamente cuando el método de la actividad devuelve resultados. Esto podría ser útil cuando desee completar la actividad de manera asíncrona, por ejemplo, manualmente una vez que se haya completado una acción realizada por una persona.

De manera predeterminada, el marco de trabajo considera que la actividad se ha completado cuando el método de la actividad devuelve resultados. Esto significa que el proceso de trabajo de actividad informa de la finalización de la tarea de actividad a Amazon SWF y le proporciona los resultados (si los hubiera). No obstante, hay casos de uso en los que no quiere que la tarea de la actividad se marque como completada cuando el método de la actividad devuelve resultados. Esto es útil especialmente cuando está modelando tareas humanas. Por ejemplo, el método de la actividad podría enviar un correo electrónico a una persona que debe completar un trabajo antes de que pueda completarse la tarea de la actividad. En estos casos, puede anotar el método de la actividad con la anotación `@ManualActivityCompletion` para indicar al proceso de trabajo de la actividad que no deberá completar la actividad automáticamente. Para completar la actividad manualmente, puede utilizar `ManualActivityCompletionClient` que se proporciona en el marco de trabajo o utilizar el método `RespondActivityTaskCompleted` en el cliente de Java de Amazon SWF que se proporciona en el SDK de Amazon SWF. Para obtener más información, consulte la AWS SDK para Java documentación.

Para completar la tarea de la actividad, tiene que proporcionar un token de tarea. Amazon SWF utiliza el token de tarea para identificar las tareas de forma única. Puede obtener acceso a este token desde `ActivityExecutionContext` en la implementación de la actividad. Debe pasar este token a la parte responsable de completar la tarea. Este token podrá recuperarse desde el `ActivityExecutionContext` llamando a `ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken()`.

La actividad `getName` del ejemplo de Hello World puede implementarse para enviar un correo electrónico para pedirle a alguien que proporcione un mensaje de saludo:

```
@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";
}
```

El siguiente objeto de código puede usarse para proporcionar el saludo y cerrar la tarea utilizando el `ManualActivityCompletionClient`. También puede producir un error en la tarea:

```
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);
    }
}
```

Implementación de AWS Lambda tareas

Temas

- Acerca de AWS Lambda
- Beneficios y limitaciones de la utilización de tareas de Lambda
- Uso de tareas Lambda en sus flujos de trabajo AWS Flow Framework para Java
- Ver la HelloLambda muestra

Acerca de AWS Lambda

AWS Lambda es un servicio informático totalmente gestionado que ejecuta el código en respuesta a eventos generados por un código personalizado o desde varios AWS servicios, como Amazon S3, DynamoDB, Amazon Kinesis, Amazon SNS y Amazon Cognito. Para obtener más información acerca de Lambda, consulte la Guía para desarrolladores de AWS Lambda.

Amazon Simple Workflow Service proporciona una tarea de Lambda que permite ejecutar funciones de Lambda en lugar de actividades de Amazon SWF tradicionales, o a la vez que ellas.

Important

Se cobrarán a su AWS cuenta las ejecuciones (solicitudes) de Lambda ejecutadas por Amazon SWF en su nombre. Para obtener más información sobre los precios de Lambda, consulte https://aws.amazon.com/lambda/ pricing/.

Beneficios y limitaciones de la utilización de tareas de Lambda

Hay una serie de beneficios derivados de la utilización de tareas de Lambda en lugar de una actividad de Amazon SWF tradicional:

- No hay necesidad de registrar ni crear versiones de tareas de Lambda a diferencia de los tipos de actividad de Amazon SWF.
- Puede utilizar cualquier función de Lambda que ya haya definido en sus flujos de trabajo.
- Amazon SWF llama directamente a las funciones de Lambda; no hay necesidad de implementar ningún proceso de trabajo para ejecutarlas, como ocurre con las actividades tradicionales.

- Lambda proporciona métricas y registros para el seguimiento y el análisis de las ejecuciones de las funciones.

También hay una serie de limitaciones relativas a las tareas Lambda que debe tener presente:

- Las tareas de Lambda solo se pueden ejecutar en AWS regiones que admiten Lambda. Para obtener más información sobre las regiones compatibles actualmente con Lambda, consulte Regiones y puntos de conexión de Lambda en la Referencia general de Amazon Web Services.
- Actualmente, las tareas Lambda solo se admiten en la API HTTP básica de SWF y en la AWS Flow Framework API para Java. Actualmente, no hay soporte para tareas Lambda en Ruby AWS Flow Framework .

Uso de tareas Lambda en sus flujos de trabajo AWS Flow Framework para Java

Existen tres requisitos para utilizar las tareas de Lambda en sus flujos de trabajo AWS Flow Framework para Java:

- Tener una función de Lambda que ejecutar. Puede utilizar cualquier función de Lambda que haya definido. Para obtener más información sobre cómo crear funciones de Lambda, consulte la Guía para desarrolladores de AWS Lambda.
- Tener un rol de IAM que permita obtener acceso para ejecutar funciones de Lambda desde los flujos de trabajo de Amazon SWF.
- Usar código para programar la tarea de Lambda desde el flujo de trabajo.

Configuración de un rol de IAM

Antes de poder invocar funciones de Lambda desde Amazon SWF, debe proporcionar un rol de IAM que permita tener acceso a Lambda desde Amazon SWF. Puede:

- elija un rol predefinido, Rol, AWSLambda para dar permiso a sus flujos de trabajo para invocar cualquier función de Lambda asociada a su cuenta.
- defina su propia política y el rol asociado para conceder a los flujos de trabajo permiso para invocar determinadas funciones de Lambda, especificadas por sus nombres ARNs de recursos de Amazon ().

Limitar permisos de un rol de IAM

Puede limitar los permisos de un rol de IAM que proporcione a Amazon SWF mediante las claves de contexto `SourceArn` y `SourceAccount` en su política de confianza para los recursos. Estas claves limitan el uso de una política de IAM para que solo se utilice en las ejecuciones de Amazon Simple Workflow Service que pertenezcan al ARN del dominio especificado. Si se utilizan ambas claves de contexto de condición global, tanto el valor `aws:SourceAccount` como la cuenta a la que se hace referencia en el valor `aws:SourceArn` deben utilizar el mismo ID de cuenta cuando se utilicen en la misma instrucción de política.

En el siguiente ejemplo, la clave de `SourceArn` contexto restringe el uso de la función de servicio de IAM únicamente en las ejecuciones de Amazon Simple Workflow Service que pertenezcan a `someDomain` la cuenta,. `123456789012`

- Declaración 1

  Director: `"Service": "swf.amazonaws.com"`

  Acción: `sts:AssumeRole`

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

En el siguiente ejemplo, la clave de `SourceAccount` contexto restringe el uso de la función de servicio de IAM únicamente en las ejecuciones de Amazon Simple Workflow Service en la cuenta,. `123456789012`

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

Proporcionar acceso a Amazon SWF para invocar cualquier rol de Lambda

Puede usar el rol predefinido, Role, para que sus flujos de trabajo de Amazon SWF puedan invocar cualquier AWSLambdafunción de Lambda asociada a su cuenta.

Para usar AWSLambda Role para dar acceso a Amazon SWF a fin de invocar funciones de Lambda

1. Abra la [consola de Amazon IAM](https://console.aws.amazon.com/iam/).
2. Elija Roles, y a continuación Create New Role.
3. Dé un nombre al rol, como `swf-lambda` y elija Next Step.
4. En Roles de servicio de AWS , seleccione Amazon SWF, y, a continuación, seleccione Paso siguiente.
5. En la pantalla Adjuntar política, seleccione AWSLambdaRol de la lista.
6. Elija Next Step y a continuación Create Role una vez que haya revisado el rol.

Definición de un rol de IAM para proporcionar acceso para invocar una función de Lambda concreta

Si desea proporcionar acceso para invocar una función de Lambda concreta desde un flujo de trabajo, tendrá que definir su propia política de IAM.

Cómo crear una política de IAM para dar acceso a una función de Lambda concreta

1. Abra la [consola de Amazon IAM](https://console.aws.amazon.com/iam/).
2. Elija Políticas, y a continuación Crear política.
3. Elija Copiar una política AWS gestionada y seleccione AWSLambdaRol en la lista. Se generará una política para usted. Tiene la opción de editar su nombre y descripción según sus necesidades.
4. En el campo Recurso del Documento de política, añada el ARN de las funciones de Lambda. Por ejemplo:
   - Recurso: `arn:aws:lambda:us-east-1:111122223333:function:hello_lambda_function`

Note

Para obtener una descripción completa de los procedimientos que se deben seguir para especificar recursos en un rol de IAM, consulte la información general sobre las políticas de IAM en la Uso de IAM.

5. Elija Create Policy para finalizar la creación de su política.

A continuación, puede seleccionar esta política al crear un nuevo rol de IAM, y utilizar ese rol para dar acceso para invocar los flujos de trabajo de Amazon SWF. Este procedimiento es muy similar a crear un rol con la política de AWSLambdarol. En su lugar, elija su propia política al crear el rol.

Cómo crear un rol de Amazon SWF mediante su política de Lambda

1. Abra la consola de Amazon IAM.
2. Elija Roles, y a continuación Create New Role.
3. Dé un nombre al rol, como `swf-lambda-function` y elija Next Step.
4. En Roles de servicio de AWS , seleccione Amazon SWF, y, a continuación, seleccione Paso siguiente.
5. En la pantalla Asociar política, seleccione de la lista la política específica de la función de Lambda.
6. Elija Next Step y a continuación Create Role una vez que haya revisado el rol.

Programación de una tarea de Lambda para su ejecución

Una vez que haya definido un rol de IAM que le permita invocar funciones de Lambda, podrá programarlas para su ejecución como parte del flujo de trabajo.

Note

Este proceso se demuestra plenamente en el HelloLambda ejemplo de AWS SDK para Java.

Cómo programar una tarea de Lambda para su ejecución

1. En la implementación de su flujo de trabajo, obtenga una instancia de `LambdaFunctionClient` llamando a `getLambdaFunctionClient()` en una instancia de `DecisionContext`.

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

2. Programe la tarea mediante el método `scheduleLambdaFunction()` en el `LambdaFunctionClient`; para ello, pásele el nombre de la función de Lambda que haya creado y cualquier dato de entrada para la tarea de Lambda.

```
// 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. En el iniciador de ejecución del flujo de trabajo, añada el rol de IAM de Lambda a las opciones de flujo de trabajo predeterminadas mediante `StartWorkflowOptions.withLambdaRole()` y, a continuación, pase las opciones al comenzar el flujo de trabajo.

```
// 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);
```

Ver la HelloLambda muestra

En el AWS SDK para Java, se incluye un ejemplo que proporciona una implementación de un flujo de trabajo que utiliza una tarea de Lambda. Para verlo y and/or ejecutarlo, descarga la fuente.

En el archivo README que se incluye con los HelloLambdaejemplos de Java se incluye una descripción completa de cómo compilar y ejecutar el AWS Flow Framework ejemplo.

Ejecución de programas escritos con AWS Flow Framework para Java

Temas

- WorkflowWorker
- ActivityWorker
- Modelo de subprocesos de proceso de trabajo
- Extensibilidad de proceso de trabajo

El marco de trabajo proporciona clases de procesos de trabajo para inicializar el tiempo de ejecución de AWS Flow Framework para Java y comunicarse con Amazon SWF. Para implementar un proceso de trabajo de flujo de trabajo o de actividad, tiene que crear e comenzar una instancia de una clase de proceso de trabajo. Estas clases de procesos de trabajo son responsables de la administración de las operaciones asíncronas en curso, de invocar métodos asíncronos que se desbloquean y de la comunicación con Amazon SWF. Pueden configurarse con implementaciones de flujo de trabajo y de actividad, el número de subprocesos, la lista de tareas para sondear, etc.

El marco de trabajo incluye dos clases de procesos de trabajo, una para actividades y otra para flujos de trabajo. Para ejecutar la lógica de flujo de trabajo, usted utiliza la clase `WorkflowWorker`. Del mismo modo, para actividades se usa la clase `ActivityWorker`. Estas clases sondean automáticamente Amazon SWF para detectar tareas de actividad e invocan los métodos apropiados en la implementación.

El siguiente ejemplo muestra cómo crear instancias de `WorkflowWorker` y comenzar a sondear tareas:

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

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

Los pasos básicos para crear una instancia del `ActivityWorker` y comenzar a sondear tareas son los siguientes:

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

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

Cuando quiera cerrar una actividad o decisor, la aplicación debería cerrar las instancias de las clases de procesos de trabajo que se estén utilizando, así como la instancia de cliente de Java de Amazon SWF. Esto garantizará que todos los recursos utilizados por las clases de procesos de trabajo se publican correctamente.

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

Para comenzar una ejecución, simplemente cree una instancia del cliente externo generado y llame al método `@Execute`.

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

WorkflowWorker

Tal y como sugiere el nombre, el uso previsto de esta clase de proceso de trabajo es la implementación de flujo de trabajo. Se configura con una lista de tareas y el tipo de implementación de flujo de trabajo. La clase del proceso de trabajo ejecuta un bucle para sondear tareas de decisión en la lista de tareas especificadas. Cuando se recibe una tarea de decisión, crea una instancia de la implementación de flujo de trabajo y llama al método `@Execute` para procesar la tarea.

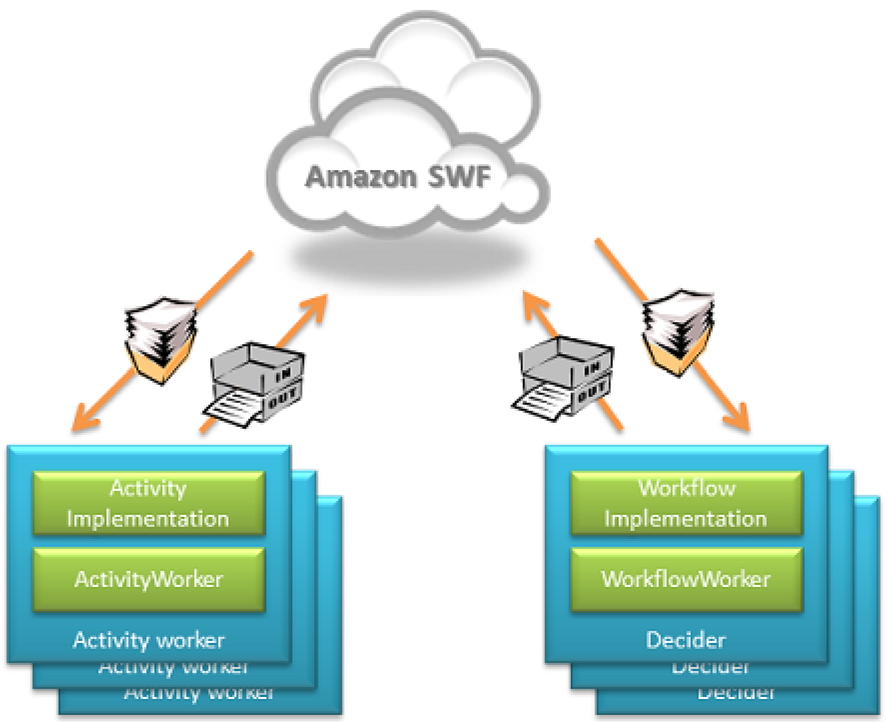
ActivityWorker

Para la implementación de procesos de trabajo de actividad, puede usar la clase `ActivityWorker` para sondear cómodamente una lista de tareas para tareas de actividad. Usted configura el proceso de trabajo de actividad con objetos de implementación de actividad. Esta clase de proceso de trabajo ejecuta un bucle para sondear tareas de actividad en la lista de tareas especificadas. Cuando se recibe una tarea de actividad, busca la implementación apropiada que proporcionó y llama al método de actividad para procesar la tarea. A diferencia de `WorkflowWorker`, que llama a la fábrica para crear una instancia nueva para cada tarea de decisión, el `ActivityWorker` simplemente utiliza el objeto que proporcionó.

La `ActivityWorker` clase utiliza las anotaciones de Java AWS Flow Framework para determinar las opciones de registro y ejecución.

Modelo de subprocesos de proceso de trabajo

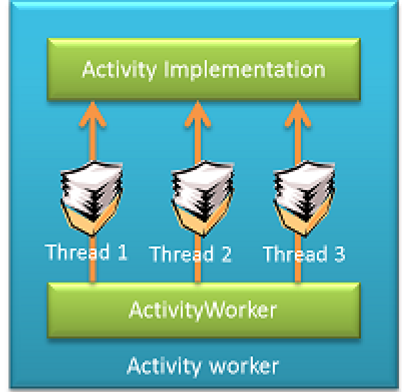
En el caso AWS Flow Framework de Java, la encarnación de una actividad o de un factor de decisión es un ejemplo de la clase obrera. Su aplicación es responsable de configurar y crear instancias del objeto del proceso de trabajo en cada máquina y proceso que deba actual como proceso de trabajo. A continuación, el objeto de proceso de trabajo recibe automáticamente tareas de Amazon SWF, las envía a la implementación de la actividad o del flujo de trabajo e informa sobre los resultados a Amazon SWF. Es posible que una sola instancia de flujo de trabajo incluya a muchos procesos de trabajo. Cuando Amazon SWF tiene una o más tareas de actividad pendientes, asigna una tarea al primer proceso de trabajo disponible, luego al siguiente y así sucesivamente. Esto permite que las tareas que pertenecen a la misma instancia de flujo de trabajo se procesen en diferentes procesos de trabajo simultáneamente.

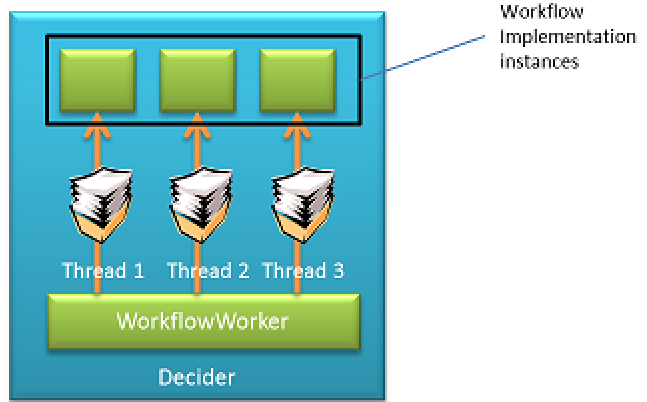


Es más, es posible configurar cada proceso de trabajo para que procese tareas en múltiples subprocesos. Esto significa que las tareas de actividad de una instancia de flujo de trabajo pueden ejecutarse simultáneamente incluso si solo hay un proceso de trabajo.

Las tareas de decisión actúan de manera similar, con la excepción de que Amazon SWF garantiza que para una ejecución de flujo de trabajo dada solo es posible ejecutar las decisiones de una en una. Una sola ejecución de flujo de trabajo exigirá habitualmente múltiples tareas de decisión; por tanto, también podría acabar ejecutando múltiples procesos y subprocesos. El decisor se configura con el tipo de implementación de flujo de trabajo. Cuando el decisor recibe una tarea de decisión, crea una instancia (objeto) de la implementación de flujo de trabajo. El marco de trabajo proporciona un patrón de fábrica extensible para la creación de estas instancias. La fábrica de flujo de trabajo predeterminada crear un objeto nuevo en cada ocasión. Puede proporcionar fábricas personalizadas para anular este comportamiento.

Al contrario de lo que ocurre con los decisores, que se configuran con tipos de implementación de flujo de trabajo, los procesos de trabajo de actividad se configuran con instancias (objetos) de implementaciones de actividad. Cuando un proceso de trabajo de actividad recibe una tarea de actividad, se envía al objeto de implementación de actividad apropiado.





El proceso de trabajo del flujo de trabajo mantiene un solo grupo de subprocesos y ejecuta el flujo de trabajo en el mismo subproceso que se utilizó para sondear la tarea en Amazon SWF. Como las actividades son de larga duración (al menos si se comparan con la lógica del flujo de trabajo), la clase Activity Worker mantiene dos grupos de subprocesos separados: uno para sondear Amazon SWF en relación con las tareas de actividad y otro para procesar las tareas mediante la ejecución de la implementación de la actividad. Esto le permite configurar el número de subprocesos para el sondeo de tareas por separado del número de subprocesos para ejecutarlos. Por ejemplo, puede tener un número pequeño de subprocesos para el sondeo y un número elevado de subprocesos para la ejecución de tareas. La clase de proceso de trabajo de actividad sondea Amazon SWF en busca de una tarea solamente cuando tiene un subproceso de sondeo libre y un subproceso libre para procesar la tarea.

Este comportamiento de subprocesos y creación de instancias implica lo siguiente:

1. Las implementaciones de actividad no tienen que tener estado. No deberá usar variables de instancia para almacenar el estado de la aplicación en objetos de actividad. Puede, no obstante, utilizar campos para almacenar recursos como por ejemplo conexiones de bases de datos.
2. Las implementaciones de actividad tienen que ser seguras para subprocesos. Como la misma instancia se puede usar para procesar tareas de diferentes subprocesos al mismo tiempo, el acceso a los recursos compartidos desde el código de la actividad debe estar sincronizado.
3. La implementación de flujo de trabajo puede ser con estado y es posible utilizar variables de instancia para almacenar el estado. Aunque se crea una instancia nueva de la implementación de flujo de trabajo para procesar cada tarea de decisión, el marco de trabajo se asegurará de que el estado se recrea correctamente. No obstante, la implementación de flujo de trabajo tiene que ser determinista. Consulte la sección Descripción de una tarea en AWS Flow Framework for Java para obtener más información.

4. Las implementaciones de flujo de trabajo no tienen que ser seguras para subprocesos cuando se utiliza la fábrica predeterminada. La implementación predeterminada garantiza que solo un subproceso utiliza una instancia de la implementación de flujo de trabajo a cada vez.

Extensibilidad de proceso de trabajo

La versión AWS Flow Framework para Java también contiene un par de clases de trabajo de bajo nivel que ofrecen un control pormenorizado y una mayor capacidad de ampliación. Al usarlas, puede personalizar por completo el registro de tipos de flujos de trabajo y de actividades, y establecer fábricas para la creación de objetos de implementación. Estos procesos de trabajo son `GenericWorkflowWorker` y `GenericActivityWorker`.

El `GenericWorkflowWorker` puede configurarse con una fábrica para la creación de fábricas de definición de flujo de trabajo. La fábrica de definición de flujo de trabajo es responsable de la creación de instancias de la implementación de flujo de trabajo y de proporcionar los ajustes de la configuración como las opciones de registro. Bajo circunstancias normales, debería utilizar la clase `WorkflowWorker` directamente. Creará y configurará automáticamente la implementación de las fábricas que se proporcionan en el marco de trabajo, `POJOWorkflowDefinitionFactoryFactory` y `POJOWorkflowDefinitionFactory`. La fábrica requiere que la clase de implementación de flujo de trabajo tenga un constructor sin ningún argumento. Este constructor se utiliza para crear instancias del objeto de flujo de trabajo en el tiempo de ejecución. La fábrica mira las anotaciones que ha utilizado en la interfaz de flujo de trabajo y la implementación para crear opciones de ejecución y registro apropiadas.

Podría proporcionar su propia implementación de las fábricas implementando `WorkflowDefinitionFactory`, `WorkflowDefinitionFactoryFactory` y `WorkflowDefinition`. La clase de proceso de trabajo utiliza la clase `WorkflowDefinition` para enviar tareas de decisión y señales. Al implementar estas clases de base, puede personalizar por completo la fábrica y el envío de solicitudes a la implementación de flujo de trabajo. Por ejemplo, puede usar estos puntos de extensibilidad para proporcionar un modelo de programación personalizado para la redacción de flujos de trabajo, por ejemplo, basado en sus propias anotaciones o generándolo a partir de WSDL en lugar del primer enfoque del código utilizado por el marco de trabajo. Para utilizar las fábricas personalizadas, tendrá que usar la clase `GenericWorkflowWorker`. Para obtener más información sobre estas clases, consulta la documentación. AWS SDK para Java

Del mismo modo, `GenericActivityWorker` le permite proporcionar una fábrica de implementación de actividad personalizada. Al implementar las clases

`ActivityImplementationFactory` y `ActivityImplementation` puede controlar por completo la creación de instancias de actividad así como personalizar las opciones de registro y ejecución. Para obtener más información sobre estas clases, consulte la AWS SDK para Java documentación.

Contexto de ejecución

Temas

- Contexto de la decisión
- Contexto de ejecución de actividad

El marco de trabajo proporciona un contexto de ambiente para las implementaciones de flujo de trabajo y de actividad. Este contexto es específico de la tarea que se está procesando y proporciona algunas utilidades que puede utilizar en su implementación. Cada vez que un proceso de trabajo procesa una tarea nueva se crea un objeto de contexto.

Contexto de la decisión

Cuando se ejecuta una tarea de decisión, el marco de trabajo proporciona el contexto para la implementación de flujo de trabajo a través de la clase `DecisionContext`. `DecisionContext` proporciona información contextual como el ID de ejecución de la ejecución del flujo de trabajo y funcionalidad de reloj y de temporizador.

Acceso DecisionContext en la implementación del flujo de trabajo

Puede obtener acceso a `DecisionContext` en la implementación de flujos de trabajo mediante la clase `DecisionContextProviderImpl`. Además, puede inyectar el contexto en un campo o propiedad de la implementación de flujo de trabajo utilizando Spring como se muestra en la sección correspondiente a capacidad de prueba e inyección de dependencia.

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

Creación de un reloj y un temporizador

El `DecisionContext` contiene una propiedad del tipo `WorkflowClock` que proporciona funcionalidad de reloj y temporizador. Como la lógica del flujo de trabajo debe ser determinista, no debe utilizar directamente el reloj del sistema en la implementación del flujo de trabajo. El método

`currentTimeMills` en el `WorkflowClock` devuelve la hora del evento de inicio de la decisión que se está procesando. Esto garantiza que obtiene el mismo valor de hora durante la reproducción, haciendo de esta manera que su lógica de flujo de trabajo sea determinista.

`WorkflowClock` también tiene un método de `createTimer` que devuelve un objeto `Promise` que estará listo después del intervalo especificado. Puede utilizar este valor como parámetro para otros métodos asíncronos para retrasar su ejecución el periodo de tiempo especificado. De esta manera, puede programar de manera efectiva un método asíncrono o una actividad para su ejecución posterior.

El ejemplo de la siguiente lista muestra cómo llamar periódicamente a una actividad.

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

En la lista de más arriba, el método asíncrono `callPeriodicActivity` llama a `activity1` y crea, a continuación, un temporizador utilizando el actual `AsyncDecisionContext`. Pasa la `Promise` devuelta como argumento a una llamada a sí mismo de manera recurrente. Esta llamada recurrente espera hasta que el temporizador se activa (1 hora en este ejemplo) antes de la ejecución.

Contexto de ejecución de actividad

Al igual que `DecisionContext` proporciona información contextual cuando se está procesando una tarea de decisión, `ActivityExecutionContext` proporciona información contextual similar cuando se está procesando una tarea de actividad. Este contexto está disponible para su código de actividad a través de la clase `ActivityExecutionContextProviderImpl`.

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

Utilizando `ActivityExecutionContext`, puede hacer lo siguiente:

Latido de una actividad de ejecución prolongada

Si la actividad es de ejecución prolongada, esta debe informar periódicamente de su progreso a Amazon SWF para dejar constancia de que la tarea sigue en curso. En ausencia de un latido de este tipo, podría agotarse el tiempo de espera de la tarea si se estableció el tiempo de espera del latido de una tarea en el registro del tipo de actividad o durante la programación de la actividad. Para enviar un latido, puede usar el método `recordActivityHeartbeat` en `ActivityExecutionContext`. El latido también proporciona un mecanismo de cancelación de las actividades en curso. Consulte la sección Gestión de errores para obtener más información y un ejemplo.

Obtenga detalles de la tarea de la actividad

Si lo desea, puede obtener todos los detalles de la tarea de actividad que pasó Amazon SWF cuando el ejecutor recibió la tarea. Esto incluye información sobre las entradas a la tarea, el tipo de tarea, el token de la tarea, etc. Si desea implementar una actividad que se completa manualmente, por ejemplo, mediante la acción de una persona, debe utilizar `ActivityExecutionContext` para recuperar el token de la tarea y pasarlo al proceso que acabará completando la tarea de actividad. Consulte la sección en Finalización manual de actividades para obtener más información.

Obtención del objeto de cliente de Amazon SWF que está utilizando el ejecutor

El objeto de cliente de Amazon SWF que está utilizando el ejecutor puede recuperarse llamando al método `getService` en `ActivityExecutionContext`. Esto es útil si se desea hacer una llamada directa al servicio de Amazon SWF.

Ejecuciones de flujo de trabajo secundario

En los ejemplos proporcionados hasta el momento, hemos comenzado la ejecución del flujo de trabajo directamente desde una aplicación. No obstante, la ejecución del flujo de trabajo puede comenzarse desde dentro de un flujo de trabajo mediante una llamada al método de punto de entrada del flujo de trabajo en el cliente generado. Cuando una ejecución de flujo de trabajo comienza desde el contexto de otra ejecución de flujo de trabajo se denomina ejecución de flujo de trabajo secundario. Esto le permite refactorizar flujos de trabajo complejos en unidades más pequeñas y, potencialmente, compartirlas entre diferentes flujos de trabajo. Por ejemplo, puede crear un flujo de trabajo de procesamiento de pagos y llamarlo desde un flujo de procesamiento de pedidos.

Semánticamente, la ejecución del flujo de trabajo secundario actúa de la misma manera que un flujo de trabajo individual salvo por las siguientes diferencias:

1. Cuando el flujo de trabajo principal termina debido a una acción explícita por parte del usuario, por ejemplo, llamando a la API `TerminateWorkflowExecution` de Amazon SWF, o se termina debido a que se agota el tiempo de espera, entonces el destino de la ejecución del flujo de trabajo secundario vendrá determinado por una política secundaria. Es posible configurar esta política secundaria para que termine, cancele o abandone (siga ejecutando) ejecuciones de flujo de trabajo secundarias.
2. El resultado del flujo de trabajo secundario (valor de retorno del método del punto de entrada) puede ser utilizado por la ejecución del flujo de trabajo principal de la misma manera que la `Promise<T>` devuelta por un método asíncrono. Esto es diferente de las ejecuciones independientes, en las que la aplicación debe obtener el resultado mediante Amazon APIs SWF.

En el siguiente ejemplo, el flujo de trabajo `OrderProcessor` crea un flujo de trabajo secundario `PaymentProcessor`:

```
@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);

}
```

Flujos de trabajo continuos

En algunos casos de uso, es posible que necesite un flujo de trabajo que se ejecute siempre o se ejecute durante periodos prolongados, por ejemplo, un flujo de trabajo que supervise el estado de una flota de servidores.

Note

Como Amazon SWF conserva el historial completo de la ejecución de un flujo de trabajo, el historial seguirá creciendo con el tiempo. El marco de trabajo recupera este historial de Amazon SWF cuando realiza una reproducción y esto será caro si el tamaño del historial es demasiado grande. En estos flujos de trabajo continuos o de ejecución prolongada, deberá cerrar periódicamente la actual ejecución y comenzar una nueva para seguir procesando.

Se trata de la continuación lógica de la ejecución de flujo de trabajo. El autocliente generado puede usarse para este fin. En la implementación de flujo de trabajo, simplemente llame al método `@Execute` en el autocliente. Una vez que se completa la actual ejecución, el marco de trabajo comenzará una ejecución nueva utilizando el mismo ID de flujo de trabajo.

También puede continuar la ejecución llamando al método `continueAsNewOnCompletion` en el `GenericWorkflowClient` que puede recuperar del actual `DecisionContext`. Por ejemplo, la siguiente implementación de flujo de trabajo establece un temporizador para disparar depués de un día y llama a su propio punto de entrada para comenzar una ejecución nueva.

```
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();
    }
}
```

Cuando un flujo de trabajo se llama a sí mismo de manera recurrente, el marco de trabajo cerrará el actual flujo de trabajo cuando se hayan completado todas las tareas pendientes y comience una nueva ejecución de flujo de trabajo. Tenga en cuenta que mientras haya tareas pendientes, la actual ejecución de flujo de trabajo no se cerrará. Esta nueva ejecución no heredará automáticamente ningún historial o datos de la ejecución original; si desea trasladar algún estado a una ejecución nueva, entonces debe pasarlo de manera explícita como entrada.

Configuración de la prioridad de las tareas en Amazon SWF

De forma predeterminada, las tareas de una lista de tareas se entregan en función de su hora de llegada: las tareas que se programan primero se suelen ejecutar primero, en la medida de lo posible. Al establecer una prioridad de las tareas opcional, puede dar prioridad a algunas tareas: Amazon SWF intentará realizar las tareas de prioridad más alta de una lista de tareas antes que las de prioridad más baja.

Puede establecer prioridades de las tareas tanto para flujos de trabajo como para actividades. La prioridad de las tareas de un flujo de trabajo no afectará a la prioridad de ninguna tarea de actividad que programe ni tampoco a ningún flujo de trabajo secundario que inicie. La prioridad predeterminada de una actividad o de un flujo de trabajo se establece (la establece el usuario, o bien Amazon SWF) durante el registro, y la prioridad de las tareas registrada siempre se utiliza a menos que se anule al programar la actividad o al iniciar una ejecución de flujo de trabajo.

Los valores de prioridad de las tareas pueden ir de "-2147483648" a "2147483647", con números más elevados que indican mayor prioridad. Si no establece la prioridad de las tareas para una actividad o flujo de trabajo, se asignará una prioridad de cero ("0").

Temas

- Establecimiento de prioridad de las tareas para flujos de trabajo
- Establecimiento de prioridad de las tareas para actividades

Establecimiento de prioridad de las tareas para flujos de trabajo

Puede establecer la prioridad de las tareas para un flujo de trabajo al registrarlo o iniciarlo. La prioridad de las tareas que se establece al registrarse el tipo de flujo de trabajo se usa como valor predeterminado de cualquier ejecución de flujo de trabajo de ese tipo, a menos que se anule al iniciar la ejecución de flujo de trabajo.

Para registrar un tipo de flujo de trabajo con una prioridad de tarea predeterminada, defina la defaultTaskPriorityopción WorkflowRegistrationOptionsal declararla:

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

Puede establecer la taskPriority para un flujo de trabajo al iniciarlo, anulando la prioridad de las tareas registradas (predeterminada).

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

PriorityWorkflowClientExternalFactory cf
    = new PriorityWorkflowClientExternalFactoryImpl(swfService, domain);

priority_workflow_client = cf.getClient();

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

Además, puede establecer la prioridad de las tareas al comenzar un flujo de trabajo secundario o continuar un flujo de trabajo como nuevo. Por ejemplo, puede configurar la opción TaskPriority en ContinueAsNewWorkflowExecutionParameterso en. StartChildWorkflowExecutionParameters

Establecimiento de prioridad de las tareas para actividades

Puede establecer la prioridad de las tareas para una actividad al registrarla o al programarla. La prioridad de las tareas que se establece al registrar un tipo de actividad se usa como prioridad predeterminada cuando se establece la actividad, a menos que se anule al programar la actividad.

Para registrar un tipo de actividad con una prioridad de tarea predeterminada, defina la defaultTaskPriorityopción ActivityRegistrationOptionsal declararla:

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

Puede establecer la taskPriority para una actividad al programarla, anulando la prioridad de las tareas registradas (predeterminada).

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

ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();

activityClient.doSomethingImportant(activityOptions);
```

DataConverters

Cuando su implementación de flujo de trabajo llama a una actividad remota, las entradas que se le han pasado y los resultados de la ejecución de la actividad se deben serializar para que puedan enviarse por cable. El marco usa la DataConverter clase para este propósito. Se trata de una clase abstracta que puede implementar para proporcionar su propio serializador. Se proporciona una implementación predeterminada basada en el serializador Jackson, `JsonDataConverter`, en el marco de trabajo. Para obtener más información, consulte la documentación de AWS SDK para Java. Consulte la documentación del procesador Jackson JSON para obtener información sobre cómo realiza Jackson la serialización así como anotaciones Jackson que pueden usarse para influir en ella. El formato del cable utilizado se considera parte del contrato. Por lo tanto, puede

especificar un `DataConverter` en sus interfaces de actividades y de flujos de trabajo estableciendo la propiedad `DataConverter` de las anotaciones `@Activities` y `@Workflow`.

El marco de trabajo creará objetos del tipo `DataConverter` especificado en la anotación `@Activities` para serializar la entradas a la actividad y para deserializar sus resultados. Del mismo modo, los objetos del tipo `DataConverter` que especifique en la anotación `@Workflow` se utilizarán para serializar los parámetros que pase al flujo de trabajo y en el caso de flujo de trabajo secundarios, para deserializar el resultado. Además de las entradas, el marco de trabajo también pasa datos adicionales a Amazon SWF, por ejemplo, detalles de la excepción. El serializador de flujo de trabajo también se utilizará para serializar estos datos.

También puede proporcionar una instancia del `DataConverter` si no desea que el marco de trabajo lo cree automáticamente. Los clientes generados tienen sobrecargas del constructor que toman un `DataConverter`.

Si no especifica un tipo `DataConverter` y no pasa un objeto `DataConverter`, se usará el `JsonDataConverter` de manera predeterminada.

Paso de datos a los métodos asíncronos

Temas

- Paso de colecciones y mapas a métodos asíncronos
- Settable <T>
- @NoWait
- Promise <Void>
- AndPromise y OrPromise

El uso de `Promise<T>` se ha explicado en las secciones anteriores. Se tratan aquí algunos casos de uso avanzados de `Promise<T>`.

Paso de colecciones y mapas a métodos asíncronos

El marco de trabajo admite el paso de matrices, colecciones y mapas como tipos `Promise` a métodos asíncronos. Por ejemplo, un método asíncrono puede tomar `Promise<ArrayList<String>>` como un argumento tal y como se muestra en la siguiente lista.

```
@Asynchronous
```

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

Semánticamente, actúa como cualquier otro parámetro escrito `Promise` y el método asíncrono esperará hasta que la colección esté disponible antes de la ejecución. Si los miembros de una colección son objetos `Promise`, entonces puede hacer que el marco de trabajo espere a que todos los miembros estén preparados tal y como se muestra en el siguiente fragmento de código. Esto hará que el método asíncrono espere hasta que cada miembro de la colección esté disponible.

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

Tenga en cuenta que debe usarse la anotación `@Wait` en el parámetro para indicar que contiene objetos `Promise`.

Tenga en cuenta también que la actividad `printActivity` toma un argumento `String` pero el método correspondiente en el cliente generado toma una Promise<String>. Estamos llamando el método en el cliente y no invocando directamente el método de la actividad.

Settable <T>

`Settable<T>` es un tipo derivado de `Promise<T>` que proporciona un método establecido que le permite establecer manualmente el valor de una `Promise`. Por ejemplo, el siguiente flujo de trabajo espera la recepción de una señal esperando en `Settable<?>`, que se establece en el método de señal:

```
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;
    }
}
```

También se puede encadenar `Settable<?>` a otra promesa individualmente. Puede utilizar `AndPromise` y `OrPromise` para agrupar promesas. Puede desencadenar un objeto `Settable` encadenado llamando al método `unchain()` que hay en él. Cuando está encadenado, `Settable<?>` está listo automáticamente cuando la promesa a la que está encadenado está lista. El encadenado es especialmente útil cuando desea utilizar una promesa devuelta desde dentro del alcance de `doTry()` en otras partes de su programa. Como `TryCatchFinally` se usa como una clase anidada, no puedes declarar a `Promise<>` en el ámbito del padre y configurarla. `doTry()` Esto es porque Java requiere que las variables se declaren en el ámbito principal y se utilicen en clases anidadas para marcarlas como finales. Por ejemplo:

```
@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;
}
```

Un `Settable` puede encadenarse a una promesa a cada vez. Puede desencadenar un objeto `Settable` encadenado llamando al método `unchain()` que hay en él.

@NoWait

Cuando pasa una `Promise` a un método asíncrono, de manera predeterminada, el marco de trabajo esperará a que la(s) `Promise`(s) estén listas antes de ejecutar el método (salvo para los tipos de colección). Puede anular este comportamiento utilizando la anotación `@NoWait` en parámetros en la declaración del método asíncrono. Esto es útil si está pasando `Settable<T>`, que será establecido por el método asíncrono.

Promise <Void>

Las dependencias en métodos asíncronos se implementan pasando la `Promise` devuelta por un método como argumento a otro. No obstante, hay casos en los que desea devolver `void` de un método, pero sigue queriendo que otros métodos asíncronos se ejecuten una vez finalizados. En esos casos, puede utilizar `Promise<Void>` como el tipo de retorno del método. La clase `Promise` proporciona un método `Void` estático que puede utilizar para crear un objeto `Promise<Void>`. Esta `Promise` estará lista cuando el método asíncrono finalice la ejecución. Puede pasar esta `Promise` a otro método asíncrono igual que cualquier otro objeto `Promise`. Si está utilizando `Settable<Void>`, entonces llame al método establecido en él con "null" para hacer que esté listo.

AndPromise y OrPromise

Con `AndPromise` y `OrPromise` puede agrupar múltiples objetos `Promise<>` en una sola promesa lógica. Una `AndPromise` está lista una vez que todas las promesas utilizadas para construirla están listas. Una `OrPromise` está lista una vez cuando cualquier promesa de la colección de promesas que se han utilizado para construirla está lista. Puede llamar a `getValues()` en `AndPromise` y `OrPromise` para recuperar la lista de valores de las promesas constituyentes.

Capacidad de realización de pruebas e inserción de dependencias

Temas

- Integración con Spring
- JUnit Integration

El marco se ha diseñado para que sea fácil de utilizar con Inversion of Control (IoC). Las implementaciones de actividades y flujos de trabajo, así como los trabajos y los objetos de contexto que proporciona el marco, se pueden configurar usando contenedores como Spring, y también se pueden crear instancias de ellos. De manera predeterminada, el marco permite integrar el marco Spring. Además, se JUnit ha proporcionado la integración con las implementaciones de flujos de trabajo y actividades de pruebas unitarias.

Integración con Spring

El paquete com.amazonaws.services.simpleworkflow.flow.spring contiene clases que facilitan el uso del marco Spring en sus aplicaciones. Estas incluyen un ámbito personalizado y trabajos de actividades y flujos de trabajo específicos de Spring: `WorkflowScope`, `SpringWorkflowWorker` y `SpringActivityWorker`. Estas clases le permiten configurar sus implementaciones de flujos de trabajo y actividades, así como trabajos, íntegramente a través de Spring.

WorkflowScope

`WorkflowScope` es una implementación personalizada del ámbito de Spring que proporciona el marco. Este ámbito le permite crear objetos en el contenedor Spring cuya vida útil está determinada por la de una tarea de decisión. Siempre que el trabajo recibe una nueva tarea de decisión, se crean instancias de los bean en este ámbito. Debe utilizar este ámbito para los bean de implementación de flujos de trabajo, así como cualquier otro bean del que dependa. Los ámbitos singleton y prototype que proporciona Spring no se deben utilizar para los bean de implementación de flujos de trabajo, ya que el marco obliga a crear un nuevo bean para cada tarea de decisión. No hacerlo dará lugar a un comportamiento inesperado.

En el siguiente ejemplo se muestra un fragmento de código de la configuración de Spring que registra `WorkflowScope` y, a continuación, lo utiliza para configurar un bean de implementación de flujo de trabajo y un bean de cliente de actividad.

```
<!-- 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>
```

La línea de configuración `<aop:scoped-proxy proxy-target-class="false" />`, que se utiliza en la configuración del bean `workflowImpl`, es necesaria, ya que `WorkflowScope` no permite usar un proxy mediante CGLIB. Deberá utilizar esta configuración para cualquier bean de `WorkflowScope` que esté conectado a otro bean de un ámbito distinto. En este caso, el bean `workflowImpl` necesita estar conectado a un bean de proceso de trabajo de un flujo de trabajo que esté en un ámbito singleton (a continuación podrá ver el ejemplo completo).

Puede obtener más información acerca del uso de ámbitos personalizados en la documentación del marco Spring.

Trabajos específicos de Spring

Cuando use Spring deberá usar las clases de trabajo específicas de Spring que proporciona el marco: `SpringWorkflowWorker` y `SpringActivityWorker`. Estos trabajos se pueden insertar en su aplicación usando Spring, tal como se muestra en el siguiente ejemplo. Los trabajos preparados para Spring implementan la interfaz `SmartLifecycle` de Spring y, de manera predeterminada, comienzan automáticamente a sondear tareas cuando se inicia el contexto de Spring. Puede desactivar esta funcionalidad estableciendo la propiedad `disableAutoStartup` del trabajo en `true`.

En el siguiente ejemplo se muestra cómo configurar un decisor. En este ejemplo se utilizan las interfaces `MyActivities` y `MyWorkflow` (que no se muestran aquí) y las correspondientes implementaciones (`MyActivitiesImpl` y `MyWorkflowImpl`). Las interfaces de cliente generadas y las implementaciones son `MyWorkflowClient`/`MyWorkflowClientImpl` y `MyActivitiesClient`/`MyActivitiesClientImpl` (que tampoco se muestran aquí).

El cliente de actividades se inserta en la implementación del flujo de trabajo usando la característica de autoconexión de Spring:

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

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

La configuración del decisor de Spring es la siguiente:

```
<?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>
```

Como `SpringWorkflowWorker` está completamente configurado en Spring y comienza a sondear automáticamente cuando se inicializa el contexto de Spring, el proceso de alojamiento para el decisor es sencillo:

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

Asimismo, el proceso de trabajo de la actividad se puede configurar del siguiente modo:

```
<?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>
```

El proceso de host del trabajo de la actividad es similar al del decisor:

```
public class ActivityHost {
   public static void main(String[] args) {
```

```
        ApplicationContext context = new FileSystemXmlApplicationContext(
        "resources/spring/ActivityHostBean.xml");
        System.out.println("Activity worker started");
    }
}
```

Inserción del contexto de decisión

Si su implementación del flujo de trabajo depende de los objetos de contexto, estos también se pueden insertar fácilmente a través de Spring. El marco registra automáticamente los bean relacionados con el contexto en el contenedor de Spring. Por ejemplo, en el siguiente fragmento de código, los distintos objetos de contexto se han conectado automáticamente. No es necesario realizar ninguna otra configuración de objetos de contexto de Spring.

```
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();
   }
}
```

Si desea configurar los objetos de contexto en la implementación del flujo de trabajo mediante la configuración XML de Spring, utilice los nombres de bean declarados en la clase `WorkflowScopeBeanNames` del paquete com.amazonaws.services.simpleworkflow.flow.spring. Por ejemplo:

```
<!-- 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>
```

También puede insertar un `DecisionContextProvider` en el bean de implementación de flujo de trabajo y utilizarlo para crear el contexto. Esto puede resultar útil si desea proporcionar implementaciones personalizadas del proveedor y contexto.

Inserción de recursos en las actividades

Puede crear instancias de implementaciones de actividades y configurarlas usando un contenedor Inversion of Control (IoC), y también puede insertar fácilmente recursos, como conexiones de bases de datos, si se declaran como propiedades de la clase de implementación de actividad. Estos recursos suelen ir en el ámbito de singleton. Tenga en cuenta que el trabajo de la actividad llama a sus implementaciones en varios subprocesos. Por tanto, debe sincronizarse el acceso a los recursos compartidos.

JUnit Integration

El marco proporciona JUnit extensiones e implementaciones de prueba de los objetos de contexto, como un reloj de prueba, que puede utilizar para escribir y ejecutar pruebas unitarias. JUnit Con estas extensiones, puede probar la implementación de su flujo de trabajo insertada y localmente.

Escritura de una prueba unitaria simple

Para escribir pruebas para su flujo de trabajo, utilice la clase `WorkflowTest` del paquete com.amazonaws.services.simpleworkflow.flow.junit. Esta clase es una JUnit `MethodRule` implementación específica del marco y ejecuta el código del flujo de trabajo de forma local, llamando a las actividades en línea en lugar de hacerlo a través de Amazon SWF. Esto le ofrece la flexibilidad de realizar pruebas con la frecuencia que desee, sin generar cargos.

Para utilizar esta clase, solo tiene que declarar un campo de tipo `WorkflowTest` y anotarlo con `@Rule`. Antes de realizar las pruebas, cree un nuevo objeto `WorkflowTest` y añádale sus implementaciones de actividades y flujos de trabajo. Después podrá utilizar la fábrica de clientes de flujo de trabajo generada para crear un cliente y comenzar a ejecutar el flujo de trabajo. El marco

también proporciona un JUnit ejecutor personalizado`FlowBlockJUnit4ClassRunner`, que debe utilizar para las pruebas de flujo de trabajo. Por ejemplo:

```
@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);
    }
}
```

Asimismo, puede especificar una lista de tareas independiente para cada implementación de actividades que añada a `WorkflowTest`. Por ejemplo, si tiene una implementación de un flujo de

trabajo que programa las actividades en listas de tareas específicas para cada host, podrá registrar la actividad en la lista de tareas de cada host:

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

Tenga en cuenta que el código en `@Test` es asíncrono. Por tanto, deberá utilizar el cliente de flujo de trabajo asíncrono para comenzar una ejecución. Con objeto de verificar los resultados de su prueba, también se proporciona una clase de ayuda `AsyncAssert`. Esta clase le permite esperar a que las promesas estén disponibles antes de comprobar los resultados. En este ejemplo, esperamos a que esté disponible el resultado de la ejecución del flujo de trabajo antes de comprobar el resultado de la prueba.

Si utiliza Spring, es posible usar la clase `SpringWorkflowTest` en lugar de la clase `WorkflowTest`. `SpringWorkflowTest` proporciona propiedades para configurar fácilmente implementaciones de actividades y flujos de trabajo a través de la configuración de Spring. Al igual que los trabajos específicos de Spring, debe utilizar `WorkflowScope` para configurar los bean de implementación de flujos de trabajo. Esto garantiza que se cree un nuevo bean de implementación de flujo de trabajo para cada tarea de decisión. Asegúrese de configurar estos beans con la proxy-target-class configuración de scoped-proxy establecida en. `false` Para obtener más información, consulte la sección Integración con Spring. Es posible cambiar el ejemplo de configuración de Spring que se muestra en la sección Integración con Spring para comprobar el flujo de trabajo usando `SpringWorkflowTest`:

```
<?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>
```

Simulación de implementaciones de actividades

Puede utilizar implementaciones de actividades reales durante las pruebas, pero si desea realizar una prueba unitaria solo de la lógica del flujo de trabajo, debe hacer una simulación de las actividades. Puede hacerlo proporcionando una implementación simulada de la interfaz de actividades en la clase `WorkflowTest`. Por ejemplo:

```
@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);
    }
}
```

También puede proporcionar una implementación simulada del cliente de actividades e insertarla en la implementación de su flujo de trabajo.

Objetos de contexto de prueba

Si la implementación del flujo de trabajo depende de los objetos de contexto del marco de trabajo (por ejemplo, `DecisionContext`), no tiene que hacer nada especial para probar dichos flujos de trabajo. Cuando se realiza una prueba con `WorkflowTest`, este inserta automáticamente objetos de contexto de prueba. Cuando la implementación del flujo de trabajo acceda a los objetos de contexto, por ejemplo, mediante `DecisionContextProviderImpl`,

se obtendrá la implementación de prueba. Puede manipular estos objetos de contexto de prueba en el código de prueba (método `@Test`) para crear casos de prueba interesantes. Por ejemplo, si su flujo de trabajo crea un temporizador, puede hacer que se active llamando al método `clockAdvanceSeconds` en la clase `WorkflowTest` para hacer adelantar el reloj. También puede adelantar el reloj para que los temporizadores se activen antes de lo normal usando la propiedad `ClockAccelerationCoefficient` en `WorkflowTest`. Por ejemplo, si su flujo de trabajo crea un temporizador para una hora, puede establecer `ClockAccelerationCoefficient` en 60 para hacer que se active al cabo de un minuto. De forma predeterminada, `ClockAccelerationCoefficient` está establecido en 1.

Para obtener más información acerca de los paquetes com.amazonaws.services.simpleworkflow.flow.test y com.amazonaws.services.simpleworkflow.flow.junit, consulte la documentación de AWS SDK para Java .

Gestión de errores

Temas

- TryCatchFinally Semántica
- Cancelación
- Anidado TryCatchFinally

La construcción `try/catch/finally` en Java facilita el control de errores y se utiliza de manera generalizada. Le permite asociar controladores de errores a un bloque de código. A nivel interno, esto se produce introduciendo metadatos adicionales sobre los controladores de errores en la pila de llamadas. Cuando se genera una excepción, el tiempo de ejecución busca en la pila de llamadas un controlador de errores asociado para invocarlo y, si no encuentra ninguno adecuado, propaga la excepción hacia arriba en la cadena de llamadas.

Esto funciona correctamente en el caso de código sincrónico, pero controlar los errores en programas asíncronos y distribuidos es más complicado. Como una llamada asíncrona se devuelve inmediatamente, la persona que llama no está en la pila de llamadas cuando se ejecuta el código asíncrono. Esto significa que el intermediario no puede controlar de la manera habitual las excepciones no controladas en el código asíncrono. Normalmente, las excepciones que se generan en el código asíncrono se controlan transfiriendo un estado de error a una devolución de llamada que se transfiere al método asíncrono. Por otro lado, si se utiliza `Future<?>`, notificará un error cuando

intente obtener acceso a él. No es la solución idónea, ya que el código que recibe la excepción (la devolución de llamada o el código que utiliza `Future<?>`) no tiene el contexto de la llamada original y quizás no pueda controlar la excepción correctamente. Además, en un sistema asíncrono distribuido, donde hay componentes que se ejecutan simultáneamente, se puede producir más de un error a la vez. Estos errores pueden ser de varios tipos y tener distintos niveles de gravedad, por lo que es preciso controlarlos de manera adecuada.

Tampoco es tarea fácil limpiar recursos después de una llamada asíncrona. A diferencia del código sincrónico, no se puede usar try/catch/finally en el código de llamada para limpiar recursos, ya que el trabajo iniciado en el bloque try puede continuar cuando se ejecute el bloque final.

El marco proporciona un mecanismo que hace que la gestión de errores en el código asíncrono distribuido sea similar y casi tan simple como la de Java. try/catch/finally

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

La clase `TryCatchFinally` y sus variantes, `TryFinally` y `TryCatch`, funcionan de un modo similar a `try/catch/finally` de Java. Al utilizarla puede asociar controladores de excepciones a bloques de código de flujo de trabajo que se pueden ejecutar como tareas asíncronas y remotas. El método `doTry()` es lógicamente equivalente al bloque `try`. El marco ejecuta automáticamente el código en `doTry()`. Se puede transferir una lista de objetos `Promise` al constructor de `TryCatchFinally`. El método `doTry` se ejecutará cuando estén listos todos los objetos `Promise` que se hayan transferido al constructor. Si genera una excepción mediante código invocado de manera asíncrona desde dentro de `doTry()`, las tareas pendientes que haya en `doTry()` se cancelan y se llama a `doCatch()` para controlar la excepción. Por ejemplo, en la lista anterior, si `downloadImage` genera una excepción, se cancelarán `createThumbnail` y `uploadImage`. Por último, se llama a `doFinally()` cuando se ha llevado a cabo todo el trabajo asíncrono (completado, erróneo o cancelado). Se puede utilizar para limpiar recursos. También es posible anidar estas clases en función de sus necesidades.

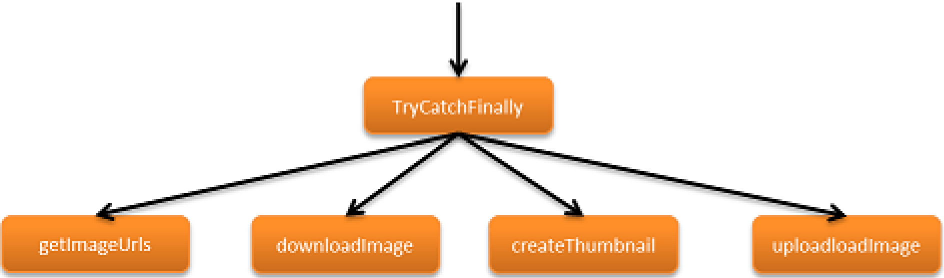
Cuando se notifica una excepción en `doCatch()`, el marco ofrece una pila de llamada lógica completa que incluye llamadas asíncronas y remotas. Esto puede resultar útil para el proceso de depuración, especialmente si hay métodos asíncronos que llaman a otros métodos asíncronos. Por ejemplo, una excepción de downloadImage generará una excepción de este tipo:

```
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 Semántica

La ejecución de un programa AWS Flow Framework para Java se puede visualizar como un árbol de ramas que se ejecutan simultáneamente. Si se llama a un método asíncrono, a una actividad y al

propio `TryCatchFinally`, se crea una nueva rama en este árbol de ejecuciones. Por ejemplo, en la siguiente figura se puede ver el flujo de trabajo de procesamiento de imágenes en forma de árbol.



Si se produce un error en una rama de ejecución, la rama volverá atrás, al igual que una excepción hace que la pila de la llamada vuelva hacia atrás en un programa de Java. El proceso de marcha atrás sigue avanzando por la rama de ejecución hasta que, bien se controla el error, o se llega a la raíz del árbol, en cuyo caso finalizaría la ejecución del flujo de trabajo.

El marco de trabajo notifica los errores que se producen al procesar las tareas como excepciones. Asocia los controladores de excepciones (métodos `doCatch()`) definidos en `TryCatchFinally` con todas las tareas que crea el código en el correspondiente `doTry()`. Si una tarea da error, por ejemplo, porque se agote el tiempo de espera o porque haya una excepción sin gestionar, se generará la excepción pertinente y se invocará al `doCatch()` correspondiente para gestionarla. Para ello, el marco de trabajo funciona de forma conjunta con Amazon SWF para propagar los errores remotos y recuperarlos como excepciones en el contexto de quien realiza la llamada.

Cancelación

Cuando se produce una excepción en el código sincrónico, el control pasa directamente al bloque `catch`, omitiendo el resto del código en el bloque `try`. Por ejemplo:

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

En este código, si `b()` genera una excepción, nunca se invocará a `c()`. Comparémoslo con un flujo de trabajo:

```
new TryCatch() {

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

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

En este caso, las llamadas a `activityA`, `activityB` y `activityC` devuelven todas valores correctamente y hacen que se creen tres tareas que se ejecutarán de forma asíncrona. Supongamos que posteriormente la tarea para `activityB` genera un error. Amazon SWF registra este error en el historial. Para controlar el error, en primer lugar el marco intentará cancelar el resto de las tareas que se originaron en el ámbito del mismo `doTry()` (en este caso, `activityA` y `activityC`). Una vez finalizadas todas estas tareas (canceladas, erróneas o completadas correctamente), se invocará al método `doCatch()` adecuado para controlar el error.

Al contrario que en el ejemplo del código sincrónico, donde nunca se ejecutó `c()`, se invocó a `activityC` y se programó una tarea para su ejecución. El marco intentará cancelarla, pero no hay garantías de que se cancele. No se puede garantizar que se cancele, porque puede que la actividad ya se haya completado, no haya tenido en cuenta la solicitud de cancelación o haya dado error. Sin embargo, el marco garantiza que se llame al método `doCatch()` únicamente cuando hayan finalizado las tareas iniciadas desde el correspondiente método `doTry()`. También garantiza que se llame al método `doFinally()` solo cuando hayan finalizado todas las tareas iniciadas desde los métodos `doTry()` y `doCatch()`. Si, por ejemplo, las actividades del ejemplo anterior dependen una de otra, supongamos que `activityB` depende de `activityA` y `activityC` de `activityB`, la cancelación de `activityC` será inmediata porque no se programará en Amazon SWF hasta que finalice `activityB`:

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

Latido de actividades

El mecanismo AWS Flow Framework de cancelación cooperativa de Java permite cancelar sin problemas las tareas que se realizan durante el vuelo. Cuando se pone en marcha una cancelación, se cancelan automáticamente las tareas que estaban bloqueadas o que estaban en espera de ser asignadas a un trabajo. Sin embargo, si ya se ha asignado la tarea a un trabajo, el marco solicitará que se cancele la actividad. La implementación de la actividad debe controlar de forma explícita estas solicitudes de cancelación. Esto se realiza a través de la notificación de los latidos de su actividad.

La notificación de los latidos permite a la implementación de la actividad notificar el progreso de una tarea de actividad en curso, lo que resulta muy útil para monitorizar y permite a la actividad comprobar la existencia de solicitudes de cancelación. El método `recordActivityHeartbeat` generará una excepción `CancellationException` si se ha solicitado una cancelación. La implementación de la actividad puede detectar esta excepción y actuar según la solicitud de cancelación, o bien puede no tener en cuenta la solicitud integrando la excepción. Para respetar la solicitud de cancelación, la actividad debe realizar la limpieza deseada, si la hubiese, y volver a generar la excepción `CancellationException`. Cuando se genera esta excepción desde la implementación de una actividad, el marco registra que la tarea de actividad ha finalizado con el estado de cancelada.

En el ejemplo siguiente se muestra una actividad que descarga y procesa imágenes. Produce latidos después de procesar cada una de las imágenes y, si se solicita una cancelación, limpia y vuelve a generar la excepción para que se confirme la cancelación.

```
@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;
            }
        }
}
```

Notificar los latidos de las actividades no es obligatorio, pero se recomienda hacerlo si su actividad se ejecuta durante mucho tiempo o si va a realizar operaciones exhaustivas que desearía cancelar en caso de error. Debería llamar a `heartbeatActivityTask` periódicamente desde la implementación de la actividad.

Si se agota el tiempo de espera de la actividad, se generará la excepción `ActivityTaskTimedOutException`, y el método `getDetails` en el objeto de la excepción devolverá los datos transferidos a la última llamada a `heartbeatActivityTask` realizada correctamente para la correspondiente tarea de actividad. La implementación del flujo de trabajo puede utilizar esta información para determinar los avances realizados antes de que se agotase el tiempo de espera de la tarea de actividad.

Note

No es conveniente aplicar latidos con excesiva frecuencia, ya que Amazon SWF podría limitar las solicitudes de latidos. Consulte la Guía para desarrolladores de Amazon Simple Workflow Service para conocer los límites establecidos en Amazon SWF.

Cancelación explícita de una tarea

Además de las condiciones de error, hay otros casos en que es preciso cancelar de forma explícita una tarea. Por ejemplo, es posible que sea necesario cancelar una actividad para procesar pagos mediante tarjeta de crédito si el usuario cancela la orden. El marco le permite cancelar explícitamente las tareas creadas en el ámbito de un método `TryCatchFinally`. En el siguiente ejemplo se cancela una tarea de pago cuando se recibe una señal mientras se procesa el pago.

```
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);
        }
    }
}
```

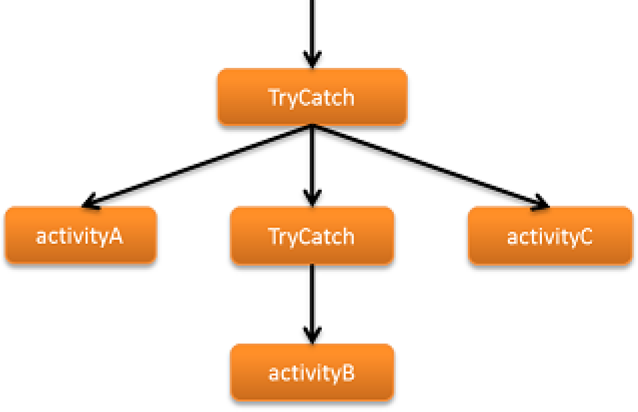
Recepción de notificaciones de tareas canceladas

Cuando una tarea finaliza con estado de cancelada, el marco informa a la lógica del flujo de trabajo generando una excepción `CancellationException`. Cuando una actividad se completa con estado de cancelada, se crea un registro en el historial y el marco llama al método `doCatch()` adecuado con una excepción `CancellationException`. Tal como se muestra en el ejemplo anterior, cuando se cancela una tarea de procesamiento de pagos, el flujo de trabajo recibe una excepción `CancellationException`.

Una excepción `CancellationException` no controlada se propaga hasta la rama de ejecución, como cualquier otra excepción. Sin embargo, el método `doCatch()` recibirá la excepción `CancellationException` únicamente si no hay otra excepción en ese ámbito, dado que hay otras excepciones con mayor prioridad que la cancelación.

Anidado TryCatchFinally

Puede anidar el método `TryCatchFinally` para adaptarlo a sus necesidades. Como cada uno de ellos `TryCatchFinally` crea una nueva rama en el árbol de ejecución, puede crear ámbitos anidados. Las excepciones en el ámbito principal producirán intentos de cancelación de todas las tareas iniciadas por el método `TryCatchFinally` anidado en su interior. Sin embargo, las excepciones contenidas en un método `TryCatchFinally` anidado no se propagan automáticamente al principal. Si desea propagar una excepción desde un `TryCatchFinally` anidado al método `TryCatchFinally` que lo contiene, deberá volver a generar la excepción en `doCatch()`. Dicho de otro modo, solo se desarrollan las excepciones no controladas, igual que `try/catch` en Java. Si cancela un `TryCatchFinally` anidado llamando al método de cancelación, se cancelará el `TryCatchFinally` anidado, pero el `TryCatchFinally` que lo contiene no se cancelará automáticamente.




```
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);
    }
};
```

Reintento de actividades con errores

En ocasiones se producen errores en las actividades por motivos efímeros, como por ejemplo la pérdida temporal de conexión. En otras ocasiones, la actividad podría tener éxito, por lo que la manera apropiada de abordar el error en la actividad consiste con frecuencia en reintentar la actividad, quizás varias veces.

Hay diferentes estrategias para reintentar actividades; la mejor depende de los detalles de su flujo de trabajo. Las estrategias se dividen en tres categorías básicas:

- La retry-until-success estrategia simplemente sigue reintentando la actividad hasta que se complete.
- La estrategia de reintento exponencial aumenta el intervalo de tiempo entre reintentos exponencialmente hasta que se completa la actividad o el proceso alcanza un punto de parada especificado, como por ejemplo un número máximo de intentos.

- La estrategia de reintento personalizada decide si se reintenta la actividad, o cómo hacerlo, después de cada intento en el que se ha producido un error.

Las siguientes secciones describen cómo implementar estas estrategias. Todos los procesos de trabajo de flujo de trabajo de ejemplo utilizan una sola actividad, `unreliableActivity`, que hace lo siguiente de manera aleatoria:

- Se completa de manera inmediata
- Produce un error de manera intencionada superando el valor del tiempo de espera
- Produce un error de manera intencionada produciendo una `IllegalStateException`

Retry-Until-Success Estrategia

La estrategia de reintento más sencilla consiste en seguir reintentando la actividad cada vez que se produce un error hasta que finalmente se ejecuta satisfactoriamente. Este es el patrón básico:

1. Implemente una clase `TryCatch` o `TryCatchFinally` anidada en el método de punto de entrada de su flujo de trabajo.
2. Ejecute la actividad en `doTry`
3. Si se produce un error en la actividad, el marco de trabajo llama a `doCatch`, que ejecuta de nuevo el método de punto de entrada.
4. Repita los pasos 2 y 3 hasta que la actividad se realiza correctamente.

El siguiente flujo de trabajo implementa la retry-until-success estrategia. La interfaz de flujo de trabajo se implementa en `RetryActivityRecipeWorkflow` y tiene un método, `runUnreliableActivityTillSuccess`, que es el punto de entrada del flujo de trabajo. El proceso de trabajo de flujo de trabajo se implementa en `RetryActivityRecipeWorkflowImpl`, de la siguiente manera:

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

El flujo de trabajo funciona de la siguiente manera:

1. `runUnreliableActivityTillSuccess` crea un objeto `Settable<Boolean>` denominado `retryActivity` que se utiliza para indicar si se ha producido un error en la actividad y debería volver a intentarse. `Settable<T>` proviene de `Promise<T>` y funciona de forma muy parecida, pero en este caso, usted establece manualmente el valor de un objeto `Settable<T>`.
2. `runUnreliableActivityTillSuccess` implementa una clase `TryCatch` anidada de manera anónima para gestionar cualquier excepción lanzada por la actividad `unreliableActivity`. Para obtener más información sobre cómo tratar excepciones lanzadas por un código asíncrono, consulte Gestión de errores.

3. doTry ejecuta la actividad unreliableActivity, que devuelve un objeto Promise<Void> denominado activityRanSuccessfully.
4. doTry llama al método asíncrono setRetryActivityToFalse, que tiene dos parámetros:
   - activityRanSuccessfully toma el objeto Promise<Void> devuelto por la actividad unreliableActivity.
   - retryActivity toma el objeto retryActivity.

   Si unreliableActivity finaliza, activityRanSuccessfully está listo y setRetryActivityToFalse establece retryActivity en "false". De lo contrario, activityRanSuccessfully nunca está listo y setRetryActivityToFalse no se ejecuta.
5. Si unreliableActivity genera una excepción, el marco de trabajo llama a doCatch y le pasa el objeto de excepción. doCatch establece retryActivity en true.
6. runUnreliableActivityTillSuccess al método asíncrono restartRunUnreliableActivityTillSuccess y le pasa el objeto retryActivity. Dado que retryActivity es un tipo de Promise<T>, restartRunUnreliableActivityTillSuccess aplaza la ejecución hasta que retryActivity esté listo, lo que ocurre después de que se completa TryCatch.
7. Cuando retryActivity está listo, restartRunUnreliableActivityTillSuccess extrae el valor.
   - Si el valor es false, el reintento ha tenido éxito. restartRunUnreliableActivityTillSuccess no hace nada y la secuencia de reintento termina.
   - Si el valor es "true", se ha producido un error en el reintento. restartRunUnreliableActivityTillSuccess llama a runUnreliableActivityTillSuccess para ejecutar de nuevo la actividad.
8. Los pasos 1 al 7 se repiten hasta que se completa unreliableActivity.

> **Note**
>
> doCatch no gestiona la excepción; simplemente establece el objeto retryActivity en "true" para indicar que se ha producido un error en la actividad. El reintento es gestionado por el método asíncrono restartRunUnreliableActivityTillSuccess, que aplaza la ejecución hasta que se completa TryCatch. El motivo de este enfoque es que, si reintenta una actividad en doCatch, no es posible cancelarla. Volver a intentar la actividad

`restartRunUnreliableActivityTillSuccess` le permite ejecutar actividades que se pueden cancelar.

Estrategia de reintento exponencial

Con la estrategia de reintento exponencial, el marco de trabajo ejecuta una actividad en la que se ha producido un error de nuevo tras un periodo de tiempo especificado, N segundos. Si se produce un error en ese intento, el marco de trabajo ejecuta de nuevo la actividad después de 2N segundos y luego tras 4N segundos, etc. Debido a que el tiempo de espera puede aumentar bastante, habitualmente interrumpe los reintentos en algún punto en lugar de continuar de manera indefinida.

El marco de trabajo ofrece tres maneras de implementar una estrategia de reintento exponencial:

- La anotación `@ExponentialRetry` es el enfoque más sencillo, pero debe establecer las opciones de configuración de los reintentos en tiempo de compilación.
- La clase `RetryDecorator` le permite establecer la configuración de reintentos en el tiempo de ejecución y cambiarla según sea necesario.
- La clase `AsyncRetryingExecutor` le permite establecer la configuración de reintentos en el tiempo de ejecución y cambiarla según sea necesario. Además, el marco de trabajo llama a un método `AsyncRunnable.run` implementado por el usuario para la ejecución de cada reintento.

Todos los enfoques admiten las siguientes opciones de configuración, en las que los valores de tiempo se muestran en segundos:

- El tiempo de espera de reintento inicial.
- El coeficiente de retardo, que se utiliza para computar los intervalos de reintento, de la siguiente manera:

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

  El valor predeterminado es 2.0.
- El número máximo de reintentos. El valor predeterminado es ilimitado.
- El intervalo máximo de reintentos. El valor predeterminado es ilimitado.

- El plazo de vencimiento. Los reintentos se detienen cuando la duración total del proceso supera este valor. El valor predeterminado es ilimitado.
- Las excepciones que dispararán el proceso de reintento. De manera predeterminada, todas las excepciones disparan el proceso de reintento.
- Las excepciones que no dispararán un reintento. De manera predeterminada, no se excluye ninguna excepción.

En las siguientes secciones se describen las distintas maneras de implementar una estrategia de reintento exponencial.

Reintento exponencial con @ ExponentialRetry

La manera más sencilla de implementar una estrategia de reintento exponencial para una actividad consiste en aplicar una anotación `@ExponentialRetry` a la actividad en la definición de interfaz. Si se produce un error en la actividad, el marco de trabajo gestiona el proceso de reintento automáticamente, en función de los valores de opciones especificados. Este es el patrón básico:

1. Aplique `@ExponentialRetry` a las actividades apropiadas y especifique la configuración de reintento.
2. Si se produce un error en una actividad anotada, el marco de trabajo reintenta automáticamente la actividad en función de la configuración especificada por los argumentos del comentario.

El proceso de trabajo del flujo de trabajo `ExponentialRetryAnnotationWorkflow` implementa la estrategia de reintento exponencial utilizando una anotación `@ExponentialRetry`. Utiliza una actividad `unreliableActivity` cuya definición de interfaz se implementa en `ExponentialRetryAnnotationActivities`, de la siguiente manera:

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

Las opciones de `@ExponentialRetry` especifican la siguiente estrategia:

- Reintentar solo si la actividad lanza `IllegalStateException`.
- Usar el tiempo de espera inicial de 5 segundos.
- No más de 5 reintentos.

La interfaz de flujo de trabajo se implementa en `RetryWorkflow` y tiene un método, `process`, que es el punto de entrada del flujo de trabajo. El proceso de trabajo de flujo de trabajo se implementa en `ExponentialRetryAnnotationWorkflowImpl`, de la siguiente manera:

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

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

El flujo de trabajo funciona de la siguiente manera:

1. `process` ejecuta el método síncrono `handleUnreliableActivity`.
2. `handleUnreliableActivity` ejecuta la actividad `unreliableActivity`.

Si se produce un error en la actividad y lanza `IllegalStateException`, el marco de trabajo ejecuta automáticamente la estrategia de reintento especificada en `ExponentialRetryAnnotationActivities`.

Reintento exponencial con la clase RetryDecorator

`@ExponentialRetry` es fácil de utilizar. No obstante, la configuración es estática y se establece en el tiempo de compilación, por lo que el marco de trabajo utiliza la misma estrategia de reintento cada vez que se produce un error en la actividad. Puede implementar una estrategia de reintento exponencial más flexible utilizando la clase `RetryDecorator` que le permite especificar la configuración en el tiempo de ejecución y cambiarla según sea necesario. Este es el patrón básico:

1. Cree y configure un objeto `ExponentialRetryPolicy` que especifique la configuración de reintento.
2. Cree un objeto `RetryDecorator` y pase el objeto `ExponentialRetryPolicy` del Paso 1 al constructor.
3. Aplique el objeto decorador a la actividad pasando el nombre de la clase del cliente de la actividad al método de decoración del objeto `RetryDecorator`.
4. Ejecute la actividad.

Si se produce un error en la actividad, el marco de trabajo reintenta la actividad en función de la configuración del objeto `ExponentialRetryPolicy`. Puede cambiar la configuración de los reintentos según sea necesario modificando este objeto.

Note

La anotación `@ExponentialRetry` y la clase `RetryDecorator` se excluyen mutuamente. No puede utilizar `RetryDecorator` para anular dinámicamente una política de reintentos especificada por una anotación `@ExponentialRetry`.

La siguiente implementación de flujo de trabajo muestra cómo usar la clase `RetryDecorator` para implementar una estrategia de reintento exponencial. Utiliza una actividad `unreliableActivity` que no tiene una anotación `@ExponentialRetry`. La interfaz de flujo de trabajo se implementa en `RetryWorkflow` y tiene un método, `process`, que es el punto de entrada del flujo de trabajo. El proceso de trabajo de flujo de trabajo se implementa en `DecoratorRetryWorkflowImpl`, de la siguiente manera:

```
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();
  }
}
```

El flujo de trabajo funciona de la siguiente manera:

1. `process` crea y configura un objeto `ExponentialRetryPolicy` de la siguiente manera:
   - Pasando el intervalo de reintentos inicial al constructor.
   - Llamando al método `withMaximumAttempts` del objeto para establecer el número máximo de intentos en 5. `ExponentialRetryPolicy` expone otros objetos `with` que se pueden utilizar para especificar otras opciones de configuración.
2. `process` crea un objeto `RetryDecorator` denominado `retryDecorator` y pasa el objeto `ExponentialRetryPolicy` del Paso 1 al constructor.
3. `process` aplica el decorador a la actividad llamando al método `retryDecorator.decorate` y pasándole el nombre de la clase del cliente de la actividad.
4. `handleUnreliableActivity` ejecuta la actividad.

Si se produce un error en la actividad, el marco de trabajo lo reintenta en función de la configuración especificada en el Paso 1.

> **Note**
>
> Varios de los métodos `with` de la clase `ExponentialRetryPolicy` tienen un método `set` correspondiente que puede llamar para modificar la opción de configuración correspondiente en cualquier momento: `setBackoffCoefficient`, `setMaximumAttempts`, `setMaximumRetryIntervalSeconds` y `setMaximumRetryExpirationIntervalSeconds`.

Reintento exponencial con la clase AsyncRetryingExecutor

La clase `RetryDecorator` ofrece más flexibilidad en la configuración del proceso de reintento que `@ExponentialRetry`, pero el marco de trabajo sigue ejecutando los reintentos automáticamente, en función de la configuración actual del objeto `ExponentialRetryPolicy`. Un enfoque más flexible consiste en usar la clase `AsyncRetryingExecutor`. Además de permitirle configurar

el proceso de reintento en el tiempo de ejecución, el marco de trabajo llama a un método `AsyncRunnable.run` implementado por el usuario para que ejecute cada reintento en lugar de simplemente ejecutar la actividad.

Este es el patrón básico:

1. Cree y configure un objeto `ExponentialRetryPolicy` para especificar la configuración de reintento.
2. Cree un objeto `AsyncRetryingExecutor` y pásele el objeto `ExponentialRetryPolicy` y una instancia del reloj del flujo de trabajo.
3. Implemente una clase `TryCatch` o `TryCatchFinally` anidada anónima.
4. Implemente una clase `AsyncRunnable` anónima y anule el método `run` para la implementación del código personalizado para la ejecución de la actividad.
5. Anule `doTry` para llamar al método `execute` del objeto `AsyncRetryingExecutor` y pasarle la clase `AsyncRunnable` del Paso 4. El objeto `AsyncRetryingExecutor` llama a `AsyncRunnable.run` para ejecutar la actividad.
6. Si se produce un error en la actividad, el objeto `AsyncRetryingExecutor` llama de nuevo al método `AsyncRunnable.run` en función de la política de reintentos especificada en el Paso 1.

El siguiente flujo de trabajo muestra cómo usar la clase `AsyncRetryingExecutor` para implementar una estrategia de reintento exponencial. Utiliza la misma actividad `unreliableActivity` que el flujo de trabajo `DecoratorRetryWorkflow` sobre el que hemos hablado antes. La interfaz de flujo de trabajo se implementa en `RetryWorkflow` y tiene un método, `process`, que es el punto de entrada del flujo de trabajo. El proceso de trabajo de flujo de trabajo se implementa en `AsyncExecutorRetryWorkflowImpl`, de la siguiente manera:

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

El flujo de trabajo funciona de la siguiente manera:

1. `process` llama al método `handleUnreliableActivity` y le pasa los ajustes de la configuración.
2. `handleUnreliableActivity` utiliza los ajustes de la configuración del Paso 1 para crear un objeto `ExponentialRetryPolicy`, `retryPolicy`.
3. `handleUnreliableActivity` crea un objeto `AsyncRetryExecutor`, `executor`, y pasa el objeto `ExponentialRetryPolicy` del Paso 2 y una instancia del reloj del flujo de trabajo al constructor.
4. `handleUnreliableActivity` implementa una clase `TryCatch` anidada de manera anónima y anula los métodos `doTry` y `doCatch` para ejecutar los reintentos y gestionar cualquier excepción.
5. `doTry` crea una clase `AsyncRunnable` anónima y anula el método `run` para la implementación del código personalizado para la ejecución de `unreliableActivity`. Para simplificar, `run` simplemente ejecuta la actividad, pero puede implementar un enfoque más sofisticado según considere apropiado.

6. `doTry` llama a `executor.execute` y le pasa el objeto `AsyncRunnable`. `execute` llama al método `run` del objeto `AsyncRunnable` para ejecutar la actividad.
7. Si se produce un error en la actividad, el ejecutor llama a `run` de nuevo, en función de la configuración del objeto `retryPolicy`.

Para obtener más información sobre cómo utilizar la clase `TryCatch` para gestionar errores, consulte AWS Flow Framework para excepciones de Java.

Estrategia de reintento personalizada

El enfoque más flexible para reintentar las actividades fallidas es una estrategia personalizada, que invoca de forma recursiva a un método asíncrono que ejecuta el reintento, al igual que la estrategia. retry-until-success No obstante, en lugar de simplemente ejecutar la actividad de nuevo, usted implementa la lógica personalizada que decide si se ejecuta cada reintento sucesivo y cómo hacerlo. Este es el patrón básico:

1. Cree un objeto de estado `Settable<T>` que se utiliza para indicar si se ha producido un error en la actividad.
2. Implemente una clase `TryCatch` o `TryCatchFinally` anidada.
3. `doTry` ejecuta la actividad.
4. Si se produce un error en la actividad, `doCatch` establece el objeto de estado para indicar que se ha producido un error en la actividad.
5. Llame al método asíncrono de gestión de errores y pásele el objeto de estado. El método aplaza la ejecución hasta que `TryCatch` o `TryCatchFinally` se completan.
6. El método de gestión de errores decide si se vuelve a intentar la actividad y, si la respuesta es afirmativa, cuándo hacerlo.

El siguiente flujo de trabajo muestra cómo implementar una estrategia de reintento personalizada. Utiliza la misma actividad `unreliableActivity` que los flujos de trabajo `DecoratorRetryWorkflow` y `AsyncExecutorRetryWorkflow`. La interfaz de flujo de trabajo se implementa en `RetryWorkflow` y tiene un método, `process`, que es el punto de entrada del flujo de trabajo. El proceso de trabajo de flujo de trabajo se implementa en `CustomLogicRetryWorkflowImpl`, de la siguiente manera:

```
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;
    }
}
```

El flujo de trabajo funciona de la siguiente manera:

1. `process` llama al método asíncrono `callActivityWithRetry`.
2. `callActivityWithRetry` crea un objeto `Settable<Throwable>` llamado failure que se utiliza para indicar si se ha producido un error en la actividad. `Settable<T>` proviene de `Promise<T>` y funciona de forma muy parecida, pero en este caso usted establece manualmente el valor de un objeto `Settable<T>`.

3. `callActivityWithRetry` implementa una clase `TryCatchFinally` anidada de manera anónima para gestionar cualquier excepción lanzada por `unreliableActivity`. Para obtener más información sobre cómo tratar excepciones lanzadas por un código asíncrono, consulte [AWS Flow Framework para excepciones de Java](#).
4. `doTry` ejecuta `unreliableActivity`.
5. Si `unreliableActivity` lanza una excepción, el marco de trabajo llama a `doCatch` y le pasa el objeto de excepción. `doCatch` establece `failure` en el objeto de excepción, lo que indica que se ha producido un error en la actividad y pone el objeto en el estado ready.
6. `doFinally` comprueba si `failure` está listo, lo que solo será "true" si `doCatch` ha establecido `failure`.
   - Si está listo`failure`, no hace nada. `doFinally`
   - Si `failure` no está listo, la actividad completada y `doFinally` establecen el error en `null`.
7. `callActivityWithRetry` llama al método asíncrono `retryOnFailure` y le pasa el error. Dado que el error es un tipo `Settable<T>`, `callActivityWithRetry` la ejecución se aplaza hasta que el error esté listo, lo que ocurre después de que se completa `TryCatchFinally`.
8. `retryOnFailure` obtiene el valor del error.
   - Si el error se establece en null, el reintento se ha realizado con éxito. `retryOnFailure` no hace nada, lo cual termina el proceso de reintento.
   - Si el error se establece en un objeto de excepción y `shouldRetry` devuelve "true", `retryOnFailure` llama a `callActivityWithRetry` para reintentar la actividad.

     `shouldRetry` implementa la lógica personalizada para decidir si vuelve a intentar una actividad en la que se ha producido un error. Para simplificar, `shouldRetry` siempre devuelve `true` y `retryOnFailure` ejecuta la actividad inmediatamente, pero puede implementar una lógica más sofisticada según considere apropiado.
9. Los pasos 2 al 8 se repiten hasta que `unreliableActivity` se completa, o bien hasta que `shouldRetry` decide interrumpir el proceso.

> **Note**
>
> `doCatch` no gestiona el proceso de reintento, simplemente establece el error para indicar que se ha producido un error en la actividad. El proceso de reintento es gestionado por el método asíncrono `retryOnFailure`, que aplaza la ejecución hasta que se completa `TryCatch`. El motivo de este enfoque es que, si reintenta una actividad en `doCatch`, no

es posible cancelarla. Volver a intentar la actividad `retryOnFailure` le permite ejecutar actividades que se pueden cancelar.

Tareas del demonio

El AWS Flow Framework para Java permite marcar ciertas tareas como`daemon`. Esto le permite crear tareas que hacen algo de trabajo en segundo plano que deberán cancelarse cuando se realiza el resto del trabajo. Por ejemplo, una tarea de monitorización de estado deberá cancelarse una vez que se haya completado el resto del flujo de trabajo. Puede hacerlo estableciendo el indicador `daemon` en un método asíncrono o una instancia de `TryCatchFinally`. En el siguiente ejemplo, el método asíncrono `monitorHealth()` está marcado como `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();
  }
}
```

En el ejemplo de arriba, cuando se completa do`UsefulWorkActivity`, se cancelará `monitoringHealth` automáticamente. Esto cancelará a su vez la bifurcación de ejecución completa enraizada en este método asíncrono. La semántica de la cancelación es la misma que en `TryCatchFinally`. De manera parecida, puede marcar un demonio `TryCatchFinally` pasando un indicador booleano al constructor.

```
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
            }
        };
    }
}
```

Una tarea de daemon iniciada dentro de `TryCatchFinally` se establece en el contexto en el que se crea, es decir, se establecerá en cualquiera de estos métodos: `doTry()`, `doCatch()` o `doFinally()`. Por ejemplo, en el siguiente ejemplo, el método asíncrono startMonitoring está marcado como demonio y se llama desde `doTry()`. La tarea que se creó para él se cancelará tan pronto como se completen las otras tareas (`doUsefulWorkActivity` en este caso) iniciadas dentro de `doTry()`.

```
public class MyWorkflowImpl implements MyWorkflow {
    MyActivitiesClient activitiesClient = new MyActivitiesClientImpl();

    @Override
    public void startMyWF(int a, String b) {
        new TryFinally() {
            @Override
            protected void doTry() throws Throwable {
                activitiesClient.doUsefulWorkActivity();
                startMonitoring();
            }

            @Override
            protected void doFinally() throws Throwable {
                // Clean up
            }
        };
    }
```

```
    @Asynchronous(daemon = true)
    void startMonitoring(){
      activitiesClient.monitoringActivity();
    }
```

AWS Flow Framework para Java Replay Behavior

Este tema trata sobre ejemplos de comportamiento de reproducción mediante ejemplos de la sección ¿Qué es AWS Flow Framework para Java?. Se tratan tanto escenarios síncronos como asíncronos.

Ejemplo 1: reproducción síncrona

Para ver un ejemplo de cómo funciona la reproducción en un flujo de trabajo sincrónico, modifique las implementaciones del HelloWorldWorkflowflujo de trabajo y de la actividad añadiendo `println` llamadas en sus respectivas implementaciones, de la siguiente manera:

```
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);
```

```
    }
}
```

Para más detalles sobre el código, consulte HelloWorldWorkflow Solicitud. Se muestra a continuación una versión editada del resultado, con comentarios que indican el comienzo de cada episodio de reproducción.

```
//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
```

El proceso de reproducción para este ejemplo funciona de la siguiente manera:

- El primer episodio programa la tarea de actividad `getName`, que no tiene dependencias.
- El segundo episodio programa la tarea de actividad `getGreeting`, que depende de `getName`.
- El tercer episodio programa la tarea de actividad `say`, que depende de `getGreeting`.
- El último episodio no programa tareas adicionales y no encuentra actividades sin completar, lo que termina la ejecución del flujo de trabajo.

Note

Se llama una vez a los tres métodos de cliente de actividades para cada episodio. No obstante, solo una de esas llamadas produce una tarea de actividad, por lo que cada tarea solo se realiza una vez.

Ejemplo 2: reproducción asíncrona

De forma parecida al ejemplo de reproducción síncrona, puede modificar HelloWorldWorkflowAsync Solicitud para ver cómo funciona una reproducción asíncrona. Produce el siguiente resultado:

```
//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 usa tres episodios de repetición porque solo hay dos actividades. La actividad `getGreeting` se ha sustituido por el método de flujo de trabajo asíncrono getGreeting que no inicia un episodio de reproducción cuando se completa.

El primer episodio no llama a `getGreeting`, porque depende de la finalización de la actividad nombre. No obstante, una vez que se completa getName, la reproducción llama a getGreeting una vez para cada episodio subsiguiente.

Véase también

- AWS Flow Framework Conceptos básicos: ejecución distribuida

Prácticas recomendadas

Utilice estas prácticas recomendadas para sacar el máximo provecho AWS Flow Framework de Java.

Temas

- Realización de cambios en el código del decisor: control de versiones y marcas de características

Realización de cambios en el código del decisor: control de versiones y marcas de características

Esta sección muestra cómo evitar cambios que no son compatibles con las versiones anteriores a un decisor mediante dos métodos:

- El control de versiones proporciona una solución básica.
- El control de versiones con indicadores de características se basa en la solución del control de versiones: no se introduce ninguna versión nueva del flujo de trabajo y no es necesario introducir código nuevo para actualizar la versión.

Antes de probar estas soluciones, familiarícese con la sección Escenario de ejemplo que explica las causas y los efectos de los cambios al decisor que no son compatibles con las versiones anteriores.

El proceso de reproducción y cambios de códigos

Cuando un AWS Flow Framework dispositivo de toma de decisiones para Java ejecuta una tarea de decisión, primero debe reconstruir el estado actual de la ejecución antes de poder añadirle pasos. El decisor lo hace utilizando un proceso denominado reproducción.

El proceso de reproducción vuelve a ejecutar el código del decisor desde el principio, al tiempo que recorre simultáneamente el historial de eventos que ya se han producido. Recorrer el historial de eventos permite al marco de trabajo reaccionar a señales o tareas finalizada y desbloquear los objetos `Promise` en el código.

Cuando el marco de trabajo ejecuta el código del decisor, asigna un ID a cada tarea programada (una actividad, una función de Lambda, un temporizador, un flujo de trabajo secundario o una señal

saliente); para ello, aumenta un contador. El marco de trabajo comunica este ID a Amazon SWF y añade el ID a los eventos del historial, como por ejemplo, `ActivityTaskCompleted`.

Para que el proceso de reproducción se realice con éxito, es importante que el código del decisor sea determinista y programar las mismas tareas en el mismo orden para todas las decisiones en cada ejecución de flujo de trabajo. Si no cumple este requisito, el marco de trabajo podría, por ejemplo, no hacer coincidir el ID en un evento `ActivityTaskCompleted` con un objeto `Promise` existente.

Escenario de ejemplo

Hay una clase de cambios de códigos que se considera que no son compatibles con las versiones anteriores. Estos cambios incluyen actualizaciones que modifican el número, tipo u orden de las tareas programadas. Considere el siguiente ejemplo:

Escribe código decisor para programar dos tareas de temporizador. Comienza la ejecución y ejecuta una decisión. Como resultado, se programan dos tareas temporizadas, con IDs `1` y. `2`

Si actualiza el código decisor para programar solo un temporizador antes de la ejecución de la siguiente decisión, durante la siguiente tarea de decisión el marco de trabajo producirá un error al reproducir el segundo evento `TimerFired`, porque el ID `2` no coincide con ninguna tarea del temporizador producida por el código.

Esquema del escenario

El siguiente esquema muestra los pasos de este escenario. El objetivo final del escenario es migrar a un sistema que solo programa un temporizador pero que no provoca errores en las ejecuciones iniciadas antes de la migración.

1. La versión inicial del decisor
   a. Escriba el decisor.
   b. Inicie el decisor.
   c. El decisor programa dos temporizadores.
   d. El decisor comienza cinco ejecuciones.
   e. Detenga el decisor.
2. Cambio de decisor no compatible con versiones anteriores
   a. Modifique el decisor.
   b. Inicie el decisor.

c. El decisor programa un temporizador.

d. El decisor comienza cinco ejecuciones.

Las siguientes secciones incluyen ejemplo de código Java que muestran como implementar este escenario. Los ejemplos de código en la sección Soluciones muestran diferentes maneras de solucionar cambios que no son compatibles con las versiones anteriores.

Note

Puede utilizar la última versión del AWS SDK para Java para ejecutar este código.

Código común

El siguiente código Java no cambia de un ejemplo a otro en este escenario.

`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;
    }

}
```

Escritura del código del decisor inicial

A continuación se muestra el código Java inicial del decisor. Está registrado como versión 1 y programa dos tareas de temporizador de cinco segundos.

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);
        }

    }
}
```

Simulación de un cambio no compatible con versiones anteriores

El siguiente código Java modificado del decisor en un buen de cambio no compatible con versiones anteriores. El código sigue estando registrado como versión 1 pero programa solo un temporizador.

`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);
        }

    }
}
```

El siguiente código Java le permite simular el problema de hacer cambios que no son compatibles con versiones anteriores ejecutando el decisor modificado.

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();
    }

}
```

Cuando ejecuta el programa, las tres ejecuciones en las que se produce un error son aquellas que comenzaron bajo la versión inicial del decisor y que continuaron después de la migración.

Soluciones

Puede utilizar las siguientes soluciones para evitar cambios que no son compatibles con las versiones anteriores. Para obtener más información, consulte Realización de cambios en el código del decisor y Escenario de ejemplo.

Uso del control de versiones

En esta solución, copia el decisor en una clase nueva, modifica el decisor y, a continuación, registra el decisor bajo una versión de flujo de trabajo nueva.

`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);
        }

    }

}
```

En el código Java actualizado, el proceso de trabajo del segundo decisor ejecuta ambas versiones del flujo de trabajo, permitiendo que las ejecuciones en tránsito sigan ejecutándose independientemente de los cambios en la versión 2.

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();
    }

}
```

Cuando ejecuta el programa, todas las ejecuciones se completan correctamente.

Uso de indicadores de características

Otra solución para los problemas de compatibilidad con versiones anteriores consiste en ramificar código para admitir dos implementaciones en la misma clase en función de los datos introducidos en lugar de las versiones de flujo de trabajo.

Cuando adopta este enfoque, añade campos a los objetos de entrada (o modifica campos existentes de estos) cada vez que introduce cambios importantes. Para ejecuciones que comienzan antes de la migración, el objeto de entrada no tendrá el campo (o tendrá un valor diferente). Por tanto, no tiene que aumentar el número de versión.

> **Note**
>
> Si añade campos nuevos, asegúrese de que el proceso de deserialización JSON es compatible con versiones anteriores. Los objetos serializados antes de la introducción del campo deberían deserializarse correctamente después de la migración. Dado que JSON establece un valor `null` siempre que falta un campo, utilice los tipos a los que se aplica conversión boxing (`Boolean` en lugar de `boolean`) y gestione los casos en los que el valor es `null`.

`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);
            }
        }

    }
}
```

En el código Java actualizado, el código para ambas versiones del flujo de trabajo sigue registrándose para la versión 1. No obstante, después de la migración, las nuevas ejecuciones comienzan con el campo `skipSecondTimer` de los datos de entrada establecidos en `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();
    }

}
```

Cuando ejecuta el programa, todas las ejecuciones se completan correctamente.

Consejos de solución de problemas y depuración AWS Flow Framework para Java

Temas

- Errores de compilación
- Fallo de recurso desconocido
- Excepciones al llamar a get () con una promesa
- Flujos de trabajo no deterministas
- Problemas debidos al control de versiones
- Solución de problemas y depuración de la ejecución de un flujo de trabajo
- Tareas perdidas
- Fallo de validación debido a restricciones de longitud de los parámetros de la API

En esta sección se describen algunos de los errores más comunes con los que se puede tropezar al desarrollar flujos de trabajo con Java. AWS Flow Framework También se ofrecen algunos consejos para ayudarle a diagnosticar y depurar problemas.

Errores de compilación

Si utiliza la opción de incorporación del tiempo de compilación de AspectJ, podría encontrarse errores en tiempo de compilación que indican que el compilador no encuentra las clases de cliente generadas para el flujo de trabajo y las actividades. La causa más probable de esos errores de compilación es que el constructor de AspectJ ha omitido los clientes generados durante la compilación. Para solucionar este problema, puede eliminar la funcionalidad de AspectJ del proyecto y luego volver a habilitarla. Tenga en cuenta que tendrá que hacer esto cada vez que cambien las interfaces de flujos de trabajo y actividades. Debido a este problema, recomendamos utilizar en su lugar la opción de incorporación del tiempo de carga. Consulte la sección Configuración del AWS Flow Framework para Java para obtener más información.

Fallo de recurso desconocido

Amazon SWF devuelve un error de recurso desconocido al intentar realizar una operación en un recurso que no está disponible. Las causas más comunes de este error son:

- Se ha configurado un proceso de trabajo con un dominio que no existe. Para solucionarlo, registre primero el dominio mediante la consola de Amazon SWF, o bien mediante la API del servicio de Amazon SWF.
- Ha intentado crear tareas de actividades o ejecuciones de flujos de trabajo de tipos que no se han registrado. Esto puede ocurrir si intenta crear la ejecución del flujo de trabajo antes de que se hayan ejecutado los procesos de trabajo. Como los trabajadores registran sus tipos cuando se ejecutan por primera vez, debes ejecutarlos al menos una vez antes de intentar iniciar las ejecuciones (o registrarlos manualmente mediante la consola o la API del servicio). Tenga en cuenta que, una vez que se hayan registrado los tipos, puede crear ejecuciones aunque no se esté ejecutando ningún proceso de trabajo.
- Un proceso de trabajo intenta completar una tarea que ya ha superado el tiempo de espera. Por ejemplo, si un trabajador tarda demasiado en procesar una tarea y supera el tiempo de espera, se producirá un UnknownResource error cuando intente completar la tarea o no la complete. Los AWS Flow Framework trabajadores seguirán sondeando Amazon SWF y procesando tareas adicionales. Sin embargo, debería pensar en la posibilidad de ajustar el tiempo de espera. Para ajustarlo, tiene que registrar una versión nueva del tipo de actividad.

Excepciones al llamar a get () con una promesa

A diferencia de Java Future, `Promise` es una construcción sin bloqueo y, llamar a `get()` en una `Promise` que no está preparada aún, generará una excepción en lugar de un bloqueo. La forma correcta de utilizar una `Promise` consiste en pasarla a un método asíncrono (o a una tarea) y obtener acceso a su valor en el método asíncrono. AWS Flow Framework para Java garantiza que solo se llame a un método asíncrono cuando estén preparados todos los argumentos de `Promise` que se hayan pasado a dicho método. Si cree que su código es correcto o si se encuentra con esto mientras ejecuta uno de los AWS Flow Framework ejemplos, lo más probable es que AspectJ no esté configurado correctamente. Para obtener más información, consulte la sección Configuración del AWS Flow Framework para Java.

Flujos de trabajo no deterministas

Tal y como se ha explicado en la sección No determinismo, la implementación del flujo de trabajo debe ser determinista. Algunos errores comunes que pueden llevar al indeterminismo son el uso del reloj del sistema, el uso de números aleatorios y la generación de. GUIDs Como estas construcciones pueden devolver valores diferentes en momentos diferentes, el flujo de control del flujo de trabajo puede tomar diferentes rutas cada vez que se ejecute (consulte las secciones AWS

Flow Framework Conceptos básicos: ejecución distribuida y Descripción de una tarea en AWS Flow Framework for Java para obtener más información). Si el marco de trabajo detecta el uso un sistema no determinista al ejecutar el flujo de trabajo, se generará una excepción.

Problemas debidos al control de versiones

Al implementar una nueva versión del flujo de trabajo o actividad, por ejemplo, cuando agrega una nueva característica, debe aumentar la versión del tipo mediante la anotación adecuada: `@Workflow`, `@Activites` o `@Activity`. Cuando se implementan nuevas versiones de un flujo de trabajo, lo más frecuente es que la versión existente ya se esté ejecutando. Por lo tanto, tiene que asegurarse de que los procesos de trabajo que tienen la versión adecuada del flujo de trabajo y las actividades reciben las tareas. Para hacer esto, puede utilizar un conjunto diferente de listas de tareas para cada versión. Por ejemplo, puede anexar el número de versión al nombre de la lista de tareas. De esta manera, se asegura de que las tareas que pertenecen a diferentes versiones del flujo de trabajo y las actividades se asignan a los procesos de trabajo adecuados.

Solución de problemas y depuración de la ejecución de un flujo de trabajo

El primer paso para solucionar los problemas de una ejecución de flujo de trabajo consiste en utilizar la consola de Amazon SWF para examinar el historial de flujos de trabajo. El historial de flujos de trabajo es un registro completo y autorizado de todos los eventos que han cambiado el estado de ejecución de los flujos de trabajo. Amazon SWF mantiene este historial, que es de gran valor para diagnosticar los problemas. La consola de Amazon SWF le permite buscar ejecuciones de flujos de trabajo y desglosarlas en eventos individuales del historial.

AWS Flow Framework proporciona una `WorkflowReplayer` clase que puede utilizar para reproducir la ejecución de un flujo de trabajo de forma local y depurarla. Con esta clase, puede depurar ejecuciones de flujo de trabajo cerradas y en ejecución. `WorkflowReplayer` utiliza el historial que está almacenado en Amazon SWF para realizar la reproducción. Puede hacer que apunte a una ejecución de flujo de trabajo de su cuenta de Amazon SWF o proporcionarle los eventos del historial (por ejemplo, puede recuperar el historial de Amazon SWF y serializarlo localmente para utilizarlo posteriormente). La reproducción de una ejecución de flujo de trabajo utilizando `WorkflowReplayer` no afecta a la ejecución del flujo de trabajo que se está realizando en su cuenta. La reproducción se realiza por completo en el cliente. Puede depurar el flujo de trabajo, crear puntos de interrupción y meterse en el código utilizando sus herramientas de depuración

habituales. Si utiliza Eclipse, considere la posibilidad de añadir filtros por pasos para filtrar AWS Flow Framework paquetes.

Por ejemplo, el siguiente fragmento de código se puede utilizar para reproducir una ejecución de flujo de trabajo:

```
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 también le permite obtener un volcado de subprocesos asíncrono de la ejecución de su flujo de trabajo. Este volcado de subprocesos le proporciona las pilas de llamadas de todas las tareas asíncronas abiertas. Esta información puede ser útil para determinar qué tareas de la ejecución siguen pendientes y posiblemente estén atascadas. Por ejemplo:

```
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);
}
```

Tareas perdidas

En ocasiones, es posible que cierre procesos de trabajo y comience otros nuevos en una rápida sucesión, pero al final descubra que las tareas se entregan a los procesos de trabajo antiguos. Esto puede ser debido a las condiciones de carrera del sistema, que se distribuyen entre varios procesos. Este problema puede aparecer también cuando ejecuta pruebas de unidades en un bucle sólido. Este problema también se puede producir en ocasiones al detener una prueba en Eclipse, porque es posible que no se llame a los controladores de cierre.

Para asegurarse de que el problema se debe a que las tareas van a los procesos de trabajo antiguos, debería examinar el historial de flujos de trabajo para determinar qué proceso ha recibido la tarea que esperaba que recibiera el nuevo proceso de trabajo. Por ejemplo, el evento `DecisionTaskStarted` del historial contiene la identidad del proceso de trabajo del flujo de trabajo que ha recibido la tarea. El identificador utilizado por Flow Framework tiene el siguiente formato: {*processId*}`@`{*host name*}. Por ejemplo, estos son los datos del evento `DecisionTaskStarted` en la consola de Amazon SWF para una ejecución de ejemplo:

Marca temporal del evento	Mon Feb 20 11:52:40 GMT-800 2012
Identidad	2276 @ip -0A6C1 DF5
Identificador de evento programado	33

Para evitar esta situación, utilice diferentes listas de tareas para cada prueba. Considere también la posibilidad de añadir un aplazamiento entre el cierre de los procesos de trabajo antiguos y el inicio de los nuevos.

Fallo de validación debido a restricciones de longitud de los parámetros de la API

Amazon SWF impone restricciones de longitud a los parámetros de la API. Recibirá un `HTTP 400` error si la implementación de su flujo de trabajo o actividad supera las restricciones. Por

ejemplo, cuando se solicita `ActivityExecutionContext` enviar un latido para una actividad `recordActivityHeartbeat` en ejecución, la cadena no debe tener más de 2048 caracteres.

Otro escenario común es cuando una actividad falla debido a una excepción. El marco informa de un error en la actividad a Amazon SWF llamando RespondActivityTaskFailedcon la excepción serializada como detalle. La llamada a la API indicará un error 400 si la excepción serializada tiene una longitud superior a 32 768 bytes. Para mitigar esta situación, puedes truncar el mensaje de excepción o las causas para cumplir con la restricción de longitud.

AWS Flow Framework para Java Reference

Temas

- AWS Flow Framework para anotaciones de Java
- AWS Flow Framework para excepciones de Java
- AWS Flow Framework para paquetes Java

AWS Flow Framework para anotaciones de Java

Temas

- @Tareas
- @Actividad
- @ActivityRegistrationOptions
- @Asynchronous
- @Execute
- @ExponentialRetry
- @GetState
- @ManualActivityCompletion
- @Signal
- @SkipRegistration
- @Wait y @ NoWait
- @Flujo de trabajo
- @WorkflowRegistrationOptions

@Tareas

Esta anotación se puede utilizar en una interfaz para declarar un conjunto de tipos de actividades. Cada método de una interfaz con esta anotación representa un tipo de actividad. Una interfaz no puede tener ambas anotaciones, `@Workflow` y `@Activities`.

Se pueden especificar los siguientes parámetros en esta anotación:

`activityNamePrefix`

Especifica el prefijo del nombre de los tipos de actividades declarados en la interfaz. Si se establece como una cadena vacía (que es el valor predeterminado), el nombre de la interfaz seguido de '.' se utilizará como prefijo.

`version`

Especifica la versión predeterminada de los tipos de actividades declarados en la interfaz. El valor predeterminado es `1.0`.

`dataConverter`

Especifica el tipo de serializing/deserializing datos que se va `DataConverter` a utilizar para crear tareas de este tipo de actividad y sus resultados. Se establece en `NullDataConverter` de manera predeterminada, lo cual indica que se debe utilizar `JsonDataConverter`.

@Actividad

Esta anotación se puede utilizar en métodos dentro de una interfaz anotada con `@Activities`.

Se pueden especificar los siguientes parámetros en esta anotación:

`name`

Especifica el nombre del tipo de actividad. El valor predeterminado es una cadena vacía, lo que indica que se debe utilizar el prefijo predeterminado y el nombre del método de la actividad para determinar el nombre del tipo de actividad (con el formato {{prefijo}{{nombre}). Tenga en cuenta que, cuando especifica un nombre en una anotación `@Activity`, el marco no le agregará automáticamente un prefijo. Puede utilizar su propio esquema de nomenclatura.

`version`

Especifica la versión del tipo de actividad. Esto anula la versión predeterminada especificada en la anotación `@Activities` en la interfaz que la contiene. El valor predeterminado es una cadena vacía.

@ActivityRegistrationOptions

Especifica las opciones de registro de un tipo de actividad. Esta anotación se puede utilizar en una interfaz con la anotación `@Activities` o los métodos en ella. Si se especifica en ambos sitios, se aplicará la anotación utilizada en el método.

Se pueden especificar los siguientes parámetros en esta anotación:

`defaultTasklist`

Especifica la lista de tareas predeterminada que se va a registrar con Amazon SWF para este tipo de actividad. Este valor predeterminado se puede anular al llamar al método de actividad en el cliente generado usando el parámetro `ActivitySchedulingOptions`. Se establece en `USE_WORKER_TASK_LIST` de manera predeterminada. Se trata de un valor especial que indica que es preciso utilizar la lista de tareas que usa el trabajo, el cual está realizando el registro.

`defaultTaskScheduleToStartTimeoutSeconds`

Especifica lo defaultTaskSchedule ToStartTimeout registrado en Amazon SWF para este tipo de actividad. Se trata del tiempo máximo que una tarea de este tipo de actividad puede esperar antes de que se asigne a un trabajo. Consulte la referencia de la API de Amazon Simple Workflow Service para obtener más información.

`defaultTaskHeartbeatTimeoutSeconds`

Especifica el valor de `defaultTaskHeartbeatTimeout` registrado con Amazon SWF para este tipo de actividad. Los procesos de trabajo de las actividades deben proporcionar un latido en este intervalo de tiempo, de lo contrario se agotará el tiempo de espera de la tarea. Establecido en -1 de manera predeterminada, se trata de un valor especial que indica que se debe deshabilitar este tiempo de espera. Consulte la referencia de la API de Amazon Simple Workflow Service para obtener más información.

`defaultTaskStartToCloseTimeoutSeconds`

Especifica lo defaultTaskStart ToCloseTimeout registrado en Amazon SWF para este tipo de actividad. Este tiempo de espera determina el tiempo máximo que un trabajo puede tardar en procesar una tarea de actividad de este tipo. Consulte la referencia de la API de Amazon Simple Workflow Service para obtener más información.

`defaultTaskScheduleToCloseTimeoutSeconds`

Especifica el valor de `defaultScheduleToCloseTimeout` registrado con Amazon SWF para este tipo de actividad. El tiempo de espera determina el intervalo de tiempo total que la tarea

puede estar abierta. Establecido en -1 de manera predeterminada, se trata de un valor especial que indica que se debe deshabilitar este tiempo de espera. Consulte la referencia de la API de Amazon Simple Workflow Service para obtener más información.

@Asynchronous

Cuando se utiliza en un método dentro de la lógica de coordinación del flujo de trabajo, indica que el método se debe ejecutar de manera asíncrona. Al realizar una llamada al método, se volverá inmediatamente, pero la ejecución real se realizará de manera asíncrona cuando estén preparados todos los parámetros Promise<> que se pasan a los métodos. Los métodos con la anotación @Asynchronous tienen que devolver el tipo Promise<> o void.

`daemon`

Indica si la tarea creada para el método asíncrono debería ser una tarea de demonio. `False` de forma predeterminada.

@Execute

Cuando se utiliza en un método dentro de una interfaz con la anotación `@Workflow`, identifica el punto de entrada del flujo de trabajo.

Important

Solo un método dentro de la interfaz podrá incluir la anotación `@Execute`.

Se pueden especificar los siguientes parámetros en esta anotación:

`name`

Especifica el nombre del tipo de flujo de trabajo. Si no se establece, de forma predeterminada el nombre es {{prefijo}{{nombre}, donde {{prefijo} es el nombre de la interfaz de flujo de trabajo seguido de un "." y {{nombre} es el nombre del método con la anotación `@Execute` en el flujo de trabajo.

`version`

Especifica la versión del tipo de flujo de trabajo.

@ExponentialRetry

Cuando se utiliza en una actividad o un método asíncrono, establece una política de reintentos exponencial si el método genera una excepción no controlada. Se hace un reintento después de un periodo de retardo, que se calcula mediante la potencia del número de intentos.

Se pueden especificar los siguientes parámetros en esta anotación:

`intialRetryIntervalSeconds`

Especifica el tiempo que hay que esperar antes de que se realice el primer reintento. Este valor no debe ser mayor que `maximumRetryIntervalSeconds` y `retryExpirationSeconds`.

`maximumRetryIntervalSeconds`

Especifica el tiempo de espera máximo entre reintentos. Una vez alcanzado, el intervalo de reintento quedará determinado por este valor. Establecido en -1 de forma predeterminada, corresponde a una duración ilimitada.

`retryExpirationSeconds`

Especifica el intervalo de tiempo tras el cual se detendrá el proceso de reintento exponencial. Establecido en -1 de forma predeterminada, corresponde a ausencia de vencimiento.

`backoffCoefficient`

Especifica el coeficiente utilizado para calcular el intervalo de reintento. Consulte Estrategia de reintento exponencial.

`maximumAttempts`

Especifica el número de intentos tras el cual se detendrá el proceso de reintento exponencial. Está establecido en -1 de forma predeterminada, lo que significa que no hay límite para el número de reintentos.

`exceptionsToRetry`

Especifica la lista de tipos de excepciones que deben activar un reintento. Las excepciones no controladas de estos tipos no se seguirán propagando y se realizará un reintento del método cuando haya transcurrido el intervalo de reintento calculado. De forma predeterminada, la lista contiene `Throwable`.

`excludeExceptions`

Especifica la lista de tipos de excepciones que no deben activar un reintento. Las excepciones no controladas de este tipo podrán propagarse. La lista está vacía de forma predeterminada.

@GetState

Cuando se utiliza en un método dentro de una interfaz con la anotación `@Workflow`, identifica que el método se utiliza para recuperar el último estado de ejecución del flujo de trabajo. Puede haber un método como máximo con esta anotación en una interfaz con la anotación `@Workflow`. Los métodos con esta anotación no deben usar ningún parámetro y deben tener un tipo de devolución distinto de `void`.

@ManualActivityCompletion

Esta anotación se puede utilizar en un método de actividad para indicar que la tarea de la actividad no debe haber finalizado cuando el método devuelva un valor. La tarea de la actividad no se completará automáticamente y deberá completarse de forma manual directamente con la API de Amazon SWF. Esto resulta útil en los casos de uso donde la tarea de la actividad se delega a algún sistema externo no automatizado o que requiera intervención humana para completarse.

@Signal

Cuando se utiliza en un método dentro de una interfaz con la anotación `@Workflow`, identifica una señal que pueden recibir las ejecuciones del tipo de flujo de trabajo que haya declarado la interfaz. El uso de esta anotación es obligatorio para definir un método de señalización.

Se pueden especificar los siguientes parámetros en esta anotación:

`name`

Especifica la parte del nombre que corresponde al nombre de la señal. Si no se especifica, se utiliza el nombre del método.

@SkipRegistration

Cuando se utiliza en una interfaz con la anotación `@Workflow`, indica que el tipo de flujo de trabajo no debe registrarse con Amazon SWF. Solo se debe utilizar una de las anotaciones

`@WorkflowRegistrationOptions` y `@SkipRegistrationOptions` en una interfaz con la anotación `@Workflow`, no ambas.

@Wait y @ NoWait

Estas anotaciones se pueden usar en un parámetro de tipo `Promise<>` para indicar si el método AWS Flow Framework para Java debe esperar a que esté listo antes de ejecutar el método. De manera predeterminada, los parámetros `Promise<>` que se transfieren a los métodos `@Asynchronous` deben estar preparados antes de que se ejecute el método. En determinadas situaciones, es necesario anular este comportamiento predeterminado. No se espera a los parámetros `Promise<>` que se pasan a los métodos `@Asynchronous` y que tienen la anotación `@NoWait`.

Los parámetros de colecciones (o subclases de parámetros) que contengan promesas, como `List<Promise<Int>>`, se deben anotar con `@Wait`. De manera predeterminada, el marco no espera a los miembros de una colección.

@Flujo de trabajo

Esta anotación se utiliza en una interfaz para declarar un tipo de flujo de trabajo. Una interfaz que incluya esta anotación debe contener exactamente un método que incluya la anotación @Execute para declarar un punto de entrada del flujo de trabajo.

Note

Una interfaz no puede tener ambas anotaciones `@Workflow` y `@Activities` declaradas al mismo tiempo, ya que se excluyen mutuamente.

Se pueden especificar los siguientes parámetros en esta anotación:

`dataConverter`

Especifica qué `DataConverter` se debe utilizar al enviar solicitudes para las ejecuciones correspondientes a este tipo de flujo de trabajo, y también al recibir los resultados.

El valor predeterminado es `NullDataConverter` que, a su vez, vuelve `JsonDataConverter` a procesar todos los datos de solicitud y respuesta como notación de JavaScript objetos (JSON).

Ejemplo

```
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

Cuando se utiliza en una interfaz con la anotación `@Workflow`, proporciona la configuración predeterminada que utiliza Amazon SWF al registrar el tipo de flujo de trabajo.

> **Note**
>
> Se debe utilizar una de las anotaciones `@WorkflowRegistrationOptions` o `@SkipRegistrationOptions` en una interfaz con la anotación `@Workflow`, pero no se pueden especificar ambas.

Se pueden especificar los siguientes parámetros en esta anotación:

Descripción

Descripción de texto opcional del tipo de flujo de trabajo.

`defaultExecutionStartToCloseTimeoutSeconds`

Especifica el valor `defaultExecutionStartToCloseTimeout` registrado con Amazon SWF para el tipo de flujo de trabajo. Se trata del tiempo total que puede tardar en completarse la ejecución de un flujo de trabajo de este tipo.

Para obtener más información acerca de los tiempos de espera de los flujos de trabajo, consulte Tipos de tiempo de espera de Amazon SWF.

`defaultTaskStartToCloseTimeoutSeconds`

Especifica el valor `defaultTaskStartToCloseTimeout` registrado con Amazon SWF para el tipo de flujo de trabajo. Especifica el tiempo que puede tardar en completarse una sola tarea de decisión para la ejecución de un flujo de trabajo de este tipo.

Si no especifica `defaultTaskStartToCloseTimeout`, se aplicará un valor predeterminado de 30 segundos.

Para obtener más información acerca de los tiempos de espera de los flujos de trabajo, consulte Tipos de tiempo de espera de Amazon SWF.

`defaultTaskList`

Se trata de la lista de tareas predeterminada que se utiliza en las tareas de decisión para las ejecuciones de este tipo de flujo de trabajo. La configuración predeterminada aquí se puede anular usando `StartWorkflowOptions` al comenzar la ejecución de un flujo de trabajo.

Si no especifica `defaultTaskList`, se establecerá en `USE_WORKER_TASK_LIST` de manera predeterminada. Esto indica que es preciso utilizar la lista de tareas que usa el trabajo que está haciendo el registro del flujo de trabajo.

`defaultChildPolicy`

Especifica la política que se debe utilizar con los flujos de trabajo secundarios si finaliza una ejecución de este tipo. El valor predeterminado es `ABANDON`. Los valores posibles son:

- `ABANDON`: permite que sigan funcionando las ejecuciones de flujos de trabajo secundarios.
- `TERMINATE`: finaliza las ejecuciones de flujos de trabajo secundarios.
- `REQUEST_CANCEL`: solicita que se cancelen las ejecuciones de flujos de trabajo secundarios.

AWS Flow Framework para excepciones de Java

AWS Flow Framework Para Java, utiliza las siguientes excepciones. En esta sección se proporciona información general sobre la excepción. Para obtener más información, consulte la AWS SDK para Java documentación de las excepciones individuales.

Temas

- ActivityFailureException
- ActivityTaskException

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

ActivityFailureException

El marco de trabajo utiliza esta excepción internamente para comunicar que se ha producido un error en una actividad. Cuando se produce un error en una actividad debido a una excepción no controlada, se encapsula en `ActivityFailureException` y se informa a Amazon SWF. Tiene que abordar esta excepción solo si usa los puntos de extensibilidad del proceso de trabajo de actividad. Su código de aplicación no tendrá nunca que enfrentarse a esta excepción.

ActivityTaskException

Esta es la clase de base para excepciones de error de tarea de actividad: `ScheduleActivityTaskFailedException`, `ActivityTaskFailedException`, `ActivityTaskTimedoutException`. Contiene el ID de la tarea y el tipo de actividad de la tarea con error. Puede detectar esta excepción en su implementación de flujo de trabajo para abordar errores de actividad de manera genérica.

ActivityTaskFailedException

Las excepciones sin administrar en actividades se notifican a la implementación de flujo de trabajo lanzando una `ActivityTaskFailedException`. La excepción original puede recuperarse de la

propiedad cause para esta excepción. La excepción también proporciona información adicional que es útil para fines de depuración, como por ejemplo el identificador de actividad única en el historial.

El marco de trabajo puede proporcionar la excepción remota serializando la excepción original del proceso de trabajo de actividad.

ActivityTaskTimedOutException

Se lanza esta excepción si Amazon SWF ha agotado el tiempo de espera de una actividad. Esto podría ocurrir si no ha sido posible asignar la tarea de la actividad al proceso de trabajo dentro del periodo de tiempo establecido o el proceso de trabajo no ha podido completarla dentro del tiempo establecido. Puede establecer estos tiempos de espera en la actividad mediante la anotación `@ActivityRegistrationOptions` o utilizando el parámetro `ActivitySchedulingOptions` al llamar al método de la actividad.

ChildWorkflowException

Clase de base para excepciones que se utiliza para informar de errores en la ejecución del flujo de trabajo secundario. La excepción contiene los ID de la ejecución de flujo de trabajo secundario así como su tipo de flujo de trabajo. Puede detectar esta excepción para abordar los errores de ejecución de flujo de trabajo secundario de manera genérica.

ChildWorkflowFailedException

Las excepciones sin administrar en flujos de trabajo secundarios se notifican a la implementación de flujo de trabajo principal lanzando una `ChildWorkflowFailedException`. La excepción original puede recuperarse de la propiedad `cause` para esta excepción. La excepción también proporciona información adicional que es útil para fines de depuración, como por ejemplo identificadores únicos de la ejecución secundaria.

ChildWorkflowTerminatedException

Esta excepción se lanza en la ejecución de flujo de trabajo principal para informar la terminación de una ejecución de flujo de trabajo secundario. Debería detectar esta excepción si desea abordar la terminación del flujo de trabajo secundario, por ejemplo, para realizar una limpieza o compensación.

ChildWorkflowTimedOutException

Esta excepción se produce en la ejecución del flujo de trabajo principal para informar de que Amazon SWF ha agotado el tiempo de espera y ha cerrado una ejecución de flujo de trabajo secundario.

Debería detectar esta excepción si desea abordar el cierre forzoso del flujo de trabajo secundario, por ejemplo, para realizar una limpieza o compensación.

DataConverterException

El marco de trabajo utiliza el componente `DataConverter` para serializar y anular la serialización de datos enviados a través del cable. Esta excepción se lanza si `DataConverter` produce un error al serializar o anular la serialización de datos. Esto podría ocurrir por diferentes motivos, por ejemplo, debido a una falta de coincidencia en los componentes `DataConverter` que se están utilizando para serializar o anular la serialización de datos.

DecisionException

Esta es la clase de base para las excepciones que representan errores para que Amazon SWF promulgue una decisión. Puede detectar esta excepción para abordar de manera genérica dichas excepciones.

ScheduleActivityTaskFailedException

Se produce esta excepción si Amazon SWF no consigue programar una tarea de actividad. Esto podría ocurrir por diferentes motivos, por ejemplo, porque la actividad se descartara, o bien porque se hubiera alcanzado algún límite de Amazon SWF en la cuenta. La propiedad `failureCause` en la excepción especifica la causa exacta del error para programar la actividad.

SignalExternalWorkflowException

Esta excepción se produce si Amazon SWF no consigue procesar una solicitud de la ejecución de flujo de trabajo para indicar otra ejecución de flujo de trabajo. Esto ocurre cuando no es posible encontrar la ejecución de flujo de trabajo de destino, es decir, cuando la ejecución del flujo de trabajo que se especificó no existe o está cerrada.

StartChildWorkflowFailedException

Se produce esta excepción si Amazon SWF no consigue iniciar una ejecución de flujo de trabajo secundario. Esto podría ocurrir por diferentes motivos, por ejemplo, porque el tipo de flujo de trabajo secundario especificado se descartara, o bien porque se hubiera alcanzado algún límite de Amazon SWF en la cuenta. La propiedad `failureCause` en la excepción especifica la causa exacta del error para comenzar la ejecución del flujo de trabajo secundario.

StartTimerFailedException

Se produce esta excepción si Amazon SWF no consigue iniciar un temporizador solicitado por la ejecución de flujo de trabajo. Esto podría ocurrir si el ID del temporizador especificado ya se está utilizando o bien si se ha alcanzado algún límite de Amazon SWF en la cuenta. La propiedad `failureCause` en la excepción especifica la causa exacta del error.

TimerException

Esta es la clase de base para excepciones relacionadas con temporizadores.

WorkflowException

El marco de trabajo utiliza esta excepción internamente para comunicar que se ha producido un error en la ejecución de un flujo de trabajo. Tiene que abordar esta excepción solo si usa un punto de extensibilidad del proceso de trabajo de flujo de trabajo.

AWS Flow Framework para paquetes Java

En esta sección se proporciona una descripción general de los paquetes incluidos en el paquete AWS Flow Framework para Java. Para obtener más información sobre cada paquete, consulte com.amazonaws.services.simpleworkflow.flow en la Referencia de la API de AWS SDK para Java.

com.amazonaws.services.simpleworkflow.flow

Contiene componentes que se integran con Amazon SWF.

com.amazonaws.services.simpleworkflow.flow.annotations

Contiene las anotaciones utilizadas por el modelo de programación AWS Flow Framework para Java.

com.amazonaws.services.simpleworkflow.flow.aspectj

Contiene AWS Flow Framework los componentes de Java necesarios para funciones como @Asynchronous y@ExponentialRetry.

com.amazonaws.services.simpleworkflow.flow.common

Contiene utilidades comunes como constantes definidas por el marco de trabajo.

com.amazonaws.services.simpleworkflow.flow.core

Contiene características esenciales como `Task` y `Promise`.

com.amazonaws.services.simpleworkflow.flow.generic

Contiene componentes esenciales, como clientes genéricos, sobre las que se basan otras características.

com.amazonaws.services.simpleworkflow.flow.interceptors

Contiene implementaciones de decoradores proporcionados por el marco de trabajo entre los que se incluye `RetryDecorator`.

com.amazonaws.services.simpleworkflow.flow.junit

Contiene componentes que proporcionan integración de Junit.

com.amazonaws.services.simpleworkflow.flow.pojo

Contiene clases que implementan definiciones de actividad y de flujo de trabajo para el modelo de programación basado en anotaciones.

com.amazonaws.services.simpleworkflow.flow.spring

Contiene componentes que proporcionan integración de Spring.

com.amazonaws.services.simpleworkflow.flow.test

Contiene clases de ayudantes, como `TestWorkflowClock`, para implementaciones de flujo de trabajo de pruebas de unidad.

com.amazonaws.services.simpleworkflow.flow.worker

Contiene implementaciones de procesos de trabajo de actividad y de flujo de trabajo.

Historial de documentos

En la siguiente tabla, se describen los cambios importantes que se han realizado en la documentación desde la última versión de la Guía para desarrolladores de AWS Flow Framework para Java.

- Versión deAPI: 2012-01-25
- Última actualización de la documentación 25 de junio de 2018

Cambio	Descripción	Fecha de modificación
Actualización	Se ha corregido un error en la descripción `backoffCoefficient` de `@ExponentialRetry` . Consulte @ExponentialRetry.	25 de junio de 2018
Actualización	Se han limpiado los ejemplos de código en toda esta guía.	5 de junio de 2017
Actualización	Se han simplificado y mejorado la organización y el contenido de esta guía.	19 de mayo de 2017
Actualización	Se ha simplificado y mejorado la sección Realización de cambios en el código del decisor: control de versiones y marcas de características.	10 de abril de 2017
Actualización	Se ha añadido la sección Prácticas recomendadas con nuevas directrices sobre cómo aplicar cambios en el código del decisor.	3 de marzo de 2017
Nueva caracterí stica	Puede especificar las tareas de Lambda, además de las tareas de actividad tradicionales en los flujos de trabajo. Para obtener más información, consulte Implementación de AWS Lambda tareas.	21 de julio de 2015

Cambio	Descripción	Fecha de modificación
Nueva caracterí stica	Amazon SWF permite establecer la prioridad de las tareas en una lista de tareas, de manera que se intente entregar las tareas con mayor prioridad antes que las de menor prioridad. Para obtener más información, consulte Configuración de la prioridad de las tareas en Amazon SWF.	17 de diciembre de 2014
Actualización	Se han realizado actualizaciones y correcciones.	1 de agosto de 2013
Actualización	• Se han realizado actualizaciones y correcciones, incluidas actualizaciones en las instrucciones de configuración para Eclipse 4.3 y AWS SDK para Java 1.4.7. • Se ha añadido un nuevo conjunto de tutoriales para comienzos	28 de junio de 2013
Nueva caracterí stica	La versión inicial de AWS Flow Framework para Java.	27 de febrero de 2012

Las traducciones son generadas a través de traducción automática. En caso de conflicto entre la traducción y la version original de inglés, prevalecerá la version en inglés.