aws

Guida per gli sviluppatori

AWS Flow Framework per Java

Versione API 2021-04-28

AWS Flow Framework per Java: Guida per gli sviluppatori

Table of Contents

Che cos'è AWS Flow Framework per Java?

Con AWS Flow Framework, puoi concentrarti sull'implementazione della logica del flusso di lavoro. Dietro le quinte, il framework utilizza le funzionalità di pianificazione, routing e gestione dello stato di Amazon SWF per gestire l'esecuzione del flusso di lavoro e renderlo scalabile, affidabile e verificabile. AWS Flow Framework i flussi di lavoro basati su di essi sono altamente simultanei. I flussi di lavoro possono essere distribuiti su più componenti, che possono essere eseguiti come processi separati su computer separati ed essere scalati indipendentemente. L'applicazione può continuare a progredire se uno dei suoi componenti è in esecuzione, il che la rende altamente tollerante ai guasti.

Cosa c'è in questa guida?

Questa guida contiene informazioni su come installare, configurare e utilizzare AWS Flow Framework per creare applicazioni Amazon SWF.

Guida introduttiva a AWS Flow Framework for Java

Se hai appena iniziato a usare la versione AWS Flow Framework per Java, leggi la Guida introduttiva a AWS Flow Framework for Java sezione. Ti guiderà attraverso il download e l'installazione di AWS Flow Framework per Java, come configurare il tuo ambiente di sviluppo e ti guiderà attraverso un semplice esempio di creazione di un flusso di lavoro.

Comprensione AWS Flow Framework di Java

Introduce Amazon SWF AWS Flow Framework e i concetti di base, descrivendo la struttura di base di AWS Flow Framework un'applicazione e come i dati vengono scambiati tra le parti di un flusso di lavoro distribuito.

AWS Flow Framework per la guida alla programmazione Java

Questo capitolo fornisce linee guida di programmazione di base per lo sviluppo di applicazioni di flusso di lavoro con Java, tra cui come registrare attività e tipi di flusso di lavoro, implementare client di flusso di lavoro, creare flussi di lavoro secondari, gestire errori e altro ancora. AWS Flow Framework

Comprensione di un task in AWS Flow Framework for Java

Questo capitolo fornisce un'analisi più approfondita del funzionamento di For Java, fornendo informazioni aggiuntive sull'ordine di esecuzione dei flussi di lavoro asincroni e una procedura logica di esecuzione di un flusso di lavoro standard. AWS Flow Framework

Suggerimenti per la risoluzione dei problemi e il debug per Java AWS Flow Framework

Questo capitolo presenta informazioni sugli errori comuni che puoi usare per risolvere i problemi dei flussi di lavoro o per imparare ad evitare gli errori comuni.

AWS Flow Framework per Java Reference

Questo capitolo è un riferimento alle annotazioni, alle eccezioni e ai pacchetti che AWS Flow Framework for Java aggiunge all'SDK per Java.

Guida introduttiva a AWS Flow Framework for Java

Questa sezione presenta una serie AWS Flow Framework di semplici applicazioni di esempio che introducono il modello di programmazione e l'API di base. Le applicazioni di esempio sono basate sull'applicazione standard Hello World, utilizzata per presentare C e i linguaggi di programmazione correlati. Ecco una tipica implementazione Java di Hello World:

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

Quella che segue è una breve descrizione delle applicazioni di esempio. Includono il codice sorgente completo per implementare ed eseguire autonomamente le applicazioni. Prima di iniziare, dovresti configurare il tuo ambiente di sviluppo e creare un progetto AWS Flow Framework per Java, come inConfigurazione di AWS Flow Framework per Java.

- HelloWorld Applicazione introduce le applicazioni di flusso di lavoro implementando Hello World come applicazione Java standard, ma strutturandola come applicazione di flusso di lavoro.
- HelloWorldWorkflow Applicazioneutilizza AWS Flow Framework for Java per la conversione HelloWorld in un flusso di lavoro Amazon SWF.
- HelloWorldWorkflowAsync Applicazione modifica `HelloWorldWorkflow` per utilizzare un metodo di flusso di lavoro asincrono.
- HelloWorldWorkflowDistributed Applicazione modifica `HelloWorldWorkflowAsync` in modo che il flusso di lavoro e i lavoratori di attività possano operare su sistemi separati.
- HelloWorldWorkflowParallel Applicazione modifica `HelloWorldWorkflow` per eseguire due attività in parallelo.

Configurazione di AWS Flow Framework per Java

Il AWS Flow Framework for Java è incluso in. AWS SDK per Java Se non l'hai ancora configurato AWS SDK per Java, consulta la sezione Guida introduttiva alla AWS SDK per Java Developer Guide per informazioni sull'installazione e la configurazione dell'SDK stesso.

Aggiungi il framework Flow con Maven

Gli strumenti di compilazione di Amazon SWF sono open source: per visualizzare o scaricare il codice o per creare gli strumenti da soli, visita il repository all'indirizzo. https://github.com/aws/aws-swf-build-tools

Amazon fornisce strumenti di compilazione Amazon SWF nel Maven Central Repository.

Per configurare il framework per Maven, aggiungi la seguente dipendenza al file `pom.xml` del tuo progetto:

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

HelloWorld Applicazione

Per presentare il modo in cui sono strutturate le applicazioni Amazon SWF, creeremo un'applicazione Java che si comporta come un flusso di lavoro, ma che viene eseguita localmente in un unico processo. Non è richiesta alcuna connessione ad Amazon Web Services.

> **Note**
>
> L'HelloWorldWorkflowesempio si basa su questo, la connessione ad Amazon SWF per gestire la gestione del flusso di lavoro.

Un'applicazione del flusso di lavoro è composta da tre componenti base:

- Un lavoratore di attività supporta un set di attività, ciascuna delle quali è un metodo che viene eseguito in modo indipendente per eseguire un determinato task.
- Un lavoratore di flusso di lavoro orchestra l'esecuzione delle attività e gestisce il flusso di dati. Si tratta della realizzazione programmatica di una topologia del flusso di lavoro che è sostanzialmente un digramma di flusso che definisce il momento in cui vengono eseguite le diverse attività, sia che vengano eseguite in modo sequenziale o contemporaneamente.
- Uno starter di flusso di lavoro avvia un'istanza di flusso di lavoro, chiamata esecuzione, e può interagire con essa durante l'esecuzione.

HelloWorld è implementato come tre classi e due interfacce correlate, descritte nelle sezioni seguenti. Prima di iniziare, è necessario configurare l'ambiente di sviluppo e creare un nuovo progetto AWS Java come descritto in Configurazione di AWS Flow Framework per Java. I pacchetti utilizzati per le seguenti procedure guidate sono stati tutti nominati `helloWorld.`*`XYZ`*. Per utilizzare questi nomi, imposta l'attributo `within` in aop.xml secondo quanto indicato di seguito:

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

Per implementarlo HelloWorld, create un nuovo pacchetto Java nel vostro progetto AWS SDK denominato `helloWorld.HelloWorld` e aggiungete i seguenti file:

- Un file di interfaccia denominato `GreeterActivities.java`
- Un file di classe denominato `GreeterActivitiesImpl.java`, che implementa il lavoratore di attività.
- Un file di interfaccia denominato `GreeterWorkflow.java`.
- Un file di classe denominato `GreeterWorkflowImpl.java`, che implementa il lavoratore di flusso di lavoro.
- Un file di classe denominato `GreeterMain.java`, che implementa lo starter di flusso di lavoro.

I dettagli sono illustrati nelle sezioni seguenti e includono il codice completo per ogni componente, che puoi aggiungere al file appropriato.

HelloWorld Attività: implementazione

HelloWorld suddivide l'operazione complessiva di stampa di un `"Hello World!"` messaggio di saluto sulla console in tre attività, ognuna delle quali viene eseguita con un metodo di attività. I metodi di attività sono definiti nell'interfaccia `GreeterActivities`, secondo quanto segue.

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

HelloWorld ha un'implementazione di attività`GreeterActivitiesImpl`, che fornisce i `GreeterActivities` metodi illustrati:

```
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);
   }
}
```

Le attività sono indipendenti una dall'altra e spesso possono essere utilizzate da diversi flussi di lavoro. Ad esempio, i flussi di lavoro che utilizzano l'attività `say` per visualizzare una stringa sulla console. I flussi di lavoro possono inoltre avere diverse implementazioni di attività e ognuna di essere esegue un set diverso di task.

HelloWorld Workflow Worker

Per stampare «Hello World!» sulla console, le attività devono essere eseguite in sequenza nell'ordine corretto con i dati corretti. L'addetto al HelloWorld workflow orchestra l'esecuzione delle attività sulla base di una semplice topologia lineare del flusso di lavoro, illustrata nella figura seguente.

Le tre attività vengono eseguite in sequenza e i dati fluiscono da un'attività a quella successiva.

L'operatore del HelloWorld workflow utilizza un unico metodo, il punto di ingresso del flusso di lavoro, definito nell'`GreeterWorkflow`interfaccia come segue:

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

```
}
```

La classe `GreeterWorkflowImpl` implementa l'interfaccia come mostrato di seguito:

```
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);
   }
}
```

Il `greet` metodo implementa la HelloWorld topologia creando un'istanza di`GreeterActivitiesImpl`, chiamando ogni metodo di attività nell'ordine corretto e passando i dati appropriati a ciascun metodo.

HelloWorld Workflow Starter

Uno starter di flusso di lavoro è un'applicazione che avvia l'esecuzione del flusso di lavoro e che può comunicare con il flusso di lavoro durante l'esecuzione. La `GreeterMain` classe implementa il HelloWorld workflow starter, come segue:

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

`GreeterMain` crea un'istanza di `GreeterWorkflowImpl` e chiama `greet` per eseguire il lavoratore di flusso di lavoro. Esegui `GreeterMain` come applicazione Java e dovresti vedere «Hello World!» nell'output della console.

HelloWorldWorkflow Applicazione

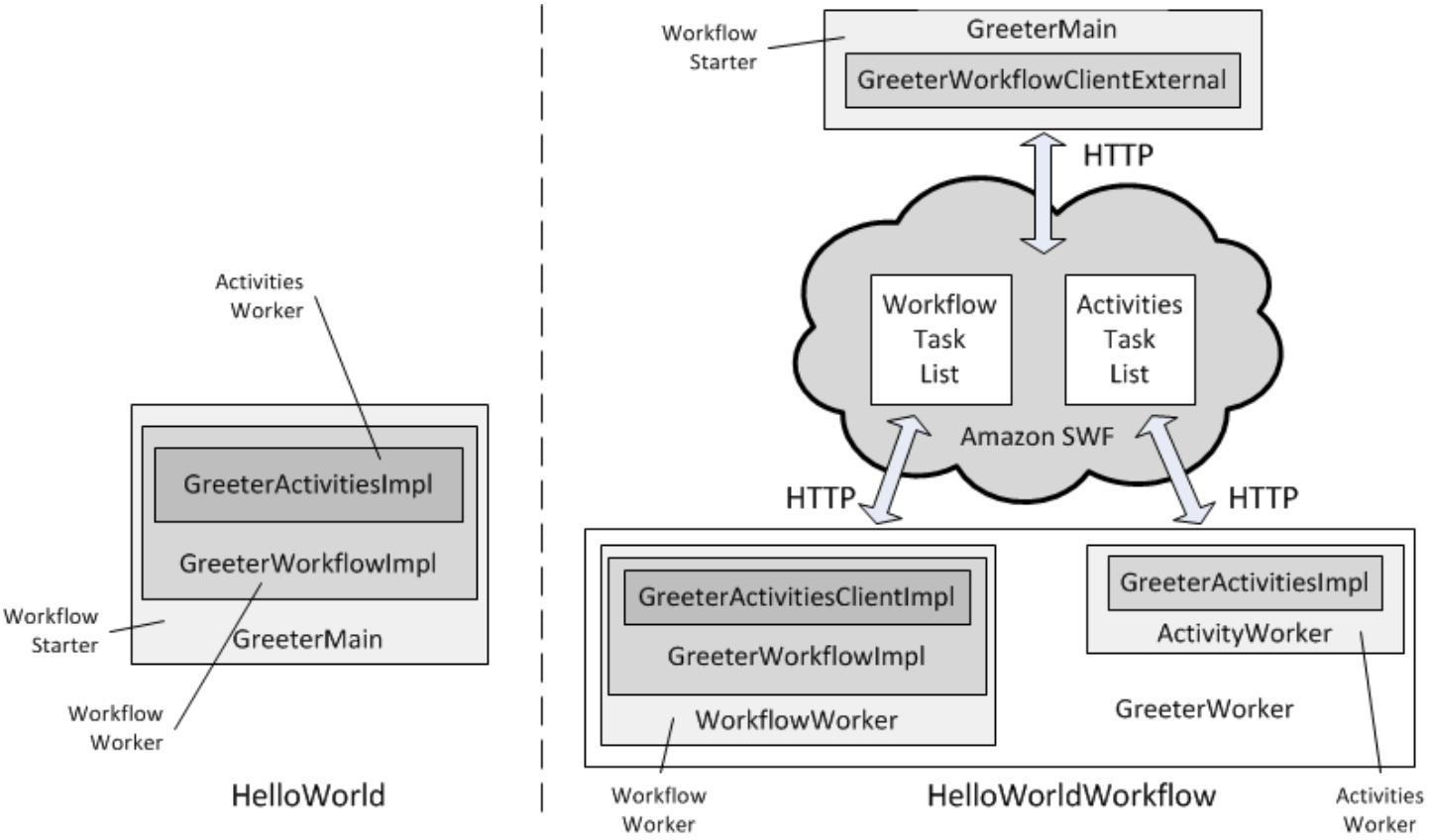
Sebbene l'HelloWorldesempio di base sia strutturato come un flusso di lavoro, si differenzia da un flusso di lavoro Amazon SWF per diversi aspetti chiave:

Applicazioni di workflow convenzionali e Amazon SWF

HelloWorld	Flusso di lavoro Amazon SWF
Viene eseguita localmente come singolo processo.	Viene eseguito come più processi che possono essere distribuiti su più sistemi, tra cui EC2 istanze Amazon, data center privati, computer client e così via. Non è necessario eseguirli sullo stesso sistema operativo.
Le attività sono metodi sincroni che vengono bloccati fino a che non risultano completati.	Le attività sono rappresentate da metodi asincroni, i quali restituiscono immediatamente un risultato e consenton o al flusso di lavoro di eseguire altri task in attesa del completamento dell'attività.
Il lavoratore di flusso di lavoro interagisce con un lavoratore di attività chiamando il metodo appropriato.	I lavoratori del flusso di lavoro interagiscono con gli addetti alle attività utilizzando richieste HTTP, con Amazon SWF che funge da intermediario.
Lo starter di flusso di lavoro interagis ce con un lavoratore di attività chiamando il metodo appropriato.	Gli avviatori di flussi di lavoro interagiscono con gli operatori del flusso di lavoro utilizzando richieste HTTP, con Amazon SWF che funge da intermediario.

Implementare un'applicazione di flusso di lavoro asincrona distribuita da zero, ad esempio, facendo in modo che il lavoratore di flusso di lavoro interagisca direttamente con un lavoratore di attività mediante chiamate di servizi Web, è possibile. Tuttavia, ciò comporterebbe l'implementazione di tutto il codice complesso necessario a gestire l'esecuzione asincrona di molteplici attività, controllare il flusso di dati, ecc. The AWS Flow Framework for Java e Amazon SWF si occupano di tutti questi dettagli, il che ti consente di concentrarti sull'implementazione della logica aziendale.

HelloWorldWorkflow è una versione modificata HelloWorld che funziona come flusso di lavoro Amazon SWF. L'illustrazione seguente riepiloga il funzionamento delle due applicazioni.



HelloWorld viene eseguito come un unico processo e starter, workflow worker e Activities Worker interagiscono utilizzando chiamate di metodo convenzionali. Con`HelloWorldWorkflow`, starter, workflow worker e activities worker sono componenti distribuiti che interagiscono tramite Amazon SWF utilizzando richieste HTTP. Amazon SWF gestisce l'interazione mantenendo elenchi di attività e flussi di lavoro, che invia ai rispettivi componenti. Questa sezione descrive come funziona il framework. HelloWorldWorkflow

HelloWorldWorkflow viene implementato utilizzando l'API AWS Flow Framework for Java, che gestisce i dettagli a volte complicati dell'interazione con Amazon SWF in background e semplifica notevolmente il processo di sviluppo. Puoi utilizzare lo stesso progetto per cui lo hai creato HelloWorld, che è già configurato per AWS Flow Framework le applicazioni Java. Tuttavia, per eseguire l'applicazione, è necessario configurare un account Amazon SWF, come segue:

- Crea un AWS account, se non ne hai già uno, su Amazon Web Services.
- Assegna l'ID di accesso e l'ID segreto del tuo account rispettivamente alle variabili AWS_ACCESS_KEY_ID e di AWS_SECRET_KEY ambiente. È vivamente sconsigliato esporre i valori di chiave letterali nel codice. L'assegnazione di tali chiavi a variabili di ambiente è un modo pratico di gestire il problema.

- Registrati per creare un account Amazon SWF su Amazon Simple Workflow Service.
- Accedi Console di gestione AWS e seleziona il servizio Amazon SWF.
- Scegli Gestisci domini nell'angolo in alto a destra e registra un nuovo dominio Amazon SWF. Un dominio è un contenitore logico per le risorse dell'applicazione, come i tipi di flusso di lavoro e attività e le esecuzioni di flusso di lavoro. Puoi utilizzare qualsiasi nome di dominio conveniente, ma nelle procedure dettagliate viene utilizzato "». helloWorldWalkthrough

Per implementare HelloWorldWorkflow, crea una copia di HelloWorld. HelloWorld pacchetto nella directory del progetto e chiamalo HelloWorld. HelloWorldWorkflow. Le seguenti sezioni descrivono come modificare il HelloWorld codice originale per utilizzarlo AWS Flow Framework per Java ed eseguirlo come applicazione di workflow Amazon SWF.

HelloWorldWorkflow Addetto alle attività

HelloWorld ha implementato le sue attività come un'unica classe. An AWS Flow Framework for Java Activities Worker ha tre componenti di base:

- I metodi di attività, che eseguono le attività effettive, sono definiti in un'interfaccia e implementati in una classe correlata.
- Una ActivityWorkerclasse gestisce l'interazione tra i metodi di attività e Amazon SWF.
- Un'applicazione host di attività registra e avvia il lavoratore di attività e gestisce la pulizia.

Questa sezione descrive i metodi di attività. Le altre due classi sono presentate in una sezione successiva.

HelloWorldWorkflow definisce l'interfaccia delle attività in`GreeterActivities`, come segue:

```
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);
}
```

Questa interfaccia non era strettamente necessaria per HelloWorld, ma lo è AWS Flow Framework per un'applicazione Java. Nota che la definizione dell'interfaccia non è cambiata. Tuttavia, è necessario applicarne due AWS Flow Framework per le annotazioni Java @ActivityRegistrationOptions e@Activities, alla definizione dell'interfaccia. Le annotazioni forniscono informazioni di configurazione e indicano al AWS Flow Framework processore di annotazioni Java di utilizzare la definizione dell'interfaccia per generare una classe client di attività, argomento discusso più avanti.

`@ActivityRegistrationOptions`sha diversi valori denominati che vengono utilizzati per configurare il comportamento delle attività. HelloWorldWorkflow specifica due timeout:

- `defaultTaskScheduleToStartTimeoutSeconds` indica per quanto tempo i task possono rimanere in coda nell'elenco di task di attività; il valore impostato è 300 secondi (5 minuti).
- `defaultTaskStartToCloseTimeoutSeconds` indica il tempo massimo di cui l'attività dispone per eseguire il task; il valore impostato è 10 secondi.

Questi timeout assicurano il completamento del task entro un tempo ragionevole. Se uno dei due timeout viene superato, il framework genera un errore e il lavoratore di flusso di lavoro deve decidere come gestire il problema. Per informazioni su come gestire tale errori, consulta Gestione errori.

`@Activities` comporta vari valori, ma in genere definisce soltanto il numero di versione delle attività, mediante il quale puoi tenere traccia di differenti generazioni di implementazioni di attività. Se modifichi un'interfaccia di attività dopo averla registrata in Amazon SWF, inclusa la modifica `@ActivityRegistrationOptions` dei valori, devi utilizzare un nuovo numero di versione.

HelloWorldWorkflow implementa i metodi di attività in`GreeterActivitiesImpl`, come segue:

```
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);
    }
}
```

Notate che il codice è identico all' HelloWorld implementazione. Fondamentalmente, un' AWS Flow Framework attività è solo un metodo che esegue del codice e forse restituisce un risultato. La differenza tra un'applicazione standard e un'applicazione di workflow Amazon SWF risiede nel modo in cui il flusso di lavoro esegue le attività, dove vengono eseguite le attività e in che modo i risultati vengono restituiti al workflow worker.

HelloWorldWorkflow Workflow Worker

Un workflow worker di Amazon SWF ha tre componenti di base.

- Un'implementazione di flusso di lavoro, ovvero una classe che esegue task correlati al flusso di lavoro.
- Una classe client di attività, che è in pratica un proxy per la classe di attività e viene utilizzata da un'implementazione di flusso di lavoro per eseguire metodi di attività in modo asincrono.
- Una WorkflowWorkerclasse che gestisce l'interazione tra il flusso di lavoro e Amazon SWF.

Questa sezione descrive l'implementazione di flusso di lavoro e il client di attività; la classe `WorkflowWorker` è descritta in una sezione successiva.

HelloWorldWorkflow definisce l'interfaccia del flusso di lavoro in`GreeterWorkflow`, come segue:

```
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();
}
```

Inoltre, questa interfaccia non è strettamente necessaria HelloWorld , ma è essenziale AWS Flow Framework per un'applicazione Java. È necessario applicarne due AWS Flow Framework

per le annotazioni Java @Flusso di lavoro e@WorkflowRegistrationOptions, per la definizione dell'interfaccia del flusso di lavoro. Le annotazioni forniscono informazioni di configurazione e indirizzano inoltre il processore di annotazioni AWS Flow Framework per Java a generare una classe client di workflow basata sull'interfaccia, come discusso più avanti.

`@Workflow`ha un parametro opzionale, DataConverter, che viene spesso utilizzato con il suo valore NullDataConverter predefinito, che indica che deve essere utilizzato. JsonDataConverter

`@WorkflowRegistrationOptions` comporta ha vari parametri facoltativi che possono essere utilizzati per configurare il lavoratore di flusso di lavoro. Qui, impostiamo, `defaultExecutionStartToCloseTimeoutSeconds` che specifica per quanto tempo può essere eseguito il flusso di lavoro, a 3600 secondi (1 ora).

La definizione dell'`GreeterWorkflow`interfaccia differisce da un aspetto importante, l' HelloWorld annotazione. @Execute Le interfacce di flusso di lavoro definiscono i metodi che possono essere chiamati dalle applicazioni come lo starter di flusso di lavoro e sono limitate a pochi metodi, ognuno con un ruolo particolare. Il framework non specifica un nome o un elenco di parametri per i metodi di interfaccia del flusso di lavoro; si utilizza un elenco di nomi e parametri adatto al flusso di lavoro e si applica un'annotazione AWS Flow Framework per Java per identificare il ruolo del metodo.

`@Execute` ha due scopi:

- Identifica `greet` come punto di ingresso del flusso di lavoro, ovvero il metodo che lo starter di flusso di lavoro chiama per avviare il flusso di lavoro. In genere, un punto di ingresso può accettare uno o più parametri, che consentono allo starter di inizializzare il flusso di lavoro, ma questo esempio non richiede l'inizializzazione.
- Definisce il numero di versione del flusso di lavoro, mediante il quale puoi tenere traccia di differenti generazioni di implementazioni di flusso di lavoro. Per modificare l'interfaccia di un flusso di lavoro dopo averla registrata in Amazon SWF, inclusa la modifica dei valori di timeout, devi utilizzare un nuovo numero di versione.

Per informazioni sugli altri metodi che possono essere inclusi in un'interfaccia di flusso di lavoro, consulta Contratti di flusso di lavoro e attività.

HelloWorldWorkflow implementa il flusso di lavoro in`GreeterWorkflowImpl`, come segue:

```
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);
   }
}
```

Il codice è simile a HelloWorld, ma presenta due importanti differenze.

- `GreeterWorkflowImpl` crea un'istanza di `GreeterActivitiesClientImpl`, il client di attività, anziché di `GreeterActivitiesImpl`, ed esegue le attività chiamando i metodi sull'oggetto client.
- Le attività relative a nome e formula di apertura restituiscono oggetti `Promise<String>` anziché oggetti `String`.

HelloWorld è un'applicazione Java standard che viene eseguita localmente come un singolo processo, quindi `GreeterWorkflowImpl` può implementare la topologia del flusso di lavoro semplicemente creando un'istanza di`GreeterActivitiesImpl`, chiamando i metodi in ordine e passando i valori restituiti da un'attività all'altra. Con un flusso di lavoro Amazon SWF, l'attività di un'attività viene comunque eseguita con un metodo di attività di. `GreeterActivitiesImpl` Tuttavia, il metodo non viene necessariamente eseguito nello stesso processo del flusso di lavoro (può addirittura non essere eseguito sullo stesso sistema) e il flusso di lavoro deve eseguire l'attività in modo asincrono. Queste condizioni comportano le seguenti problematiche:

- Come eseguire un metodo di attività che può essere eseguito in un processo differente ed eventualmente su un sistema differente.
- Come eseguire un metodo di attività in modo asincrono.
- Come gestire i valori di input e restituiti delle attività. Ad esempio, se il valore restituito dell'Attività A è un input all'Attività B, devi assicurarti che l'Attività B non venga eseguita fino a che l'Attività A non risulta completata.

Il flusso di controllo dell'applicazione ti consente di implementare varie topologie di flusso di lavoro mediante l'utilizzo del controllo di flusso Java standard combinato con il client di attività e `Promise<T>`.

Client di attività

`GreeterActivitiesClientImpl` è fondamentalmente un proxy per `GreeterActivitiesImpl` che consente a un'implementazione di flusso di lavoro di eseguire i metodi `GreeterActivitiesImpl` in modo asincrono.

Le classi `GreeterActivitiesClient` e `GreeterActivitiesClientImpl` sono generate automaticamente utilizzando le informazioni fornite nelle annotazioni applicate alla classe `GreeterActivities`. Non devi quindi implementarle personalmente.

> **Note**
>
> Eclipse genera queste classi quando salvi il progetto. Puoi visualizzare il codice generato nella sottodirectory `.apt_generated` della directory del progetto.
> Per evitare errori di compilazione nella classe `GreeterWorkflowImpl`, è consigliabile spostare la directory `.apt_generated` nella parte superiore della scheda Order and Export (Ordina ed esporta) della finestra di dialogo Java Build Path (Percorso di compilazione Java).

Un lavoratore di flusso di lavoro esegue un'attività chiamando il metodo di client corrispondente. Il metodo è asincrono e restituisce immediatamente un oggetto `Promise<T>`, dove T è il tipo restituito dell'attività. L'oggetto `Promise<T>` restituito è in pratica un segnaposto per il valore che il metodo di attività restituirà.

- Quando il metodo di client di attività restituisce un risultato, lo stato dell'oggetto `Promise<T>` è inizialmente non pronto, a indicare che l'oggetto non rappresenta ancora un valore restituito valido.
- Quando il metodo di attività corrispondente completa il relativo task e restituisce un risultato, il framework assegna il valore restituito all'oggetto `Promise<T>`, il cui stato diventa pronto.

Tipo di Promise <T>

Lo scopo primario degli oggetti `Promise<T>` è gestire il flusso di dati tra i componenti asincroni e controllare quando vengono eseguiti. Grazie a questi oggetti, la tua applicazione non deve gestire in modo esplicito la sincronizzazione o dipendere da meccanismi come i timer per impedire l'esecuzione prematura dei componenti asincroni. Quando chiami un metodo di client di attività, questo restituisce immediatamente un risultato ma il framework ritarda l'esecuzione del metodo di attività corrispondente fino a che un oggetto `Promise<T>` di input è pronto e rappresenta dati validi.

Dalla prospettiva `GreeterWorkflowImpl`, i tre metodi di client di attività restituiscono un risultato immediatamente. Dalla prospettiva `GreeterActivitiesImpl`, il framework chiama `getGreeting` solo quando `name` risulta completato e chiama `say` solo quando `getGreeting` risulta completato.

L'utilizzo di `Promise<T>` per passare dati da un'attività a quella successiva consente a `HelloWorldWorkflow` di impedire ai metodi di attività di tentare di utilizzare dati non validi, ma anche di determinare quando le attività vengono eseguite e di definire implicitamente la topologia di flusso di lavoro. Il passaggio del valore restituito `Promise<T>` di ogni attività all'attività successiva richiede l'esecuzione in sequenza delle attività, definendo la topologia lineare descritta precedentemente. Con AWS Flow Framework for Java, non è necessario utilizzare alcun codice di modellazione speciale per definire topologie anche complesse, ma solo il controllo di flusso Java standard e. `Promise<T>` Per un esempio di implementazione di una topologia parallela semplice, consulta HelloWorldWorkflowParallel Attività: Lavoratore.

> **Note**
>
> Quando un metodo di attività come `say` non restituisce un valore, il metodo di client corrispondente restituisce un oggetto `Promise<Void>`. L'oggetto non rappresenta dati, ma è inizialmente non pronto e diventa pronto quando l'attività è completata. Puoi quindi passare un oggetto `Promise<Void>` a altri metodi di client di attività per assicurarti che questi differiscano l'esecuzione fino al completamento dell'attività originale.

`Promise<T>` consente a un'implementazione di flusso di lavoro di utilizzare metodi di client di attività e i relativi valori restituiti come con i metodi sincroni. Devi tuttavia prestare attenzione riguardo all'accesso al valore di un oggetto `Promise<T>`. A differenza del tipo Java Future<T>, il framework gestisce la sincronizzazione per `Promise<T>`, non l'applicazione. Se chiami `Promise<T>.get` e l'oggetto non è pronto, `get` genera un'eccezione. Nota che `HelloWorldWorkflow` non accede mai direttamente a un oggetto `Promise<T>`, ma passa semplicemente gli oggetti da un'attività a quella successiva. Quando un oggetto diventa pronto, il framework estrae il valore e lo passa al metodo di attività come tipo standard.

L'accesso agli oggetti `Promise<T>` deve avvenire solo tramite codice asincrono, dove il framework garantisce che l'oggetto è pronto e rappresenta un valore valido. `HelloWorldWorkflow` gestisce questa condizione passando gli oggetti `Promise<T>` solo a metodi di client di attività. Puoi accedere al valore di un oggetto `Promise<T>` nell'implementazione di flusso di lavoro passando l'oggetto a un metodo di flusso di lavoro asincrono, il cui comportamento è simile a quello di un'attività. Per vedere un esempio, consulta HelloWorldWorkflowAsync Applicazione.

HelloWorldWorkflow Implementazione del workflow e delle attività

Le implementazioni del flusso di lavoro e delle attività hanno classi di lavoro associate ActivityWorkere WorkflowWorker. Gestiscono la comunicazione tra Amazon SWF e le attività e le implementazioni del flusso di lavoro analizzando l'elenco di attività di Amazon SWF appropriato, eseguendo il metodo appropriato per ogni attività e gestendo il flusso di dati. Per maggiori dettagli, consulta AWS Flow Framework Concetti di base: struttura dell'applicazione.

Per associare le implementazioni di flusso di lavoro e attività agli oggetti lavoratore corrispondenti, devi implementare una o più applicazioni lavoratore che:

- Registra flussi di lavoro o attività con Amazon SWF.
- Creano oggetti lavoratore e li associano alle implementazioni di lavoratore di attività o di flusso di lavoro.
- Indirizza gli oggetti di lavoro affinché inizino a comunicare con Amazon SWF.

Se intendi eseguire il flusso di lavoro e le attività come processi distinti, devi implementare host lavoratore di flusso di lavoro e attività distinti. Per vedere un esempio, consulta HelloWorldWorkflowDistributed Applicazione. Per semplicità, HelloWorldWorkflow implementa un singolo host di lavoro che gestisce le attività e i lavoratori del flusso di lavoro nello stesso processo, come segue:

```
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`non ha una HelloWorld controparte, quindi è necessario aggiungere una classe Java denominata `GreeterWorker` al progetto e copiare il codice di esempio in quel file.

Il primo passaggio consiste nel creare e configurare un AmazonSimpleWorkflowClientoggetto, che richiama i metodi di servizio Amazon SWF sottostanti. A questo proposito, `GreeterWorker`:

1. Crea un ClientConfigurationoggetto e specifica un timeout del socket di 70 secondi. Questo valore definisce il tempo di attesa per il trasferimento dei dati via una connessione aperta stabilita prima della chiusura del socket.
2. Crea un AWSCredentials oggetto Basic per identificare l' AWS account e passa le chiavi dell'account al costruttore. Per comodità e per evitare di esporle come testo normale nel codice, le chiavi sono memorizzate come variabili di ambiente.
3. Crea un AmazonSimpleWorkflowClientoggetto per rappresentare il flusso di lavoro e passa gli `ClientConfiguration` oggetti `BasicAWSCredentials` and al costruttore.
4. Imposta l'URL dell'endpoint del servizio dell'oggetto client. Amazon SWF è attualmente disponibile in tutte le AWS regioni.

Per comodità, `GreeterWorker` definisce due costanti di stringa.

- `domain`è il nome di dominio Amazon SWF del flusso di lavoro, che hai creato quando hai configurato il tuo account Amazon SWF. `HelloWorldWorkflow`presuppone che stiate eseguendo il flusso di lavoro nel dominio "»helloWorldWalkthrough.

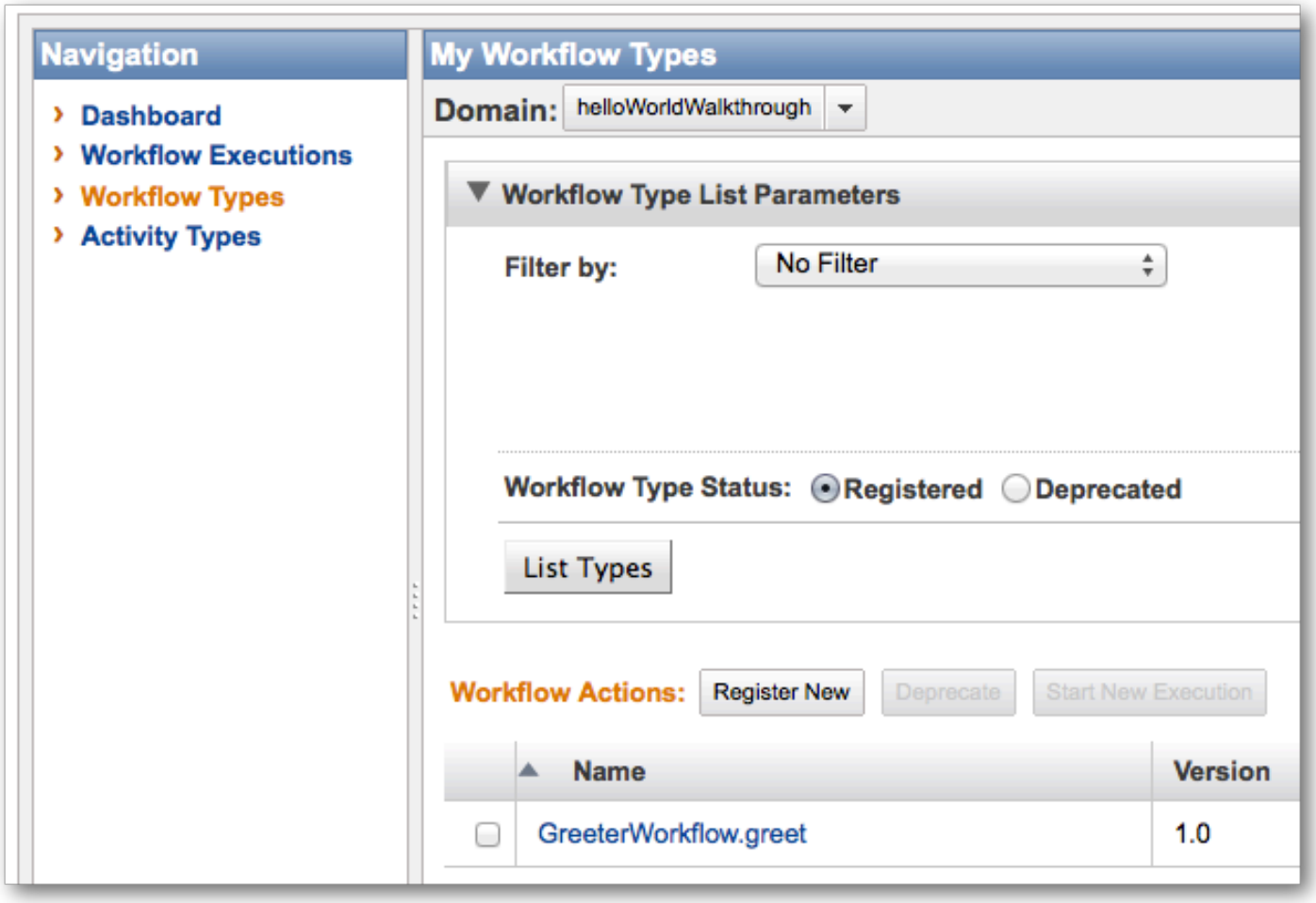
- `taskListToPoll`è il nome degli elenchi di attività utilizzati da Amazon SWF per gestire la comunicazione tra gli addetti al flusso di lavoro e alle attività. Puoi impostare il nome su qualsiasi stringa conveniente. HelloWorldWorkflow utilizza "HelloWorldList" sia per il flusso di lavoro che per gli elenchi delle attività. I nomi terminano con spazi dei nomi differenti, di conseguenza gli elenchi di task sono distinti.

`GreeterWorker`utilizza le costanti di stringa e l'AmazonSimpleWorkflowClientoggetto per creare oggetti di lavoro, che gestiscono l'interazione tra le attività e le implementazioni dei worker e Amazon SWF. In particolare, gli oggetti lavoratore gestiscono il task di polling dei task nell'elenco di task appropriato.

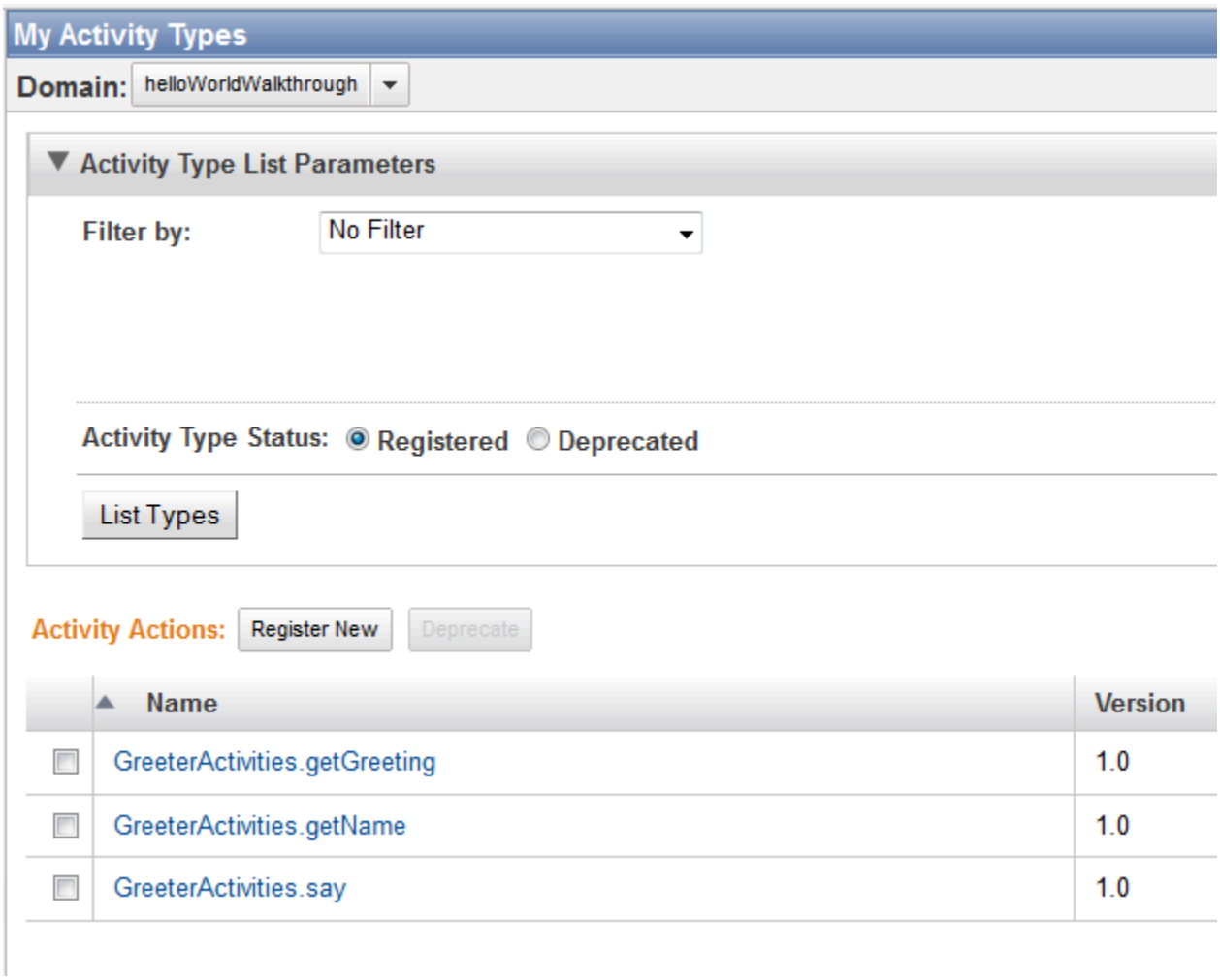
`GreeterWorker` crea un oggetto `ActivityWorker` e lo configura per gestire `GreeterActivitiesImpl` aggiungendo una nuova istanza della classe. `GreeterWorker` chiama quindi il metodo `start` dell'oggetto `ActivityWorker`, che indica all'oggetto di avviare il polling nell'elenco di task di attività specificato.

`GreeterWorker` crea un oggetto `WorkflowWorker` e lo configura per gestire `GreeterWorkflowImpl` aggiungendo un nome di file di classe, `GreeterWorkflowImpl.class`. Chiama quindi il metodo `start` dell'oggetto `WorkflowWorker`, che indica all'oggetto di avviare il polling dell'elenco di task di flusso di lavoro specificato.

A questo punto, puoi eseguire `GreeterWorker` senza problemi. Registra il flusso di lavoro e le attività con Amazon SWF e avvia gli oggetti di lavoro analizzando i rispettivi elenchi di attività. Per verificarlo, esegui `GreeterWorker` e accedi alla console Amazon SWF e seleziona `helloWorldWalkthrough` dall'elenco dei domini. Se scegli Workflow Types (Tipi di flusso di lavoro) nel riquadro Navigation (Navigazione), `GreeterWorkflow.greet` dovrebbe essere visualizzato nella finestra:



Se scegli Activity Types (Tipi di attività), vengono visualizzati i metodi `GreeterActivities`:



Tuttavia, se scegli Workflow Executions (Esecuzioni di flusso di lavoro), non verrà visualizzata alcuna esecuzione attiva. Sebbene i lavoratori di flusso di lavoro e di attività eseguano il polling di task, non abbiamo ancora avviato un'esecuzione di flusso di lavoro.

HelloWorldWorkflow Antipasto

L'ultimo pezzo del puzzle consiste nell'implementare uno starter di flusso di lavoro, ovvero un'applicazione che avvia l'esecuzione di flusso di lavoro. Lo stato di esecuzione viene memorizzato da Amazon SWF, in modo da poterne visualizzare la cronologia e lo stato di esecuzione. HelloWorldWorkflow implementa un sistema di avvio del flusso di lavoro modificando la `GreeterMain` classe nel modo seguente:

```
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 oggetto `AmazonSimpleWorkflowClient` utilizzando lo stesso codice di `GreeterWorker`. Crea quindi un oggetto `GreeterWorkflowClientExternal` che agisce come proxy per il flusso di lavoro nello stesso modo in cui il client di attività creato in `GreeterWorkflowClientImpl` agisce come proxy per i metodi di attività. Anziché creare un oggetto client di flusso di lavoro utilizzando `new` devi:

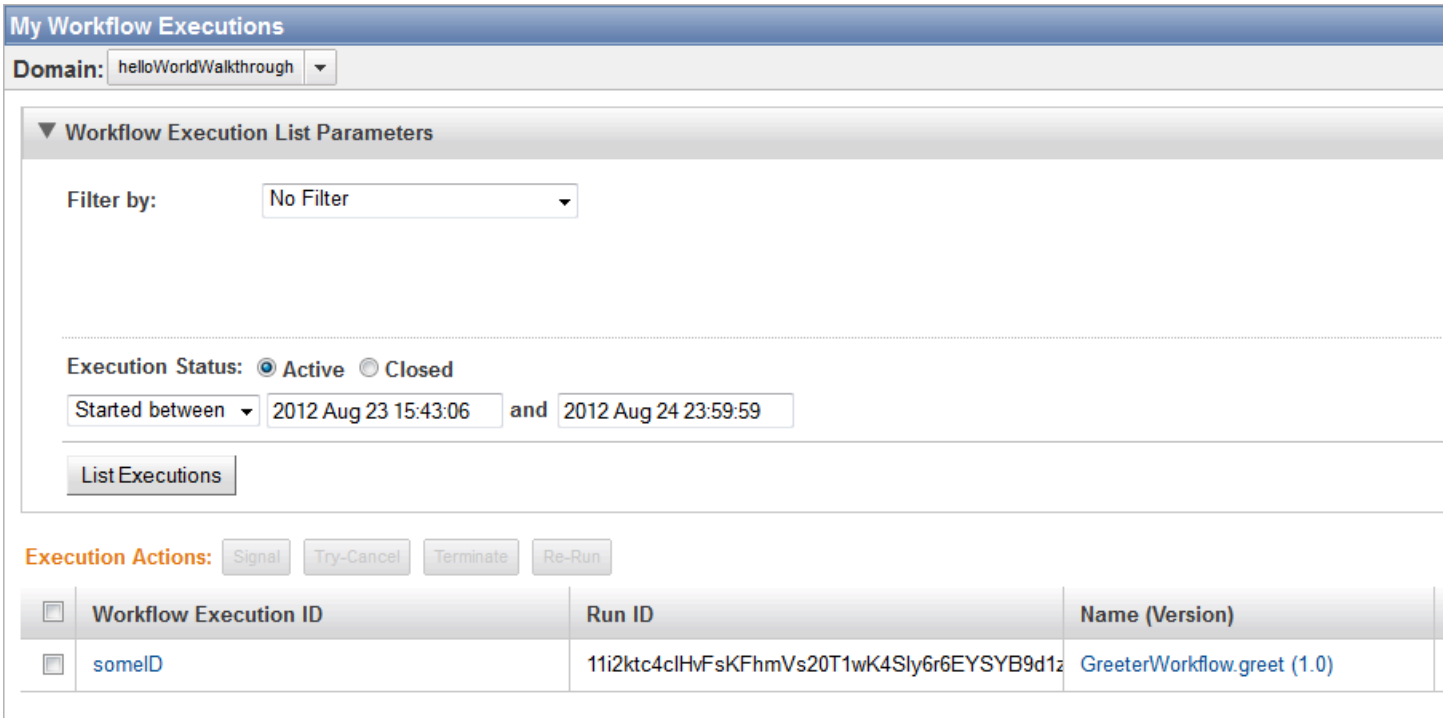
1. Crea un oggetto client factory esterno e passa l'`AmazonSimpleWorkflowClient`oggetto e il nome di dominio Amazon SWF al costruttore. L'oggetto client factory viene creato dal processore di annotazioni del framework, che crea il nome dell'oggetto semplicemente aggiungendo "ClientExternalFactoryImpl" al nome dell'interfaccia del flusso di lavoro.
2. Crea un oggetto client esterno chiamando il `getClient` metodo dell'oggetto factory, che crea il nome dell'oggetto aggiungendo "ClientExternal" al nome dell'interfaccia del flusso di lavoro. Facoltativamente, puoi passare `getClient` una stringa che Amazon SWF utilizzerà per identificare questa istanza del flusso di lavoro. Altrimenti, Amazon SWF rappresenta un'istanza di flusso di lavoro utilizzando un GUID generato.

Il client restituito dalla fabbrica creerà solo flussi di lavoro denominati con la stringa passata al metodo getClient (il client restituito dalla fabbrica ha già lo stato in Amazon SWF). Per eseguire un flusso di lavoro con un ID differente, nella factory devi creare un nuovo client con l'ID differente specificato.

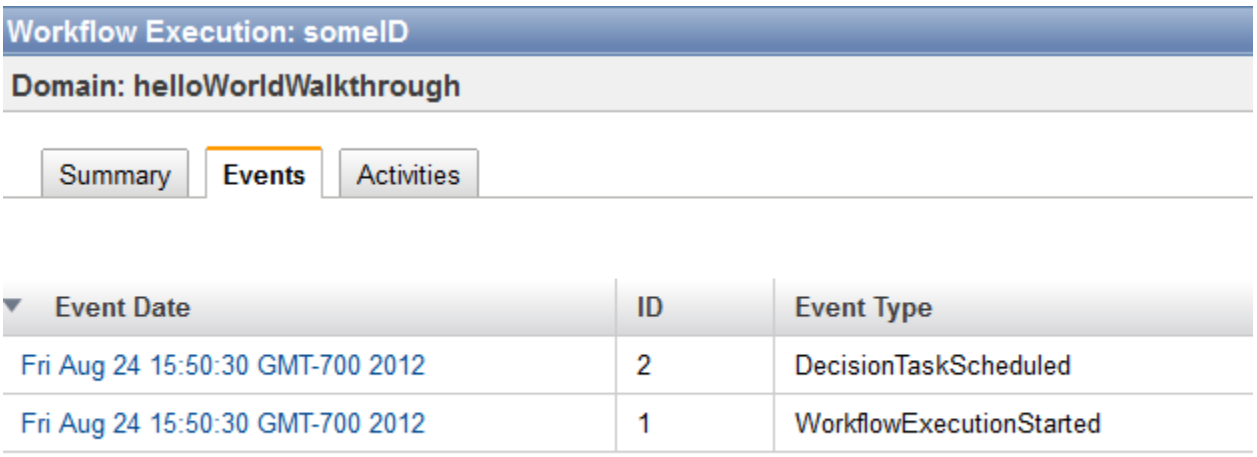
Il client di flusso di lavoro espone un metodo `greet` che `GreeterMain` chiama per iniziare il flusso di lavoro, in quanto `greet()` era il metodo specificato con l'annotazione `@Execute`.

> **Note**
>
> Il processore di annotazione crea anche un oggetto client factory interno utilizzato per creare flussi di lavoro figlio. Per informazioni dettagliate, consultare Esecuzioni del flusso di lavoro figlio.

Chiudi `GreeterWorker` se è ancora in esecuzione ed esegui `GreeterMain`. Ora dovresti vedere SomeID nell'elenco delle esecuzioni di flussi di lavoro attive della console Amazon SWF:.



Se scegli `someID` e quindi la scheda Events (Eventi), gli eventi vengono visualizzati:

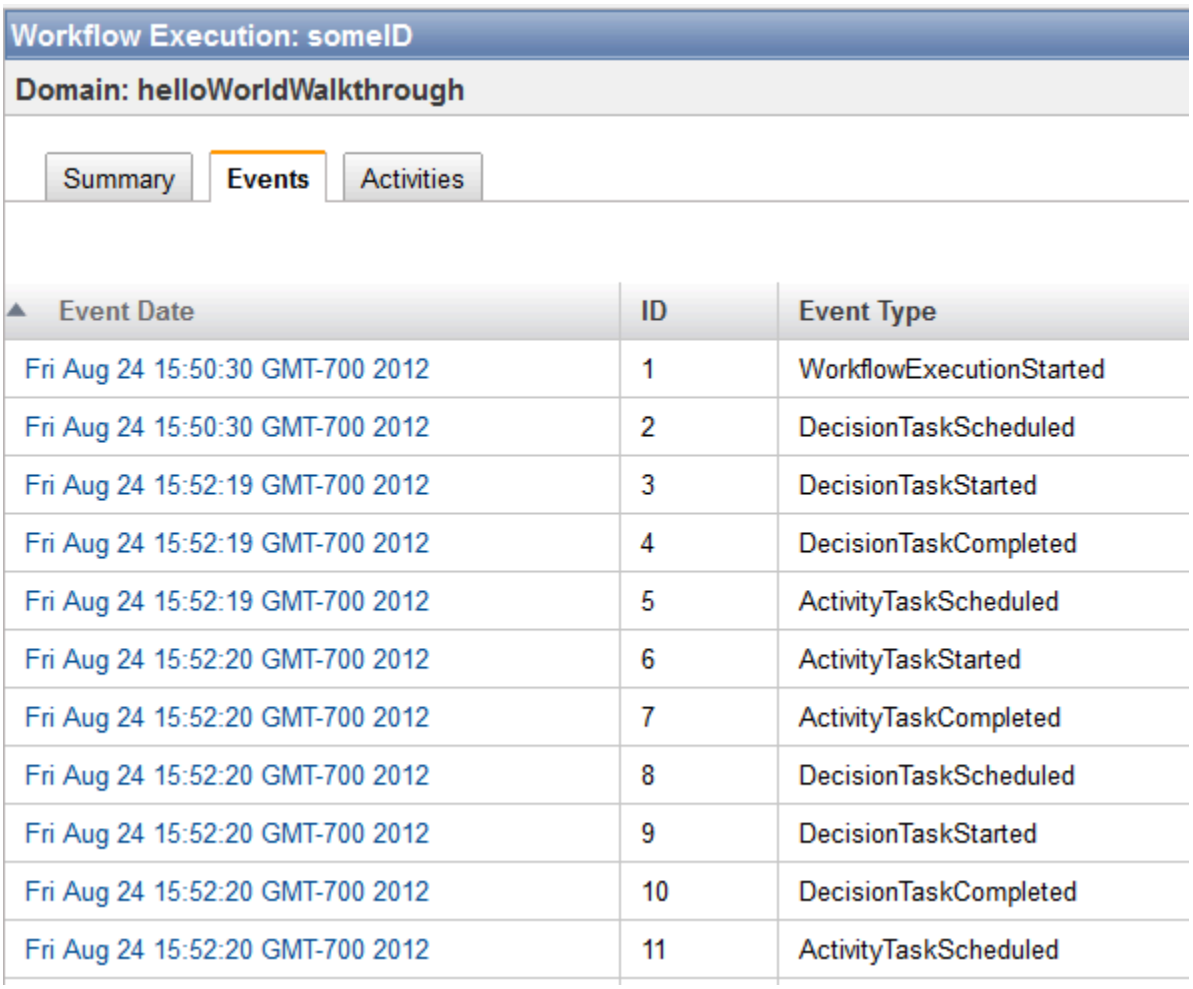


> **Note**
>
> Se in precedenza hai avviato `GreeterWorker` ed è ancora in esecuzione, l'elenco di eventi sarà più lungo per i motivi che indicheremo più avanti. Chiudi `GreeterWorker` ed esegui di nuovo `GreaterMain`.

Nella scheda Events (Eventi) sono elencati solo due eventi:

- `WorkflowExecutionStarted` indica che l'esecuzione del flusso di lavoro è stata avviata.
- `DecisionTaskScheduled`indica che Amazon SWF ha messo in coda la prima operazione decisionale.

Il motivo per cui il flusso di lavoro è bloccato a livello del primo task di decisione è che il flusso di lavoro è distribuito su due applicazioni, `GreeterMain` e `GreeterWorker`. `GreeterMain` ha avviato l'esecuzione di flusso di lavoro, ma `GreeterWorker` non è in esecuzione. Di conseguenza, i lavoratori non effettuano il polling negli elenchi e non eseguono i task. Puoi eseguire l'una o l'altra delle applicazioni indipendentemente, ma hai bisogno di entrambe affinché l'esecuzione di flusso di lavoro continui oltre il primo task di decisione. Se quindi a questo punto esegui `GreeterWorker`, i lavoratori di flusso di lavoro e attività avvieranno il polling, i task saranno completati rapidamente e nella scheda `Events` verrà visualizzato il primo batch di eventi.



Workflow Execution: someID

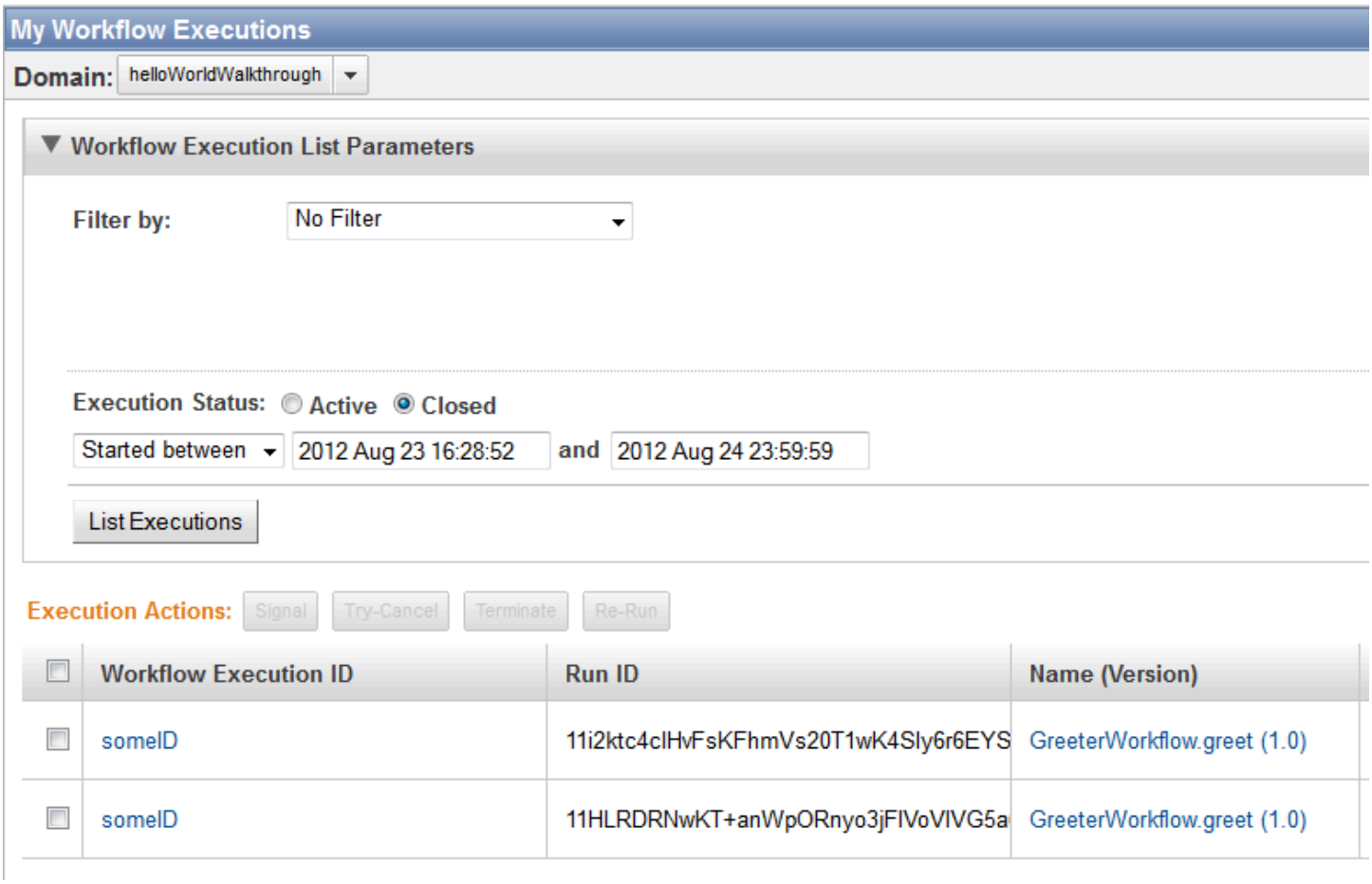
Domain: helloWorldWalkthrough

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

Puoi scegliere singoli eventi per visualizzare ulteriori informazioni sugli stessi. Quando avrai finito di cercare, il flusso di lavoro dovrebbe avere stampato «Hello World!» sulla tua console.

Una volta completato, il flusso di lavoro non è più visibile nell'elenco di esecuzioni attive. Tuttavia, se vuoi esaminarlo, scegli Closed (Chiuse) in Execution Status (Stato esecuzione), quindi scegli List Executions (Elenca esecuzioni). In questo modo, vengono visualizzate tutte le istanze di flusso di lavoro completate nel dominio specificato (`helloWorldWalkthrough`) che non hanno superato il relativo periodo di retention impostato alla creazione del dominio.



Nota che ogni istanza di flusso di lavoro ha un valore Run ID (ID di esecuzione) univoco. È possibile utilizzare lo stesso ID di workflow per diverse istanze di workflow, ma solo per un'esecuzione attiva alla volta.

HelloWorldWorkflowAsync Applicazione

A volte, è preferibile avere un flusso di lavoro che esegue determinati task localmente anziché utilizzare un'attività. Tuttavia, i task di flusso di lavoro spesso comportano l'elaborazione dei valori rappresentati dagli oggetti `Promise<T>`. Se passi un oggetto `Promise<T>` a un metodo di flusso di lavoro sincrono, il metodo viene eseguito immediatamente ma non può accedere al valore dell'oggetto `Promise<T>` fino a che l'oggetto non è pronto. In realtà, sarebbe possibile eseguire il polling di `Promise<T>.isReady` fino a che non restituisce `true`, ma questa soluzione non è efficace e potrebbe comportare il blocco del metodo per un lungo periodo di tempo. Un miglior approccio consiste nell'utilizzare un metodo asincrono.

Un metodo asincrono viene implementato in modo molto simile a un metodo standard, spesso come membro della classe di implementazione del flusso di lavoro, e viene eseguito nel contesto

dell'implementazione del flusso di lavoro. Per designarlo come metodo asincrono, è necessario applicare un'annotazione `@Asynchronous`, la quale indica al framework di considerarlo come un'attività.

- Quando un'implementazione di flusso di lavoro chiama un metodo asincrono, restituisce immediatamente un risultato. I metodi asincroni in genere restituiscono un oggetto `Promise<T>` che diventa pronto al completamento del metodo.
- Se a un metodo asincrono passi uno o più oggetti `Promise<T>`, ritarda l'esecuzione fino a che tutti gli oggetti di input sono pronti. Un metodo asincrono può quindi accedere ai relativi valori `Promise<T>` di input senza rischiare un'eccezione.

> **Note**
>
> A causa del modo in cui AWS Flow Framework for Java esegue il flusso di lavoro, i metodi asincroni in genere vengono eseguiti più volte, quindi è consigliabile utilizzarli solo per attività rapide con costi generali ridotti. Per eseguire task di lunga durata, come calcoli voluminosi, è consigliabile utilizzare le attività. Per informazioni dettagliate, vedi AWS Flow Framework Concetti di base: esecuzione distribuita.

Questo argomento è una guida dettagliata di HelloWorldWorkflowAsync, una versione modificata sostituisce una delle attività con un metodo HelloWorldWorkflow asincrono. Per implementare l'applicazione, crea una copia di HelloWorld. HelloWorldWorkflow pacchetto nella directory del progetto e chiamalo HelloWorld. HelloWorldWorkflowAsync.

> **Note**
>
> Questo argomento si basa sui concetti e sui file presentati negli argomenti HelloWorld Applicazione e HelloWorldWorkflow Applicazione. Approfondisci il file e concetti presentati in tali argomenti prima di continuare.

Le sezioni seguenti descrivono come modificare il HelloWorldWorkflow codice originale per utilizzare un metodo asincrono.

HelloWorldWorkflowAsync Attività Implementazione

HelloWorldWorkflowAsync implementa la sua interfaccia di lavoro per le attività in`GreeterActivities`, come segue:

```
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);
}
```

Questa interfaccia è simile a quella utilizzata da HelloWorldWorkflow, con le seguenti eccezioni:

- Omette l'attività `getGreeting`; quel task è ora gestito da un metodo asincrono.
- Il numero di versione è impostato su 2.0. Dopo aver registrato un'interfaccia di attività con Amazon SWF, non puoi modificarla a meno che non cambi il numero di versione.

Le restanti implementazioni del metodo di attività sono identiche a. HelloWorldWorkflow Elimina semplicemente `getGreeting` da `GreeterActivitiesImpl`.

HelloWorldWorkflowAsync implementazione del flusso di lavoro

HelloWorldWorkflowAsync definisce l'interfaccia del flusso di lavoro come segue:

```
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();
```

```
}
```

L'interfaccia è identica a HelloWorldWorkflow parte un nuovo numero di versione. Come per le attività, se intendi modificare un flusso di lavoro registrato, devi modificarne la versione.

HelloWorldWorkflowAsync implementa il flusso di lavoro come segue:

```
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 sostituisce l'`getGreeting`attività con un metodo `getGreeting` asincrono, ma il `greet` metodo funziona più o meno allo stesso modo:

1. Esegue l'attività `getName`, la quale restituisce immediatamente un oggetto `Promise<String>` `name`che rappresenta il nome.
2. Chiama il metodo asincrono `getGreeting` e gli passa l'oggetto `name`. `getGreeting` restituisce immediatamente un oggetto `Promise<String>`, ovvero `greeting`, che rappresenta la formula di apertura.
3. Esegue l'attività `say` e le passa l'oggetto `greeting`.
4. Al completamento di `getName`, `name` diventa pronto e `getGreeting` utilizza il relativo valore per costruire la formula di apertura.
5. Al completamento di `getGreeting`, `greeting` diventa pronto e `say` stampa la stringa sulla console.

La differenza è che, anziché chiamare il client di attività per eseguire un'attività `getGreeting`, greet chiama il metodo asincrono `getGreeting`. Il risultato è lo stesso, ma il funzionamento del metodo `getGreeting` è un po' differente dall'attività `getGreeting`.

- Il lavoratore di flusso di lavoro utilizza la semantica delle chiamate di funzione standard per eseguire `getGreeting`. Tuttavia, l'esecuzione asincrona dell'attività è mediata da Amazon SWF.
- `getGreeting` viene eseguito nel processo dell'implementazione di flusso di lavoro.
- `getGreeting` restituisce un oggetto `Promise<String>` anziché un oggetto `String`. Per ottenere il valore String incluso in `Promise`, devi chiamare il relativo metodo `get()`. Tuttavia, poiché l'attività viene eseguita in modo asincrono, il suo valore restituito potrebbe non essere pronto immediatamente; genererà un'eccezione finché non `get()` sarà disponibile il valore restituito dal metodo asincrono.

  Per ulteriori informazioni sul funzionamento di `Promise`, consulta AWS Flow Framework Concetti di base: Data Exchange tra attività e flussi di lavoro.

`getGreeting` crea un valore restituito passando la stringa della formula di apertura al metodo `Promise.asPromise` statico. Questo metodo crea un oggetto `Promise<T>` del tipo appropriato, imposta il valore e ne attiva lo stato pronto.

HelloWorldWorkflowAsync Workflow e Activities Host and Starter

HelloWorldWorkflowAsync implementa `GreeterWorker` come classe host per le implementazioni del flusso di lavoro e delle attività. È identico all' HelloWorldWorkflowimplementazione tranne per il `taskListToPoll` nome, che è impostato su "»`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 il workflow starter in`GreeterMain`; è identico all' HelloWorldWorkflow implementazione.

Per eseguire il flusso di lavoro, esegui `GreeterWorker` e`GreeterMain`, proprio come con. HelloWorldWorkflow

HelloWorldWorkflowDistributed Applicazione

Con HelloWorldWorkflow e HelloWorldWorkflowAsync, Amazon SWF media l'interazione tra il flusso di lavoro e le implementazioni delle attività, ma vengono eseguite localmente come un unico processo. `GreeterMain`è in un processo separato, ma viene comunque eseguito sullo stesso sistema.

Una caratteristica fondamentale di Amazon SWF è il supporto di applicazioni distribuite. Ad esempio, puoi eseguire il workflow worker su un' EC2 istanza Amazon, il workflow starter su un computer del data center e le attività su un computer desktop client. Puoi anche eseguire attività diverse su sistemi diversi.

L' HelloWorldWorkflowDistributed applicazione si estende HelloWorldWorkflowAsync per distribuire l'applicazione su due sistemi e tre processi.

- Il flusso di lavoro e lo starter operano come processi separati su un solo sistema.
- Le attività operano su un sistema separato.

Per implementare l'applicazione, crea una copia di HelloWorld. HelloWorldWorkflowAsync pacchetto nella directory del progetto e chiamalo HelloWorld. HelloWorldWorkflowDistributed. Le sezioni seguenti descrivono come modificare il HelloWorldWorkflowAsync codice originale per distribuire l'applicazione su due sistemi e tre processi.

Non devi modificare il flusso di lavoro o le implementazioni di attività per eseguirli su sistemi separati, e neanche i numeri di versione. Non devi neanche modificare `GreeterMain`. Tutto quello che devi cambiare è l'host delle attività e del flusso di lavoro.

Con HelloWorldWorkflowAsync, una singola applicazione funge da host del flusso di lavoro e delle attività. Per eseguire su sistemi separati il flusso di lavoro e le implementazioni delle attività, devi implementare applicazioni separate. Elimina GreeterWorker dal progetto e aggiungi due nuovi file di classe GreeterWorkflowWorker e GreeterActivitiesWorker.

HelloWorldWorkflowDistributed implementa le sue attività ospitate in GreeterActivitiesWorker, come segue:

```
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 il proprio host di workflow in`GreeterWorkflowWorker`, come segue:

```
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();
   }
}
```

Ricorda che `GreeterActivitiesWorker` è solo `GreeterWorker` senza il codice `WorkflowWorker` e che `GreeterWorkflowWorker` è solo `GreeterWorker` senza il codice `ActivityWorker`.

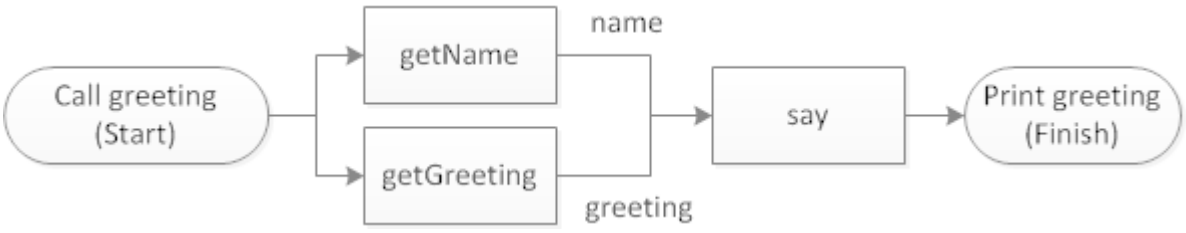
Per eseguire il flusso di lavoro:

1. Crea un file JAR eseguibile con `GreeterActivitiesWorker` come punto di ingresso.
2. Copia il file JAR della Fase 1 su un altro sistema, che abbia qualsiasi sistema operativo che supporta Java.
3. Assicurati che AWS le credenziali con accesso allo stesso dominio Amazon SWF siano disponibili sull'altro sistema.
4. Esegui il file JAR.
5. Nel sistema di sviluppo, utilizza Eclipse per eseguire `GreeterWorkflowWorker` e `GreeterMain`.

Oltre al fatto che le attività vengono eseguite su un sistema diverso da quello di Workflow Worker e Workflow Starter, il flusso di lavoro funziona esattamente nello stesso modo di. HelloWorldAsync Tuttavia, poiché la `println` chiamata stampa «Hello World!» se la console è `say` attiva, l'output verrà visualizzato sul sistema su cui è in esecuzione l'Activities Worker.

HelloWorldWorkflowParallel Applicazione

Le versioni precedenti di Hello World! tutti utilizzare un flusso di lavoro lineare topologia. Tuttavia, Amazon SWF non si limita alle topologie lineari. L' HelloWorldWorkflowParallel applicazione è una versione modificata HelloWorldWorkflow che utilizza una topologia parallela, come illustrato nella figura seguente.



Con HelloWorldWorkflowParallel, `getName` e `getGreeting` corri in parallelo e ognuno restituisce una parte del saluto. `say`quindi unisce le due stringhe in un messaggio di saluto e lo stampa sulla console.

Per implementare l'applicazione, crea una copia di HelloWorld. HelloWorldWorkflow pacchetto nella directory del progetto e chiamalo HelloWorld. HelloWorldWorkflowParallel. Le sezioni seguenti descrivono come modificare il HelloWorldWorkflow codice originale per l'esecuzione `getName` e `getGreeting` in parallelo.

HelloWorldWorkflowParallel Attività: Lavoratore

L'interfaccia HelloWorldWorkflowParallel delle attività è implementata in`GreeterActivities`, come illustrato nell'esempio seguente.

```
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);
}
```

L'interfaccia è simile a HelloWorldWorkflow, con le seguenti eccezioni:

- `getGreeting` non accetta alcun input, ma restituisce semplicemente una stringa di formula di apertura.
- `say` accetta due stringhe di input, la formula di apertura e il nome.
- L'interfaccia ha un nuovo numero di versione, necessario ogni volta che modifichi un'interfaccia registrata.

HelloWorldWorkflowParallel implementa le attività in`GreeterActivitiesImpl`, come segue:

```
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);
   }
}
```

Ora `getName` e `getGreeting` restituiscono semplicemente metà della stringa di formula di apertura. `say` concatena le due parti per generare la frase completa e la stampa sulla console.

HelloWorldWorkflowParallel Workflow Worker

L'interfaccia del HelloWorldWorkflowParallel flusso di lavoro è implementata in`GreeterWorkflow`, come segue:

```
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 classe è identica alla HelloWorldWorkflow versione, tranne per il fatto che il numero di versione è stato modificato per corrispondere all'operatore delle attività.

Il flusso di lavoro è implementato in `GreeterWorkflowImpl`, come mostrato di seguito:

```
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 prima vista, questa implementazione sembra molto simile alle HelloWorldWorkflow tre attività eseguite in sequenza dai metodi client. Tuttavia, ciò non avviene per le attività.

- HelloWorldWorkflow passato `name` a. `getGreeting` Poiché `name` era un oggetto `Promise<T>`, `getGreeting` ha posticipato l'esecuzione dell'attività fino al completamento di `getName` e le due attività sono state eseguite in sequenza.
- HelloWorldWorkflowParallel non trasmette alcun input `getName` o`getGreeting`. Nessuno dei due metodi posticipa l'esecuzione e i metodi di attività associati sono eseguiti immediatamente e in parallelo.

L'attività `say` accetta `greeting` e `name` come parametri di input. Poiché sono oggetti `Promise<T>`, `say` posticipa l'esecuzione fino al completamento di entrambe le attività e quindi costruisce e stampa la formula di apertura.

Si noti che HelloWorldWorkflowParallel non utilizza alcun codice di modellazione speciale per definire la topologia del flusso di lavoro. Lo fa implicitamente utilizzando il controllo di flusso Java standard e sfruttando le proprietà degli oggetti. `Promise<T>` AWS Flow Framework per le applicazioni Java è possibile implementare anche topologie complesse semplicemente utilizzando `Promise<T>` oggetti insieme ai costrutti di flusso di controllo Java convenzionali.

HelloWorldWorkflowParallel Workflow e attività Host and Starter

HelloWorldWorkflowParallel implementa `GreeterWorker` come classe host per le implementazioni del flusso di lavoro e delle attività. È identico all' HelloWorldWorkflow implementazione tranne per il `taskListToPoll` nome, che è impostato su "»HelloWorldParallelList.

`HelloWorldWorkflowParallel`implementa il workflow starter in `GreeterMain` ed è identico all' HelloWorldWorkflow implementazione.

Per eseguire il flusso di lavoro, esegui `GreeterWorker` e `GreeterMain` esattamente come con `HelloWorldWorkflow`.

Comprensione AWS Flow Framework di Java

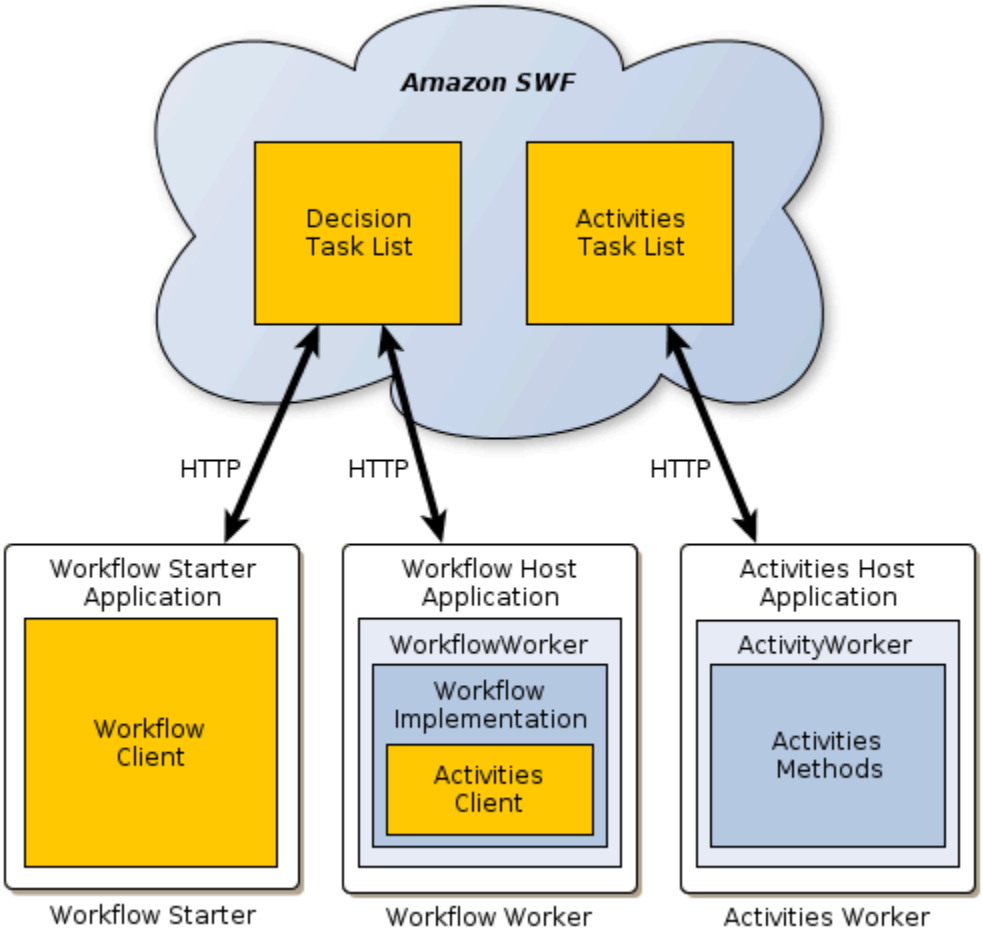
The AWS Flow Framework for Java funziona con Amazon SWF per semplificare la creazione di applicazioni scalabili e con tolleranza ai guasti per eseguire attività asincrone che possono essere di lunga durata, remote o entrambe. Il programma «Hello World!» alcuni esempi Che cos'è AWS Flow Framework per Java? hanno introdotto le nozioni di base su come utilizzarlo per AWS Flow Framework implementare applicazioni di flusso di lavoro di base. Questa sezione fornisce informazioni concettuali sul funzionamento AWS Flow Framework delle applicazioni. La prima sezione riassume la struttura di base di un' AWS Flow Framework applicazione, mentre le sezioni rimanenti forniscono ulteriori dettagli sul funzionamento AWS Flow Framework delle applicazioni.

Argomenti

AWS Flow Framework Concetti di base: struttura dell'applicazione

Concettualmente, un' AWS Flow Framework applicazione è composta da tre componenti di base: chi avvia il flusso di lavoro, gli addetti al flusso di lavoro e gli addetti alle attività. Una o più applicazioni host sono responsabili della registrazione dei lavoratori (flusso di lavoro e attività) con Amazon SWF, dell'avvio dei lavoratori e della gestione della pulizia. I lavoratori gestiscono i meccanismi di esecuzione del flusso di lavoro e possono essere implementati su vari host.

Questo diagramma rappresenta un'applicazione di base: AWS Flow Framework



> **Note**
>
> L'implementazione di questi componenti in tre applicazioni distinte è vantaggiosa da un punto di vista concettuale, ma puoi comunque creare applicazioni per implementare questa funzionalità in vari modi. Ad esempio, puoi utilizzare una singola applicazione host per i lavoratori di flusso di lavoro e di attività oppure host di flusso di lavoro e di attività distinti. Puoi inoltre avere molteplici lavoratori di attività, ognuno dei quali gestisce un set di attività differente su host distinti, ecc.

I tre AWS Flow Framework componenti interagiscono indirettamente inviando richieste HTTP ad Amazon SWF, che gestisce le richieste. Amazon SWF esegue le seguenti operazioni:

- Gestione di uno o più elenchi di task di decisione, i quali determinano l'operazione successiva che deve essere eseguita da un lavoratore di flusso di lavoro.

- Gestione di uno o più elenchi di task di attività, i quali determinano quali task saranno eseguiti da un lavoratore di attività.
- Mantiene una step-by-step cronologia dettagliata dell'esecuzione del flusso di lavoro.

Con AWS Flow Framework, il codice dell'applicazione non deve gestire direttamente molti dei dettagli mostrati nella figura, come l'invio di richieste HTTP ad Amazon SWF. Basta chiamare AWS Flow Framework i metodi e il framework gestisce i dettagli dietro le quinte.

Ruolo del lavoratore di attività

Il lavoratore di attività esegue i vari task che il flusso di lavoro deve realizzare e comprende quanto segue:

- L'implementazione di attività, che include un set di metodi di attività che eseguono task particolari per il flusso di lavoro.
- Un ActivityWorkeroggetto, che utilizza richieste HTTP long poll per eseguire il polling di Amazon SWF per le attività da eseguire. Quando è necessaria un'attività, Amazon SWF risponde alla richiesta inviando le informazioni necessarie per eseguire l'attività. L'ActivityWorkeroggetto chiama quindi il metodo di attività appropriato e restituisce i risultati ad Amazon SWF.

Ruolo del lavoratore di flusso di lavoro

Il lavoratore di flusso di lavoro orchestra l'esecuzione di varie attività e gestisce il flusso di dati e le attività non riuscite. e comprende quanto segue:

- L'implementazione di flusso di lavoro, che include la logica di orchestrazione delle attività, gestisce le attività non riuscite, ecc.
- Un client di attività, che funge da proxy per il lavoratore di attività e consente al lavoratore di flusso di lavoro di pianificare le attività da eseguire in modo asincrono.
- Un WorkflowWorkeroggetto che utilizza richieste HTTP long poll per eseguire il polling di Amazon SWF per attività decisionali. Se nell'elenco delle attività del flusso di lavoro sono presenti attività, Amazon SWF risponde alla richiesta restituendo le informazioni necessarie per eseguire l'attività. Il framework esegue quindi il flusso di lavoro per eseguire l'operazione e restituisce i risultati ad Amazon SWF.

Ruolo dello starter di flusso di lavoro

Lo starter di flusso di lavoro avvia un'istanza di flusso di lavoro, denominata anche esecuzione di flusso di lavoro, e può interagire con un'istanza durante l'esecuzione per passare ulteriori dati al lavoratore di flusso di lavoro o ottenere lo stato corrente del flusso di lavoro.

Lo starter di flusso di lavoro utilizza un client di flusso di lavoro per avviare l'esecuzione di flusso di lavoro, interagisce con il flusso di lavoro come necessario durante l'esecuzione e gestisce la pulizia. Lo starter del flusso di lavoro potrebbe essere un'applicazione eseguita localmente, un'applicazione Web, o anche il. AWS CLI Console di gestione AWS

In che modo Amazon SWF interagisce con la tua applicazione

Amazon SWF media l'interazione tra i componenti del flusso di lavoro e mantiene una cronologia dettagliata del flusso di lavoro. Amazon SWF non avvia la comunicazione con i componenti; attende le richieste HTTP dai componenti e gestisce le richieste come richiesto. Per esempio:

- Se la richiesta proviene da un lavoratore, che analizza le attività disponibili, Amazon SWF risponde direttamente al lavoratore se un'attività è disponibile. Per ulteriori informazioni sul polling, consulta Polling delle attività nella Guida per sviluppatori di Amazon Simple Workflow Service.
- Se la richiesta è una notifica da parte di un operatore di attività che indica il completamento di un'attività, Amazon SWF registra le informazioni nella cronologia di esecuzione e aggiunge un'attività all'elenco delle attività decisionali per informare l'operatore del flusso di lavoro che l'attività è completa, consentendogli di procedere alla fase successiva.
- Se la richiesta di esecuzione di un'attività proviene dall'operatore del flusso di lavoro, Amazon SWF registra le informazioni nella cronologia di esecuzione e aggiunge un'attività all'elenco delle attività per indirizzare un lavoratore di attività a eseguire il metodo di attività appropriato.

Questo approccio consente agli operatori di lavorare su qualsiasi sistema dotato di una connessione Internet, tra cui EC2 istanze Amazon, data center aziendali, computer client e così via. Non è nemmeno necessario che siano eseguiti sullo stesso sistema operativo. Poiché le richieste HTTP provengono dai lavoratori, non sono richieste porte visibili esternamente; i lavoratori possono essere eseguiti protetti da un firewall.

Ulteriori informazioni

Per una discussione più approfondita sul funzionamento di Amazon SWF, consulta la Amazon Simple Workflow Service Developer Guide.

AWS Flow Framework Concetti di base: esecuzione affidabile

Le applicazioni distribuite asincrone devono risolvere problemi di affidabilità a cui non sono soggette le applicazioni convenzionali, tra cui:

- Come assicurare una comunicazione affidabile tra componenti distribuiti asincroni, come i componenti a esecuzione prolungata su sistemi remoti.
- Come impedire la perdita dei risultati in caso di errore o di disconnessione di un componente, in particolare nelle applicazioni a esecuzione prolungata.
- Come gestire i componenti distribuiti con errori.

Le applicazioni possono fare affidamento su Amazon SWF per gestire questi problemi. AWS Flow Framework Esploreremo in che modo Amazon SWF fornisce meccanismi per garantire che i flussi di lavoro funzionino in modo affidabile e prevedibile, anche quando sono di lunga durata e dipendono da attività asincrone eseguite computazionalmente e con l'interazione umana.

Assicurare una comunicazione affidabile

AWS Flow Framework fornisce una comunicazione affidabile tra un operatore del flusso di lavoro e i relativi addetti alle attività utilizzando Amazon SWF per inviare attività a lavoratori con attività distribuite e restituire i risultati al lavoratore del flusso di lavoro. Amazon SWF utilizza i seguenti metodi per garantire una comunicazione affidabile tra un lavoratore e le sue attività:

- Amazon SWF archivia in modo duraturo le attività pianificate e le attività del flusso di lavoro e garantisce che vengano eseguite al massimo una volta.
- Amazon SWF garantisce che un'attività venga completata correttamente e restituisca un risultato valido oppure notificherà all'operatore del flusso di lavoro che l'attività non è riuscita.
- Amazon SWF archivia in modo duraturo il risultato di ogni attività completata o, per le attività non riuscite, memorizza le informazioni di errore pertinenti.

AWS Flow Framework Quindi utilizza i risultati dell'attività di Amazon SWF per determinare come procedere con l'esecuzione del flusso di lavoro.

Impedire la perdita dei risultati

Gestione della cronologia del flusso di lavoro

Un'attività che esegue un'operazione di data mining su un petabyte di dati può durare varie ore e un'attività che richiede a un lavoratore umano di eseguire un task complesso può durare vari giorni o addirittura settimane.

Per adattarsi a scenari come questi, il completamento AWS Flow Framework dei flussi di lavoro e delle attività può richiedere tempi arbitrari: fino al limite di un anno per l'esecuzione di un flusso di lavoro. L'esecuzione affidabile di processi a esecuzione prolungata necessita di un meccanismo per archiviare in modo permanente e continuo la cronologia di esecuzione del flusso di lavoro.

AWS Flow Framework Gestisce questo problema dipendendo da Amazon SWF, che mantiene una cronologia di esecuzione di ogni istanza del flusso di lavoro. La cronologia del flusso di lavoro fornisce un record completo e attendibile dell'avanzamento del flusso di lavoro, inclusi tutti i task di flusso di lavoro e di attività che sono stati pianificati e completati, nonché le informazioni restituite dalle attività completate o non riuscite.

AWS Flow Framework le applicazioni di solito non hanno bisogno di interagire direttamente con la cronologia del flusso di lavoro, sebbene possano accedervi se necessario. Nella maggior parte dei casi, le applicazioni possono semplicemente lasciare che il framework interagisca con la cronologia del flusso di lavoro in background. Per una discussione completa sulla cronologia del flusso di lavoro, consulta Workflow History nella Amazon Simple Workflow Service Developer Guide.

Esecuzione stateless

La cronologia delle esecuzioni consente ai lavoratori di flusso di lavoro di essere stateless. Se disponi di più istanze di un lavoratore di attività o di flusso di lavoro, qualsiasi lavoratore può eseguire qualsiasi task. Il lavoratore riceve tutte le informazioni sullo stato necessarie per eseguire l'attività da Amazon SWF.

Questo approccio rende i flussi di lavoro più affidabili. Ad esempio, se un lavoratore di attività non riesce, non è necessario riavviare il flusso di lavoro. È sufficiente riavviare il lavoratore, il quale eseguirà il polling nell'elenco dei task ed elaborerà tutti i task nell'elenco, indipendentemente dal momento in cui si è verificato l'errore. Puoi rendere l'intero flusso di lavoro a tolleranza di errore utilizzando due o più lavoratori di flusso di lavoro e di attività, eventualmente su sistemi distinti. In questo modo, in caso di errore in uno dei lavoratori, l'altro continuerà a gestire i task pianificati senza alcuna interruzione nell'avanzamento del flusso di lavoro.

Gestire componenti distribuiti con errori

Le attività spesso non hanno esito positivo per motivi effimeri, come una breve disconnessione, di conseguenza una strategia comune per la gestione delle attività non riuscite consiste nel ripetere l'attività. Anziché gestire un nuovo tentativo implementando complesse strategie di passaggio di messaggi, le applicazioni possono utilizzare AWS Flow Framework. Fornisce diversi meccanismi per riprovare le attività non riuscite e fornisce un meccanismo integrato di gestione delle eccezioni che funziona con l'esecuzione asincrona e distribuita delle attività in un flusso di lavoro.

AWS Flow Framework Concetti di base: esecuzione distribuita

Un'istanza di workflow è essenzialmente un thread di esecuzione virtuale che può comprendere le attività e la logica di orchestrazione in esecuzione su più computer remoti. Amazon SWF e la AWS Flow Framework funzione come sistema operativo che gestisce le istanze del flusso di lavoro su una CPU virtuale tramite:

- Mantenendo lo stato di esecuzione di ciascuna istanza.
- Passando da un'istanza all'altra.
- Riprendendo l'esecuzione di un'istanza dal punto in cui è stata interrotta.

Riproduzione dei flussi di lavoro

Dato che le attività possono essere di lunga durata, il blocco del flusso di lavoro fino al completamento non è consigliabile. AWS Flow Framework Gestisce invece l'esecuzione del flusso di lavoro utilizzando un meccanismo di riproduzione, che si basa sulla cronologia del flusso di lavoro gestita da Amazon SWF per eseguire il flusso di lavoro in episodi.

Ciascun episodio riproduce la logica del flusso di lavoro in modo da eseguire ogni attività solo una volta e ritarda l'esecuzione delle attività e dei metodi asincroni fino a quando i loro oggetti Promise non sono pronti.

Lo starter del flusso di lavoro avvia il primo episodio di riproduzione quando inizia l'esecuzione del flusso di lavoro. Il framework chiama il metodo del punto di ingresso del flusso di lavoro e:

1. Esegue tutti i task del flusso di lavoro che non dipendono dal completamento dell'attività, inclusa la chiamata di tutti i metodi client di attività.

2. Fornisce ad Amazon SWF un elenco di attività e attività da pianificare per l'esecuzione. Per il primo episodio, l'elenco consiste solo delle attività che non dipendono da Promise e possono essere eseguite immediatamente.
3. Notifica ad Amazon SWF che l'episodio è completo.

Amazon SWF memorizza le attività nella cronologia del flusso di lavoro e ne pianifica l'esecuzione inserendole nell'elenco delle attività. I lavoratori di attività eseguono il polling dell'elenco ed eseguono i task.

Quando un activity worker completa un'attività, restituisce il risultato ad Amazon SWF, che lo registra nella cronologia di esecuzione del flusso di lavoro e pianifica una nuova attività del flusso di lavoro per l'operatore del flusso di lavoro inserendola nell'elenco delle attività del flusso di lavoro. Il lavoratore esegue il polling dell'elenco e quando riceve il task esegue l'episodio di riproduzione successivo, nel modo seguente:

1. Il framework esegue nuovamente il metodo del punto di ingresso del flusso di lavoro e:
   - Esegue tutti i task del flusso di lavoro che non dipendono dal completamento dell'attività, inclusa la chiamata di tutti i metodi client di attività. Tuttavia, il framework verifica la cronologia delle esecuzioni e non pianifica doppioni dello stesso task di attività.
   - Verifica la cronologia per vedere quali task di attività sono stati completati ed esegue i metodi asincroni del flusso di lavoro che dipendono da quelle attività.
2. Quando tutte le attività del flusso di lavoro che possono essere eseguite sono state completate, il framework riporta ad Amazon SWF:
   - Fornisce ad Amazon SWF un elenco di tutte le attività i cui `Promise<T>` oggetti di input sono pronti dall'ultimo episodio e possono essere pianificati per l'esecuzione.
   - Se l'episodio non ha generato attività aggiuntive ma ci sono ancora attività non completate, il framework notifica ad Amazon SWF che l'episodio è completo. Attende quindi il completamento di un'altra attività, avviando il successivo episodio di riproduzione.
   - Se l'episodio non ha generato attività aggiuntive e tutte le attività sono state completate, il framework notifica ad Amazon SWF che l'esecuzione del flusso di lavoro è completa.

Per esempi di comportamento di riproduzione, consulta AWS Flow Framework per Java Replay Behavior.

Riproduzione e metodi di flusso di lavoro asincroni

I metodi di flusso di lavoro asincroni sono spesso utilizzati come attività, perché il metodo ritarda l'esecuzione fino a che tutti gli oggetti `Promise<T>` di input sono pronti. Tuttavia, il meccanismo di riproduzione gestisce i metodi asincroni in modo diverso rispetto alle attività.

- La riproduzione non garantisce che un metodo asincrono venga eseguito solo una volta. Ritarda l'esecuzione di un metodo asincrono fino a quando i suoi oggetti Promise di input sono pronti, ma poi esegue quel metodo per tutti gli episodi successivi.
- Quando un metodo asincrono viene completato, non avvia un nuovo episodio.

Un esempio di riproduzione di un flusso di lavoro asincrono si trova in AWS Flow Framework per Java Replay Behavior.

Riproduzione e implementazione del flusso di lavoro

Per la maggior parte, non occorre preoccuparsi dei dettagli del meccanismo di riproduzione. In sostanza è qualcosa che accade dietro le quinte. Tuttavia, la riproduzione ha due importanti effetti per l'implementazione di un flusso di lavoro.

- Non utilizzare metodi di flusso di lavoro per eseguire task di lunga durata, perché la riproduzione ripete i task più volte. Anche i metodi asincroni in genere si ripetono più di una volta. Utilizza invece le attività per i task di lunga durata; la riproduzione esegue le attività solo una volta.
- La logica del flusso di lavoro deve essere totalmente deterministica; ogni episodio deve accettare lo stesso percorso del flusso di controllo. Ad esempio, il percorso del flusso di controllo non deve dipendere dall'ora corrente. Per una descrizione dettagliata della riproduzione e dei requisiti deterministici, consulta Non determinismo.

AWS Flow Framework Concetti di base: elenchi di attività ed esecuzione delle attività

Amazon SWF gestisce i flussi di lavoro e le attività pubblicandoli in elenchi denominati. Amazon SWF mantiene almeno due elenchi di attività, uno per i lavoratori del flusso di lavoro e uno per gli addetti alle attività.

> **Note**
>
> Puoi specificare il numero di elenchi di task che preferisci, con lavoratori diversi assegnati a ogni elenco. Non vi è alcun limite al numero di elenchi di task. Solitamente, puoi specificare un elenco di task del lavoratore nell'applicazione host del lavoratore quando crei l'oggetto del lavoratore.

Il seguente estratto dall'applicazione host `HelloWorldWorkflow` crea un nuovo lavoratore di attività e lo assegna all'elenco di task di attività `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();
    ...
  }
}
```

Per impostazione predefinita, Amazon SWF pianifica le attività del lavoratore nell'`HelloWorldList`elenco. In seguito il lavoratore analizza l'elenco alla ricerca di task. Puoi assegnare all'elenco di task il nome che preferisci. Puoi anche utilizzare lo stesso nome per gli elenchi di flusso di lavoro e attività. Internamente, Amazon SWF inserisce i nomi degli elenchi di attività e flussi di lavoro in namespace diversi, quindi i due elenchi saranno distinti.

Se non specifichi un elenco di attività, AWS Flow Framework specifica un elenco predefinito quando il lavoratore registra il tipo con Amazon SWF. Per ulteriori informazioni, consulta Registrazione dei tipi di flusso di lavoro e di attività.

A volte è utile che un lavoratore o un gruppo di lavoratori specifici eseguano determinati task. Ad esempio, un flusso di lavoro di elaborazione delle immagini potrebbe utilizzare un'attività per scaricare un'immagine e un'altra per elaborarla. È più efficiente eseguire entrambi i task sullo stesso sistema ed evitare costi legati al trasferimento di file di grandi dimensioni all'interno della rete.

Per supportare tali scenari, puoi specificare in modo esplicito un elenco di task quando chiami un metodo client di attività utilizzando un overload che include un parametro `schedulingOptions`. È possibile specificare l'elenco delle attività passando al metodo un oggetto configurato in modo appropriato. `ActivitySchedulingOptions`

Ad esempio, supponiamo che l'attività `say` dell'applicazione `HelloWorldWorkflow` sia ospitata da un lavoratore di attività diverso da `getName` e `getGreeting`. Il seguente esempio mostra come garantire che `say` utilizzi lo stesso elenco di task di `getName` e `getGreeting`, anche se originariamente sono stati assegnati a elenchi diversi.

```
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);
  }
}
```

Il metodo asincrono `runSay` ottiene l'elenco di task `getGreeting` dall'oggetto client. Poi crea e configura un oggetto `ActivitySchedulingOptions` che garantisca che `say` analizzi lo stesso elenco di task di `getGreeting`.

> **Note**
>
> Quando passi un parametro `schedulingOptions` a un metodo client di attività, questo sovrascrive l'elenco di task originale soltanto per l'esecuzione di quell'attività. Se richiami nuovamente il metodo activities client senza specificare un elenco di attività, Amazon SWF assegna l'attività all'elenco originale e l'activity worker analizzerà tale elenco.

AWS Flow Framework Concetti di base: applicazioni scalabili

Amazon SWF ha due caratteristiche chiave che semplificano la scalabilità di un'applicazione di workflow per gestire il carico corrente:

- Una cronologia completa delle esecuzioni dei flussi di lavoro, che permette di implementare un'applicazione stateless.
- Una pianificazione dei task con legami deboli alla loro esecuzione, che semplifica la scalabilità dell'applicazione per soddisfare le esigenze attuali.

Amazon SWF pianifica le attività pubblicandole in elenchi di attività allocati dinamicamente, non comunicando direttamente con gli addetti al flusso di lavoro e alle attività. I lavoratori utilizzano invece richieste HTTP per eseguire il polling dei rispettivi elenchi di task. Questo approccio associa vagamente la pianificazione delle attività all'esecuzione delle attività e consente ai lavoratori di funzionare su qualsiasi sistema adatto, tra cui EC2 istanze Amazon, data center aziendali, computer client e così via. Poiché le richieste HTTP provengono dai worker, non sono necessarie porte visibili esternamente, il che consente agli operatori di funzionare anche dietro un firewall.

Il meccanismo long polling utilizzato dai lavoratori per eseguire il polling dei task assicura che i lavoratori non vengano sovraccaricati. Anche se c'è un picco nei task pianificati, i lavoratori estraggono i task secondo le loro esigenze. Tuttavia, poiché il lavoratori sono stateless, puoi scalare dinamicamente un'applicazione per soddisfare un maggiore carico avviando istanze lavoratore aggiuntive. Anche se operano su sistemi diversi, ciascuna istanza esegue il polling dello stesso elenco di task e la prima istanza lavoratore disponibile esegue ciascun task, indipendentemente dalla posizione o dal momento di inizio del lavoratore. Quando il carico diminuisce, si può ridurre di conseguenza il numero di lavoratori.

AWS Flow Framework Concetti di base: Data Exchange tra attività e flussi di lavoro

Quando chiami un metodo client di attività asincrono, restituisce immediatamente un oggetto Promessa (noto anche come Futuro) che rappresenta il valore restituito del metodo di attività. Inizialmente, la Promessa è in uno stato non pronto e il valore restituito è indefinito. Dopo che il metodo di attività ha completato il task e viene restituito, il framework esegue il marshalling del valore restituito nella rete al lavoratore di flusso di lavoro, che assegna un valore alla Promessa e fa entrare l'oggetto in uno stato pronto.

Anche se il metodo di attività non ha un valore restituito, puoi ancora utilizzare la Promessa per gestire l'esecuzione del flusso di lavoro. Se passi una Promessa restituita a un metodo client di attività o a un metodo flusso di lavoro, questa ritarda l'esecuzione fino a quando l'oggetto è pronto.

Se passi una o più promesse a un metodo client di attività, il framework mette in coda il task ma ritarda la pianificazione fino a quando tutti gli oggetti di input sono pronti. Poi estrae i dati da ogni Promessa ed ne esegue il marshalling su internet al lavoratore di attività, che li passa al metodo di attività come tipo standard.

> **Note**
>
> Se devi trasferire grandi quantità di dati tra i lavoratori di flusso di lavoro e attività, l'approccio consigliato è archiviare i dati in una posizione comoda e passare le informazioni di recupero. Ad esempio, puoi archiviare i dati in un bucket Amazon S3 e passare l'URL associato.

La promessa <T> Tipo

Il tipo `Promise<T>` è simile per alcuni aspetti al tipo Java `Future<T>`. Entrambi i tipi rappresentano valori restituiti da metodi asincroni e sono inizialmente non definiti. Puoi accedere al valore di un oggetto chiamando il suo metodo `get`. Al di là di ciò, i due tipi si comportano in modo diverso.

- `Future<T>` è un costrutto di sincronizzazione che permette a un'applicazione di attendere il completamento di un metodo asincrono. Se chiami `get` e l'oggetto non è pronto, si blocca fino a quando l'oggetto è pronto.
- Con `Promise<T>`, la sincronizzazione è gestita dal framework. Se chiami `get` e l'oggetto non è pronto, `get` genera un'eccezione.

Lo scopo primario di `Promise<T>` è gestire il flusso di dati da un'attività a un'altra. Garantisce che un'attività non venga eseguita fino a quando i dati di input sono validi. In molti casi, i lavoratori di flusso di lavoro non devono accedere agli oggetti `Promise<T>` direttamente; passano semplicemente gli oggetti da un'attività a un'altra e lasciano che i lavoratori di framework e attività gestiscano i dettagli. Per accedere al valore dell'oggetto `Promise<T>` in un lavoratore di flusso di lavoro, devi essere certo che l'oggetto sia pronto prima di chiamare il suo metodo `get`.

- L'approccio consigliato è passare l'oggetto `Promise<T>` a un metodo flusso di lavoro asincrono ed elaborare i valori al suo interno. Un metodo asincrono ritarda l'esecuzione fino a quando tutti gli oggetti di input `Promise<T>` sono pronti, il che ti garantisce l'accesso sicuro ai valori.

- `Promise<T>` espone un metodo `isReady` che restituisce `true` se l'oggetto è pronto. Non è consigliato utilizzare `isReady` per analizzare un oggetto `Promise<T>`, ma `isReady` è utile in alcune circostanze.

Il AWS Flow Framework for Java include anche un `Settable<T>` tipo, che è derivato da `Promise<T>` e ha un comportamento simile. La differenza è che il framework di solito imposta il valore di un `Promise<T>` oggetto e l'operatore del flusso di lavoro è responsabile dell'impostazione del valore di a`Settable<T>`.

Ci sono alcune circostanze in cui un lavoratore di flusso di lavoro deve creare un oggetto `Promise<T>` e definire il suo valore. Ad esempio, un metodo asincrono che restituisce un oggetto `Promise<T>` deve creare un valore restituito.

- Per creare un oggetto che rappresenta un valore tipizzato, chiama il metodo statico `Promise.asPromise` che crea un oggetto `Promise<T>` del tipo appropriato, definisce il suo valore e lo fa entrare in uno stato pronto.
- Per creare un oggetto `Promise<Void>`, chiama il metodo statico `Promise.Void`.

> **Note**
>
> `Promise<T>` può rappresentare qualunque tipo valido. Tuttavia, se bisogna eseguire il marshalling dei dati su Internet, il tipo deve essere compatibile con il convertitore di dati. Per ulteriori informazioni, consulta la prossima sezione.

Convertitore e marshalling dei dati

AWS Flow Framework Gestisce i dati su Internet utilizzando un convertitore di dati. Per impostazione predefinita, il framework utilizza un convertitore di dati che è basato sul processore Jackson JSON. Tuttavia, il convertitore ha dei limiti. Ad esempio, non può effettuare il marshalling delle mappe che non utilizzano le stringhe come chiavi. Se il convertitore predefinito non è sufficiente per la tua applicazione, puoi implementare un convertitore di dati personalizzato. Per informazioni dettagliate, consulta DataConverters.

AWS Flow Framework Concetti di base: Data Exchange tra applicazioni ed esecuzioni di flussi di lavoro

Un metodo del punto di ingresso del flusso di lavoro può avere uno o più parametri, che permettono allo starter di trasferire i dati iniziali al flusso di lavoro. Può essere utile anche per fornire al flusso di lavoro dati aggiuntivi durante l'esecuzione. Ad esempio, se un cliente modifica l'indirizzo di spedizione, puoi avvisare il flusso di lavoro di elaborazione dell'ordine affinché apporti le opportune modifiche.

Amazon SWF consente ai flussi di lavoro di implementare un metodo di segnale, che consente ad applicazioni come Workflow Starter di trasferire dati al flusso di lavoro in qualsiasi momento. Un metodo segnale può avere tutti i nomi e parametri opportuni. Lo designi come metodo segnale includendolo nella definizione dell'interfaccia del flusso di lavoro e applicando un'annotazione `@Signal` alla dichiarazione del metodo.

L'esempio seguente mostra l'interfaccia del flusso di lavoro per l'elaborazione di un ordine che dichiara un metodo segnale, `changeOrder`, che permette allo starter di modificare l'ordine originale dopo l'avvio del flusso di lavoro.

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

Il processore di annotazione del framework crea un metodo client del flusso di lavoro con lo stesso nome del metodo segnale e lo starter chiama il metodo client per trasferire i dati al flusso di lavoro. Per un esempio, consulta Recipes AWS Flow Framework

Tipi di timeout di Amazon SWF

Per garantire che le esecuzioni dei flussi di lavoro vengano eseguite correttamente, puoi impostare diversi tipi di timeout con Amazon SWF. Alcuni timeout specificano la durata totale del flusso di lavoro. Altri timeout specificano quanto impiegano le attività prima di essere assegnate a un lavoratore e quanto ci vuole a completarle dal momento in cui sono state pianificate. Tutti i timeout

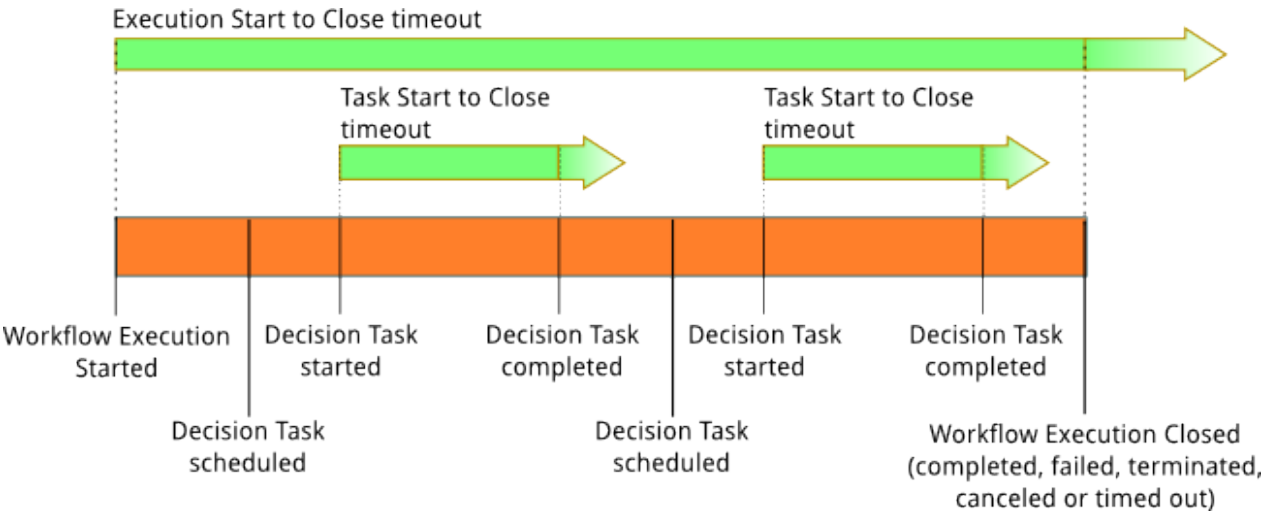
nell'API Amazon SWF sono specificati in secondi. Amazon SWF supporta anche la stringa `NONE` come valore di timeout, che indica l'assenza di timeout.

Per i timeout relativi alle attività decisionali e alle attività, Amazon SWF aggiunge un evento alla cronologia di esecuzione del flusso di lavoro. Gli attributi dell'evento forniscono informazioni sul tipo di timeout verificatosi e su quale attività decisionale o attività è stata influenzata. Amazon SWF pianifica anche un'attività decisionale. Quando il decisore riceve il nuovo compito decisionale, vedrà l'evento di timeout nella cronologia e intraprenderà l'azione appropriata richiamando l'azione. RespondDecisionTaskCompleted

Un task si considera aperto dal momento in cui è pianificato fino alla sua chiusura. Perciò un task è segnalato come aperto quando un lavoratore lo sta elaborando. Un task è chiuso quando un lavoratore lo segnala come completato, annullato o non riuscito. Un'attività può anche essere chiusa da Amazon SWF a seguito di un timeout.

I timeout nel flusso di lavoro e i task di decisione

Il diagramma seguente mostra la correlazione tra i timeout del flusso di lavoro e di decisione e il ciclo di vita di un flusso di lavoro:



Esistono due tipi di timeout che interessano i task del flusso di lavoro e di decisione:

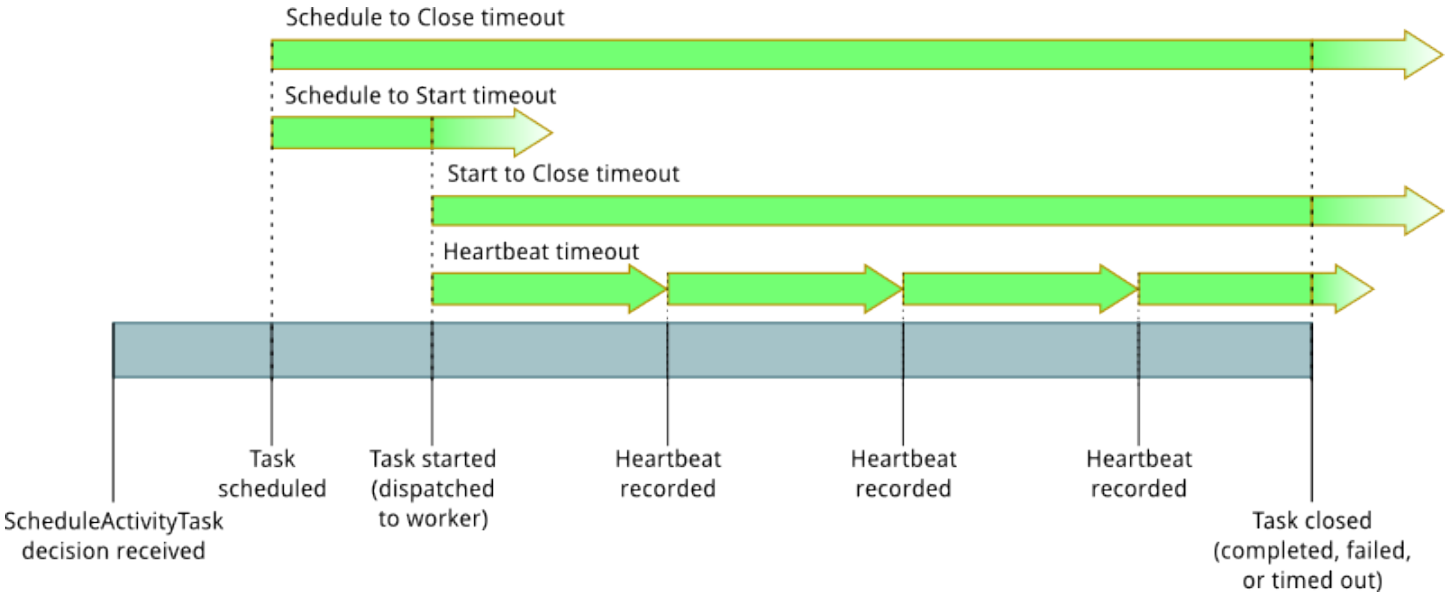
- Workflow Start to Close (**`timeoutType: START_TO_CLOSE`**): questo timeout specifica il tempo massimo necessario per completare l'esecuzione di un flusso di lavoro. È impostato come predefinito durante la registrazione del flusso di lavoro, ma può essere sovrascritto con un valore diverso quando il flusso di lavoro inizia. Se questo timeout viene superato, Amazon SWF chiude l'esecuzione del flusso di lavoro e aggiunge un evento di WorkflowExecutionTimedOuttipo alla

cronologia di esecuzione del flusso di lavoro. Oltre al `timeoutType`, gli attributi dell'evento specificano la `childPolicy` valida per l'esecuzione del flusso di lavoro. La policy figlio specifica in che modo vengono gestite le esecuzioni del flusso di lavoro figlio se quella padre scade o termina in altro modo. Ad esempio, se la `childPolicy` è impostata su TERMINATA, allora le esecuzioni del flusso di lavoro figlio verranno terminate. Una volta scaduta un'esecuzione del flusso di lavoro, non potrai più intervenire se non con chiamate di visibilità.

- Inizio e chiusura dell'attività decisionale (**`timeoutType: START_TO_CLOSE`**): questo timeout specifica il tempo massimo che il decisore corrispondente può impiegare per completare un'attività decisionale. Viene impostato durante la registrazione del tipo di flusso di lavoro. Se questo timeout viene superato, l'attività viene contrassegnata come scaduta nella cronologia di esecuzione del flusso di lavoro e Amazon SWF aggiunge un evento di tipo DecisionTaskTimedOutalla cronologia del flusso di lavoro. Gli attributi dell'evento includeranno gli eventi che corrispondono a quando questo task decisionale è stato pianificato (`scheduledEventId`) e quando è stato avviato (). IDs `startedEventId` Oltre ad aggiungere l'evento, Amazon SWF pianifica anche una nuova attività decisionale per avvisare il decisore che tale attività decisionale è scaduta. Dopo che si verifica questo timeout, il tentativo di completare il task di decisione scaduto utilizzando `RespondDecisionTaskCompleted` non andrà a buon fine.

Timeout nei task di attività

Il diagramma seguente mostra la correlazione tra i timeout e il ciclo di vita di un task di attività:



Esistono quattro tipi di timeout che interessano i task di attività:

- Inizio attività da inizio a chiusura (**`timeoutType: START_TO_CLOSE`**): questo timeout specifica il tempo massimo che un addetto all'attività può impiegare per elaborare un'attività dopo che il lavoratore ha ricevuto l'attività. Tenta di chiudere un'attività scaduta utilizzando RespondActivityTaskCanceledRespondActivityTaskCompleted, e RespondActivityTaskFailedavrà esito negativo.
- Activity Task Heartbeat (**`timeoutType: HEARTBEAT`**): questo timeout specifica il tempo massimo di esecuzione di un'attività prima che il relativo avanzamento nel corso dell'azione possa avvenire. `RecordActivityTaskHeartbeat`
- Activity Task Schedule to Start (**`timeoutType: SCHEDULE_TO_START`**): questo timeout specifica per quanto tempo Amazon SWF attende prima di scadere il timeout dell'attività se non sono disponibili lavoratori per eseguire l'attività. Una volta scaduto, il task non verrà assegnato ad altri lavoratori.
- Activity Task Schedule to Close (**`timeoutType: SCHEDULE_TO_CLOSE`**): questo timeout specifica quanto tempo può impiegare l'attività dal momento in cui è pianificata al momento in cui viene completata. Come procedura ottimale, questo valore non deve essere maggiore della somma del timeout dell'attività e del schedule-to-start timeout dell'attività. start-to-close

> **Note**
>
> Ciascun tipo di timeout ha un valore predefinito, generalmente impostato su `NONE` (infinito). In ogni caso, il tempo massimo per l'esecuzione delle attività è un anno.

In fase di registrazione del tipo di attività si impostano valori predefiniti, ma puoi sovrascriverli con nuovi valori quando pianifichi il task di attività. Quando si verifica uno di questi timeout, Amazon SWF aggiungerà un evento di ActivityTaskTimedOuttipo alla cronologia del flusso di lavoro. L'attributo del valore `timeoutType` di questo evento specifica quale di questi timeout si è verificato. Per ciascuno dei timeout, il valore del `timeoutType` è indicato tra parentesi. Gli attributi dell'evento includeranno anche gli eventi che corrispondono a quando l'attività è stata pianificata (`scheduledEventId`) e quando è stata avviata (). IDs `startedEventId` Oltre ad aggiungere l'evento, Amazon SWF pianifica anche una nuova attività decisionale per avvisare chi decide che si è verificato il timeout.

Comprensione di un task in AWS Flow Framework for Java

Argomenti

Attività

La primitiva sottostante utilizzata da AWS Flow Framework for Java per gestire l'esecuzione del codice asincrono è la classe. `Task` Un oggetto di tipo `Task` rappresenta il lavoro che deve essere eseguito in modo asincrono. Quando chiami un metodo asincrono, il framework crea un `Task` per eseguire il codice in quel metodo e lo inserisce in un elenco per essere eseguito successivamente. Analogamente, quando richiami `Activity`, viene creato un `Task` apposito. Dopo questa operazione la chiamata del metodo ritorna, solitamente restituendo un `Promise<T>` come risultato futuro della chiamata.

La classe `Task` è pubblica e può essere utilizzata direttamente. Ad esempio, possiamo riscrivere l'esempio di Hello World per utilizzare un `Task` anziché un metodo asincrono.

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

Il framework chiama il metodo `doExecute()` quando tutti i `Promise` passati al costruttore di `Task` sono pronti. Per maggiori dettagli sulla `Task` classe, consultate la documentazione. AWS SDK per Java

Il framework include anche una classe chiamata `Functor` che rappresenta un `Task` che è anche un `Promise<T>`. L'oggetto `Functor` è pronto quando `Task` è completato. Nel seguente esempio, un `Functor` viene creato per ottenere il messaggio di saluto:

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

Ordine di esecuzione

I task possono essere eseguiti soltanto quando tutti i parametri `Promise<T>` digitati, passati al metodo o all'attività asincroni corrispondenti, sono pronti. Un `Task` pronto per l'esecuzione viene logicamente spostato su una coda pronta. In altre parole, è pianificato per l'esecuzione. La classe di lavoro esegue l'attività richiamando il codice che hai scritto nel corpo del metodo asincrono o pianificando un'attività in Amazon Simple Workflow Service (AWS) nel caso di un metodo di attività.

Man mano che i task vengono eseguiti e producono risultati, altri task sono pronti e l'esecuzione del programma continua il suo ciclo. Il modo in cui il framework esegue i task è importante per comprendere in che ordine viene eseguito il codice asincrono. Il codice che appare in sequenza all'interno del tuo programma potrebbe non essere eseguito in quell'ordine.

```
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!");
}
```

Il codice nell'elenco sopra visualizzerà i seguenti dati:

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

Questo non è il risultato che ti aspetti, ma può essere facilmente spiegato pensando al modo in cui vengono eseguiti i task per i metodi asincroni:

1. La chiamata a `getUserName` crea `Task`. Chiamiamolo `Task1`. Perché `getUserName` non accetta alcun parametro, `Task1` viene immediatamente messo nella coda pronta.
2. Successivamente, la chiamata a `printHelloName` crea `Task` che deve aspettare il risultato di `getUserName`. Chiamiamolo `Task2`. Poiché il valore richiesto non è ancora pronto, `Task2` viene inserito nella lista di attesa.
3. In seguito viene creato un task per `printHelloWorld` e aggiunto alla coda pronta. Chiamiamolo `Task3`.
4. La `println` dichiarazione stampa quindi «Hello, Amazon!» alla console.
5. A questo punto, `Task1` e `Task3` sono inseriti nella coda pronta e `Task2` nell'elenco di attesa.
6. Il lavoratore esegue `Task1` e il risultato rende `Task2` pronto. `Task2` viene aggiunto alla coda pronta dietro `Task3`.
7. `Task3` e `Task2` vengono poi eseguiti in quell'ordine.

L'esecuzione delle attività segue lo stesso schema. Quando chiami un metodo sul client di attività, ne crea uno `Task` che, al momento dell'esecuzione, pianifica un'attività in Amazon SWF.

Il framework si basa su caratteristiche come la generazione del codice e i proxy dinamici per immettere la logica che converte le chiamate di metodo in richiami di attività e in task asincroni nel tuo programma.

Esecuzione del flusso di lavoro

L'esecuzione dell'implementazione del flusso di lavoro viene gestita dalla classe di lavoratore. Quando chiami un metodo sul client di workflow, questo chiama Amazon SWF per creare un'istanza di workflow. Le attività in Amazon SWF non devono essere confuse con le attività del framework.

Un'attività in Amazon SWF può essere un'attività o un'attività decisionale. L'esecuzione dei task di attività è semplice. La classe activity worker riceve attività da Amazon SWF, richiama il metodo di attività appropriato nell'implementazione e restituisce il risultato ad Amazon SWF.

L'esecuzione dei task di decisione è più complesso. L'addetto al flusso di lavoro riceve attività decisionali da Amazon SWF. Un task di decisione è effettivamente una richiesta per sapere dalla logica di flusso di lavoro quali sono i passaggi successivi. Il primo task di decisione viene generato per un'istanza di flusso di lavoro quando viene iniziata sul client di flusso di lavoro. Dopo aver ricevuto questo task di decisione, il framework inizia a eseguire il codice nel metodo di flusso di lavoro annotato con `@Execute`. Questo metodo esegue la logica di coordinamento che pianifica le attività. Quando lo stato dell'istanza del flusso di lavoro cambia, ad esempio quando un'attività viene completata, vengono pianificate ulteriori attività decisionali. A questo punto, la logica di flusso di lavoro può decidere di eseguire un'azione in base ai risultati dell'attività; ad esempio, potrebbe decidere di pianificare un'altra attività.

Il framework nasconde tutti questi dettagli allo sviluppatore traducendo in modo perfetto i task di decisione nella logica di flusso di lavoro. Dal punto di vista dello sviluppatore, il codice assomiglia a un normale programma. Sotto le copertine, il framework lo associa alle chiamate ad Amazon SWF e alle attività decisionali utilizzando la cronologia gestita da Amazon SWF. Quando giunge un task di decisione, il framework riproduce l'esecuzione del programma inserendo i risultati delle attività completate fino a quel momento. I metodi e le attività asincroni che stavano aspettando i risultati vengono sbloccati e l'esecuzione del programma prosegue.

L'esecuzione del flusso di lavoro di elaborazione di immagini e la relativa cronologia vengono mostrate nella seguente tabella.

Esecuzione del flusso di lavoro di anteprima

Esecuzione del programma di flusso di lavoro	Cronologia gestita da Amazon SWF
Esecuzione iniziale	
1. Invia loop 2. getImageUrls 3. downloadImage 4. createThumbnail (task nella coda di attesa) 5. uploadImage (task nella coda di attesa) 6. <prossima iterazione del loop>	1. Avvio dell'istanza di flusso di lavoro, id="1" 2. downloadImage pianificato

Esecuzione del programma di flusso di lavoro	Cronologia gestita da Amazon SWF
Riproduci di nuovo	
1. Invia loop 2. getImageUrls 3. percorso downloadImage image ="foo" 4. createThumbnail 5. uploadImage (task nella coda di attesa) 6. <prossima iterazione del loop>	1. Avvio dell'istanza di flusso di lavoro, id="1" 2. downloadImage pianificato 3. downloadImage completato, restituisce="foo" 4. createThumbnail pianificato
Riproduci di nuovo	
1. Invia loop 2. getImageUrls 3. percorso downloadImage image ="foo" 4. percorso miniatura createThumbnail="bar" 5. uploadImage 6. <prossima iterazione del loop>	1. Avvio dell'istanza di flusso di lavoro, id="1" 2. downloadImage pianificato 3. downloadImage completato, restituisce="foo" 4. createThumbnail pianificato 5. createThumbnail completato, restituis ce="bar" 6. uploadImage pianificato
Riproduci di nuovo	
1. Invia loop 2. getImageUrls 3. percorso downloadImage image ="foo" 4. percorso miniatura createThumbnail="bar" 5. uploadImage 6. <prossima iterazione del loop>	1. Avvio dell'istanza di flusso di lavoro, id="1" 2. downloadImage pianificato 3. downloadImage completato, restituisce="foo" 4. createThumbnail pianificato 5. createThumbnail completato, restituis ce="bar" 6. uploadImage pianificato 7. uploadImage completato ...

Quando `processImage` viene effettuata una chiamata a, il framework crea una nuova istanza del flusso di lavoro in Amazon SWF. Rappresenta un record duraturo del momento in cui viene iniziata un'istanza di flusso di lavoro. Il programma viene eseguito fino alla chiamata all'`downloadImage`attività, che richiede ad Amazon SWF di pianificare un'attività. Il flusso di lavoro viene eseguito ulteriormente e crea attività per le attività successive, che però non possono essere eseguite fino al completamento dell'`downloadImage`attività; pertanto, questo episodio di replay termina. Amazon SWF invia l'`downloadImage`attività per l'esecuzione e, una volta completata, viene registrato nella cronologia insieme al risultato. Il flusso di lavoro è ora pronto per andare avanti e Amazon SWF genera un'attività decisionale. Il framework riceve il task di decisione e riproduce il flusso di lavoro inserendo il risultato dell'immagine scaricata registrato nella cronologia. Questo sblocca l'attività e `createThumbnail` l'esecuzione del programma prosegue ulteriormente pianificando l'`createThumbnail`attività in Amazon SWF. Lo stesso processo si ripete per `uploadImage`. L'esecuzione del programma prosegue in questo modo fino a quando il flusso di lavoro ha elaborato tutte le immagini e non ci sono più task in sospeso. Poiché nessuno stato di esecuzione viene memorizzato localmente, ogni attività decisionale può essere potenzialmente eseguita su una macchina diversa. Questa operazione ti permette di scrivere programmi che siano tolleranti ai guasti e facilmente scalabili.

Non determinismo

Poiché il framework si basa sulla replay, è importante che il codice di orchestrazione (tutto il codice del flusso di lavoro ad eccezione delle implementazioni delle attività) sia deterministico. Ad esempio, il flusso di controllo del programma non deve dipendere da un numero casuale o dall'ora corrente. Poiché queste cose cambieranno tra le chiamate, il replay potrebbe non seguire lo stesso percorso attraverso la logica di orchestrazione. Ciò potrebbe portare a risultati o errori imprevisti. Il framework offre un `WorkflowClock` che puoi utilizzare per individuare l'ora corrente in modo deterministico. Per ulteriori informazioni, consulta la sezione su Contesto di esecuzione.

> **Note**
>
> Il cablaggio Spring non corretto degli oggetti di implementazione del flusso di lavoro può condurre al non determinismo. I bean di implementazione del flusso di lavoro e i bean da cui dipendono devono essere inclusi nell'ambito del flusso di lavoro (`WorkflowScope`). Ad esempio, cablare un bean di implementazione del flusso di lavoro a un bean che mantiene il proprio stato e si trova nel contesto globale porterà a un comportamento imprevisto. Per ulteriori informazioni, consulta la sezione Integrazione di Spring.

AWS Flow Framework per la guida alla programmazione Java

Questa sezione fornisce dettagli su come utilizzare le funzionalità di AWS Flow Framework for Java per implementare applicazioni di flusso di lavoro.

Argomenti

Implementazione di applicazioni di workflow con AWS Flow Framework

I passaggi tipici necessari per lo sviluppo di un flusso di lavoro con AWS Flow Framework sono:

1. Definizione dei contratti di attività e flusso di lavoro. Analizza i requisiti della tua applicazione, quindi determina le attività e la topologia di flusso di lavoro necessarie. Le attività gestiscono i task di elaborazione richiesti mentre la topologia di flusso di lavoro definisce la logica di business e la struttura di base del flusso di lavoro.

   Ad esempio, è possibile che per un'applicazione di elaborazione di contenuti multimediali sia necessario scaricare ed elaborare un file e quindi caricarlo in un bucket Amazon Simple Storage Service (S3). Questa procedura potrebbe essere suddivisa in quattro task di attività:

   1. Download del file da un server
   2. Elaborazione del file (ad esempio, transcodificandolo in un formato multimediale differente)
   3. Caricamento del file nel bucket S3
   4. Pulizia con eliminazione dei file locali

   Questo flusso di lavoro avrebbe un metodo del punto di ingresso e implementerebbe una topologia lineare semplice che esegue le attività in sequenza, come l'HelloWorldWorkflow Applicazione.
2. Implementazione delle interfacce di attività e flusso di lavoro. I contratti di flusso di lavoro e attività sono definiti dalle interfacce Java, rendendo le relative convenzioni di chiamata previsibili con SWF e fornendo flessibilità nell'implementazione della logica di flusso di lavoro e dei task di attività. Le differenti parti del programma possono agire da consumer dei dati delle altre parti, ma non devono essere necessariamente a conoscenza dei dettagli di implementazione delle altre parti.

   Ad esempio, puoi definire un'interfaccia `FileProcessingWorkflow` e fornire differenti implementazioni di flusso di lavoro per codifica di video, compressione, anteprime e così via. Ognuno di questi flussi di lavoro può avere differenti flussi di controllo nonché chiamare differenti metodi di attività e non è necessario che lo starter di flusso di lavoro ne sia a conoscenza. Grazie alle interfacce, risulta semplice anche testare i flussi di lavoro utilizzando implementazioni fittizie che possono essere sostituite in seguito con codice funzionale.
3. Generazione di client di attività e flusso di lavoro. AWS Flow Framework Elimina la necessità di implementare i dettagli relativi alla gestione dell'esecuzione asincrona, all'invio di richieste HTTP, allo smistamento dei dati e così via. In effetti, lo starter di flusso di lavoro esegue un'istanza di flusso di lavoro chiamando un metodo sul client di flusso di lavoro e l'implementazione di flusso di lavoro esegue le attività chiamando metodi sul client di attività. Il framework gestisce i dettagli di queste interazioni in background.

Se utilizzi Eclipse e hai configurato il tuo progetto, ad esempio, il processore di AWS Flow Framework annotazioni utilizza le definizioni dell'interfaccia per generare automaticamente client di flusso di lavoro e attività che espongono lo stesso set di metodi dell'interfaccia corrispondente. Configurazione di AWS Flow Framework per Java

4. Implementazione delle applicazioni host di attività e flusso di lavoro. Le implementazioni del flusso di lavoro e delle attività devono essere incorporate in applicazioni host che eseguono il polling di Amazon SWF per le attività, gestiscono tutti i dati e utilizzano i metodi di implementazione appropriati. AWS Flow Framework for Java include ActivityWorkerclassi che semplificano WorkflowWorkere semplificano l'implementazione delle applicazioni host.
5. Metti alla prova il tuo flusso di lavoro. AWS Flow Framework per Java offre un' JUnit integrazione che puoi utilizzare per testare i flussi di lavoro in linea e localmente.
6. Distribuzione dei lavoratori. Puoi distribuire i tuoi lavoratori in modo appropriato, ad esempio su istanze EC2 Amazon o sui computer del tuo data center. Una volta implementate e avviate, i lavoratori iniziano a interrogare Amazon SWF per le attività e le gestiscono secondo necessità.
7. Avvio delle esecuzioni. Un'applicazione avvia un'istanza di flusso di lavoro utilizzando il client di flusso di lavoro per chiamare il punto di ingresso del flusso di lavoro. Puoi anche avviare flussi di lavoro utilizzando la console Amazon SWF. Indipendentemente da come avvii un'istanza di flusso di lavoro, puoi utilizzare la console Amazon SWF per monitorare l'istanza del flusso di lavoro in esecuzione ed esaminare la cronologia del flusso di lavoro per le istanze in esecuzione, completate e non riuscite.

AWS SDK per JavaInclude un set AWS Flow Framework di esempi Java che puoi sfogliare ed eseguire seguendo le istruzioni nel file readme.html nella directory principale. È inoltre disponibile una serie di ricette, semplici applicazioni, che mostrano come gestire una serie di problemi di programmazione specifici, disponibili su AWS Flow Framework Recipes.

Contratti di flusso di lavoro e attività

Le interfacce Java sono utilizzate per dichiarare le firme dei flussi di lavoro e delle attività. L'interfaccia forma il contratto tra l'implementazione di flusso di lavoro (o attività) e il client del flusso di lavoro (o attività). Ad esempio, un tipo di flusso di lavoro `MyWorkflow` è definito usando un'interfaccia che è annotata con l'annotazione `@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();
}
```

Il contratto non dispone di impostazioni specifiche per l'implementazione. Questo utilizzo dei contratti neutrali rispetto alle implementazioni permette ai client di essere dissociati dall'implementazione e quindi offre la flessibilità per modificare i dettagli dell'implementazione senza spezzare il client. Viceversa, potresti cambiare il client senza dover modificare il flusso di lavoro o l'attività. Ad esempio, il client può essere modificato per chiamare un'attività in modo asincrono utilizzando le promesse (`Promise<T>`) senza richiedere una modifica all'implementazione di attività. Allo stesso modo, l'implementazione dell'attività può essere modificata in modo da essere completata in modo asincrono, ad esempio da una persona che invia un'e-mail, senza richiedere la modifica dei client dell'attività.

Nell'esempio riportato sopra, l'interfaccia di flusso di lavoro `MyWorkflow` contiene un metodo, `startMyWF`, per avviare una nuova esecuzione. Questo metodo è annotato con l'annotazione `@Execute` e deve avere un tipo restituito `void` o `Promise<>`. In un'interfaccia di flusso di lavoro data, può al massimo essere annotato un metodo con questa annotazione. Questo metodo è il punto di ingresso della logica di flusso di lavoro e il framework chiama questo metodo per eseguire la logica di flusso di lavoro quando viene ricevuto un task di decisione.

L'interfaccia di flusso di lavoro definisce inoltre i segnali che possono essere inviati al flusso di lavoro. Il metodo di segnale viene invocato quando un segnale con un nome corrispondente viene ricevuto dall'esecuzione del flusso di lavoro. Ad esempio, l'interfaccia `MyWorkflow` dichiara un metodo di segnale `signal1`, annotato con l'annotazione `@Signal`.

L'annotazione `@Signal` è richiesta sui metodi di segnale. Il tipo restituito del metodo di segnale deve essere `void`. Un'interfaccia di flusso di lavoro potrebbe avere zero o più metodi di segnali definiti al proprio interno. Potresti dichiarare un'interfaccia di flusso di lavoro senza un metodo `@Execute`

e alcuni metodi `@Signal` per generare client che non possono avviare la propria esecuzione ma inviare segnali per effettuare le esecuzioni.

Metodi annotati con le annotazioni `@Execute` e `@Signal` possono avere numeri di parametri di ogni tipo eccetto `Promise<T>` o le sue derivate. Questa funzionalità ti permette di passare input fortemente tipizzati a un'esecuzione di flusso di lavoro dall'avvio e durante la sua esecuzione. Il tipo restituito del metodo `@Execute` deve essere `void` o `Promise<>`.

Inoltre, puoi anche dichiarare un metodo nell'interfaccia di flusso di lavoro per segnalare l'ultimo stato dell'esecuzione del flusso di lavoro, ad esempio il metodo `getState` nel precedente esempio. Questo stato non è l'intero stato di applicazione del flusso di lavoro. Lo scopo di questa funzionalità è permetterti di archiviare fino a 32 KB di data per indicare l'ultimo stato dell'esecuzione. Ad esempio, in un flusso di lavoro di elaborazione dell'ordine, potresti archiviare una stringa che indica che l'ordine è stato ricevuto, elaborato o annullato. Questo metodo viene chiamato dal framework ogni volta che un task di decisione viene completato per ottenere l'ultimo stato. Lo stato è memorizzato in Amazon Simple Workflow Service (Amazon SWF) e può essere recuperato utilizzando il client esterno generato. Questo ti permette di verificare l'ultimo stato di esecuzione del flusso di lavoro. I metodi annotati con `@GetState` non devono acquisire argomenti e non devono avere un tipo restituito `void`. Da questo metodo puoi restituire qualunque tipo che si adatti alle tue esigenze. Nell'esempio citato prima, un oggetto di `MyWorkflowState` (vedi definizione riportata sotto) viene restituito dal metodo utilizzato per archiviare uno stato della stringa e una percentuale numerica completati. Il metodo dovrebbe eseguire l'accesso di sola lettura dell'oggetto dell'implementazione del flusso di lavoro e viene richiamato in modo sincronico, il che non permette l'utilizzo di operazioni asincrone come i metodi di chiamata con `@Asynchronous`. Può essere annotato al massimo un metodo in un'interfaccia di flusso di lavoro con l'annotazione `@GetState`.

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

Analogamente, un set di iniziative viene definito usando un'interfaccia che è annotata con l'annotazione `@Activities`. Ogni metodo nell'interfaccia corrisponde a un'attività, ad esempio:

```
@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);
}
```

L'interfaccia ti permette di raggruppare un set di iniziative relazionate. Puoi definire qualunque numero di attività all'interno dell'interfaccia delle attività e puoi definire il numero di interfacce delle attività che desideri. Analogamente ai metodi `@Execute` e `@Signal`, i metodi di attività possono acquisire qualunque numero di argomenti di qualunque tipo tranne `Promise<T>` o le sue derivate. Il tipo restituito di un'attività non deve essere `Promise<T>` o le sue derivate.

Registrazione dei tipi di flusso di lavoro e di attività

Amazon SWF richiede la registrazione dei tipi di attività e flussi di lavoro prima di poterli utilizzare. Il framework registra automaticamente i flussi di lavoro e le attività nelle implementazioni che aggiungi al lavoratore. Il framework cerca i tipi che implementano flussi di lavoro e attività e li registra con Amazon SWF. Per impostazione predefinita, il framework utilizza le definizioni di interfaccia per dedurre le opzioni di registrazione per i tipi di flusso di lavoro e attività. Tutte le interfacce di flusso di lavoro devono avere l'annotazione `@WorkflowRegistrationOptions` o `@SkipRegistration`. Il lavoratore di flusso di lavoro registra tutti i tipi di flusso di lavoro con cui è configurato che hanno l'annotazione `@WorkflowRegistrationOptions`. Inoltre, ogni metodo di attività deve avere l'annotazione `@ActivityRegistrationOptions` o `@SkipRegistration` oppure una di queste annotazioni deve essere presente nell'interfaccia `@Activities`. Il lavoratore di attività registra tutti i tipi di attività con cui è configurato e a cui si applica un'annotazione `@ActivityRegistrationOptions`. La registrazione è eseguita automaticamente all'avvio di uno dei lavoratori. I tipi di flusso di lavoro e attività che hanno l'annotazione `@SkipRegistration` non sono registrati. Le annotazioni `@ActivityRegistrationOptions` e `@SkipRegistration` hanno la semantica di override e la più specifica viene applicata a un tipo di attività.

Tieni presente che Amazon SWF non consente di registrare nuovamente o modificare il tipo una volta registrato. Il framework tenterà di registrare tutti i tipi, ma se il tipo è già registrato, la registrazione non verrà ripetuta e non verrà segnalato alcun errore.

Se hai la necessità di modificare le impostazioni registrate, devi registrate una nuova versione del tipo. Puoi anche eseguire l'override delle impostazioni registrate all'avvio di una nuova esecuzione o quando chiami un'attività che utilizza i client generati.

La registrazione richiede un nome di tipo e altre opzioni di registrazione. L'implementazione di default le determina come descritto di seguito.

Nome e versione del tipo di flusso di lavoro

Il framework determina il nome del tipo di flusso di lavoro a partire dall'interfaccia di flusso di lavoro. La forma del nome del tipo di flusso di lavoro predefinito è {*prefix*} {*name*}. Il {*prefix*} è impostato sul nome dell'`@Workflow`interfaccia seguito da un '.' e il {*name*} è impostato sul nome del `@Execute` metodo. Il nome di default del tipo di flusso di lavoro nell'esempio precedente è `MyWorkflow.startMyWF`. Puoi eseguire l'override del nome di default utilizzando il parametro name del metodo `@Execute`. Il nome di default del tipo di flusso di lavoro nell'esempio è `startMyWF`. Il nome non deve essere una stringa vuota. Nota che quando esegui l'override del nome utilizzando `@Execute`, il framework non aggiunge automaticamente un prefisso davanti al nome. Sei libero di usare il tuo schema di denominazione.

La versione del flusso di lavoro viene specificata utilizzando il parametro `version` dell'annotazione `@Execute`. Non esiste un valore di default per `version` e quindi deve essere specificato in modo esplicito; `version` è una stringa in formato libero e sei libero di utilizzare uno schema di controllo delle versioni personalizzato.

Nome del segnale

Il nome del segnale può essere specificato utilizzando il parametro name dell'annotazione `@Signal`. Se non è specificato, per impostazione predefinita viene utilizzato il nome del metodo del segnale.

Nome e versione del tipo di attività

Il framework determina il nome del tipo di attività a partire dall'interfaccia di attività. La forma del nome del tipo di attività predefinito è {*prefix*} {*name*}. Il {*prefix*} è impostato sul nome dell'`@Activities`interfaccia seguito da un '.' e il {*name*} è impostato sul nome del metodo. Il valore predefinito {*prefix*} può essere sovrascritto nell'`@Activities`annotazione sull'interfaccia delle attività. Puoi anche specificare il nome del tipo di attività utilizzando l'annotazione `@Activity` nel metodo di attività. Nota che quando esegui l'ovveride del nome utilizzando `@Activity`, il framework non aggiungerà automaticamente un prefisso davanti al nome. Sei libero di utilizzare il tuo schema di denominazione.

La versione dell'attività viene specificata utilizzando il parametro version dell'annotazione `@Activities`. Questa versione è utilizzata come valore di default per tutte le attività definite nell'interfaccia ed è possibile eseguire l'override per una singola attività utilizzando l'annotazione `@Activity`.

Elenco di task predefinito

L'elenco di task di default può essere configurato utilizzando le annotazioni `@WorkflowRegistrationOptions` e `@ActivityRegistrationOptions` e impostando il parametro `defaultTaskList`. Per impostazione predefinita, è impostato su `USE_WORKER_TASK_LIST`. Questo è un valore speciale che indica al framework di utilizzare l'elenco di task configurato sull'oggetto lavoratore utilizzato per registrare il tipo di attività o di flusso di lavoro. Puoi anche scegliere di non registrare un elenco di task di default impostando `NO_DEFAULT_TASK_LIST`, nel caso tu voglia che l'elenco di task sia specificato al runtime. Se non è stato registrato alcun elenco di task di default, devi specificarlo all'avvio del flusso di lavoro o quando chiami il metodo di attività utilizzando i parametri `StartWorkflowOptions` e `ActivitySchedulingOptions` sull'overload del client generato per i rispettivi metodi.

Altre opzioni di registrazione

Tutte le opzioni di registrazione del flusso di lavoro e del tipo di attività consentite dall'API Amazon SWF possono essere specificate tramite il framework.

Per un elenco completo delle opzioni di registrazione di flusso di lavoro, consulta quanto segue:

- @Flusso di lavoro
- @Execute
- @WorkflowRegistrationOptions
- @Signal

Per un elenco completo delle opzioni di registrazione di attività, consulta quanto segue:

- @Activity
- @Activities
- @ActivityRegistrationOptions

Se desideri avere un controllo completo sulla registrazione dei tipi, consulta Estensibilità dei lavoratori.

Client di attività e flusso di lavoro

Client di flusso di lavoro e attività generati dal framework in base alle interfacce `@Workflow` e `@Activities`. Vengono generate interfacce del client separate che contengono metodi e impostazioni che si riferiscono solo al client. Se stai sviluppando utilizzando Eclipse, questa operazione viene eseguita dal plug-in Amazon SWF Eclipse ogni volta che salvi il file contenente l'interfaccia appropriata. Il codice generato viene posizionato nella directory di origine generata nel progetto all'interno dello stesso pacchetto dell'interfaccia.

> **Note**
>
> Il nome predefinito della directory utilizzato da Eclipse è .apt_generated. Eclipse non mostra le directory nome che inizia con un '.' in Package Explorer. Utilizza un nome diverso della directory se desideri visualizzare i file generati all'interno di Project Explorer. In Eclipse, fai clic sul tasto destro su Package Explorer e poi scegli Properties (Proprietà), Java Compiler (Compilatore Java), Annotation processing (Elaborazione delle annotazioni) e modifica le impostazioni Generate source directory (Genera directory di origine).

Client di flusso di lavoro

Gli artefatti generati per il flusso di lavoro contengono tre interfacce lato client e le classi che le implementano. I client generati includono:

- Un client asincrono che deve essere utilizzato dall'interno dell'implementazione del flusso di lavoro che offre metodi asincroni per avviare le esecuzioni del flusso di lavoro e inviare segnali
- Un client esterno che può essere utilizzato per avviare le esecuzioni, inviare segnali e recuperare lo stato del flusso di lavoro dall'esterno dell'ambito dell'implementazione del flusso di lavoro
- Un client autogenerato che può essere utilizzato per creare flussi di lavoro continui

Ad esempio, le interfacce del client generato per l'interfaccia di esempio `MyWorkflow` sono:

```
//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);
```

Le interfacce hanno effettuato l'overloading dei metodi che corrispondono a ciascun metodo nell'interfaccia `@Workflow` che hai dichiarato.

Il client esterno riflette i metodi sull'interfaccia `@Workflow` con un overload aggiuntivo del metodo `@Execute` che accetta `StartWorkflowOptions`. Puoi usare l'overload per passare opzioni aggiuntive quando avvii una nuova esecuzione del flusso di lavoro. Queste opzioni ti permettono di sovrascrivere l'elenco di task predefinito, le impostazioni di timeout e i tag associati all'esecuzione del flusso di lavoro.

Invece, il client asincrono dispone di metodi che ti consentono l'invocazione asincrona del metodo `@Execute`. I seguenti overload del metodo vengono generati nell'interfaccia del client per il metodo `@Execute` nell'interfaccia del flusso di lavoro:

1. Un overload che accetta gli argomenti originali "così come sono". Il tipo restituito dell'overload sarà `Promise<Void>` se il metodo originale ha restituito `void`; altrimenti sarà `Promise<>` come dichiarato nel metodo originale. Per esempio:

   Metodo originale:

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

   Metodo generato:

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

   Questo overload deve essere usato quando tutti gli argomenti del flusso di lavoro sono disponibili e non devono essere attesi.

2. Un overload che accetta gli argomenti originali "così come sono" e argomenti variabili aggiuntivi del tipo `Promise<?>`. Il tipo restituito dell'overload sarà `Promise<Void>` se il metodo originale ha restituito `void`; altrimenti sarà `Promise<>` come dichiarato nel metodo originale. Per esempio:

   Metodo originale:

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

   Metodo generato:

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

   Questo overload deve essere usato quando tutti gli argomenti del flusso di lavoro sono disponibili e non devono essere attesi, ma desideri attendere che altre promesse siano pronte. L'argomento variabile può essere usato per passare gli oggetti `Promise<?>` che non sono stati dichiarati come argomenti, ma desideri attendere prima di eseguire la chiamata.

3. Un overload che accetta gli argomenti originali "così come sono", un argomento aggiuntivo del tipo `StartWorkflowOptions` e argomenti variabili aggiuntivi del tipo `Promise<?>`. Il tipo restituito

dell'overload sarà `Promise<Void>` se il metodo originale ha restituito `void`; altrimenti sarà `Promise<>` come dichiarato nel metodo originale. Per esempio:

Metodo originale:

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

Metodo generato:

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

Questo overload deve essere usato quando tutti gli argomenti del flusso di lavoro sono disponibili e non devono essere attesi, quando desideri sovrascrivere le impostazioni predefinite usate per avviare l'esecuzione del flusso di lavoro o quando desideri attendere che altre promesse siano pronte. L'argomento variabile può essere usato per passare gli oggetti `Promise<?>` che non sono stati dichiarati come argomenti, ma desideri attendere prima di eseguire la chiamata.

4. Un overload in cui ogni argomento nel metodo originale viene sostituito con un wrapper `Promise<>`. Il tipo restituito dell'overload sarà `Promise<Void>` se il metodo originale ha restituito `void`; altrimenti sarà `Promise<>` come dichiarato nel metodo originale. Per esempio:

   Metodo originale:

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

   Metodo generato:

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

   Questo overload deve essere usato quando gli argomenti da passare all'esecuzione del flusso di lavoro devono essere valutati in modo asincrono. Non verrà eseguita una chiamata all'overload del metodo fino a quando tutti gli argomenti passati non diventano pronti.

Se alcuni degli argomenti sono già pronti, allora puoi convertirli in un oggetto `Promise` che è già pronto attraverso il metodo `Promise.asPromise(`*`value`*`)`. Per esempio:

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

5. Un overload in cui ogni argomento nel metodo originale viene sostituito con un wrapper `Promise<>`. L'overload dispone anche di argomenti variabili aggiuntivi del tipo `Promise<?>`. Il tipo restituito dell'overload sarà `Promise<Void>` se il metodo originale ha restituito `void`; altrimenti sarà `Promise<>` come dichiarato nel metodo originale. Per esempio:

   Metodo originale:

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

   Metodo generato:

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

   Questo overload deve essere usato quando gli argomenti da passare all'esecuzione del flusso di lavoro devono essere valutati in modo asincrono e desideri attendere che altre promesse siano pronte. Non verrà eseguita una chiamata all'overload del metodo fino a quando tutti gli argomenti passati non diventano pronti.

6. Un overload in cui ogni argomento nel metodo originale viene sostituito con un wrapper `Promise<?>`. L'overload dispone anche di un argomento aggiuntivo del tipo `StartWorkflowOptions` e di argomenti variabili del tipo `Promise<?>`. Il tipo restituito dell'overload sarà `Promise<Void>` se il metodo originale ha restituito `void`; altrimenti sarà `Promise<>` come dichiarato nel metodo originale. Per esempio:

   Metodo originale:

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

   Metodo generato:

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

Utilizza questo overload quando gli argomenti da passare all'esecuzione del flusso di lavoro verranno valutati in modo asincrono e desideri sovrascrivere le impostazioni predefinite utilizzate per avviare l'esecuzione del flusso di lavoro. Non verrà eseguita una chiamata all'overload del metodo fino a quando tutti gli argomenti passati non diventano pronti.

Viene inoltre generato un metodo corrispondente a ciascun segnale nell'interfaccia del flusso di lavoro, ad esempio:

Metodo originale:

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

Metodo generato:

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

Il client asincrono non contiene un metodo che corrisponde al metodo annotato con `@GetState` nell'interfaccia originale. Poiché il recupero dello stato richiede una chiamata al servizio Web, non è adatto all'uso all'interno di un flusso di lavoro. Quindi, viene fornito soltanto attraverso il client esterno.

Il client autogenerato deve essere utilizzato dall'interno di un flusso di lavoro per avviare una nuova esecuzione dopo il completamento di quella attuale. I metodi su questo client sono simili a quelli sul client asincrono, ma restituiscono `void`. Questo client non dispone di metodi che corrispondono ai metodi annotati con `@Signal` e `@GetState`. Per ulteriori dettagli, consulta Flussi di lavoro continui.

I client generati derivano da interfacce di base: `WorkflowClient` e `WorkflowClientExternal`, rispettivamente, che forniscono metodi che puoi utilizzare per annullare o terminare l'esecuzione del flusso di lavoro. Per ulteriori dettagli su queste interfacce, consulta la documentazione AWS SDK per Java .

I client generati ti permettono di interagire con le esecuzioni del flusso di lavoro in modo fortemente tipizzato. Una volta creata, un'istanza di un client generato è legata a un'esecuzione del flusso di lavoro specifica e può essere utilizzata soltanto per quell'esecuzione. Inoltre, il framework fornisce client dinamici che non sono specifici per un tipo o un'esecuzione del flusso di lavoro. I client generati si basano su questo client. Puoi anche usare direttamente questi client. Consulta la sezione su Client dinamici.

Il framework genera inoltre factory per creare client fortemente tipizzati. Le factoy del client generato per l'interfaccia di esempio `MyWorkflow` sono:

```
//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);
}
```

L'interfaccia di base `WorkflowClientFactory` è:

```
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);
}
```

Devi utilizzare queste factory per creare istanze del client. La factory ti permette di configurare il client generico (il client generico deve essere utilizzato per eseguire l'implementazione personalizzata del client) e il `DataConverter` utilizzato dal client per effettuare il marshalling dei dati, oltre alle opzioni utilizzate per avviare l'esecuzione del flusso di lavoro. Per ulteriori dettagli, consulta le sezioni DataConverters e Esecuzioni del flusso di lavoro figlio. `StartWorkflowOptions`Contiene impostazioni che è possibile utilizzare per ignorare le impostazioni predefinite, ad esempio i timeout, specificate al momento della registrazione. Per maggiori dettagli sulla classe, consultate la documentazione. `StartWorkflowOptions` AWS SDK per Java

Il client esterno può essere utilizzato per avviare le esecuzioni del flusso di lavoro dall'esterno dell'ambito di un flusso di lavoro mentre il client asincrono può essere utilizzato per avviare un'esecuzione del flusso di lavoro dal codice all'interno di un flusso di lavoro. Per avviare un'esecuzione, devi semplicemente usare il client generato per chiamare il metodo che corrisponde al metodo annotato con `@Execute` nell'interfaccia del flusso di lavoro.

Il framework genera inoltre classi di implementazioni per le interfacce del client. Questi client creano e inviano richieste ad Amazon SWF per eseguire l'azione appropriata. La versione client del `@Execute` metodo avvia un'esecuzione di un nuovo flusso di lavoro o crea un'esecuzione del flusso di lavoro secondario utilizzando Amazon SWF APIs. Analogamente, la versione client del `@Signal` metodo utilizza Amazon SWF APIs per inviare un segnale.

> **Note**
>
> Il client di workflow esterno deve essere configurato con il client e il dominio Amazon SWF. Puoi utilizzare il costruttore client factory che li accetta come parametri o passare un'implementazione client generica già configurata con il client e il dominio Amazon SWF.

Il framework percorre la gerarchia del tipo dell'interfaccia del flusso di lavoro e inoltre genera interfacce del client per le interfacce del flusso di lavoro padre e deriva da esse.

Client di attività

Analogamente al client del flusso di lavoro, viene generato un client per ogni interfaccia annotata con `@Activities`. Gli artefatti generati includono un'interfaccia lato client e una classe client. L'interfaccia generata per l'interfaccia di esempio `@Activities` sopra indicata (`MyActivities`) è la seguente:

```
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);
}
```

L'interfaccia contiene un set di metodi su cui è stato effettuato l'overloading che corrispondono a ciascun metodo di attività nell'interfaccia `@Activities`. Tali overload sono forniti per comodità e permettono di chiamare le attività in modo asincrono. I seguenti overload del metodo vengono generati nell'interfaccia del client per ogni metodo di attività nell'interfaccia `@Activities`:

1. Un overload che accetta gli argomenti originali "così come sono". Il tipo restituito per questo overload è `Promise<`*T*`>`, dove *T* è il tipo restituito del metodo originale. Per esempio:

   Metodo originale:

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

Metodo generato:

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

Questo overload deve essere usato quando tutti gli argomenti del flusso di lavoro sono disponibili e non devono essere attesi.

2. Un overload che accetta gli argomenti originali "così come sono", un argomento del tipo `ActivitySchedulingOptions` e argomenti variabili aggiuntivi del tipo `Promise<?>`. Il tipo restituito per questo overload è `Promise<`*T*`>`, dove *T* è il tipo restituito del metodo originale. Per esempio:

   Metodo originale:

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

   Metodo generato:

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

   Questo overload deve essere usato quando tutti gli argomenti del flusso di lavoro sono disponibili e non devono essere attesi, quando desideri sovrascrivere le impostazioni predefinite o quando desideri attendere che un'altra `Promise` sia pronta. Gli argomenti variabili possono essere usati per passare gli oggetti `Promise<?>` aggiuntivi che non sono stati dichiarati come argomenti, ma desideri attendere prima di eseguire la chiamata.

3. Un overload in cui ogni argomento nel metodo originale viene sostituito con un wrapper `Promise<>`. Il tipo restituito per questo overload è `Promise<`*T*`>`, dove *T* è il tipo restituito del metodo originale. Per esempio:

   Metodo originale:

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

Metodo generato:

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

Questo overload deve essere usato quando gli argomenti da passare all'attività verranno valutati in modo asincrono. Non verrà eseguita una chiamata all'overload del metodo fino a quando tutti gli argomenti passati non diventano pronti.

4. Un overload in cui ogni argomento nel metodo originale viene sostituito con un wrapper `Promise<>`. L'overload dispone anche di un argomento aggiuntivo del tipo `ActivitySchedulingOptions` e di argomenti variabili del tipo `Promise<?>`. Il tipo restituito per questo overload è `Promise<`*T*`>`, dove *T* è il tipo restituito del metodo originale. Per esempio:

   Metodo originale:

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

   Metodo generato:

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

   Questo overload deve essere usato quando tutti gli argomenti da passare all'attività verranno valutati in modo asincrono, quando desideri sovrascrivere le impostazioni predefinite registrate con il tipo o quando desideri attendere che un altro oggetto `Promise` sia pronto. Non verrà eseguita una chiamata all'overload del metodo fino a quando tutti gli argomenti passati non diventano pronti. La classe del client generata implementa questa interfaccia. L'implementazione di ogni metodo di interfaccia crea e invia una richiesta ad Amazon SWF per pianificare un'attività del tipo appropriato utilizzando Amazon SWF. APIs

5. Un overload che accetta gli argomenti originali "così come sono" e argomenti variabili aggiuntivi del tipo `Promise<?>`. Il tipo restituito per questo overload è `Promise<`*T*`>`, dove *T* è il tipo restituito del metodo originale. Per esempio:

   Metodo originale:

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

Metodo generato:

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

Questo overload deve essere usato quando tutti gli argomenti dell'attività sono disponibili e non devono essere attesi, ma desideri attendere che altri oggetti `Promise` siano pronti.

6. Un overload in cui ogni argomento del metodo originale viene sostituito con un wrapper `Promise` e argomenti variabili aggiuntivi del tipo `Promise<?>`. Il tipo restituito per questo overload è `Promise<`*T*`>`, dove *T* è il tipo restituito del metodo originale. Per esempio:

   Metodo originale:

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

   Metodo generato:

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

   Questo overload deve essere usato quando tutti gli argomenti dell'attività verranno attesi in modo asincrono e desideri attendere che altre `Promise` siano pronte. Verrà eseguita una chiamata asincrona all'overload del metodo fino a quando tutti gli oggetti `Promise` passati non diventano pronti.

Il client di attività generato dispone inoltre di un metodo protetto che corrisponde a ogni metodo di attività, nominato `{`*`activity method name`*`}Impl()`, a cui tutti gli overload dell'attività eseguono una chiamata. Puoi sovrascrivere questo metodo per creare implementazioni del client fittizie. Questo metodo accetta come argomenti tutti gli argomenti per il metodo originale nei wrapper `Promise<>`, `ActivitySchedulingOptions` e argomenti variabili del tipo `Promise<?>`. Per esempio:

Metodo originale:

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

Metodo generato:

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

Opzioni di programmazione

Il client di attività generato ti permette di passare in `ActivitySchedulingOptions` come argomento. La `ActivitySchedulingOptions` struttura contiene impostazioni che determinano la configurazione dell'attività che il framework pianifica in Amazon SWF. Queste impostazioni sovrascrivono quelle predefinite specificate come opzioni di registrazione. Per specificare le opzioni di pianificazione in modo dinamico, crea un oggetto `ActivitySchedulingOptions`, configuralo come preferisci e passalo al metodo di attività. Nell'esempio seguente abbiamo specificato l'elenco di task che deve essere utilizzato per il task di attività. Questa operazione sovrascrive l'elenco di task registrato predefinito per l'invocazione dell'attività.

```
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);
    }
```

```
}
```

Client dinamici

Oltre ai client generati, il framework fornisce anche client generici `DynamicActivityClient` che puoi utilizzare per avviare dinamicamente esecuzioni di flussi di lavoro, inviare segnali, pianificare attività, ecc. `DynamicWorkflowClient` Ad esempio, potresti voler pianificare un'attività il cui tipo non è noto in fase di progettazione. Puoi utilizzare `DynamicActivityClient` per pianificare questo tipo di task di attività. Analogamente, puoi pianificare in modo dinamico un'esecuzione del flusso di lavoro figlio utilizzando `DynamicWorkflowClient`. Nel seguente esempio, il flusso di lavoro cerca l'attività da un database e utilizza il client dell'attività dinamico per pianificarla:

```
//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);
}
```

Per ulteriori dettagli, consulta la documentazione. AWS SDK per Java

Segnalare e annullare le esecuzioni del flusso di lavoro

Il client del flusso di lavoro generato dispone di metodi corrispondenti a ogni segnale che possono essere inviati al flusso di lavoro. Puoi usarli dall'interno di un flusso di lavoro per inviare segnali ad

altre esecuzioni del flusso di lavoro. Questa operazione fornisce un meccanismo tipizzato per l'invio dei segnali. Tuttavia, a volte può essere necessario determinare dinamicamente il nome del segnale, ad esempio quando il nome del segnale viene ricevuto in un messaggio. Puoi usare il client del flusso di lavoro dinamico per inviare segnali in modo dinamico a qualunque esecuzione del flusso di lavoro. Analogamente, puoi usare il client per richiedere l'annullamento di un'altra esecuzione del flusso di lavoro.

Nel seguente esempio, il flusso di lavoro cerca l'esecuzione per inviare un segnale da un database e invia il segnale in modo dinamico tramite il client del flusso di lavoro dinamico.

```
//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);
}
```

Implementazione del flusso di lavoro

Per implementare un flusso di lavoro, scrivi una classe che implementa l'interfaccia `@Workflow` desiderata. Ad esempio, l'interfaccia del flusso di lavoro di esempio (`MyWorkflow`) può essere implementata come segue:

```
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);
  }
}
```

Il metodo `@Execute` in questa classe è il punto di ingresso della logica del flusso di lavoro. Poiché il framework utilizza il replay per ricostruire lo stato dell'oggetto quando deve essere elaborata un'attività decisionale, viene creato un nuovo oggetto per ogni attività decisionale.

L'utilizzo di `Promise<`*T*`>` come parametro non è consentito nel metodo `@Execute` in un'interfaccia `@Workflow`. Questo perché una chiamata asincrona è una decisione esclusiva dell'intermediario. L'implementazione del flusso di lavoro in sé non dipende dalla modalità di invocazione (sincrona o asincrona). Di conseguenza, l'interfaccia client generata ha overload che accettano i parametri `Promise<`*T*`>` in modo che questi metodi possano essere chiamati in modo asincrono.

Il tipo di restituzione di un metodo `@Execute` può essere solo `void` o `Promise<`*T*`>`. Ricorda che un tipo di restituzione del client esterno corrispondente è `void` e non `Promise<>`. Poiché il client esterno non è progettato per essere utilizzato dal codice asincrono, il client esterno non restituisce oggetti. `Promise` Per ottenere i risultati delle esecuzioni dei flussi di lavoro dichiarati esternamente, è possibile progettare il flusso di lavoro in modo che aggiorni lo stato in un archivio dati esterno tramite un'attività. La visibilità di Amazon SWF APIs può essere utilizzata anche per recuperare il risultato di un flusso di lavoro a fini diagnostici. Non è consigliabile utilizzare la visibilità APIs per recuperare i risultati delle esecuzioni dei flussi di lavoro come pratica generale, poiché queste chiamate API potrebbero essere limitate da Amazon SWF. La visibilità APIs richiede l'identificazione dell'esecuzione del flusso di lavoro utilizzando una struttura. `WorkflowExecution` Puoi ottenere questa struttura dal client di flusso di lavoro generato chiamando il metodo `getWorkflowExecution`. Questo metodo restituisce la struttura `WorkflowExecution` corrispondente all'esecuzione del flusso di lavoro a cui il client è legato. Consulta il riferimento all'API di Amazon Simple Workflow Service per maggiori dettagli sulla visibilità APIs.

Quando chiami le attività dall'implementazione del flusso di lavoro, devi utilizzare il client di attività generato. Analogamente, per inviare segnali, devi utilizzare i client di flusso di lavoro generati.

Contesto di decisione

Il framework fornisce un contesto di ambiente ogni volta che il codice del flusso di lavoro viene eseguito dal framework. Questo contesto offre funzionalità specifiche a cui puoi accedere nell'implementazione del flusso di lavoro, ad esempio la creazione di un timer. Consulta la sezione relativa a Contesto di esecuzione per ulteriori informazioni.

Esposizione dello stato dell'esecuzione

Amazon SWF ti consente di aggiungere uno stato personalizzato nella cronologia del flusso di lavoro. L'ultimo stato riportato dall'esecuzione del flusso di lavoro ti viene restituito tramite chiamate di visibilità al servizio Amazon SWF e nella console Amazon SWF. Ad esempio, in un flusso di lavoro di elaborazione dell'ordine, puoi segnalare lo stato dell'ordine in fasi diverse come "ordine ricevuto", ordine spedito" e così via. In Java, ciò avviene tramite un metodo sull'interfaccia del flusso di lavoro che viene annotato con l'annotazione. AWS Flow Framework `@GetState` Quando il decisore ha terminato l'elaborazione di un task di decisione, chiama il metodo per ricevere l'ultimo stato dall'implementazione del flusso di lavoro. A parte le chiamate di visibilità, lo stato può essere recuperato anche utilizzando il client esterno generato (che utilizza internamente le chiamate API di visibilità).

L'esempio seguente mostra come configurare il contesto di esecuzione.

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

Si può utilizzare il client esterno generato per recuperare in qualsiasi momento l'ultimo stato dell'esecuzione del flusso di lavoro.

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

Nell'esempio precedente, lo stato di esecuzione è segnalato in varie fasi. Quando inizia l'istanza del flusso di lavoro, `periodicWorkflow` segnala lo stato iniziale come "Appena iniziata". Ogni chiamata a `callPeriodicActivity` aggiorna lo stato del flusso di lavoro. Una volta che `activity1` è stata chiamata 100 volte, il metodo esegue la restituzione e l'istanza del flusso di lavoro è completata.

Locali del flusso di lavoro

A volte, puoi avere la necessità di utilizzare le variabili statiche nell'implementazione del flusso di lavoro. Ad esempio, puoi voler archiviare un contatore a cui è stato effettuato l'accesso da vari posti (forse classi diverse) nell'implementazione del flusso di lavoro. Tuttavia, non puoi affidarti a variabili statiche nei flussi di lavoro, perché le variabili statiche sono condivise tra i thread, il che rappresenta un problema, perché un lavoratore elabora task di decisione diversi su thread diversi nello stesso momento. In alternativa, puoi archiviare questo stato in un campo dell'implementazione del flusso di lavoro, ma poi devi trasferire l'oggetto dell'implementazione. A questo scopo, il framework fornisce una classe `WorkflowExecutionLocal<?>`. Ogni stato che deve avere una variabile statica come semantica deve essere mantenuto come istanza locale utilizzando `WorkflowExecutionLocal<?>`. Puoi dichiarare e utilizzare una variabile statica di questo tipo. Ad esempio, nel seguente frammento di codice, un `WorkflowExecutionLocal<String>` viene utilizzato per archiviare un nome utente.

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

Implementazione di attività

Le attività vengono implementate fornendo un'implementazione dell'interfaccia `@Activities`. The AWS Flow Framework for Java utilizza le istanze di implementazione delle attività configurate sul worker per elaborare le attività in fase di esecuzione. Il lavoratore trova automaticamente l'implementazione di attività del tipo corretto.

Puoi utilizzare proprietà e campi per trasferire le risorse alle istanze di attività, ad esempio le connessioni del database. Poiché è possibile accedere all'oggetto di implementazione dell'attività da più thread, le risorse condivise devono essere thread-safe.

Ricorda che l'implementazione di attività non accetta parametri di tipo `Promise<>` e non restituisce oggetti di quel tipo. Questo perché l'implementazione di attività non deve dipendere dal modo in cui è stata invocata (sincrono o asincrono).

L'interfaccia delle attività mostrata in precedenza può essere implementata in questo modo:

```
public class MyActivitiesImpl implements MyActivities {

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

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

L'attività di implementazione ha a disposizione un contesto locale di thread per recuperare l'oggetto del task, l'oggetto del convertitore di dati in uso ecc. Si può accedere al contesto attuale tramite `ActivityExecutionContextProvider.getActivityExecutionContext()`. Per maggiori dettagli, consulta la AWS SDK per Java documentazione `ActivityExecutionContext` e la sezione dedicataContesto di esecuzione.

Completamento manuale della attività

L'annotazione `@ManualActivityCompletion` nell'esempio precedente è opzionale. È consentita solo sui metodi che implementano un'attività e viene utilizzata per configurare l'attività perché non sia completata automaticamente in fase di restituzione del metodo di attività. Ciò può essere utile quando si desidera completare l'attività in modo asincrono, ad esempio manualmente dopo il completamento di un'azione umana.

Per impostazione predefinita, il framework considera l'attività completata alla fase di restituzione del metodo di attività. Ciò significa che l'addetto all'attività segnala il completamento dell'attività ad Amazon SWF e gli fornisce i risultati (se presenti). Tuttavia, ci sono casi d'uso in cui non è consigliabile che il task di attività sia indicato come completato in questa fase. Questo è particolarmente utile quando stai modellando task umani. Ad esempio, il metodo di attività può inviare una e-mail alla persona che deve completare una parte del lavoro prima del completamento del task di attività. In questi casi, puoi annotare il metodo di attività con `@ManualActivityCompletion` per comunicare al lavoratore di attività che non la deve completare automaticamente. Per completare l'attività manualmente, puoi utilizzare il metodo `ManualActivityCompletionClient` fornito nel

framework o utilizzare il `RespondActivityTaskCompleted` metodo sul client Java Amazon SWF fornito nell'SDK Amazon SWF. Per ulteriori dettagli, consulta la documentazione. AWS SDK per Java

Per completare il task di attività, devi fornire un token del task. Il task token viene utilizzato da Amazon SWF per identificare in modo univoco le attività. Puoi accedere al token dal `ActivityExecutionContext` nell'implementazione di attività. Devi trasferire il token alla parte responsabile del completamento del task. Il token può essere recuperato dal `ActivityExecutionContext` chiamando `ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken()`.

L'attività `getName` dell'esempio di Hello World può essere implementata per inviare un'e-mail in cui si chiede a qualcuno di esprimere un messaggio di saluto:

```
@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";
}
```

Si può utilizzare il seguente frammento di codice per il saluto e chiudere il task utilizzando il `ManualActivityCompletionClient`. In alternativa, il task può anche non andare a buon fine:

```
public class CompleteActivityTask {

    public void completeGetNameActivity(String taskToken) {

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

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

AWS Lambda Attività di implementazione

Argomenti

- Informazioni su AWS Lambda
- Vantaggi e limiti dell'utilizzo delle attività Lambda
- Utilizzo delle attività Lambda nei flussi di lavoro AWS Flow Framework per Java
- Visualizza l'esempio HelloLambda

Informazioni su AWS Lambda

AWS Lambda è un servizio di elaborazione completamente gestito che esegue il codice in risposta a eventi generati da codice personalizzato o da vari AWS servizi come Amazon S3, DynamoDB, Amazon Kinesis, Amazon SNS e Amazon Cognito. Per ulteriori informazioni su Lambda, consulta la Guida per gli sviluppatori di AWS Lambda.

Amazon Simple Workflow Service fornisce un task Lambda che consente di eseguire funzioni Lambda al posto o insieme alle attività tradizionali di Amazon SWF.

> **Important**
>
> Sul tuo AWS account verranno addebitate le esecuzioni (richieste) Lambda eseguite da Amazon SWF per tuo conto. Per informazioni dettagliate sui prezzi di Lambda, consulta https://aws.amazon.com/lambda/ pricing/.

Vantaggi e limiti dell'utilizzo delle attività Lambda

L'utilizzo delle attività Lambda al posto di un'attività tradizionale di Amazon SWF offre numerosi vantaggi:

- Le attività Lambda non devono essere registrate o sottoposte a versioni come i tipi di attività di Amazon SWF.
- Puoi utilizzare qualsiasi funzione Lambda esistente che hai già definito nei tuoi flussi di lavoro.
- Le funzioni Lambda vengono richiamate direttamente da Amazon SWF; non è necessario implementare un programma di lavoro per eseguirle come è necessario fare con le attività tradizionali.
- Lambda fornisce metriche e log per tracciare e analizzare le esecuzioni delle funzioni.

L'utilizzo di task Lambda comporta anche alcuni limiti che è necessario conoscere:

- Le attività Lambda possono essere eseguite solo nelle AWS regioni che forniscono supporto per Lambda. Consulta le regioni e gli endpoint Lambda nel riferimento generale di Amazon Web Services per dettagli sulle regioni attualmente supportate per Lambda.
- Le attività Lambda sono attualmente supportate solo dall'API HTTP SWF di base e in Java. AWS Flow Framework Al momento non è disponibile alcun supporto per le attività Lambda in AWS Flow Framework for Ruby.

Utilizzo delle attività Lambda nei flussi di lavoro AWS Flow Framework per Java

Esistono tre requisiti per utilizzare le attività Lambda nei flussi di lavoro AWS Flow Framework per Java:

- Una funzione Lambda da eseguire. Puoi usare qualsiasi funzione Lambda che hai definito. Per ulteriori informazioni su come creare funzioni Lambda, consulta la Guida per gli AWS Lambda sviluppatori.
- Un ruolo IAM che fornisce l'accesso per eseguire funzioni Lambda dai flussi di lavoro Amazon SWF.
- Codice per pianificare l'attività Lambda dall'interno del flusso di lavoro.

Configurazione di un ruolo IAM

Prima di poter richiamare le funzioni Lambda da Amazon SWF, devi fornire un ruolo IAM che fornisca l'accesso a Lambda da Amazon SWF. Puoi eseguire una delle seguenti operazioni:

- scegli un ruolo predefinito, Ruolo, AWSLambda per autorizzare i flussi di lavoro a richiamare qualsiasi funzione Lambda associata al tuo account.
- definisci la tua politica e il ruolo associato per autorizzare i flussi di lavoro a richiamare particolari funzioni Lambda, specificate dai rispettivi Amazon Resource Names (). ARNs

Limita le autorizzazioni su un ruolo IAM

Puoi limitare le autorizzazioni su un ruolo IAM che fornisci ad Amazon SWF utilizzando `SourceArn` le chiavi `SourceAccount` e context nella tua policy di attendibilità delle risorse. Queste chiavi limitano l'utilizzo di una policy IAM in modo che venga utilizzata solo dalle esecuzioni di Amazon Simple Workflow Service che appartengono all'ARN del dominio specificato. Se utilizzi entrambe le chiavi di contesto della condizione globale, il `aws:SourceAccount` valore e l'account a cui si fa riferimento nel `aws:SourceArn` valore devono utilizzare lo stesso ID account quando vengono utilizzati nella stessa dichiarazione politica.

Nel seguente esempio, la chiave di `SourceArn` contesto limita l'utilizzo del ruolo del servizio IAM solo nelle esecuzioni di Amazon Simple Workflow Service che appartengono `someDomain` all'account,. `123456789012`

- Dichiarazione 1

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

  Operazione: `sts:AssumeRole`

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

Nell'esempio seguente, la chiave di `SourceAccount` contesto limita l'utilizzo del ruolo del servizio IAM solo nelle esecuzioni di Amazon Simple Workflow Service nell'account,. `123456789012`

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

Fornire ad Amazon SWF l'accesso per richiamare qualsiasi ruolo Lambda

Puoi utilizzare il ruolo predefinito, Role, per dare ai flussi di lavoro Amazon SWF la possibilità di richiamare qualsiasi AWSLambdafunzione Lambda associata al tuo account.

Utilizzare AWSLambda Role per consentire ad Amazon SWF l'accesso per richiamare le funzioni Lambda

1. Apri la console Amazon IAM.
2. Scegli Roles (Ruoli), quindi Create New Role (Crea nuovo ruolo).
3. Assegna un nome al ruolo, come `swf-lambda`, quindi scegli Next Step (Fase successiva).
4. In AWS Service Roles, scegli Amazon SWF e scegli Next Step.
5. Nella schermata Allach Policy, scegli AWSLambdaRuolo dall'elenco.
6. Scegli Next Step (Fase successiva), quindi Create Role (Crea ruolo) dopo aver esaminato il ruolo.

Definizione di un ruolo IAM per fornire l'accesso per richiamare una funzione Lambda specifica

Se desideri fornire l'accesso per richiamare una funzione Lambda specifica dal tuo flusso di lavoro, dovrai definire la tua policy IAM.

Creare una policy IAM per fornire l'accesso a una particolare funzione Lambda

1. Apri la console Amazon IAM.
2. Scegli Policies (Policy), quindi Create Policy (Crea policy).
3. Scegli Copia una policy AWS gestita e seleziona AWSLambdaRuolo dall'elenco. Viene generata una policy. Se necessario, modificane il nome e la descrizione.
4. Nel campo Resource del Policy Document, aggiungi l'ARN delle tue funzioni Lambda. Per esempio:

- Risorsa: `arn:aws:lambda:us-east-1:111122223333:function:hello_lambda_function`

> **Note**
>
> Per una descrizione completa di come specificare le risorse in un ruolo IAM, consulta Panoramica delle politiche IAM nell'uso di IAM.

5. Scegli Create Policy (Crea policy) per completare la creazione della policy.

Puoi quindi selezionare questa policy quando crei un nuovo ruolo IAM e utilizzarlo per concedere a invoke l'accesso ai tuoi flussi di lavoro Amazon SWF. Questa procedura è molto simile alla creazione di un ruolo con la politica AWSLambdaRole. Scegli invece la tua policy quando crei il ruolo.

Per creare un ruolo Amazon SWF utilizzando la tua policy Lambda

1. Apri la console Amazon IAM.
2. Scegli Roles (Ruoli), quindi Create New Role (Crea nuovo ruolo).
3. Assegna un nome al ruolo, come `swf-lambda-function`, quindi scegli Next Step (Fase successiva).
4. In AWS Service Roles, scegli Amazon SWF e scegli Next Step.
5. Nella schermata Allega policy, scegli la policy specifica per la funzione Lambda dall'elenco.
6. Scegli Next Step (Fase successiva), quindi Create Role (Crea ruolo) dopo aver esaminato il ruolo.

Pianifica l'esecuzione di un'attività Lambda

Dopo aver definito un ruolo IAM che ti consente di richiamare le funzioni Lambda, puoi pianificarne l'esecuzione come parte del tuo flusso di lavoro.

> **Note**
>
> Questo processo è ampiamente dimostrato dall'HelloLambda esempio contenuto in. AWS SDK per Java

Per pianificare l'esecuzione di un'attività Lambda

1. Nell'implementazione di flusso di lavoro, ottieni un'istanza di `LambdaFunctionClient` chiamando `getLambdaFunctionClient()` su un'istanza `DecisionContext`.

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

2. Pianifica l'attività utilizzando il `scheduleLambdaFunction()` metodo su`LambdaFunctionClient`, passandole il nome della funzione Lambda che hai creato e tutti i dati di input per l'attività 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. Nel programma di avvio dell'esecuzione del workflow, aggiungi il ruolo IAM lambda alle opzioni di workflow predefinite utilizzando`StartWorkflowOptions.withLambdaRole()`, quindi passa le opzioni all'avvio del flusso di lavoro.

```
// 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);
```

Visualizza l'esempio HelloLambda

Un esempio che fornisce un'implementazione di un flusso di lavoro che utilizza un'attività Lambda è fornito in. AWS SDK per Java Per visualizzarlo, and/or eseguilo, scarica il codice sorgente.

Una descrizione completa di come creare ed eseguire l'HelloLambdaesempio è fornita nel file README fornito con AWS Flow Framework gli esempi Java.

Esecuzione di programmi scritti con AWS Flow Framework for Java

Argomenti

- WorkflowWorker
- ActivityWorker
- Modello di threading di lavoratore
- Estensibilità dei lavoratori

Il framework fornisce classi di lavoro per inizializzare il runtime AWS Flow Framework for Java e comunicare con Amazon SWF. Per implementare un lavoratore di attività o di flusso di lavoro, devi creare e avviare un'istanza di una classe di lavoratore. Queste classi di lavoratori sono responsabili della gestione delle operazioni asincrone in corso, dell'utilizzo di metodi asincroni che vengono sbloccati e della comunicazione con Amazon SWF. Possono essere configurate con implementazioni di flusso di lavoro e attività, il numero di thread, l'elenco di task da sottoporre a polling e così via.

Il framework include due classi di lavoratore, una per le attività e l'altra per i flussi di lavoro. Per eseguire la logica di flusso di lavoro, devi utilizzare la classe `WorkflowWorker`. Per le attività, viene invece utilizzata la classe `ActivityWorker`. Queste classi eseguono automaticamente il polling di Amazon SWF per le attività e richiamano i metodi appropriati nella tua implementazione.

L'esempio seguente mostra come creare un'istanza di `WorkflowWorker` e avviare il polling dei task:

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

```
worker.addWorkflowImplementationType(MyWorkflowImpl.class);

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

La procedura di base per creare un'istanza di `ActivityWorker` e avviare il polling dei task è la seguente:

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

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

Quando desideri chiudere un'attività o un dispositivo decisionale, l'applicazione deve chiudere le istanze delle classi di lavoro utilizzate e l'istanza del client Java Amazon SWF. In questo modo, tutte le risorse utilizzate dalle classi di lavoratore vengono rilasciate correttamente.

```

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

```

Per avviare un'esecuzione, crea semplicemente un'istanza del client esterno generato e chiama il metodo `@Execute`.

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

WorkflowWorker

Come suggerisce il nome, questa classe di lavoratore è utilizzata dall'implementazione di flusso di lavoro. È configurata con un elenco di task e con il tipo di implementazione di flusso di lavoro. La classe di lavoratore esegue un ciclo per il polling dei task di decisione nell'elenco di task specificato.

Quando un task di decisione viene ricevuto, crea un'istanza dell'implementazione di flusso di lavoro e chiama il metodo `@Execute` per elaborare il task.

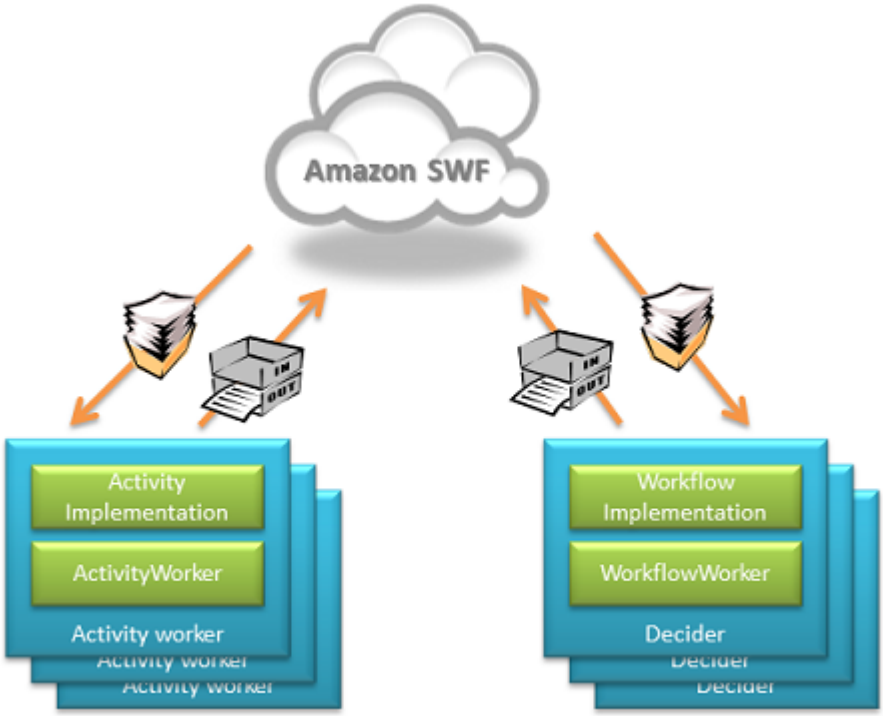
ActivityWorker

Per implementare dei lavoratori di attività, puoi utilizzare la classe `ActivityWorker` per eseguire efficacemente il polling dei task di attività in un elenco di task. Configura quindi il lavoratore di attività con oggetti di implementazione di attività. Questa classe di lavoratore esegue un ciclo per il polling dei task di attività nell'elenco di task specificato. Quando si riceve un task di attività, cerca l'implementazione appropriata che hai fornito e chiama il metodo di attività per elaborare il task. A differenza di `WorkflowWorker`, che chiama la factory per creare una nuova istanza per ogni task di decisione, `ActivityWorker` utilizza semplicemente l'oggetto che hai fornito.

La `ActivityWorker` classe utilizza le annotazioni AWS Flow Framework for Java per determinare le opzioni di registrazione ed esecuzione.

Modello di threading di lavoratore

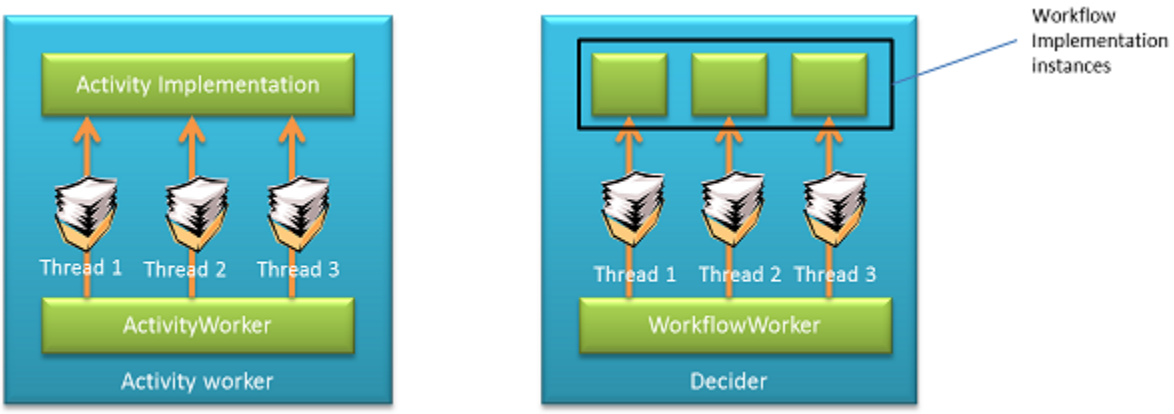
In Java, AWS Flow Framework l'incarnazione di un'attività o di un decisore è un'istanza della classe operaia. La tua applicazione è responsabile della configurazione e della creazione di un'istanza dell'oggetto lavoratore su ogni macchina nonché del processo che agisce come lavoratore. L'oggetto worker riceve quindi automaticamente le attività da Amazon SWF, le invia all'implementazione dell'attività o del flusso di lavoro e riporta i risultati ad Amazon SWF. Una singola istanza di flusso di lavoro può interessare molti lavoratori. Quando Amazon SWF ha una o più attività in sospeso, assegna un'attività al primo lavoratore disponibile, quindi a quello successivo e così via. In questo modo, i task che appartengono alla stessa istanza di flusso di lavoro possono essere elaborati su differenti lavoratori contemporaneamente.



Inoltre, ogni lavoratore può essere configurato per elaborare task su più thread. Ciò significa che i task di attività di un'istanza di flusso di lavoro possono essere eseguiti simultaneamente anche se vi è un solo lavoratore.

Le attività decisionali si comportano in modo simile, con l'eccezione che Amazon SWF garantisce che per un determinato flusso di lavoro possa essere eseguita solo una decisione alla volta. Una singola esecuzione di flusso di lavoro richiede in genere più task di decisione ed è quindi possibile che venga eseguita su più processi e thread. Il decisore è configurato con il tipo di implementazione di flusso di lavoro. Quando riceve un task di decisione, crea un'istanza (oggetto) dell'implementazione di flusso di lavoro. Il framework fornisce un modello factory estensibile per la creazione di queste istanze. La factory di flusso di lavoro di default crea un nuovo oggetto ogni volta. Puoi fornire factory personalizzate per annullare questo comportamento.

Contrariamente ai decisori, che sono configurati con tipi di implementazione di flusso di lavoro, i lavoratori di attività sono configurati con istanze (oggetti) delle implementazioni di attività. Quando un lavoratore di attività riceve un task di attività, questo è inviato all'oggetto di implementazione di attività appropriato.



L'operatore del flusso di lavoro gestisce un unico pool di thread ed esegue il flusso di lavoro sullo stesso thread utilizzato per eseguire il polling di Amazon SWF per l'attività. Poiché le attività durano a lungo (almeno rispetto alla logica del flusso di lavoro), la classe Activity Worker gestisce due pool di thread separati: uno per il polling di Amazon SWF per le attività e l'altro per l'elaborazione delle attività eseguendo l'implementazione dell'attività. Ciò ti consente di configurare il numero di thread per il polling dei task indipendentemente dal numero di thread per eseguirli. Ad esempio, puoi avere un numero ridotto di thread per il polling e un numero elevato di thread per l'esecuzione dei task. L'activity worker class interroga Amazon SWF per un'attività solo quando dispone di un thread di sondaggio gratuito e di un thread libero per l'elaborazione dell'attività.

Questo comportamento di threading e creazione di istanze implica quanto segue:

1. Le implementazioni di attività devono essere stateless. Non devi utilizzare variabili di istanza per archiviare lo stato dell'applicazione in oggetti attività. Puoi comunque utilizzare dei campi per archiviare risorse come le connessioni di database.
2. Le implementazioni di attività devono essere thread-safe. Poiché la stessa istanza può essere utilizzata per elaborare attività da thread diversi contemporaneamente, l'accesso alle risorse condivise dal codice di attività deve essere sincronizzato.
3. L'implementazione di flusso di lavoro può essere stateful e le variabili di istanza possono essere utilizzate per archiviare lo stato. Anche se viene creata una nuova istanza dell'implementazione di flusso di lavoro per elaborare ogni task di decisione, il framework assicurerà la corretta ricreazione dello stato. Tuttavia, l'implementazione di flusso di lavoro deve essere deterministica. Per ulteriori informazioni, consulta la sezione Comprensione di un task in AWS Flow Framework for Java.
4. Le implementazioni di flusso di lavoro non devono essere thread-safe quando si utilizza la factory di default. L'implementazione di default garantisce che un'istanza dell'implementazione di flusso di lavoro è utilizzata da un solo thread alla volta.

Estensibilità dei lavoratori

The AWS Flow Framework for Java contiene anche un paio di classi di lavoro di basso livello che offrono controllo ed estensibilità dettagliati. Mediante tali classi, puoi personalizzare completamente la registrazione dei tipi di flusso di lavoro e di attività e impostare factory per la creazione di oggetti di implementazione. Questi lavoratori sono `GenericWorkflowWorker` e `GenericActivityWorker`.

Il lavoratore `GenericWorkflowWorker` può essere configurato con una factory per creare factory di definizione di flusso di lavoro. Il ruolo di una factory di definizione di flusso di lavoro è di creare istanze dell'implementazione di flusso di lavoro e di fornire impostazioni di configurazione come le opzioni di registrazione. In circostanze normali, devi utilizzare la classe `WorkflowWorker` direttamente. Questa creerà e configurerà automaticamente l'implementazione delle factory fornite nel framework, ovvero `POJOWorkflowDefinitionFactoryFactory` e `POJOWorkflowDefinitionFactory`. La factory richiede che la classe di implementazione di flusso di lavoro abbia un costruttore senza argomenti. Questo costruttore è utilizzato per creare istanze dell'oggetto di flusso di lavoro al runtime. La factory analizza le annotazioni utilizzate nell'interfaccia e nell'implementazione di flusso di lavoro per creare opzioni di registrazione ed esecuzione appropriate.

Puoi fornire una tua implementazione delle factory mediante `WorkflowDefinitionFactory`, `WorkflowDefinitionFactoryFactory` e `WorkflowDefinition`. La classe `WorkflowDefinition` è utilizzata dalla classe di lavoratore per inviare task di decisione e segnali. Implementando queste classi di base, puoi personalizzare completamente la factory e l'invio di richieste all'implementazione di flusso di lavoro. Ad esempio, puoi utilizzare questi punti di estensibilità per fornire un modello di programmazione personalizzato per la scrittura di flussi di lavoro, ad esempio, basato sulle tue annotazioni o generato a partire da WSDL anziché mediante l'approccio Code First utilizzato dal framework. Per utilizzare le tue factory personalizzate, dovrai servirti della classe `GenericWorkflowWorker`. Per maggiori dettagli su queste classi, consulta la documentazione. AWS SDK per Java

Allo stesso modo, `GenericActivityWorker` ti consente di fornire una factory di implementazione di attività personalizzata. Implementando le classi `ActivityImplementationFactory` e `ActivityImplementation`, puoi controllare completamente la creazione di istanze di attività nonché personalizzare opzioni di registrazione ed esecuzione. Per maggiori dettagli su queste classi, consulta la AWS SDK per Java documentazione.

Contesto di esecuzione

Argomenti

Il framework fornisce un contesto di ambiente alle implementazioni di flusso di lavoro e attività. Questo contesto è specifico del task in corso di elaborazione e fornisce alcune utilità che puoi utilizzare nella tua implementazione. Un oggetto di contesto è creato ogni volta che il lavoratore elabora un nuovo task.

Contesto di decisione

Quando un'attività viene eseguita, la decisione quadro fornisce il contesto per l'implementazione di flussi di lavoro attraverso la `DecisionContext` classe `DecisionContext` fornisce informazioni sensibili al contesto quali l'esecuzione del flusso di lavoro eseguire Id e orologio timer e funzionalità.

Accesso nell'implementazione del flusso DecisionContext di lavoro

Puoi accedere a `DecisionContext` nell'implementazione di flusso di lavoro utilizzando la classe `DecisionContextProviderImpl`. In alternativa, puoi inserire il contesto in un campo o in una proprietà di tale implementazione utilizzando Spring come mostrato nella sezione relativa alla testabilità e all'inserimento delle dipendenze.

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

Creazione di un orologio e di un timer

La classe `DecisionContext` contiene una proprietà di tipo `WorkflowClock` che fornisce la funzionalità di orologio e timer. Poiché la logica del flusso di lavoro deve essere deterministica, non è necessario utilizzare direttamente l'orologio di sistema nell'implementazione del flusso di lavoro. Il metodo `currentTimeMills` su `WorkflowClock` restituisce l'ora dell'evento di avvio della decisione in corso di elaborazione. In questo modo, ottieni lo stesso valore di ora durante la riproduzione, rendendo di conseguenza deterministica la logica di flusso di lavoro.

`WorkflowClock` include inoltre un metodo `createTimer` che restituisce un oggetto `Promise` che diventa pronto dopo l'intervallo specificato. Puoi utilizzare questo valore come parametro per altri metodi asincroni allo scopo di ritardarne l'esecuzione in base al periodo di tempo specificato. In questo modo, puoi pianificare efficacemente un'attività o un metodo asincrono per un'esecuzione successiva.

L'esempio nel listato seguente mostra come chiamare periodicamente un'attività.

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

Nel listato precedente, il metodo asincrono `callPeriodicActivity` chiama `activity1` e quindi crea un timer utilizzando la classe `AsyncDecisionContext` corrente. Passa l'oggetto `Promise` restituito come argomento a una chiamata ricorsiva a se stesso. Questa chiamata attende fino all'attivazione del timer (1 ora i questo esempio) prima dell'esecuzione.

Contesto di esecuzione di attività

Esattamente come `DecisionContext` fornisce informazioni di contesto quando un task di decisione è in corso di elaborazione, `ActivityExecutionContext` fornisce informazioni di contesto simili durante l'elaborazione di un task di attività. Questo contesto è disponibile per il tuo codice delle attività mediante la classe `ActivityExecutionContextProviderImpl`.

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

Utilizzando `ActivityExecutionContext`, puoi eseguire le seguenti operazioni:

Heartbeat di un'attività a esecuzione prolungata

Se l'attività è di lunga durata, deve segnalarne periodicamente l'avanzamento ad Amazon SWF per informarlo che l'attività sta ancora facendo progressi. In assenza di tale heartbeat, è possibile che si verifichi il timeout del task se un timeout di heartbeat è stato impostato alla registrazione del tipo di attività o durante la pianificazione dell'attività. Per inviare un heartbeat, puoi utilizzare il metodo `recordActivityHeartbeat` su `ActivityExecutionContext`. L'heartbeat fornisce inoltre un meccanismo per annullare le attività in corso. Per informazioni dettagliate e un esempio, consulta la sezione Gestione errori.

Ottenimento dei dettagli del task di attività

Se lo desideri, puoi ottenere tutti i dettagli dell'attività che sono stati trasmessi da Amazon SWF quando l'esecutore ha ricevuto l'attività. Sono incluse le informazioni relative agli input al task, il tipo di task, il token del task, ecc. Se desideri implementare un'attività che viene completata manualmente, ad esempio da un'azione umana, devi utilizzare il per recuperare il token dell'attività e passarlo `ActivityExecutionContext` al processo che alla fine completerà l'attività. Per ulteriori informazioni, consulta la sezione su Completamento manuale della attività.

Ottieni l'oggetto client Amazon SWF utilizzato dall'esecutore

L'oggetto client Amazon SWF utilizzato dall'esecutore può essere recuperato chiamando method on. `getService` `ActivityExecutionContext` Ciò è utile se desideri effettuare una chiamata diretta al servizio Amazon SWF.

Esecuzioni del flusso di lavoro figlio

Negli esempi riportati finora, abbiamo iniziato l'esecuzione del flusso di lavoro direttamente da un'applicazione. Tuttavia, un'esecuzione del flusso di lavoro può essere avviata dall'interno di un flusso di lavoro chiamando il metodo del punto di ingresso del flusso di lavoro sul client generato. Quando un'esecuzione del flusso di lavoro viene avviata dal contesto di un'altra esecuzione del flusso di lavoro viene chiamata esecuzione del flusso di lavoro figlio. Questa operazione ti permette di eseguire il refactoring dei flussi di lavoro complessi in unità più piccole e condividerle potenzialmente su diversi flussi di lavoro. Ad esempio, puoi creare un flusso di elaborazione dei pagamenti e chiamarlo da un flusso di lavoro di elaborazione di un ordine.

Da un punto di vista semantico, l'esecuzione del flusso di lavoro figlio si comporta analogamente al flusso di lavoro standalone tranne che per le seguenti caratteristiche:

1. Quando il flusso di lavoro principale termina a causa di un'azione esplicita da parte dell'utente, ad esempio chiamando l'API Amazon SWF`TerminateWorkflowExecution`, o viene interrotto a causa di un timeout, il destino dell'esecuzione del flusso di lavoro secondario sarà determinato da una policy secondaria. Puoi impostare la policy figlio in modo che termini, annulli o abbandoni (mantenere in esecuzione) le esecuzioni del flusso di lavoro figlio.
2. L'output del flusso di lavoro figlio (valore restituito del metodo del punto di ingresso) può essere utilizzato dall'esecuzione del flusso di lavoro padre come l'oggetto `Promise<T>` restituito da un metodo asincrono. Ciò è diverso dalle esecuzioni autonome in cui l'applicazione deve ottenere l'output utilizzando Amazon SWF. APIs

Nell'esempio seguente, il flusso di lavoro `OrderProcessor` crea un flusso di lavoro figlio `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);

}
```

Flussi di lavoro continui

In alcuni casi d'uso, può servire un flusso di lavoro di durata lunga o eterna, ad esempio uno che monitori l'integrità di una flotta di server.

> **Note**
>
> Poiché Amazon SWF conserva l'intera cronologia dell'esecuzione di un flusso di lavoro, la cronologia continuerà a crescere nel tempo. Il framework recupera la cronologia da Amazon SWF quando esegue una riproduzione; questo può diventare costoso se le dimensioni della cronologia sono troppo grandi. Nei flussi di lavoro di lunga durata o continui, devi chiudere periodicamente l'esecuzione in corso e avviarne una nuova per poter proseguire.

Questo è un proseguimento logico dell'esecuzione del flusso di lavoro. A questo scopo si può usare un self client generato. Nell'implementazione del flusso di lavoro, basta chiamare il metodo `@Execute` sul self client. Una volta completata l'esecuzione corrente, il framework avvia una nuova esecuzione utilizzando lo stesso ID del flusso di lavoro.

Puoi anche proseguire l'esecuzione chiamando il metodo `continueAsNewOnCompletion` nel `GenericWorkflowClient` che puoi recuperare dal `DecisionContext` corrente. Ad esempio, la seguente implementazione del flusso di lavoro imposta un timer perché si attivi dopo un giorno e chiama il suo punto di ingresso per avviare una nuova esecuzione.

```
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();
    }
```

```
}
```

Quando un flusso di lavoro si chiama ricorsivamente, il framework chiude il flusso di lavoro in corso al completamento dei task in sospeso e avvia una nuova esecuzione. Ricorda che fino a quando ci sono task in sospeso, l'esecuzione corrente non può essere chiusa. La nuova esecuzione non eredita automaticamente la cronologia o i dati da quella originale; se vuoi esportare qualche stato sulla nuova esecuzione, dovrai trasferirlo esplicitamente come input.

Impostazione della priorità delle attività in Amazon SWF

Per impostazione predefinita, i task in un elenco di task sono consegnati in base alla relativa ora di arrivo. Per quanto possibile, i task pianificati per primi vengono eseguiti per primi. Impostando una priorità opzionale, puoi dare priorità a determinate attività: Amazon SWF cercherà di fornire attività con priorità più alta in un elenco di attività prima di quelle con priorità inferiore.

Puoi impostare priorità di task per flussi di lavoro e attività. La priorità di task di un flusso di lavoro non ha alcuna incidenza sulla priorità di task di attività che pianifica e nemmeno sui flussi di lavoro figlio che avvia. La priorità predefinita per un'attività o un flusso di lavoro viene impostata (da te o da Amazon SWF) durante la registrazione e la priorità dell'attività registrata viene sempre utilizzata a meno che non venga sostituita durante la pianificazione dell'attività o l'avvio di un'esecuzione del flusso di lavoro.

I valori della priorità di task possono andare da "-2147483648" a "2147483647", con i numeri più alti indicanti la priorità più elevata. Se non imposti la priorità di task per un'attività o un flusso di lavoro, verrà assegnata la priorità zero ("0").

Argomenti

- Impostazione della priorità di task per flussi di lavoro
- Impostazione della priorità di task per attività

Impostazione della priorità di task per flussi di lavoro

Puoi impostare la priorità di task per un flusso di lavoro durante la registrazione o l'avvio dello stesso. La priorità di task impostata alla registrazione del flusso di lavoro è utilizzata come impostazione di default per qualsiasi esecuzione di flusso di lavoro di quel tipo, a meno che non venga sovrascritta all'avvio dell'esecuzione di flusso di lavoro.

Per registrare un tipo di flusso di lavoro con una priorità di attività predefinita, imposta l'defaultTaskPriorityopzione in WorkflowRegistrationOptionsquando lo dichiari:

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

Puoi anche impostare taskPriority per un flusso di lavoro quando avvii quest'ultimo, sovrascrivendo la priorità di task (di default) registrata.

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

PriorityWorkflowClientExternalFactory cf
    = new PriorityWorkflowClientExternalFactoryImpl(swfService, domain);

priority_workflow_client = cf.getClient();

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

Puoi inoltre impostare la priorità di task all'avvio di un flusso di lavoro figlio o quando si continua un flusso di lavoro come nuovo. Ad esempio, è possibile impostare l'opzione ContinueAsNewWorkflowExecutionParametersTaskPriority in o in. StartChildWorkflowExecutionParameters

Impostazione della priorità di task per attività

Puoi impostare la priorità di task per un attività durante la registrazione o la pianificazione della stessa. La priorità di task impostata quando si registra un tipo di attività è utilizzata come priorità di default all'esecuzione dell'attività, a meno che non venga sovrascritta quando si pianifica l'attività.

Per registrare un tipo di attività con una priorità di attività predefinita, imposta l'defaultTaskPriorityopzione in ActivityRegistrationOptionsquando la dichiari:

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

Puoi anche impostare taskPriority per un'attività durante la pianificazione, sovrascrivendo la priorità di task (di default) registrata.

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

ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();

activityClient.doSomethingImportant(activityOptions);
```

DataConverters

Quando l'implementazione di flusso di lavoro chiama un'attività remota, l'input passato e il risultato dell'esecuzione dell'attività devono essere serializzati per essere trasmessi. Il framework utilizza la DataConverter classe per questo scopo. Si tratta di una classe astratta che puoi implementare per fornire un serializzatore personalizzato. Nel framework viene fornita un'implementazione predefinita basata sul serializzatore Jackson. `JsonDataConverter` Per ulteriori dettagli, consulta la documentazione di AWS SDK per Java. Fai riferimento alla documentazione del processore Jackson JSON per informazioni dettagliate sul modo in cui Jackson esegue la serializzazione e sulle annotazioni che possono essere utilizzate per modificarla. Il formato di trasmissione è considerato come parte del contratto. Di conseguenza, puoi specificare una classe `DataConverter` sulle interfacce di attività e di flusso di lavoro impostando la proprietà `DataConverter` delle annotazioni `@Activities` e `@Workflow`.

Il framework creerà oggetti del tipo `DataConverter` specificato sull'annotazione `@Activities` per serializzare gli input all'attività e per deserializzarne il risultato. Analogamente, gli oggetti del tipo `DataConverter` specificato sull'annotazione `@Workflow` saranno utilizzati per serializzare i parametri che passi al flusso di lavoro e, nel caso di un flusso di lavoro figlio, per deserializzare il risultato. Oltre agli input, il framework trasmette anche dati aggiuntivi ad Amazon SWF, ad esempio

i dettagli delle eccezioni, il serializzatore del flusso di lavoro verrà utilizzato anche per serializzare questi dati.

Puoi anche fornire un'istanza di `DataConverter` se non vuoi che venga creata automaticamente dal framework. I client generati hanno overload di costruttore che accettano un oggetto `DataConverter`.

Se non specifichi un tipo di `DataConverter` e non passi un oggetto `DataConverter`, `JsonDataConverter` sarà utilizzato per impostazione predefinita.

Passaggio di dati a metodi asincroni

Argomenti

L'utilizzo di `Promise<`*T*`>` è stato descritto nelle sezioni precedenti. In questa, vengono presentati alcuni casi d'uso avanzati di `Promise<`*T*`>`.

Passaggio di raccolte e mappe a metodi asincroni

Il framework supporta il passaggio di matrici, raccolte e mappe come tipi `Promise` a metodi asincroni. Ad esempio, un metodo asincrono può accettare `Promise<ArrayList<String>>` come argomento come mostrato nel listato seguente.

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

Sul piano semantico, il comportamento è quello di qualsiasi altro parametro di tipo `Promise` e il metodo asincrono attenderà fino a che la raccolta diventa disponibile prima di avviare l'esecuzione.

Se i membri di una raccolta sono oggetti `Promise`, il framework può attendere che tutti i membri diventino pronti come mostrato nel frammento seguente. In questo modo, il metodo asincrono attende che ogni membro della raccolta diventi disponibile.

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

Nota che l'annotazione `@Wait` deve essere utilizzata nel parametro per indicare che contiene oggetti `Promise`.

Considera inoltre che l'attività `printActivity` accetta un argomento `String` ma il metodo corrispondente nel client generato accetta Promise<String>. Stiamo chiamando il metodo sul client e non richiamando il metodo dell'attività direttamente.

impostabile <T>

`Settable<T>` è un tipo derivato di `Promise<T>` che fornisce un metodo set con cui impostare manualmente il valore di un oggetto `Promise`. Ad esempio, il seguente flusso di lavoro attende la ricezione di un segnale attendendo `Settable<?>`, impostato nel metodo del segnale:

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

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

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

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

```
    }
}
```

`Settable<?>` può inoltre essere concatenato a un'altra promessa alla volta. Puoi utilizzare `AndPromise` e `OrPromise` per raggruppare le promesse. Puoi annullare la concatenazione di `Settable` chiamando il metodo `unchain()`. Quando concatenato, `Settable<?>` diventa automaticamente pronto quando la promessa a cui è concatenato diventa pronta. La concatenazione è particolarmente utile quando desideri utilizzare una promessa restituita dall'ambito di un metodo `doTry()` in altre parti del programma. Poiché `TryCatchFinally` viene utilizzata come classe annidata, non è possibile dichiarare una `Promise<>` nell'ambito del genitore e impostarla. `doTry()` Questo perché Java richiede variabili che devono essere dichiarate nell'ambito padre e utilizzate in classi nidificate per essere contrassegnate come final. Esempio:

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

    new TryFinally() {

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

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

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

    return result;
}
```

`Settable` può essere concatenato a una promessa alla volta. Puoi annullare la concatenazione di `Settable` chiamando il metodo `unchain()`.

@NoWait

Quando passi un oggetto `Promise` a un metodo asincrono, per impostazione predefinita il framework attende che gli oggetti `Promise` diventino pronti prima di eseguire il metodo (ad eccezione dei tipi di raccolta). Puoi eseguire l'override di questo comportamento utilizzando l'annotazione `@NoWait` sui parametri nella dichiarazione del metodo asincrono. Ciò è utile se passi `Settable<T>`, che verrà impostato dal metodo asincrono stesso.

Promise <Void>

Le dipendenze nei metodi asincroni sono implementate passando l'oggetto `Promise` restituito da un metodo come argomento a un altro metodo. Possono tuttavia esserci casi in cui vuoi che un metodo restituisca `void` e che altri metodi asincroni siano eseguiti dopo il completamento di quel metodo. Per quei casi, puoi utilizzare `Promise<Void>` come tipo restituito del metodo. La classe `Promise` fornisce un metodo `Void` statico che puoi utilizzare per creare un oggetto `Promise<Void>`. Questo oggetto `Promise` diventerà pronto al termine dell'esecuzione del metodo asincrono. Puoi passare questo oggetto `Promise` a un altro metodo asincrono come qualsiasi altro oggetto `Promise`. Se utilizzi `Settable<Void>`, chiama il metodo set con null per renderlo pronto.

AndPromise e OrPromise

`AndPromise` e `OrPromise` ti consentono di raggruppare molteplici oggetti `Promise<>` in un'unica promessa logica. Un oggetto `AndPromise` diventa pronto quanto tutte le promesse utilizzate per costruirlo diventano pronte. Un oggetto `OrPromise` diventa pronto quando qualsiasi promessa nella raccolta di promesse utilizzata per costruirla diventa pronta. Puoi chiamare `getValues()` su `AndPromise` e `OrPromise` per recuperare l'elenco di valori delle promesse costituenti.

Testabilità e inserimento delle dipendenze

Argomenti

- Integrazione di Spring
- JUnit Integrazione

Il framework è progettato per essere compatibile con l'Inversione del controllo (Inversion of Control, IoC). Le implementazioni di flussi di lavoro e di attività, nonché i lavoratori e gli oggetti di contesto forniti dal framework, si possono configurare e creare come istanze tramite contenitori come Spring.

Il framework offre un'integrazione immediata con Spring Framework. Inoltre, JUnit è stata fornita l'integrazione con per le implementazioni del flusso di lavoro e delle attività di unit testing.

Integrazione di Spring

Il pacchetto com.amazonaws.services.simpleworkflow.flow.spring contiene classi che semplificano l'utilizzo di Spring framework nelle applicazioni. Comprendono lavoratori di flusso di lavoro e di attività compatibili con Scope e Spring: `WorkflowScope`, `SpringWorkflowWorker` e `SpringActivityWorker`. Queste classi ti permettono di configurare le implementazioni di attività e flusso di lavoro, nonché i lavoratori interamente tramite Spring.

WorkflowScope

`WorkflowScope` è una implementazione in ambito Spring personalizzata fornita dal framework. Lo scope ti permette di creare oggetti nel contenitore Spring la cui durata è limitata a quella di un task di decisione. I bean nello scope sono creati come istanze ogni volta che un lavoratore riceve un task di decisione. Devi utilizzare questo scope per i bean di implementazione del flusso di lavoro e per ogni altro bean da cui dipende. Per i bean di implementazione del flusso di lavoro non si devono usare gli scopes singleton e prototype forniti da Spring, perché il framework richiede la creazione di un nuovo bean per ciascun task di decisione. In caso contrario si verifica un comportamento inatteso.

Il seguente esempio mostra un frammento di codice della configurazione Spring che registra il `WorkflowScope` e poi lo utilizza per configurare un bean di implementazione del flusso di lavoro e un bean di client dell'attività.

```
<!-- 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 riga di configurazione: `<aop:scoped-proxy proxy-target-class="false" />`, utilizzata nella configurazione del bean `workflowImpl`, è obbligatoria perché `WorkflowScope` non supporta il proxy tramite CGLIB. Devi utilizzare questa configurazione per tutti i bean in `WorkflowScope` collegati a un altro bean in uno scope diverso. In questo caso, il bean `workflowImpl` deve essere collegato a un bean del lavoratore di flusso di lavoro in scope singleton (vedi l'esempio completo in basso).

Puoi approfondire l'utilizzo degli scope personalizzati nella documentazione di Spring Framework.

Lavoratori compatibili con Spring

Quando usi Spring, devi utilizzare le classi di lavoratori compatibili con Spring fornite dal framework: `SpringWorkflowWorker` e `SpringActivityWorker`. Questi lavoratori possono essere inseriti in un'applicazione tramite Spring, come illustrato nel prossimo esempio. I lavoratori compatibili con Spring implementano l'interfaccia `SmartLifecycle` di Spring e per impostazione predefinita iniziano automaticamente a eseguire il polling dei task quando viene avviato il contesto Spring. Puoi disattivare questa funzionalità impostando la proprietà `disableAutoStartup` del lavoratore su `true`.

L'esempio seguente mostra come configurare un decisore. Questo esempio utilizza le interfacce `MyActivities` e `MyWorkflow` (non mostrate qui) e le relative implementazioni, `MyActivitiesImpl` e `MyWorkflowImpl`. Le interfacce client e le implementazioni generate sono `MyWorkflowClient`/`MyWorkflowClientImpl` e `MyActivitiesClient`/`MyActivitiesClientImpl` (anch'esse non mostrate qui).

Il client delle attività viene introdotto nell'implementazione del flusso di lavoro utilizzando la funzionalità di collegamento automatico di Spring:

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

   @Override
```

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

La configurazione Spring per il decisore è la seguente:

```
<?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>
```

Poiché `SpringWorkflowWorker` è completamente configurato in Spring e avvia automaticamente il polling quando il contesto Spring viene inizializzato, il processo host per il decisore è semplice:

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

```
}
```

Analogamente, il lavoratore di attività può essere configurato nel modo seguente:

```
<?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>
```

Il processo di hosting del lavoratore di attività è simile a quello del decisore:

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

Contesto di decisione dell'introduzione

Se l'implementazione del flusso di lavoro dipende dagli oggetti del contesto, puoi introdurli facilmente utilizzando Spring come nel caso precedente. Il framework registra automaticamente i bean relativi al contesto nel contenitore Spring. Ad esempio, nel frammento di codice seguente, i diversi oggetti del contesto sono stati collegati automaticamente. Non è richiesta nessun'altra configurazione Spring degli oggetti del contesto.

```
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();
   }
}
```

Se vuoi configurare gli oggetti del contesto nell'implementazione del flusso di lavoro tramite la configurazione Spring XML, utilizza i nomi di bean dichiarati nella classe `WorkflowScopeBeanNames` del pacchetto com.amazonaws.services.simpleworkflow.flow.spring. Per esempio:

```
<!-- 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>
```

In alternativa, puoi introdurre un `DecisionContextProvider` nel bean di implementazione del flusso di lavoro e utilizzarlo per creare il contesto. Può essere utile se vuoi fornire implementazioni personalizzate del provider e del contesto.

Introdurre le risorse nelle attività

Puoi creare come istanze e configurare implementazioni di attività utilizzando un contenitore di inversione di controllo (Inversion of Control, IoC) e introdurre facilmente risorse, come le connessioni

di database, dichiarandole come proprietà della classe di implementazione delle attività. Queste risorse verranno in genere assegnate come singleton. Ricorda che le implementazioni di attività sono chiamate dal lavoratore su più thread. Di conseguenza, l'accesso alle risorse condivise deve essere sincronizzato.

JUnit Integrazione

Il framework fornisce JUnit estensioni e implementazioni di test degli oggetti di contesto, come un orologio di test, che è possibile utilizzare per scrivere ed eseguire test unitari. JUnit Con queste estensioni, puoi testare localmente e inline l'implementazione del flusso di lavoro.

Scrivere un semplice unit test

Per scrivere test per il flusso di lavoro, utilizza la classe `WorkflowTest` nel pacchetto com.amazonaws.services.simpleworkflow.flow.junit. Questa classe è un' JUnit `MethodRule`implementazione specifica del framework ed esegue il codice del flusso di lavoro localmente, chiamando le attività in linea anziché tramite Amazon SWF. Questo ti dà la flessibilità per eseguire i test con la frequenza che preferisci senza alcun addebito.

Per utilizzare questa classe, dichiara semplicemente un campo di tipo `WorkflowTest` e arricchiscilo con l'annotazione `@Rule`. Prima di eseguire i test, crea un nuovo oggetto `WorkflowTest` e aggiungi ad esso le implementazioni di attività e del flusso di lavoro. Puoi utilizzare la client factory del flusso di lavoro generata per creare un client e avviare un'esecuzione del flusso di lavoro. Il framework fornisce anche un JUnit runner personalizzato, `FlowBlockJUnit4ClassRunner` da utilizzare per i test del flusso di lavoro. Per esempio:

```
@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);
    }
}
```

Puoi anche specificare un elenco separato di task per ciascuna implementazione di attività aggiunta a `WorkflowTest`. Ad esempio, se hai un'implementazione del flusso di lavoro che pianifica attività in elenchi di task specifici dell'host, puoi registrare l'attività nell'elenco di task di ciascun host:

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

Tieni presente che il codice in `@Test` è asincrono. Devi quindi utilizzare il client di flusso di lavoro asincrono per avviare un'esecuzione. Per verificare i risultati dei test, viene anche fornita una classe di aiuto `AsyncAssert`. Questa classe ti permette di attendere che le promesse siano pronte prima di verificare i risultati. In questo esempio, attendiamo che sia pronto il risultato dell'esecuzione del flusso di lavoro prima di verificare l'output del test.

Se utilizzi Spring, si può usare la classe `SpringWorkflowTest` invece di quella `WorkflowTest`. `SpringWorkflowTest` fornisce proprietà che puoi utilizzare per configurare facilmente le implementazioni di attività e di flusso di lavoro tramite la configurazione di Spring. Esattamente

come per i lavoratori compatibili con Spring, devi utilizzare `WorkflowScope` per configurare i bean di implementazione del flusso di lavoro. In questo modo siamo sicuri che venga creato un nuovo bean di implementazione del flusso di lavoro per ogni task di decisione. Assicurati di configurare questi bean con l'impostazione scoped-proxy proxy-target-class impostata su. `false` Consulta la sezione Integrazione di Spring per maggiori dettagli. La configurazione Spring di esempio mostrata nella sezione Integrazione di Spring può essere modificata per testare il flusso di lavoro utilizzando `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>
```

Implementazioni di attività fittizie

Durante i test puoi usare implementazioni di attività reali, ma se vuoi eseguire unit test solo della logica del flusso di lavoro, puoi simulare le attività. Questo avviene fornendo un'implementazione fittizia dell'interfaccia delle attività alla classe `WorkflowTest`. Per esempio:

```
@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);
    }
}
```

In alternativa, puoi fornire un'implementazione fittizia del client delle attività e introdurla nell'implementazione del flusso di lavoro.

Testare gli oggetti contesto

Se l'implementazione del flusso di lavoro dipende dagli oggetti del contesto del framework, ad esempio, non `DecisionContext` è necessario fare nulla di speciale per testare tali flussi di lavoro. Quando viene eseguito un test tramite `WorkflowTest`, questo introduce automaticamente oggetti contesto di test. Quando l'implementazione del flusso di lavoro accede agli oggetti di contesto, ad esempio utilizzando`DecisionContextProviderImpl`, otterrà l'implementazione di test. Puoi manipolare questi oggetti contesto di test nel codice di test (metodo `@Test`) per creare casi interessanti di test. Ad esempio, se il flusso di lavoro crea un timer, puoi attivarlo chiamando il metodo `clockAdvanceSeconds` nella classe `WorkflowTest` per muovere l'orologio in avanti. Puoi anche accelerare l'orologio per attivare i timer in anticipo rispetto al normale utilizzando la proprietà `ClockAccelerationCoefficient` su `WorkflowTest`. Ad esempio, se il flusso di lavoro crea un timer per un ora, puoi impostare `ClockAccelerationCoefficient` su 60 per attivare il timer in un minuto. Per impostazione predefinita, `ClockAccelerationCoefficient` è impostato su 1.

Per ulteriori dettagli sui pacchetti com.amazonaws.services.simpleworkflow.flow.test e com.amazonaws.services.simpleworkflow.flow.junit, consulta la documentazione AWS SDK per Java .

Gestione errori

Argomenti

Il costrutto `try/catch/finally` in Java semplifica la gestione degli errori ed è quindi utilizzato diffusamente. Consente di associare gestori di errori a un blocco di codice. Internamente, ciò avviene aggiungendo ulteriori metadati sui gestori di errori allo stack di chiamate. Quando viene generata un'eccezione, il runtime cerca un gestore di errori associato nello stack di chiamate e lo richiama; se non lo trova, propaga l'eccezione fino alla catena di chiamate.

Questo processo è appropriato per il codice sincrono, ma la gestione degli errori in programmi distribuiti e asincroni è più complesso. Poiché una chiamata asincrona ritorna immediatamente, il chiamante non è presente nello stack di chiamate quando viene eseguito il codice asincrono. Ciò significa che le eccezioni non gestite nel codice asincrono non possono essere gestite dal chiamante nel modo usuale. In genere, le eccezioni generate nel codice asincrono sono gestite passando lo stato di errore a un callback che viene passato a un metodo asincrono. Se in alternativa si utilizza `Future<?>`, viene restituito un errore quando tenti di accedervi. Questo processo non è ideale in quanto il codice che riceve l'eccezione (il callback o il codice che utilizza `Future<?>`) non dispone del contesto della chiamata originale e può non essere in grado di gestire l'eccezione in modo adeguato. Inoltre, in un sistema asincrono distribuito in cui i componenti sono eseguiti simultaneamente, possono verificarsi più errori contemporaneamente. Questi errori possono essere di tipo e gravità differenti e devono essere gestiti in modo appropriato.

Anche la pulizia delle risorse dopo una chiamata asincrona risulta alquanto complessa. A differenza del codice sincrono, non è possibile utilizzarlo try/catch/finally nel codice chiamante per ripulire le risorse perché il lavoro iniziato nel blocco try potrebbe essere ancora in corso quando viene eseguito il blocco finally.

Il framework fornisce un meccanismo che rende la gestione degli errori nel codice asincrono distribuito simile e quasi altrettanto semplice di quella di 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();
    }
  };
}
```

Il funzionamento della classe `TryCatchFinally` e delle relative varianti, ovvero `TryFinally` e `TryCatch`, è simile a quello dei blocchi Java `try`/`catch`/`finally`. Tale classe consente di associare i gestori di eccezioni a blocchi di codice di flusso di lavoro che possono essere eseguiti come task asincroni e remoti. Il metodo `doTry()` è equivalente, a livello di logica, al blocco `try`. Il framework esegue automaticamente il codice in `doTry()`. Un elenco di oggetti `Promise` può essere passato al costruttore di `TryCatchFinally`. Il metodo `doTry` sarà eseguito quanto tutti gli oggetti `Promise` passati al costruttore diventano pronti. Se un'eccezione viene generata dal codice richiamato in modo asincrono da `doTry()`, tutto il lavoro in sospeso in `doTry()` viene annullato e `doCatch()` viene chiamato per gestire l'eccezione. Ad esempio, nell'elenco qui sopra, se `downloadImage` genera un'eccezione, `createThumbnail` e `uploadImage` verranno annullati.

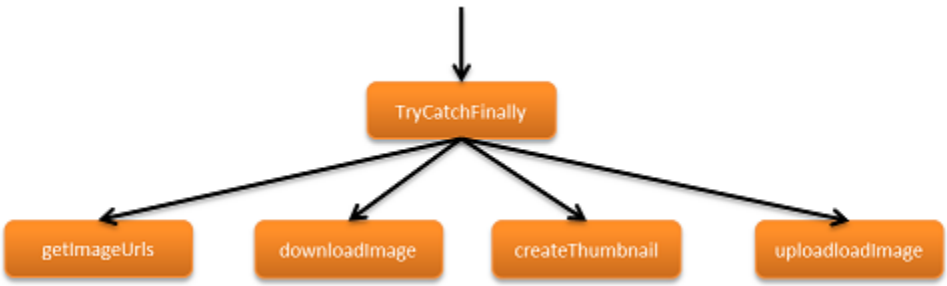
Infine, `doFinally()` viene chiamato quando tutto il lavoro asincrono risulta terminato (completato, non riuscito o annullato). Questo metodo può essere utilizzato per la pulizia delle risorse. Puoi inoltre nidificare queste classi in base alle esigenze aziendali.

Quando un'eccezione è restituita in `doCatch()`, il framework fornisce uno stack di chiamate logiche che include chiamate asincrone e remote. Ciò può rivelarsi utile per il debug, soprattutto se hai dei metodi asincroni che chiamano altri metodi asincroni. Ad esempio, un'eccezione da downloadImage genererà un'eccezione come quella riportata di seguito:

```
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 Semantica

L'esecuzione di un programma AWS Flow Framework per Java può essere visualizzata come un albero di rami in esecuzione simultanea. Una chiamata a un metodo asincrono, a un'attività e a `TryCatchFinally` crea un nuovo ramo in tale struttura. Ad esempio, il flusso di lavoro di elaborazione di immagini può essere rappresentato dalla struttura ad albero illustrata di seguito.



Un errore in un ramo dell'esecuzione comporterà la rimozione di quel ramo, proprio come un'eccezione provoca la rimozione dello stack di chiamate in un programma Java. La rimozione risale lungo il ramo di esecuzione fino a che l'errore viene gestito o viene raggiunta la radice della struttura ad albero, nel qual caso l'esecuzione di flusso di lavoro viene terminata.

Il framework segnala gli errori che si verificano durante l'elaborazione di task come eccezioni. Associa i gestori di eccezioni (metodi `doCatch()`) definiti in `TryCatchFinally` a tutti i task creati dal codice nel metodo `doTry()` corrispondente. Se un'attività fallisce, ad esempio a causa di un

timeout o di un'eccezione non gestita, verrà sollevata l'eccezione appropriata e verrà invocata la corrispondente per gestirla. `doCatch()` A tal fine, il framework collabora con Amazon SWF per propagare gli errori remoti e li resuscita come eccezioni nel contesto del chiamante.

Annullamento

Quando si verifica un'eccezione nel codice sincrono, il controllo passa direttamente al blocco `catch`, ignorando il codice rimanente nel blocco `try`. Per esempio:

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

In questo codice, se `b()` genera un'eccezione, `c()` non viene mai richiamato. Facciamo un raffronto con un flusso di lavoro:

```
new TryCatch() {

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

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

In questo caso, le chiamate a `activityA`, `activityB` e `activityC` hanno esito positivo e comportano la creazione di tre task che vengono eseguiti in modo asincrono. Supponiamo che successivamente il task per `activityB` restituisca un errore. Questo errore viene registrato nella cronologia da Amazon SWF. Per gestirlo, il framework dapprima tenterà di annullare tutti gli altri task

originati nell'ambito dello stesso `doTry()`; in questo caso, `activityA` e `activityC`. Quanto tutti i task risultano terminati (annullati, non riusciti o completati), il metodo `doCatch()` appropriato verrà richiamato per gestire l'errore.

A differenza dell'esempio sincrono, dove `c()` non è mai stato eseguito, `activityC` è stato richiamato e un task è stato pianificato per l'esecuzione. Di conseguenza, il framework effettuerà un tentativo per annullarlo, ma non è garantito che tale operazione riesca. L'annullamento non è certo in quanto l'attività può essere già stata completata, può ignorare la richiesta di annullamento o può non riuscire a causa di un errore. Il framework garantisce tuttavia che la chiamata del metodo `doCatch()` verrà effettuata solo dopo il completamento di tutti i task avviati dal metodo `doTry()` corrispondente. Garantisce inoltre la chiamata di `doFinally()` solo dopo il completamento di tutti i task avviati da `doTry()` e `doCatch()`. Se, ad esempio, le attività dell'esempio precedente dipendono l'una dall'altra, ad esempio `activityB` dipende da `activityA` e `activityC` da`activityB`, l'annullamento `activityC` sarà immediato perché non è programmato in Amazon SWF fino al `activityB` completamento:

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

Heartbeat dell'attività

Il meccanismo di cancellazione cooperativa di AWS Flow Framework for Java consente di annullare senza problemi le attività in corso. Quando si avvia l'annullamento, i task bloccati o in attesa di essere assegnati a un lavoratore vengono annullati automaticamente. Se, tuttavia, un task è già assegnato a un lavoratore, il framework richiederà all'attività di annullarlo. L'implementazione di attività deve gestire in modo esplicito queste richieste di annullamento. Ciò viene eseguito mediante la segnalazione dell'heartbeat dell'attività.

La segnalazione dell'heartbeat consente all'implementazione di attività di comunicare l'avanzamento di un task di attività in corso, il che è utile per il monitoraggio, e all'attività di verificare l'esistenza di richieste di annullamento. Il metodo `recordActivityHeartbeat` genera un'eccezione `CancellationException` se un annullamento è stato richiesto. L'implementazione di attività può rilevare questa eccezione e agire sulla richiesta di annullamento oppure può ignorare la richiesta non tenendo conto dell'eccezione. Per soddisfare la richiesta di cancellazione, l'attività deve eseguire l'eventuale pulizia desiderata e quindi generare di nuovo `CancellationException`. Quando questa eccezione viene generata a partire da un'implementazione di attività, il framework registra che il task di attività è stato completato con lo stato annullato.

L'esempio seguente mostra un'attività che scarica ed elabora immagini. L'attività genera l'heartbeat dopo l'elaborazione di ogni immagine e se viene richiesto l'annullamento, esegue la pulizia e genera di nuovo l'eccezione per confermare l'annullamento.

```
@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;
        }
    }
}
```

La segnalazione dell'heartbeat dell'attività non è necessaria, ma è consigliata se l'attività è a esecuzione prolungata o se esegue operazioni dispendiose che intendi annullare in condizioni di errore. Devi chiamare `heartbeatActivityTask` periodicamente a partire dall'implementazione di attività.

In caso di timeout dell'attività, verrà generata l'eccezione `ActivityTaskTimedOutException` e `getDetails` sull'oggetto eccezione restituirà i dati passati all'ultima chiamata a `heartbeatActivityTask` riuscita per il task di attività corrispondente. L'implementazione di flusso

di lavoro può utilizzare queste informazioni per determinare l'avanzamento prima del timeout del task di attività.

> **Note**
>
> Non è consigliabile eseguire il battito cardiaco troppo frequentemente perché Amazon SWF può limitare le richieste di heartbeat. Consulta la Amazon Simple Workflow Service Developer Guide per conoscere i limiti imposti da Amazon SWF.

Annullamento esplicito di un task

Oltre alle condizioni di errore, vi sono altri casi in cui puoi annullare esplicitamente un task. Ad esempio, è possibile che un'attività per l'elaborazione di pagamenti mediante una carta di credito debba essere annullata se l'utente annulla l'ordine. Il framework ti consente di annullare esplicitamente i task creati nell'ambito di una classe `TryCatchFinally`. Nell'esempio seguente, il task di pagamento viene annullato se si riceve un segnale durante l'elaborazione del pagamento.

```
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);
        }
    }
}
```

Ricezione di notifiche relative a task annullati

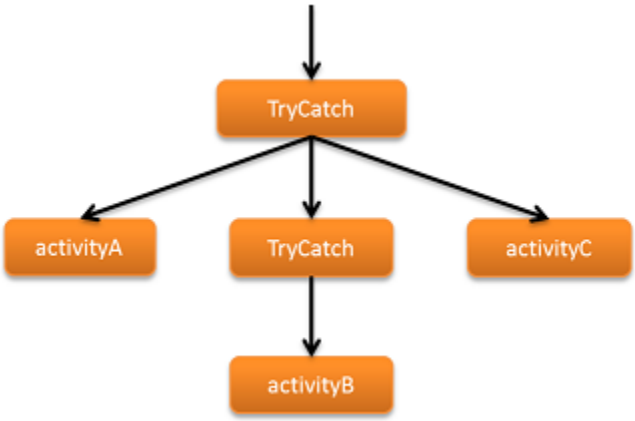
Se un task viene completato quando lo stato è annullato, il framework informa la logica di flusso di lavoro generando un'eccezione `CancellationException`. Se un'attività viene completata quando lo stato è annullata, un record viene creato nella cronologia e il framework chiama il metodo `doCatch()` appropriato con un'eccezione `CancellationException`. Come mostrato nell'esempio precedente, quando il task di elaborazione del pagamento viene annullato, il workflow riceve un'eccezione `CancellationException`.

Un'eccezione `CancellationException` non gestita viene propagata nel ramo di esecuzione come avviene con qualsiasi altra eccezione. Tuttavia, il metodo `doCatch()` riceverà l'eccezione `CancellationException` solo se non vi sono altre eccezioni nell'ambito, in quanto la priorità delle altre eccezioni è superiore a quella dell'annullamento.

Annidato TryCatchFinally

Puoi nidificare la classe `TryCatchFinally` in funzione delle tue esigenze. Poiché ognuno `TryCatchFinally` crea un nuovo ramo nell'albero di esecuzione, è possibile creare ambiti annidati. Le eccezioni nell'ambito padre comporteranno tentativi di annullamento di tutti i task avviati dalle classi `TryCatchFinally` nidificate nell'ambito. Tuttavia, le eccezioni in una classe `TryCatchFinally` nidificata non vengono propagate automaticamente al padre. Se desideri

propagare un'eccezione da una classe `TryCatchFinally` nidificata alla classe `TryCatchFinally` che la contiene, devi generare di nuovo l'eccezione in `doCatch()`. In altre parole, solo le eccezioni non gestite sono propagate, esattamente come i blocchi Java `try/catch`. Se annulli una classe `TryCatchFinally` nidificata chiamando il metodo Cancel, la classe `TryCatchFinally` nidificata verrà annullata ma non la classe `TryCatchFinally` che la contiene.




```
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);
    }
};
```

Ripetere le attività non andate a buon fine

A volte le attività non vanno a buon fine per ragioni effimere, ad esempio una perdita temporanea della connessione. In altri casi l'attività va a buon fine, quindi il modo corretto di gestire l'errore è spesso quello di ripetere l'attività, anche più volte.

Esiste una serie di strategie per ripetere le attività; la migliore dipende dai dettagli del flusso di lavoro. Tali strategie rientrano in tre categorie di base:

- La retry-until-success strategia continua semplicemente a riprovare l'attività fino al suo completamento.
- La strategia di ripetizione esponenziale aumenta esponenzialmente l'intervallo di tempo tra i tentativi fino al completamento dell'attività o fino a quando il processo raggiunge un punto di arresto specifico, come un numero massimo di tentativi.
- La strategia di ripetizione personalizzata decide se o come ripetere l'attività dopo ciascun tentativo non andato a buon fine.

Le sezioni seguenti descrivono come implementare queste strategie. I lavoratori del flusso di lavoro di esempio utilizzano tutti una singola attività, `unreliableActivity`, che esegue casualmente una delle seguenti operazioni:

- Viene completata immediatamente
- Non va a buon fine intenzionalmente superando il valore di timeout
- Non va a buon fine intenzionalmente generando `IllegalStateException`

Retry-Until-Success Strategia

La strategia più semplice è quella di ripetere l'attività ogni volta che ha esito negativo fino al buon esito. Il modello di base è:

1. Implementare una classe nidificata `TryCatch` o `TryCatchFinally` nel metodo del punto di ingresso del flusso di lavoro.
2. Eseguire l'attività in `doTry`.
3. Se l'attività non va a buon fine, il framework chiama `doCatch`, che esegue nuovamente il metodo del punto di ingresso.
4. Ripetere le fasi 2 e 3 fino al completamento con esito positivo dell'attività.

Il seguente flusso di lavoro implementa la retry-until-success strategia. L'interfaccia del flusso di lavoro è implementata in `RetryActivityRecipeWorkflow` e ha un metodo, `runUnreliableActivityTillSuccess`, che è il punto di ingresso del flusso di lavoro. Il lavoratore del flusso di lavoro viene implementato in `RetryActivityRecipeWorkflowImpl`, nel seguente modo:

```
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();
        }
    }
```

```
}
```

Il flusso di lavoro funziona come segue:

1. `runUnreliableActivityTillSuccess` crea un oggetto `Settable<Boolean>` denominato `retryActivity` che viene usato per indicare se l'attività non è riuscita e deve essere ritentata. `Settable<T>` è derivato da `Promise<T>` e funziona allo stesso modo ma il valore dell'oggetto `Settable<T>` viene impostato manualmente.
2. `runUnreliableActivityTillSuccess` implementa una classe annidata anonima `TryCatch` per gestire le eccezioni generate dall'attività `unreliableActivity`. Per ulteriori discussioni su come gestire le eccezioni generate da un codice asincrono, consulta Gestione errori.
3. `doTry` esegue l'attività `unreliableActivity`, che restituisce un oggetto `Promise<Void>` di nome `activityRanSuccessfully`.
4. `doTry` chiama il metodo asincrono `setRetryActivityToFalse`, che ha due parametri:
   - `activityRanSuccessfully` accetta l'oggetto `Promise<Void>` restituito dall'attività `unreliableActivity`.
   - `retryActivity` accetta l'oggetto `retryActivity`.

   Se `unreliableActivity` viene completato, `activityRanSuccessfully` diventa pronto e `setRetryActivityToFalse` imposta `retryActivity` su false. In caso contrario, `activityRanSuccessfully` non diventa mai pronto e `setRetryActivityToFalse` non viene eseguito.
5. Se `unreliableActivity` genera un'eccezione, il framework chiama `doCatch` e lo trasferisce all'oggetto dell'eccezione. `doCatch` imposta `retryActivity` su true.
6. `runUnreliableActivityTillSuccess` chiama il metodo asincrono `restartRunUnreliableActivityTillSuccess` e lo trasferisce all'oggetto `retryActivity`. Poiché `retryActivity` è un tipo `Promise<T>`, `restartRunUnreliableActivityTillSuccess` ritarda l'esecuzione fin quando `retryActivity` è pronto, il che si verifica dopo il completamento di `TryCatch`.
7. Quando `retryActivity` è pronto, `restartRunUnreliableActivityTillSuccess` estrae il valore.
   - Se il valore è `false`, il nuovo tentativo è andato a buon fine. `restartRunUnreliableActivityTillSuccess` non è operativo e la sequenza di ripetizione termina.

- Se il valore è true, il nuovo tentativo non è andato a buon fine. `restartRunUnreliableActivityTillSuccess` chiama `runUnreliableActivityTillSuccess` per eseguire nuovamente l'attività.

8. Si ripetono le fasi 1-7 fino al completamento di `unreliableActivity`.

> **Note**
>
> `doCatch` non gestisce l'eccezione; imposta semplicemente l'oggetto `retryActivity` su true per indicare l'esito negativo dell'attività. La ripetizione è gestita dal metodo asincrono `restartRunUnreliableActivityTillSuccess`, che ritarda l'esecuzione fino al completamento di `TryCatch`. Il motivo di questo approccio è che se riprovi un'attività in `doCatch` non puoi annullarla. Ripetere l'attività in `restartRunUnreliableActivityTillSuccess` ti permette di eseguire attività annullabili.

Strategia di ripetizione esponenziale

Con la strategia di ripetizione esponenziale, il framework esegue nuovamente un'attività non andata a buon fine dopo un periodo di tempo specifico, N secondi. Se il tentativo ha esito negativo, il framework esegue nuovamente l'attività dopo 2N secondi, 4N secondi e così via. Poiché il tempo di attesa può essere lungo, in genere i tentativi si arrestano a un certo punto invece che proseguire all'infinito.

Il framework prevede tre modi per implementare una strategia di ripetizione esponenziale:

- L'annotazione `@ExponentialRetry` è l'approccio più semplice, ma devi impostare le opzioni di configurazione della ripetizione al momento della compilazione.
- La classe `RetryDecorator` ti permette di impostare la configurazione della ripetizione in fase di runtime e di modificarla in base alle necessità.
- La classe `AsyncRetryingExecutor` ti permette di impostare la configurazione della ripetizione in fase di runtime e di modificarla in base alle necessità. Inoltre, il framework chiama un metodo `AsyncRunnable.run` implementato dall'utente per eseguire ogni tentativo di ripetizione.

Tutti gli approcci supportano le seguenti opzioni di configurazione, in cui i valori di tempo sono espressi in secondi:

- Il tempo di attesa per la ripetizione iniziale.
- Il coefficiente di backoff, che viene utilizzato per calcolare gli intervalli di ripetizione, nel modo seguente:

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

  Il valore predefinito è 2.0.
- Il numero massimo di tentativi di ripetizione. Il valore predefinito è illimitato.
- L'intervallo massimo di ripetizione. Il valore predefinito è illimitato.
- Il tempo di scadenza. I tentativi si arrestano quando la durata totale del processo supera questo valore. Il valore predefinito è illimitato.
- Le eccezioni che attivano il processo di ripetizione. Per impostazione predefinita, tutte le eccezioni attivano il processo di ripetizione.
- Le eccezioni che non attivano tentativi di ripetizione. Per impostazione predefinita, non è esclusa alcuna eccezione.

Le sezioni seguenti descrivono i vari modi in cui è possibile implementare una strategia di ripetizione esponenziale.

Riprova esponenziale con @ ExponentialRetry

Il modo più semplice per implementare una strategia di ripetizione esponenziale per un'attività è applicare un'annotazione `@ExponentialRetry` all'attività nella definizione dell'interfaccia. Se l'attività non va a buon fine, il framework gestisce automaticamente il processo di ripetizione in base ai valori opzionali specificati. Il modello di base è:

1. Applica `@ExponentialRetry` alle attività in questione e specifica la configurazione di ripetizione.
2. Se l'attività annotata non va a buon fine, il framework la recupera automaticamente secondo la configurazione specificata dagli argomenti dell'annotazione.

Il lavoratore del flusso di lavoro `ExponentialRetryAnnotationWorkflow` implementa la strategia di ripetizione esponenziale utilizzando un'annotazione `@ExponentialRetry`. Utilizza un'attività `unreliableActivity` la cui definizione dell'interfaccia è implementata in `ExponentialRetryAnnotationActivities`, nel modo seguente:

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

Le opzioni di `@ExponentialRetry` specificano la seguente strategia:

- Ripeti sono se l'attività genera `IllegalStateException`.
- Utilizza un tempo di attesa iniziale di 5 secondi.
- Non più di 5 tentativi di ripetizione.

L'interfaccia del flusso di lavoro è implementata in `RetryWorkflow` e ha un metodo, `process`, che è il punto di ingresso del flusso di lavoro. Il lavoratore del flusso di lavoro viene implementato in `ExponentialRetryAnnotationWorkflowImpl`, nel seguente modo:

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

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

Il flusso di lavoro funziona come segue:

1. `process` esegue il metodo asincrono `handleUnreliableActivity`.
2. `handleUnreliableActivity` esegue l'attività `unreliableActivity`.

Se l'attività non va a buon fine generando `IllegalStateException`, il framework esegue automaticamente la strategia di ripetizione specificata in `ExponentialRetryAnnotationActivities`.

Riprova esponenziale con la classe RetryDecorator

`@ExponentialRetry` è semplice da usare. Tuttavia, la configurazione è statica e impostata al momento della compilazione, in modo che il framework utilizzi la stessa strategia di ripetizione ogni volta che l'attività non va a buon fine. Puoi implementare una strategia di ripetizione esponenziale più flessibile utilizzando la classe `RetryDecorator`, che ti permette di specificare la configurazione in fase di runtime e di modificarla in base alle necessità. Il modello di base è:

1. Crea e configura un oggetto `ExponentialRetryPolicy` che specifichi la configurazione della ripetizione.
2. Crea un oggetto `RetryDecorator` e trasferisci l'oggetto `ExponentialRetryPolicy` della Fase 1 al costruttore.
3. Applica l'oggetto decorator all'attività trasferendo il nome della classe del client di attività al metodo decorato dell'oggetto `RetryDecorator`.
4. Esegui l'attività.

Se l'attività non va a buon fine, il framework la ripete secondo la configurazione dell'oggetto `ExponentialRetryPolicy`. Puoi modificare la configurazione della ripetizione in base alla necessità cambiando l'oggetto.

> **Note**
>
> L'annotazione `@ExponentialRetry` e la classe `RetryDecorator` sono reciprocamente esclusive. Non puoi utilizzare `RetryDecorator` per sovrascrivere dinamicamente una policy di ripetizione specificata da un'annotazione `@ExponentialRetry`.

La seguente implementazione del flusso di lavoro mostra come utilizzare la classe `RetryDecorator` per implementare una strategia di ripetizione esponenziale. Utilizza un'attività `unreliableActivity` priva dell'annotazione `@ExponentialRetry`. L'interfaccia del flusso di lavoro è implementata in `RetryWorkflow` e ha un metodo, `process`, che è il punto di ingresso del flusso di lavoro. Il lavoratore del flusso di lavoro viene implementato in `DecoratorRetryWorkflowImpl`, nel seguente modo:

```
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();
  }
}
```

Il flusso di lavoro funziona come segue:

1. `process` crea e configura un oggetto `ExponentialRetryPolicy` nel seguente modo:
   - Trasferendo al costruttore l'intervallo di ripetizione iniziale.
   - Chiamando il metodo `withMaximumAttempts` dell'oggetto per impostare il numero massimo di tentativi a 5. `ExponentialRetryPolicy` espone altri oggetti `with` che è possibile usare per specificare altre opzioni di configurazione.
2. `process` crea un oggetto `RetryDecorator` con nome `retryDecorator` e trasferisce l'oggetto `ExponentialRetryPolicy` della Fase 1 al costruttore.
3. `process` applica l'elemento decorator all'attività chiamando il metodo `retryDecorator.decorate` e trasferendolo al nome della classe del client di attività.
4. `handleUnreliableActivity` esegue l'attività.

Se l'attività non va a buon fine, il framework la ripete secondo la configurazione specificata nella Fase 1.

> **Note**
>
> Molti dei metodi `with` della classe `ExponentialRetryPolicy` hanno un metodo corrispondente `set` che puoi chiamare per modificare l'opzione di

configurazione corrispondente in qualsiasi momento: `setBackoffCoefficient`, `setMaximumAttempts`, `setMaximumRetryIntervalSeconds` e `setMaximumRetryExpirationIntervalSeconds`.

Riprova esponenziale con la classe AsyncRetryingExecutor

La classe `RetryDecorator` offre più flessibilità nella configurazione del processo di ripetizione rispetto a `@ExponentialRetry`, ma il framework esegue comunque automaticamente i tentativi di ripetizione, in base alla attuale configurazione dell'oggetto `ExponentialRetryPolicy`.
Un approccio più flessibile prevede l'utilizzo della classe `AsyncRetryingExecutor`. Oltre a permetterti di configurare il processo di ripetizione in fase di runtime, il framework chiama un metodo `AsyncRunnable.run` implementato dall'utente per eseguire ogni tentativo di ripetizione invece che eseguire semplicemente l'attività.

Il modello di base è:

1. Crea e configura un oggetto `ExponentialRetryPolicy` per specificare la configurazione della ripetizione.
2. Crea un oggetto `AsyncRetryingExecutor` e trasferiscigli l'oggetto `ExponentialRetryPolicy` e un'istanza dell'orologio del flusso di lavoro.
3. Implementa una classe annidata anonima `TryCatch` o `TryCatchFinally`.
4. Implementa una classe anonima `AsyncRunnable` e sovrascrivi il metodo `run` per implementare il codice personalizzato per eseguire l'attività.
5. Sovrascrivi `doTry` per chiamare il metodo `execute` dell'oggetto `AsyncRetryingExecutor` e trasferirlo alla classe `AsyncRunnable` dalla fase 4. L'oggetto `AsyncRetryingExecutor` chiama `AsyncRunnable.run` per eseguire l'attività.
6. Se l'attività non va a buon fine, l'oggetto `AsyncRetryingExecutor` chiama nuovamente il metodo `AsyncRunnable.run` secondo la policy di ripetizione specificata nella Fase 1.

Il flusso di lavoro seguente mostra come utilizzare la classe `AsyncRetryingExecutor` per implementare una strategia di ripetizione esponenziale. Utilizza la stessa attività `unreliableActivity` del flusso di lavoro `DecoratorRetryWorkflow` discusso in precedenza. L'interfaccia del flusso di lavoro è implementata in `RetryWorkflow` e ha un metodo, `process`, che è il punto di ingresso del flusso di lavoro. Il lavoratore del flusso di lavoro viene implementato in `AsyncExecutorRetryWorkflowImpl`, nel seguente modo:

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

Il flusso di lavoro funziona come segue:

1. `process` chiama il metodo `handleUnreliableActivity` e lo trasferisce alle impostazioni della configurazione.
2. `handleUnreliableActivity` utilizza le impostazioni di configurazione della Fase 1 per creare un oggetto `ExponentialRetryPolicy`, l'oggetto `retryPolicy`.

3. `handleUnreliableActivity` crea un oggetto `AsyncRetryExecutor`, `executor` e trasferisce l'oggetto `ExponentialRetryPolicy` della Fase 2 e un'istanza dell'orologio del flusso di lavoro al costruttore
4. `handleUnreliableActivity` implementa una classe annidata anonima `TryCatch` e sovrascrive i metodi `doTry` e `doCatch` per eseguire i tentativi di ripetizione e gestire le eventuali eccezioni.
5. `doTry` crea una classe anonima `AsyncRunnable` e sovrascrive il metodo `run` per implementare il codice personalizzato per eseguire `unreliableActivity`. Per semplicità, `run` esegue semplicemente l'attività, ma puoi implementare approcci più sofisticati in base alle necessità.
6. `doTry` chiama `executor.execute` e lo trasferisce all'oggetto `AsyncRunnable`. `execute` chiama il metodo `AsyncRunnable` dell'oggetto `run` per eseguire l'attività.
7. Se l'attività non va a buon fine, l'esecutore chiama di nuovo `run` in base alla configurazione dell'oggetto `retryPolicy`.

Per ulteriori discussioni su come utilizzare la classe `TryCatch` per gestire gli errori, consulta AWS Flow Framework per le eccezioni Java.

Strategia di ripetizione personalizzata

L'approccio più flessibile per riprovare le attività non riuscite è una strategia personalizzata, che richiama ricorsivamente un metodo asincrono che esegue il tentativo di nuovo tentativo, proprio come la strategia. retry-until-success Tuttavia, invece che rieseguire semplicemente l'attività, implementi una logica personalizzata che decide se e come eseguire i successivi tentativi di ripetizione. Il modello di base è:

1. Crea un oggetto di stato `Settable<T>`, che viene utilizzato per indicare se l'attività non è andata a buon fine.
2. Implementa una classe annidata `TryCatch` o `TryCatchFinally`.
3. `doTry` esegue l'attività.
4. Se l'attività non va a buon fine, `doCatch` imposta l'oggetto di stato per indicare che l'attività ha avuto esito negativo.
5. Chiama un metodo asincrono di gestione dell'errore e trasferiscilo all'oggetto di stato. Il metodo ritarda l'esecuzione fino al completamento di `TryCatch` o `TryCatchFinally`.
6. Il metodo di gestione dell'errore decide se e quando ripetere l'attività.

Il flusso di lavoro seguente mostra come implementare una strategia di ripetizione personalizzata. Utilizza la stessa attività `unreliableActivity` dei flussi di lavoro `DecoratorRetryWorkflow` e `AsyncExecutorRetryWorkflow`. L'interfaccia del flusso di lavoro è implementata in `RetryWorkflow` e ha un metodo, `process`, che è il punto di ingresso del flusso di lavoro. Il lavoratore del flusso di lavoro viene implementato in `CustomLogicRetryWorkflowImpl`, nel seguente modo:

```
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;
  }
}
```

Il flusso di lavoro funziona come segue:

1. `process` chiama il metodo asincrono `callActivityWithRetry`.
2. `callActivityWithRetry` crea un errore di oggetto `Settable<Throwable>` denominato che viene utilizzato per indicare se l'attività non è andata a buon fine. `Settable<T>` deriva da `Promise<T>` e funziona quasi allo stesso modo, ma il valore dell'oggetto `Settable<T>` è impostato manualmente.
3. `callActivityWithRetry` implementa una classe annidata anonima `TryCatchFinally` per gestire le eccezioni generate da `unreliableActivity`. Per ulteriori discussioni su come gestire le eccezioni generate da un codice asincrono, consulta AWS Flow Framework per le eccezioni Java.
4. `doTry` esegue `unreliableActivity`.
5. Se `unreliableActivity` genera un'eccezione, il framework chiama `doCatch` e lo trasferisce all'oggetto dell'eccezione. `doCatch` imposta `failure` sull'oggetto dell'eccezione, il che indica che l'attività non è andata a buon fine e mette l'oggetto in stato di pronto.
6. `doFinally` verifica se `failure` è pronto, che sarà vero solo se `failure` è stato impostato da `doCatch`.
   - Se è pronto, non fa nulla. `failure` `doFinally`
   - Se `failure` non è pronto, l'attività viene completata e `doFinally` imposta l'errore su `null`.
7. `callActivityWithRetry` chiama il metodo asincrono `retryOnFailure` e vi trasferisce l'errore. Poiché l'errore è un tipo `Settable<T>`, `callActivityWithRetry` ritarda l'esecuzione fin quando l'errore è pronto, il che si verifica dopo il completamento di `TryCatchFinally`.
8. `retryOnFailure` riceve il valore dall'errore.
   - Se l'errore è impostato su null, il tentativo di ripetizione è andato a buon fine. `retryOnFailure` non fa alcunché, il che termina il processo di ripetizione.
   - Se l'errore è impostato su un oggetto di eccezione e `shouldRetry` restituisce il valore true, `retryOnFailure` chiama `callActivityWithRetry` per riprovare l'attività.

     `shouldRetry` implementa una logica personalizzata per decidere se ripetere un'attività dall'esito negativo. Per semplicità, `shouldRetry` restituisce sempre il valore `true` e `retryOnFailure` esegue immediatamente l'attività, ma puoi implementare una logica più sofisticata in base alle necessità.
9. I passaggi da 2 a 8 si ripetono fino al `unreliableActivity` completamento o alla `shouldRetry` decisione di interrompere il processo.

> **Note**
>
> `doCatch` non gestisce il processo di ripetizione; imposta semplicemente l'errore per indicare l'esito negativo dell'attività. Il processo di ripetizione è gestito dal metodo asincrono `retryOnFailure`, che ritarda l'esecuzione fino al completamento di `TryCatch`. Il motivo di questo approccio è che se riprovi un'attività in `doCatch` non puoi annullarla. Ripetere l'attività in `retryOnFailure` ti permette di eseguire attività annullabili.

Task Daemon

Il AWS Flow Framework for Java consente di contrassegnare determinate attività come`daemon`. Questa funzionalità ti permette di creare task che effettuano un lavoro di background che deve essere annullato quando tutti gli altri lavori sono stati eseguiti. Ad esempio, un task di monitoraggio dello stato deve essere annullato quando il resto del flusso di lavoro è completato. Puoi farlo impostando il contrassegno `daemon` su un metodo asincrono o un'istanza di `TryCatchFinally`. Nell'esempio seguente, il metodo asincrono `monitorHealth()` è contrassegnato come `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();
  }
}
```

Nell'esempio riportato sopra, quando `doUsefulWorkActivity` viene completato, `monitoringHealth` viene automaticamente annullato. Questa operazione annulla l'intero ramo di esecuzione radicato nel metodo asincrono. La semantica dell'annullamento è identica a quella di `TryCatchFinally`. Analogamente, puoi contrassegnare un daemon `TryCatchFinally` passando un contrassegno Boolean al costruttore.

```
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
            }
        };
    }
}
```

Un'operazione daemon avviata all'interno di a `TryCatchFinally` è limitata al contesto in cui è stata creata, ovvero sarà limitata ai metodi, o. `doTry()` `doCatch()` `doFinally()` Nel seguente esempio il metodo asincrono startMonitoring viene contrassegnato come daemon e chiamato da `doTry()`. Il task creato verrà annullato non appena gli altri task (`doUsefulWorkActivity` in questo caso) avviati entro `doTry()` saranno completati.

```
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 per Java Replay Behavior

Questo argomento presenta alcuni esempi relativi al comportamento di riproduzione, in base a quanto descritto nella sezione Che cos'è AWS Flow Framework per Java?. Gli esempi forniti riguardano la riproduzione sincrona e a quella asincrona.

Esempio 1: riproduzione sincrona

Per un esempio di come funziona la replay in un flusso di lavoro sincrono, modificate le implementazioni del flusso di HelloWorldWorkflowlavoro e delle attività aggiungendo `println` chiamate all'interno delle rispettive implementazioni, come segue:

```
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);
    }
}
```

Per dettagli sul codice, consulta HelloWorldWorkflow Applicazione. Quanto segue è una versione modificata dell'output, con commenti che indicano l'avvio di ogni episodio di riproduzione.

```
//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
```

Il processo di riproduzione in questo esempio è il seguente:

- Il primo episodio pianifica il task di attività `getName`, che non ha dipendenze.
- Il secondo episodio pianifica il task di attività `getGreeting`, che dipende da `getName`.

- Il terzo episodio pianifica il task di attività `say`, che dipende da `getGreeting`.
- L'episodio finale non pianifica altri task e non trova alcuna attività non completata, di conseguenza l'esecuzione di flusso di lavoro risulta completata.

> **Note**
>
> I tre metodi di client di attività vengono chiamati una volta per ogni episodio. Tuttavia, solo una di queste chiamate genera un task di attività, quindi ogni task viene eseguito una sola volta.

Esempio 2: riproduzione asincrona

Come per l'esempio di riproduzione sincrona, puoi modificare HelloWorldWorkflowAsync Applicazione per osservare il funzionamento della riproduzione asincrona. Viene generato il seguente output:

```
//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 utilizza tre episodi di replay perché ci sono solo due attività. L'attività `getGreeting` è stata sostituita dal metodo di flusso di lavoro asincrono getGreeting, che, quando completato, non avvia un episodio di riproduzione.

Il primo episodio non chiama `getGreeting` poiché dipende dal completamento dell'attività name. Tuttavia, dopo il completamento di getName, la riproduzione chiama getGreeting una volta per ogni episodio successivo.

Vedi anche

- AWS Flow Framework Concetti di base: esecuzione distribuita

Best practice

Utilizza queste best practice per sfruttare al meglio le funzionalità AWS Flow Framework di Java.

Argomenti

Apportare modifiche al codice del decisore: la funzione Versioni multiple e gli Indicatori di caratteristiche

Questa sezione mostra come evitare modifiche non retrocompatibili a un decisore tramite due metodi:

- la funzione Versioni multiple fornisce una soluzione di base.
- la funzione Versioni multiple con gli Indicatori di caratteristiche è uno sviluppo della funzione Versioni multiple: non vengono introdotte nuove versioni del flusso di lavoro e non c'è bisogno di un nuovo codice per aggiornare la versione.

Prima di provare queste soluzioni, prendi familiarità con la sezione Scenario di esempio, che spiega le cause e gli effetti delle modifiche non retrocompatibili al codice del decisore.

Il processo di riproduzione e le modifiche del codice

Quando un decisore AWS Flow Framework per Java esegue un'attività decisionale, deve prima ricostruire lo stato corrente dell'esecuzione prima di potervi aggiungere passaggi. Il decisore compie questa operazione con un processo chiamato riproduzione.

Il processo di riproduzione riesegue dall'inizio il codice del decisore, esaminando al contempo la cronologia degli eventi che si sono già verificati. Questo esame permette al framework di reagire ai segnali o al completamento di task e di sbloccare gli oggetti `Promise` nel codice.

Quando il framework esegue il codice decisore, assegna un ID a ogni attività pianificata (un'attività, una funzione Lambda, un timer, un flusso di lavoro secondario o un segnale in uscita) incrementando un contatore. Il framework comunica questo ID ad Amazon SWF e lo aggiunge agli eventi della cronologia, ad esempio. `ActivityTaskCompleted`

Affinché il processo di riproduzione vada a buon fine, è importante che il codice del decisore sia deterministico e pianifichi gli stessi task nello stesso ordine per ogni decisione in ogni esecuzione del flusso di lavoro. Se non rispetti questo requisito, il framework potrebbe, ad esempio, non riuscire a far corrispondere l'ID di un evento `ActivityTaskCompleted` con un oggetto `Promise` esistente.

Scenario di esempio

Esiste una classe di modifiche del codice che è considerata non retrocompatibile. Queste modifiche includono gli aggiornamenti che modificano il numero, il tipo o l'ordine dei task pianificati. Considera il seguente esempio:

Scrivi un codice del decisore per pianificare due task di timer. Avvii un'esecuzione ed esegui una decisione. Di conseguenza, vengono pianificate due attività con timer, con IDs 1 e. 2

Se aggiorni il codice del decisore per pianificare un solo timer prima che venga eseguita la decisione successiva, nel prossimo task di decisione il framework non riuscirà a riprodurre il secondo evento `TimerFired`, perché l'ID 2 non corrisponde a nessun task di timer prodotto dal codice.

Struttura dello scenario

La struttura seguente mostra le fasi di questo scenario. L'obiettivo finale dello scenario è quello di effettuare la migrazione a un sistema che pianifichi solo un timer ma che non comprometta le esecuzioni avviate prima della migrazione.

1. La versione iniziale del decisore
   a. Scrivi il decisore.
   b. Avvia il decisore.
   c. Il decisore pianifica due timer.
   d. Il decisore avvia cinque esecuzioni.
   e. Arresta il decisore.
2. Una modifica del decisore non retrocompatibile
   a. Modifica il decisore.
   b. Avvia il decisore.
   c. Il decisore pianifica un timer.
   d. Il decisore avvia cinque esecuzioni.

La seguente sezione include esempi di un codice Java che mostrano come implementare questo scenario. Gli esempi di codice nella sezione Soluzioni mostrano vari modi per correggere le modifiche non retrocompatibili.

> **Note**
>
> Puoi utilizzare la versione più recente di AWS SDK per Java per eseguire questo codice.

Codice comune

Il seguente codice Java non cambia tra gli esempi di questo scenario.

`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;
    }

}
```

Scrivere il codice iniziale del decisore

Di seguito è riportato il codice Java iniziale del decisore. Viene registrato come versione 1 e pianifica due task di timer da cinque secondi.

`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);
        }

    }
}
```

Simulazione di una modifica non retrocompatibile

Il seguente codice Java modificato del decisore è un buon esempio di modifica non retrocompatibile. Il codice è ancora registrato come versione 1, ma pianifica solo un timer.

`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);
        }

    }
}
```

Il seguente codice Java ti permette di simulare il problema di apportare modifiche non retrocompatibili eseguendo il decisore modificato.

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();
    }

}
```

Quando esegui il programma, le tre esecuzioni che non vanno a buon fine sono quelle avviate secondo la versione iniziale del decisore e proseguite dopo la migrazione.

Soluzioni

Puoi utilizzare le seguenti soluzioni per evitare le modifiche non retrocompatibili. Per ulteriori informazioni, consulta Apportare modifiche al codice del decisore e Scenario di esempio.

Uso della funzione Versioni multiple

In questa soluzione, copi il decisore su una nuova classe, lo modifichi e lo registri in una nuova versione del flusso di lavoro.

`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);
        }

    }

}
```

Nel codice Java aggiornato, il secondo lavoratore del decisore esegue entrambe le versioni del flusso di lavoro, permettendo alle esecuzioni in transito di operare indipendentemente dalle modifiche alla versione 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();
    }

}
```

Quando esegui il programma, tutte le esecuzioni hanno esito positivo.

Utilizzo degli Indicatori di caratteristiche

Un'altra soluzione per i problemi di mancata retrocompatibilità è ramificare il codice per supportare due implementazioni nella stessa classe basata sui dati dell'input invece che sulle versioni del flusso di lavoro.

Seguendo questo approccio, ogni volta che introduci modifiche sensibili aggiungi campi agli oggetti dell'input o ne modifichi i campi esistenti. Per le esecuzioni avviate prima della migrazione, l'oggetto dell'input sarà privo di campo (o avrà un valore diverso). In questo modo non sei obbligato ad aumentare il numero della versione.

> **Note**
>
> Se aggiungi nuovi campi, verifica che il processo di deserializzazione JSON sia retrocompatibile. Gli oggetti serializzati prima dell'introduzione del campo devono comunque essere deserializzati dopo la migrazione. Poiché JSON imposta un valore `null` ogni volta che manca un campo, utilizza sempre tipi "boxed" (`Boolean` invece di `boolean`) e gestisci i casi in cui valore è `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);
            }
        }

    }
}
```

Nel codice Java aggiornato, il codice per entrambe le versioni del flusso di lavoro è comunque registrato per la versione 1. Tuttavia, dopo la migrazione, le nuove esecuzioni vengono avviate con il campo `skipSecondTimer` dei dati dell'input impostato su `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();
    }

}
```

Quando esegui il programma, tutte le esecuzioni hanno esito positivo.

Suggerimenti per la risoluzione dei problemi e il debug per Java AWS Flow Framework

Argomenti

Questa sezione descrive alcune insidie comuni che potresti incontrare durante lo sviluppo di flussi di lavoro utilizzando for Java. AWS Flow Framework Fornisce inoltre alcuni suggerimenti su come diagnosticare i problemi ed eseguirne il debug.

Errori di compilazione

Se utilizzi l'opzione di tessitura in fase di compilazione di AspectJ, è possibile che si verifichino errori di compilazione in cui il compilatore non riesce a trovare le classi client generate per il flusso di lavoro e le attività. La causa probabile di tali errori di compilazione è che il generatore AspectJ ha ignorato i client generati durante la compilazione. Puoi risolvere questo problema rimuovendo AspectJ dal progetto e riattivandolo. Nota che dovrai procedere in tal modo ogni volta che le interfacce di flusso di lavoro o di attività cambiano. A causa di questo problema, ti consigliamo di utilizzare l'opzione di tessitura in fase di caricamento. Per ulteriori informazioni, consulta la sezione Configurazione di AWS Flow Framework per Java.

Errore di risorsa sconosciuto

Amazon SWF restituisce un errore di risorsa sconosciuto quando tenti di eseguire un'operazione su una risorsa che non è disponibile. Le cause più comuni di questo errore sono:

- Configuri un lavoratore con un dominio inesistente. Per risolvere questo problema, registra innanzitutto il dominio utilizzando la console Amazon SWF o l'API del servizio Amazon SWF.
- Tenti di creare task di esecuzione di flusso di lavoro o di attività che non sono stati registrati. Ciò può accadere se cerchi di creare l'esecuzione di flusso di lavoro prima che i lavoratori vengano eseguiti. Poiché i worker registrano i propri tipi quando vengono eseguiti per la prima volta, è necessario eseguirli almeno una volta prima di tentare di avviare le esecuzioni (o registrare manualmente i tipi utilizzando la console o l'API del servizio). Nota che dopo la registrazione dei tipi, puoi creare le esecuzioni anche se non vi sono lavoratori in esecuzione.
- Un lavoratore tenta di completare un task di cui si è già verificato il timeout. Ad esempio, se un lavoratore impiega troppo tempo per elaborare un'attività e supera un timeout, riceverà un UnknownResource errore quando tenta di completare o fallire l'attività. I AWS Flow Framework lavoratori continueranno a sondare Amazon SWF ed elaborare attività aggiuntive. ma è comunque consigliabile modificare il timeout. A questo proposito, devi registrate una nuova versione del tipo di attività.

Eccezioni quando si chiama get () su una promessa

A differenza di Java Future, `Promise` è un costrutto non bloccante e la chiamata di `get()` su un argomento `Promise` non ancora pronto genererà un'eccezione anziché un blocco. Il modo corretto di usare a `Promise` è passarlo a un metodo asincrono (o a un'attività) e accedere al suo valore nel metodo asincrono. AWS Flow Framework for Java garantisce che un metodo asincrono venga chiamato solo quando tutti gli argomenti passati ad esso sono pronti. `Promise` Se ritieni che il tuo codice sia corretto o se ti imbatti in questo mentre esegui uno degli AWS Flow Framework esempi, probabilmente è dovuto al fatto che AspectJ non è configurato correttamente. Per ulteriori informazioni, consulta la sezione Configurazione di AWS Flow Framework per Java.

Flussi di lavoro non deterministici

Come descritto nella sezione Non determinismo, l'implementazione del tuo flusso di lavoro deve essere deterministica. Alcuni errori comuni che possono portare al non determinismo sono l'uso dell'orologio di sistema, l'uso di numeri casuali e la generazione di. GUIDs Poiché questi costrutti possono restituire valori diversi in momenti diversi, il flusso di controllo del flusso di lavoro può seguire percorsi diversi ogni volta che viene eseguito (consulta le sezioni AWS Flow Framework Concetti di base: esecuzione distribuita e Comprensione di un task in AWS Flow Framework for Java per i dettagli). Se il framework rileva una condizione di non determinismo durante l'esecuzione del flusso di lavoro, verrà generata un'eccezione.

Problemi dovuti al controllo delle versioni

Quando si implementa una nuova versione del flusso di lavoro o dell'attività, ad esempio quando si aggiunge una nuova funzionalità, è necessario aumentare la versione del tipo utilizzando l'annotazione appropriata:, o. `@Workflow` `@Activites` `@Activity` In genere, quando vengono distribuite nuove versioni di un flusso di lavoro, alcune esecuzioni della versione esistente sono già in corso. Di conseguenza, devi assicurarti che i task siano trasmessi ai lavoratori con la versione appropriata del flusso di lavoro e delle attività. A questo proposito, devi utilizzare un set di elenchi di task differente per ogni versione. Ad esempio, puoi aggiungere il numero di versione al nome dell'elenco di task. In questo modo, i task appartenenti a differenti versioni del flusso di lavoro e delle attività sono assegnati ai lavoratori appropriati.

Risoluzione dei problemi e debug dell'esecuzione di un flusso di lavoro

Il primo passaggio per la risoluzione dei problemi di esecuzione di un flusso di lavoro consiste nell'utilizzare la console Amazon SWF per esaminare la cronologia del flusso di lavoro. La cronologia del flusso di lavoro è un record completo e attendibile di tutti gli eventi che hanno modificato lo stato dell'esecuzione di flusso di lavoro. Questa cronologia è gestita da Amazon SWF ed è preziosa per la diagnosi dei problemi. La console Amazon SWF ti consente di cercare esecuzioni di flussi di lavoro e approfondire i singoli eventi della cronologia.

AWS Flow Framework fornisce una `WorkflowReplayer` classe che puoi usare per riprodurre localmente l'esecuzione di un flusso di lavoro ed eseguirne il debug. Utilizzando questa classe, è possibile eseguire il debug di esecuzioni di workflow chiuse e in esecuzione. `WorkflowReplayer`si affida alla cronologia memorizzata in Amazon SWF per eseguire la riproduzione. Puoi indirizzarlo all'esecuzione di un flusso di lavoro nel tuo account Amazon SWF o fornirgli gli eventi della cronologia (ad esempio, puoi recuperare la cronologia da Amazon SWF e serializzarla localmente per un uso successivo). La riproduzione di un'esecuzione di flusso di lavoro con `WorkflowReplayer` non ha alcun impatto sull'esecuzione in corso nel tuo account. L'intera riproduzione viene eseguita sul client. Puoi eseguire il debug del flusso di lavoro, creare punti di interruzione ed eseguire istruzioni utilizzando gli strumenti di debug abituali. Se utilizzi Eclipse, prendi in considerazione l'aggiunta di filtri Step ai pacchetti di filtri. AWS Flow Framework

Ad esempio, il seguente frammento di codice può essere utilizzato per riprodurre un'esecuzione di flusso di lavoro:

```
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 consente inoltre di ottenere un dump asincrono dei thread dell'esecuzione del flusso di lavoro. Questo dump fornisce gli stack di chiamate di tutti i task asincroni aperti. Queste informazioni possono essere utili per determinare quali task nell'esecuzione sono in sospeso e possibilmente bloccati. Esempio:

```
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);
}
```

Attività perse

A volte, è possibile che tu chiuda dei lavoratori e che ne avvii di nuovi in rapida successione per infine scoprire che i task sono recapitati ai vecchi lavoratori. Ciò può avvenire a seguito di condizioni di competizione nel sistema, il quale è ripartito su vari processi. Il problema può verificarsi anche quando esegui unit test in un ciclo ridotto oppure a seguito dell'arresto di un test in Eclipse nel caso in cui i gestori di chiusura non siano chiamati.

Per avere la certezza che il problema è in effetti dovuto al fatto che sono i vecchi lavoratori a ricevere i task, dovresti esaminare la cronologia del flusso di lavoro per determinare quale processo ha ricevuto il task che doveva essere recapitato al nuovo lavoratore. Ad esempio, l'evento `DecisionTaskStarted` nella cronologia contiene l'identità del lavoratore di flusso di lavoro che ha ricevuto il task. L'id utilizzato da Flow Framework ha il formato: {*`processId`*} `@` {*`host name`*}. Ad esempio, di seguito sono riportati i dettagli dell'`DecisionTaskStarted`evento nella console Amazon SWF per un'esecuzione di esempio:

Timestamp di evento	Mon Feb 20 11:52:40 GMT-800 2012
Identità	2276 @ip -0A6C1 DF5
ID evento pianificato	33

Per evitare questa situazione, utilizza elenchi di task differenti per ogni test. Valuta inoltre la possibilità di aggiungere un ritardo tra la chiusura dei vecchi lavoratori e l'avvio dei nuovi.

Errore di convalida dovuto a vincoli di lunghezza dei parametri API

Amazon SWF impone vincoli di lunghezza sui parametri delle API. Riceverai un `HTTP 400` errore se l'implementazione del flusso di lavoro o dell'attività supera i vincoli. Ad esempio, quando si chiama `recordActivityHeartbeat` on `ActivityExecutionContext` per inviare un battito cardiaco per un'attività in corso, la stringa non deve superare i 2048 caratteri.

Un altro scenario comune è quando un'attività fallisce a causa di un'eccezione. Il framework segnala un errore di attività ad Amazon SWF chiamando RespondActivityTaskFailedcon l'eccezione serializzata come dettagli. La chiamata API segnalerà un errore 400 se l'eccezione serializzata ha una lunghezza superiore a 32.768 byte. Per mitigare questa situazione, è possibile troncare il messaggio di eccezione o le cause per renderle conformi al vincolo di lunghezza.

AWS Flow Framework per Java Reference

Argomenti

AWS Flow Framework per Java Annotations

Argomenti

@Activities

Questa annotazione può essere usata su un'interfaccia per dichiarare un set di tipi di attività. Ciascun metodo in un'interfaccia che abbia questa annotazione rappresenta un tipo di attività. Un'interfaccia non può avere contemporaneamente l'annotazione `@Workflow` e quella `@Activities`-

Su questa annotazione possono essere specificati i seguenti parametri:

`activityNamePrefix`

Specifica il prefisso del nome dei tipi di attività dichiarati nell'interfaccia. Se impostato su una stringa vuota (valore predefinito), il nome dell'interfaccia seguito da '.' viene utilizzato come prefisso.

`version`

Specifica la versione predefinita dei tipi di attività dichiarati nell'interfaccia. Il valore predefinito è `1.0`.

`dataConverter`

Speciifica il tipo di serializing/deserializing dati `DataConverter` da utilizzare per la creazione di attività di questo tipo di attività e i relativi risultati. Impostato come predefinito su `NullDataConverter`, che indica che deve essere utilizzato `JsonDataConverter`.

@Activity

Questa annotazione può essere usata sui metodi in un'interfaccia che abbia l'annotazione `@Activities`.

Su questa annotazione possono essere specificati i seguenti parametri:

`name`

Specifica il nome del tipo di attività. Il valore predefinito è una stringa vuota, che indica che per stabilire il nome del tipo di attività, (che è in formato {prefisso}{nome}) occorre utilizzare il prefisso predefinito e il nome del metodo dell'attività. Ricorda che quando specifichi un nome in un'annotazione `@Activity`, il framework non vi aggiunge automaticamente un prefisso. Sei libero di usare il tuo schema di denominazione.

`version`

Specifica la versione del tipo di attività. Sovrascrive la versione predefinita specificata nell'annotazione `@Activities` nell'interfaccia che la contiene. L'impostazione predefinita è una stringa vuota.

@ActivityRegistrationOptions

Specifica le opzioni di registrazione di un tipo di attività. Questa annotazione può essere usata su un'interfaccia annotata con `@Activities` o sui metodi all'interno. Se specificata in entrambe le posizioni, prevale l'annotazione usata sul metodo.

Su questa annotazione possono essere specificati i seguenti parametri:

`defaultTasklist`

Speciifica l'elenco di attività predefinito da registrare con Amazon SWF per questo tipo di attività. Il valore predefinito può essere sovrascritto quando si chiama il metodo dell'attività sul client generato utilizzando il parametro `ActivitySchedulingOptions`. Impostato come predefinito su `USE_WORKER_TASK_LIST`. Questo è un valore speciale che indica che va utilizzato l'elenco di task usato dal lavoratore che esegue la registrazione.

`defaultTaskScheduleToStartTimeoutSeconds`

Speciifica le informazioni defaultTaskSchedule ToStartTimeout registrate con Amazon SWF per questo tipo di attività. Questo è il tempo massimo di attesa di un task di questo tipo di attività prima che venga assegnato a un lavoratore. Per ulteriori dettagli, consulta il riferimento all'API di Amazon Simple Workflow Service.

`defaultTaskHeartbeatTimeoutSeconds`

Speciifica le informazioni `defaultTaskHeartbeatTimeout` registrate con Amazon SWF per questo tipo di attività. In questo periodo i lavoratori di attività devono fornire messaggi di heartbeat; in caso contrario, il task scade. Impostato come predefinito a -1, un valore speciale che indica che il timeout deve essere disattivato. Per ulteriori dettagli, consulta il riferimento all'API di Amazon Simple Workflow Service.

`defaultTaskStartToCloseTimeoutSeconds`

Speciifica le informazioni defaultTaskStart ToCloseTimeout registrate con Amazon SWF per questo tipo di attività. Il timeout determina il tempo massimo che un lavoratore può impiegare per elaborare un task di attività di questo tipo. Per ulteriori dettagli, consulta il riferimento all'API di Amazon Simple Workflow Service.

`defaultTaskScheduleToCloseTimeoutSeconds`

Speciifica le informazioni `defaultScheduleToCloseTimeout` registrate con Amazon SWF per questo tipo di attività. Questo timeout determina il tempo totale in cui il task può rimanere nello

stato aperto. Impostato come predefinito a -1, un valore speciale che indica che il timeout deve essere disattivato. Per ulteriori dettagli, consulta il riferimento all'API di Amazon Simple Workflow Service.

@Asincrona

Se usata su un metodo nella logica di coordinamento del flusso di lavoro, indica che il metodo deve essere eseguito in modo asincrono. La chiamata al metodo viene restituita immediatamente, ma l'esecuzione in corso accade in modo asincrono quando sono pronti tutti i parametri Promise<> trasferiti ai metodi. I metodi annotati con @Asynchronous devono avere un tipo di restituzione Promise<> o void.

`daemon`

Indica se il task creato per il metodo asincrono deve essere di tipo daemon. `False` per impostazione predefinita.

@Execute

Se usata su un metodo in'un interfaccia annotata con `@Workflow`, identifica il punto di ingresso del flusso di lavoro.

> **Important**
>
> L'annotazione `@Execute` può essere applicata a un solo metodo nell'interfaccia.

Su questa annotazione possono essere specificati i seguenti parametri:

`name`

Specifica il nome del tipo di flusso di lavoro. Se non è impostato, il nome predefinito è {prefisso} {nome}, dove {prefisso} è il nome dell'interfaccia di flussi di lavoro seguito da un '.' e {nome} è il nome del metodo con annotazione `@Execute` nel flusso di lavoro.

`version`

Specifica la versione del tipo di flusso di lavoro.

@ExponentialRetry

se usata su un'attività o su un metodo asincrono, imposta una policy di ripetizione esponenziale nel caso in cui il metodo genera un'eccezione non gestita. Un tentativo di ripetizione viene effettuato dopo un periodo di backoff, calcolato in base all'efficacia del numero dei tentativi.

Su questa annotazione possono essere specificati i seguenti parametri:

`intialRetryIntervalSeconds`

Specifica il tempo di attesa prima del primo tentativo di ripetizione. Il valore non deve essere maggiore di `maximumRetryIntervalSeconds` e di `retryExpirationSeconds`.

`maximumRetryIntervalSeconds`

Specifica il tempo massimo tra i tentativi di ripetizione. Una volta raggiunto, l'intervallo tra i tentativi di ripetizione sarà limitato a questo valore. Impostato come predefinito a -1, il che significa una durata illimitata.

`retryExpirationSeconds`

Specifica il tempo dopo il quale la ripetizione esponenziale si arresta. Impostato come predefinito a -1, il che significa che non c'è scadenza.

`backoffCoefficient`

Specifica il coefficiente utilizzato per calcolare l'intervallo di ripetizione. Consultare Strategia di ripetizione esponenziale.

`maximumAttempts`

Specifica il numero di tentativi dopo il quale la ripetizione esponenziale si arresta. Impostato come predefinito a -1, il che significa che non c'è limite al numero di tentativi di ripetizioni.

`exceptionsToRetry`

Specifica l'elenco dei tipi di eccezione che attivano una ripetizione. L'eccezione non gestita di questi tipi non verrà propagata ulteriormente e il metodo sarà ripetuto dopo l'intervallo di ripetizione calcolato. Per impostazione predefinita, l'elenco contiene `Throwable`.

`excludeExceptions`

Specifica l'elenco dei tipi di eccezione che non attivano una ripetizione. Le eccezioni non gestite di questo tipo possono propagarsi. Per impostazione predefinita, l'elenco è vuoto.

@GetState

Se usata su un metodo in'un interfaccia annotata con `@Workflow`, identifica che il metodo è utilizzato per recuperare lo stato dell'ultima esecuzione del flusso di lavoro. Ci può essere al massimo un metodo con questa annotazione in un'interfaccia con l'annotazione `@Workflow`. I metodi così annotati non devono acquisire parametri e devono avere un tipo di restituzione diverso da `void`.

@ManualActivityCompletion

Questa annotazione può essere utilizzata su un metodo di attività per indicare che il task di attività non deve essere completato alla restituzione del metodo. L'attività non verrà completata automaticamente e dovrà essere completata manualmente direttamente utilizzando l'API Amazon SWF. Questo è utile per casi d'uso in cui il task di attività è delegato a un sistema esterno non automatizzato o che richiede l'intervento umano per il completamento.

@Signal

Se usata su un metodo in'un interfaccia annotata con `@Workflow`, identifica un segnale che può essere ricevuto dalle esecuzioni del tipo di flusso di lavoro dichiarato dall'interfaccia. L'uso di questa annotazione è obbligatorio per definire un metodo di segnale.

Su questa annotazione possono essere specificati i seguenti parametri:

`name`

Specifica la porzione nominale del nome del segnale. Se non è impostato, si usa il nome del metodo.

@SkipRegistration

Se utilizzato su un'interfaccia annotata con l'`@Workflow`annotazione, indica che il tipo di flusso di lavoro non deve essere registrato con Amazon SWF. Una delle annotazioni `@WorkflowRegistrationOptions` e `@SkipRegistrationOptions` devono essere usate su un'interfaccia con annotazione `@Workflow`, ma non entrambe.

@Wait e @ NoWait

Queste annotazioni possono essere utilizzate su un parametro di tipo `Promise<>` per indicare se AWS Flow Framework for Java deve attendere che sia pronto prima di eseguire il metodo. Per

impostazione predefinita, i parametri `Promise<>` trasferiti sui metodi `@Asynchronous` devono diventare pronti prima che si verifichi l'esecuzione del metodo. In alcuni casi, è necessario ignorare questo comportamento predefinito. I parametri `Promise<>` trasferiti sui metodi `@Asynchronous` con l'annotazione `@NoWait` non sono attesi.

I parametri (o le sottoclassi) di raccolta che contengono promesse, come `List<Promise<Int>>`, devono essere arricchiti con l'annotazione `@Wait`. Per impostazione predefinita, il framework non attende i membri di una raccolta.

@Flusso di lavoro

Questa annotazione viene usata su un'interfaccia per dichiarare un tipo di flusso di lavoro. Un'interfaccia decorata con questa annotazione deve contenere esattamente un metodo decorato con l'annotazione @Execute per dichiarare un punto di ingresso del flusso di lavoro.

> **Note**
>
> Un'interfaccia non può avere le annotazioni `@Workflow` e `@Activities` dichiarate simultaneamente; sono reciprocamente esclusive.

Su questa annotazione possono essere specificati i seguenti parametri:

`dataConverter`

Specifica quali `DataConverter` utilizzare nell'invio di richieste alle esecuzioni (e nella ricezione di risultati dalle esecuzioni) di questo tipo di flusso di lavoro.

L'impostazione predefinita è `NullDataConverter` che, a sua volta, ritorna `JsonDataConverter` a elaborare tutti i dati di richiesta e risposta come JavaScript Object Notation (JSON).

Esempio

```
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

Se utilizzato su un'interfaccia annotata con`@Workflow`, fornisce le impostazioni predefinite utilizzate da Amazon SWF per la registrazione del tipo di flusso di lavoro.

> **Note**
>
> O `@WorkflowRegistrationOptions` o `@SkipRegistrationOptions` devono essere usate su un'interfaccia con annotazione `@Workflow`, ma non puoi specificarle entrambe.

Su questa annotazione possono essere specificati i seguenti parametri:

Descrizione

Una descrizione di testo facoltativa del tipo di flusso di lavoro.

`defaultExecutionStartToCloseTimeoutSeconds`

Speciifica il tipo di flusso di lavoro `defaultExecutionStartToCloseTimeout` registrato con Amazon SWF. Il tempo totale che l'esecuzione di un flusso di lavoro di questo tipo può impiegare per il completamento.

Per ulteriori informazioni sui timeout del flusso di lavoro, consulta Tipi di timeout di Amazon SWF.

`defaultTaskStartToCloseTimeoutSeconds`

Speciifica il tipo di flusso di lavoro `defaultTaskStartToCloseTimeout` registrato con Amazon SWF. Specifica il tempo che un solo task di decisione di un'esecuzione di un flusso di lavoro di questo tipo può impiegare per il completamento.

Se non specifichi `defaultTaskStartToCloseTimeout`, per impostazione predefinita sarà di 30 secondi.

Per ulteriori informazioni sui timeout del flusso di lavoro, consulta Tipi di timeout di Amazon SWF.

`defaultTaskList`

L'elenco predefinito di task utilizzato per i task di decisione per le esecuzioni di questo tipo di flusso di lavoro. L'impostazione predefinita qui può essere sovrascritta utilizzando `StartWorkflowOptions` in fase di avvio di un'esecuzione del flusso di lavoro.

Se non specifichi `defaultTaskList`, verrà impostato su `USE_WORKER_TASK_LIST` come impostazione predefinita. Indica che va utilizzato l'elenco di task usato dal lavoratore che esegue la registrazione del flusso di lavoro.

`defaultChildPolicy`

Specifica la policy da utilizzare per i flussi di lavoro figli se un'esecuzione di questo tipo è terminata. Il valore predefinito è `ABANDON`. I valori possibili sono:

- `ABANDON`— Consenti alle esecuzioni secondarie del flusso di lavoro di continuare a funzionare
- `TERMINATE`— Interrompere le esecuzioni dei flussi di lavoro secondari
- `REQUEST_CANCEL`— Richiedere l'annullamento delle esecuzioni dei flussi di lavoro secondari

AWS Flow Framework per le eccezioni Java

Le seguenti eccezioni vengono utilizzate da AWS Flow Framework for Java. In questa sezione viene fornita una panoramica di ogni eccezione. Per ulteriori dettagli, consulta la AWS SDK per Java documentazione delle singole eccezioni.

Argomenti

- ActivityFailureException
- ActivityTaskException
- ActivityTaskFailedException
- ActivityTaskTimedOutException
- ChildWorkflowException
- ChildWorkflowFailedException
- ChildWorkflowTerminatedException
- ChildWorkflowTimedOutException
- DataConverterException
- DecisionException

- ScheduleActivityTaskFailedException
- SignalExternalWorkflowException
- StartChildWorkflowFailedException
- StartTimerFailedException
- TimerException
- WorkflowException

ActivityFailureException

Questa eccezione è utilizzata internamente dal framework per comunicare l'esito negativo di un'attività. Quando un'attività fallisce a causa di un'eccezione non gestita, viene inclusa `ActivityFailureException` e segnalata ad Amazon SWF. Devi gestire questa eccezione solo se utilizzi i punti di estensibilità del lavoratore di attività. Il codice dell'applicazione non dovrà mai gestire questa eccezione.

ActivityTaskException

Questa è la classe di base per le eccezioni di errore dei task di attività: `ScheduleActivityTaskFailedException`, `ActivityTaskFailedException`, `ActivityTaskTimedoutException`. Contiene l'ID di task e il tipo di attività del task non riuscito. Puoi rilevare questa eccezione nella tua implementazione di flusso di lavoro per gestire gli errori nelle attività in modo generico.

ActivityTaskFailedException

Le eccezioni non gestite nelle attività sono restituite all'implementazione di flusso di lavoro generando `ActivityTaskFailedException`. L'eccezione originale può essere recuperata dalla proprietà cause di questa eccezione. L'eccezione fornisce inoltre altre informazioni utili per il debug, come l'identificatore di attività univoco nella cronologia.

Il framework può fornire l'eccezione remota serializzando l'eccezione originale dal lavoratore di attività.

ActivityTaskTimedOutException

Questa eccezione viene generata se un'attività è stata interrotta da Amazon SWF. Ciò può verificarsi se il task di attività non viene assegnato al lavoratore o completato dal lavoratore entro

il periodo di tempo stabilito. Puoi impostare questi timeout per l'attività utilizzando l'annotazione `@ActivityRegistrationOptions` o il parametro `ActivitySchedulingOptions` durante la chiamata del metodo di attività.

ChildWorkflowException

La classe di base per le eccezioni utilizzate per segnalare errori nell'esecuzione di flusso di lavoro figlio. L'eccezione contiene gli ID dell'esecuzione di flusso di lavoro figlio nonché il tipo di flusso di lavoro. Puoi rilevare questa eccezione per gestire gli errori nelle esecuzioni di flusso di lavoro figlio in modo generico.

ChildWorkflowFailedException

Le eccezioni non gestite nei flussi di lavoro figlio sono restituite all'implementazione di flusso di lavoro padre generando `ChildWorkflowFailedException`. L'eccezione originale può essere recuperata dalla proprietà `cause` di questa eccezione. L'eccezione fornisce inoltre altre informazioni utili per il debug, come gli identificatori univoci dell'esecuzione figlio.

ChildWorkflowTerminatedException

Questa eccezione viene generata nell'esecuzione di flusso di lavoro padre per segnalare la terminazione di un'esecuzione di flusso di lavoro figlio. Devi rilevare questa eccezione se intendi gestire la terminazione del flusso di lavoro figlio, ad esempio, per eseguire la pulizia o la compensazione.

ChildWorkflowTimedOutException

Questa eccezione viene generata nell'esecuzione del flusso di lavoro principale per segnalare che l'esecuzione di un flusso di lavoro secondario è stata interrotta e chiusa da Amazon SWF. Devi rilevare questa eccezione se intendi gestire la chiusura forzata del flusso di lavoro figlio, ad esempio per eseguire la pulizia o la compensazione.

DataConverterException

Il framework utilizza il componente `DataConverter` per eseguire il marshalling e l'unmarshalling dei dati trasmessi. Questa eccezione viene generata se `DataConverter` non riesce a eseguire il marshalling o l'unmarshalling dei dati. L'errore potrebbe verificarsi per vari motivi, ad esempio, a seguito di una mancata corrispondenza tra i componenti `DataConverter` utilizzati per eseguire il marshalling e l'unmarshalling dei dati.

DecisionException

Questa è la classe base per le eccezioni che rappresentano la mancata attuazione di una decisione di Amazon SWF. Puoi rilevare questa eccezione per gestire tali eccezioni in modo generico.

ScheduleActivityTaskFailedException

Questa eccezione viene generata se Amazon SWF non riesce a pianificare un'attività. Ciò potrebbe accadere per vari motivi, ad esempio se l'attività è stata dichiarata obsoleta o è stato raggiunto un limite Amazon SWF sul tuo account. La proprietà `failureCause` nell'eccezione specifica la causa esatta dell'errore di pianificazione dell'attività.

SignalExternalWorkflowException

Questa eccezione viene generata se Amazon SWF non riesce a elaborare una richiesta dell'esecuzione del flusso di lavoro per segnalare l'esecuzione di un altro flusso di lavoro. Ciò si verifica se non è stato possibile trovare l'esecuzione del flusso di lavoro di destinazione, ovvero se l'esecuzione del flusso di lavoro specificata non esiste o si trova in uno stato chiuso.

StartChildWorkflowFailedException

Questa eccezione viene generata se Amazon SWF non riesce ad avviare l'esecuzione di un workflow secondario. Ciò può accadere per vari motivi, ad esempio, il tipo di flusso di lavoro secondario specificato è obsoleto o è stato raggiunto un limite Amazon SWF sul tuo account. La proprietà `failureCause` nell'eccezione specifica la causa esatta dell'errore di avvio dell'esecuzione di flusso di lavoro figlio.

StartTimerFailedException

Questa eccezione viene generata se Amazon SWF non riesce ad avviare un timer richiesto dall'esecuzione del flusso di lavoro. Ciò potrebbe accadere se l'ID timer specificato è già in uso o se è stato raggiunto un limite Amazon SWF sul tuo account. La proprietà `failureCause` nell'eccezione specifica la causa esatta dell'errore.

TimerException

Questa è la classe di base per le eccezioni relative ai timer.

WorkflowException

Questa eccezione viene utilizzata internamente dal framework per segnalare errori nell'esecuzione di flusso di lavoro. Devi gestire tale eccezione solo se utilizzi un punto di estensibilità del lavoratore di flusso di lavoro.

AWS Flow Framework per pacchetti Java

Questa sezione fornisce una panoramica dei pacchetti inclusi in Java. AWS Flow Framework Per ulteriori informazioni su ciascun pacchetto, consulta com.amazonaws.services.simpleworkflow.flow nella Guida di riferimento all'API.AWS SDK per Java

com.amazonaws.services.simpleworkflow.flow

Contiene componenti che si integrano con Amazon SWF.

com.amazonaws.services.simpleworkflow.flow.annotations

Contiene le annotazioni utilizzate dal modello di programmazione for Java. AWS Flow Framework

com.amazonaws.services.simpleworkflow.flow.aspectj

Contiene i componenti Java necessari per funzionalità come e. AWS Flow Framework @Asincrona @ExponentialRetry

com.amazonaws.services.simpleworkflow.flow.common

Contiene utilità comuni come costanti definite dal framework.

com.amazonaws.services.simpleworkflow.core

Contiene funzionalità di base come `Task` e `Promise`.

com.amazonaws.services.simpleworkflow.flow.generic

Contiene componenti principali, come client generici, su cui si basano altre funzionalità.

com.amazonaws.services.simpleworkflow.flow.interceptors

Contiene implementazioni degli elementi Decorator forniti dal framework tra cui `RetryDecorator`.

com.amazonaws.services.simpleworkflow.junit

Contiene componenti che forniscono l'integrazione JUnit.

com.amazonaws.services.simpleworkflow.pojo

Contiene classi che implementano definizioni di attività e di flusso di lavoro per il modello di programmazione basato su annotazioni.

com.amazonaws.services.simpleworkflow.flow.spring

Contiene componenti che forniscono l'integrazione Spring.

com.amazonaws.services.simpleworkflow.flow.test

Contiene classi helper, come `TestWorkflowClock`, per gli unit test di implementazioni di flusso di lavoro.

com.amazonaws.services.simpleworkflow.flow.worker

Contiene implementazioni di lavoratori di attività e flusso di lavoro.

Cronologia dei documenti

La tabella seguente descrive le modifiche importanti alla documentazione dall'ultima versione della AWS Flow Framework for Java Developer Guide.

- Versione API: 25-01-2012
- Ultimo aggiornamento della documentazione: 25 giugno 2018

Modifica	Descrizione	Data della modifica
Aggiornamento	Corretto errore nella descrizione `backoffCoefficient` per `@ExponentialRetry` . Consultare @ExponentialRetry.	25 giugno 2018
Aggiornamento	Pulizia degli esempi di codice all'interno di questa guida.	5 giugno 2017
Aggiornamento	Semplificazione e miglioramento dell'organizzazione e dei contenuti della guida.	19 maggio 2017
Aggiornamento	Semplificazione e miglioramento della sezione Apportare modifiche al codice del decisore: la funzione Versioni multiple e gli Indicatori di caratteristiche.	10 aprile 2017
Aggiornamento	Aggiunta della nuova sezione Best practice con una nuova guida alle modifiche del codice del decisore.	3 marzo 2017
Nuova caratteristica	Puoi specificare attività Lambda oltre alle tradizionali attività nei tuoi flussi di lavoro. Per ulteriori informazioni, consulta AWS Lambda Attività di implementazione.	21 luglio 2015
Nuova caratteristica	Amazon SWF include il supporto per l'impostazione della priorità delle attività in un elenco di attività, tentando di fornire le attività con una priorità più alta prima delle attività con priorità inferiore. Per ulteriori informazioni, consulta Impostazione della priorità delle attività in Amazon SWF.	17 dicembre 2014

Modifica	Descrizione	Data della modifica
Aggiornamento	Apporto di aggiornamenti e correzioni.	1 agosto 2013
Aggiornamento	• Apporto di aggiornamenti e correzioni, compresi aggiornamenti delle istruzioni per configurare per Eclipse 4.3 e AWS SDK per Java 1.4.7. • Aggiunta di un nuovo set di tutorial per creare scenari di avvio	28 giugno 2013
Nuova caratteristica	La versione iniziale di AWS Flow Framework per Java.	27 febbraio 2012

Le traduzioni sono generate tramite traduzione automatica. In caso di conflitto tra il contenuto di una traduzione e la versione originale in Inglese, quest'ultima prevarrà.