aws

Guide du développeur

AWS Flow Framework pour Java

Version de l'API 2021-04-28

AWS Flow Framework pour Java: Guide du développeur

Table of Contents

Qu'est-ce que le AWS Flow Framework pour Java ? 1
Présentation de ce guide 1
Démarrage 3
Configuration de l'infrastructure 3
Ajouter le framework de flux avec Maven 4
HelloWorld Demande 4
HelloWorld Mise en œuvre des activités 5
HelloWorld Travailleur du workflow 6
HelloWorld Démarreur de workflow 7
HelloWorldWorkflow Demande 8
HelloWorldWorkflow Travailleur des activités 10
HelloWorldWorkflow Travailleur du workflow 12
HelloWorldWorkflow Mise en œuvre des flux de travail et activités 17
HelloWorldWorkflow Démarreur 21
HelloWorldWorkflowAsyncDemande 26
HelloWorldWorkflowAsync Mise en œuvre des activités 27
HelloWorldWorkflowAsync Mise en œuvre du workflow 28
HelloWorldWorkflowAsyncWorkflow et activités Host and Starter 30
HelloWorldWorkflowDistributed Demande 31
HelloWorldWorkflowParallelDemande 34
HelloWorldWorkflowParallelTravailleur des activités 35
HelloWorldWorkflowParallelTravailleur du workflow 36
HelloWorldWorkflowParallel Workflow et activités Host and Starter 38
Compréhension AWS Flow Framework 39
Structure d'application 39
Rôle de l'exécuteur d'activité 41
Rôle de l'exécuteur de flux de travail 41
Rôle du démarreur de flux de travail 42
Comment Amazon SWF interagit avec votre application 42
Pour en savoir plus 43
Exécution fiable 43
Assurer une communication fiable 43
S'assurer qu'aucun résultat n'est perdu 44
Gestion des composants distribués ayant échoué 45

Exécution distribuée 45
Reproduction des flux de travail 45
Reproduction et méthodes de flux de travail asynchrones 47
Implémentation de reproduction et de flux de travail 47
Listes et exécution de tâches 47
Applications scalables 50
Échange de données entre les activités et les flux de travail 51
Le Promesse <T> Type 51
Convertisseurs de données et regroupement 53
Échange de données entre les applications et les exécutions de flux de travail 53
Types de délai 54
Délais liés au flux de travail et aux tâches de décision 54
Délais des tâches d'activité 56
Comprendre les tâches 58
Tâche 58
Ordre d'exécution 59
Exécution de flux de travail 61
Non-déterminisme 63
Guide de programmation 65
Implémentation des applications de flux de travail 65
Contrats de flux de travail et d'activité 67
Enregistrement des types de flux de travail et d'activité 70
Nom et version de type de flux de travail 71
Nom du signal 71
Nom et version de type de flux d'activité 71
Default Task List 72
Autres options d'enregistrement 72
Clients d'activité et de flux de travail 73
Clients de flux de travail 73
Clients d'activité 82
Options de planification 86
Clients dynamiques 87
Implémentation de flux de travail 89
Contexte décisionnel 90
Exposition de l'état d'exécution 90
Locales de flux de travail 93

Implémentation d'activité 94
Finalisation manuelle des activités 95
Implémentation de tâches Lambda 96
À propos AWS Lambda 97
Avantages et limites de l'utilisation des tâches Lambda 97
Utilisation de tâches Lambda dans vos flux de travail AWS Flow Framework pour Java 98
Voir l' HelloLambda échantillon 102
Exécution de programmes écrits avec le AWS Flow Framework pour Java 103
WorkflowWorker 104
ActivityWorker 104
Modèle de thread d'exécuteur 105
Extensibilité de l'exécuteur 107
Contexte d'exécution 108
Contexte décisionnel 108
Contexte d'exécution d'une activité 111
Exécutions de flux de travail enfant 112
Flux de travail continus 114
Définition de la priorité des tâches 115
Définition d'une priorité de tâche pour les flux de travail 116
Définition d'une priorité de tâche pour les activités 117
DataConverters 118
Transmission des données aux méthodes asynchrones 119
Transmission des collections et des cartes aux méthodes asynchrones 119
Définissable <T> 120
@NoWait 121
Promets- <Vide> 122
AndPromise et OrPromise 122
Testabilité et injection de dépendances 122
Intégration de Spring 123
JUnit Integration 130
Gestion des erreurs 136
TryCatchFinally Sémantique 138
Annulation 139
Imbriqué TryCatchFinally 144
Relance des activités ayant échoué 145
Retry-Until-Success Stratégie 146

Stratégie de nouvelle tentative exponentielle 148
Stratégie de nouvelle tentative personnalisée 156
Tâches démon 158
Comportement de reproduction 160
Exemple 1 : Reproduction synchrone 161
Exemple 2 : Reproduction asynchrone 163
consultez aussi 163
Bonnes pratiques 164
Modifications du code décideur 164
Le processus de reproduction et les modifications de code 164
Exemple de scénario 165
Solutions 172
Résolution des problèmes 177
Erreurs de compilation 177
Défaillance de ressource inconnue 177
Exceptions lors de l'appel à get () sur une promesse 178
Workflows non déterministes 178
Problèmes liés à la gestion des versions 179
Résolution des problèmes et débogage de l'exécution d'un flux de travail 179
Tâches perdues 181
Échec de validation dû à des contraintes de longueur des paramètres de l'API 181
Référence 183
Annotations 183
@Activités 183
@Activité 184
@ActivityRegistrationOptions 184
@Asynchrone 186
@Execute 186
@ExponentialRetry 186
@GetState 187
@ManualActivityCompletion 188
@Signal 188
@SkipRegistration 188
@Wait et @ NoWait 188
@Flux de travail 189
@WorkflowRegistrationOptions 190

Exceptions 191
ActivityFailureException 192
ActivityTaskException 192
ActivityTaskFailedException 192
ActivityTaskTimedOutException 192
ChildWorkflowException 193
ChildWorkflowFailedException 193
ChildWorkflowTerminatedException 193
ChildWorkflowTimedOutException 193
DataConverterException 193
DecisionException 194
ScheduleActivityTaskFailedException 194
SignalExternalWorkflowException 194
StartChildWorkflowFailedException 194
StartTimerFailedException 194
TimerException 194
WorkflowException 195
Packages 195
Historique du document 197
..... cxcix

Qu'est-ce que le AWS Flow Framework pour Java ?

Vous pouvez ainsi vous concentrer sur la AWS Flow Framework mise en œuvre de votre logique de flux de travail. Dans les coulisses, le framework utilise les fonctionnalités de planification, de routage et de gestion des états d'Amazon SWF pour gérer l'exécution de votre flux de travail et le rendre évolutif, fiable et auditable. AWS Flow Framework les flux de travail basés sur la technologie sont hautement simultanés. Les flux de travail peuvent être répartis sur plusieurs composants, qui peuvent être exécutés en tant que processus distincts sur des ordinateurs distincts et être dimensionnés indépendamment. L'application peut continuer à progresser si l'un de ses composants est en cours d'exécution, ce qui la rend très tolérante aux pannes.

Présentation de ce guide

Ce guide contient des informations sur l'installation, la configuration et l'utilisation AWS Flow Framework pour créer des applications Amazon SWF.

Commencer à utiliser le AWS Flow Framework pour Java

Si vous débutez avec le AWS Flow Framework pour Java, lisez la Commencer à utiliser le AWS Flow Framework pour Java section. Il vous expliquera comment télécharger et installer le AWS Flow Framework pour Java, comment configurer votre environnement de développement et vous expliquera un exemple simple de création d'un flux de travail.

Comprendre AWS Flow Framework Java

Présente Amazon SWF et ses AWS Flow Framework concepts de base, décrivant la structure de base d'une AWS Flow Framework application et la manière dont les données sont échangées entre les différentes parties d'un flux de travail distribué.

AWS Flow Framework pour le guide de programmation Java

Ce chapitre fournit des conseils de programmation de base AWS Flow Framework pour le développement d'applications de flux de travail avec Java, notamment comment enregistrer les types d'activité et de flux de travail, implémenter des clients de flux de travail, créer des flux de travail enfants, gérer les erreurs, etc.

Comprendre une tâche dans AWS Flow Framework for Java

Ce chapitre fournit un aperçu plus détaillé du AWS Flow Framework fonctionnement de Java, en vous fournissant des informations supplémentaires sur l'ordre d'exécution des flux de travail asynchrones et une étape logique de l'exécution d'un flux de travail standard.

Conseils de dépannage et de débogage AWS Flow Framework pour Java

Ce chapitre fournit des informations sur les erreurs courantes que vous pouvez utiliser pour dépanner vos flux de travail, ou pour apprendre à éviter les erreurs courantes.

AWS Flow Framework pour Java Reference

Ce chapitre fait référence aux annotations, exceptions et packages que le AWS Flow Framework for Java ajoute au SDK pour Java.

Commencer à utiliser le AWS Flow Framework pour Java

Cette section les présente AWS Flow Framework en vous présentant une série d'exemples d'applications simples qui présentent le modèle de programmation de base et l'API. Les exemples d'applications sont basés sur l'application standard Hello World, utilisée pour présenter le langage C et des langages de programmation associés. Voici une implémentation Java classique de Hello World :

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

Les informations suivantes constituent une brève description des exemples d'applications. Elles incluent le code source complet, de sorte que vous puissiez implémenter et exécuter vous-même les applications. Avant de commencer, vous devez d'abord configurer votre environnement de développement et créer un projet AWS Flow Framework pour Java, comme dansConfiguration du AWS Flow Framework pour Java.

- HelloWorld Demande présente des applications de flux de travail via l'implémentation de Hello World en tant qu'application Java standard, mais structurée comme une application de flux de travail.
- HelloWorldWorkflow Demandeutilise le AWS Flow Framework for Java pour le HelloWorld convertir en un flux de travail Amazon SWF.
- HelloWorldWorkflowAsyncDemande modifie `HelloWorldWorkflow` afin d'utiliser une méthode de flux de travail asynchrone.
- HelloWorldWorkflowDistributed Demande modifie `HelloWorldWorkflowAsync` de sorte que les objets exécuteur de flux de travail et d'activité puissent être exécutés sur des systèmes distincts.
- HelloWorldWorkflowParallelDemande modifie `HelloWorldWorkflow` afin d'exécuter deux activités en parallèle.

Configuration du AWS Flow Framework pour Java

Le AWS Flow Framework pour Java est inclus dans le AWS SDK pour Java. Si vous n'avez pas encore configuré le AWS SDK pour Java, consultez la section Getting Started du guide du AWS SDK

pour Java développeur pour obtenir des informations sur l'installation et la configuration du SDK lui-même.

Ajouter le framework de flux avec Maven

Les outils de génération Amazon SWF sont open source. Pour consulter ou télécharger le code ou pour créer les outils vous-même, consultez le référentiel à l'adresse. https://github.com/aws/aws-swf-build-tools

Amazon fournit des outils de création Amazon SWF dans le référentiel central Maven.

Pour configurer l'infrastructure de flux de travail pour Maven, ajoutez la dépendance suivante au fichier `pom.xml` de votre projet :

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

HelloWorld Demande

Pour présenter la structure des applications Amazon SWF, nous allons créer une application Java qui se comporte comme un flux de travail, mais qui s'exécute localement dans le cadre d'un seul processus. Aucune connexion à Amazon Web Services n'est requise.

> **Note**
>
> L'HelloWorldWorkflowexemple s'appuie sur celui-ci, en se connectant à Amazon SWF pour gérer le flux de travail.

Une application de flux de travail consiste en trois composants de base :

- Un exécuteur d'activité prend en charge un ensemble d'activités, dont chacune est une méthode qui s'exécute indépendamment pour effectuer une tâche particulière.
- Un exécuteur d'activité orchestre l'exécution des activités et gère le flux de données. Il s'agit de la réalisation par programmation d'une topologie de flux de travail, qui est essentiellement

un diagramme de flux qui définit le moment où les différentes activités s'exécutent, qu'elles s'exécutent dans l'ordre ou simultanément, etc.

- Un démarreur de flux de travail lance une instance de flux de travail, appelée exécution, et peut interagir avec elle pendant l'opération.

HelloWorld est implémenté sous la forme de trois classes et de deux interfaces associées, décrites dans les sections suivantes. Avant de commencer, vous devez configurer votre environnement de développement et créer un nouveau projet AWS Java comme décrit dansConfiguration du AWS Flow Framework pour Java. Les packages utilisés pour les procédures suivantes sont tous nommés `helloWorld.`*`XYZ`*. Pour utiliser ces noms, définissez l'attribut `within` dans aop.xml comme suit :

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

Pour l'implémenter HelloWorld, créez un nouveau package Java dans votre projet AWS SDK nommé `helloWorld.HelloWorld` et ajoutez les fichiers suivants :

- Un fichier d'interface nommé `GreeterActivities.java`
- Un fichier de classe nommé `GreeterActivitiesImpl.java` qui implémente l'exécuteur d'activité.
- Un fichier d'interface nommé `GreeterWorkflow.java`.
- Un fichier de classe nommé `GreeterWorkflowImpl.java` qui implémente l'exécuteur de flux de travail.
- Un fichier de classe nommé `GreeterMain.java` qui implémente le démarreur de flux de travail.

Les détails sont présentés dans les sections suivantes et incluent le code complet de chaque composant, que vous pouvez ajouter au fichier approprié.

HelloWorld Mise en œuvre des activités

HelloWorld divise la tâche globale d'impression d'un `"Hello World!"` message d'accueil sur la console en trois tâches, chacune étant exécutée par une méthode d'activité. Les méthode d'activité sont définies dans l'interface `GreeterActivities`, comme suit.

```
public interface GreeterActivities {
```

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

HelloWorld a une implémentation d'activité`GreeterActivitiesImpl`, qui fournit les `GreeterActivities` méthodes indiquées :

```
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);
    }
}
```

Les activités sont indépendantes les unes des autres et peuvent être souvent utilisées par différents flux de travail. Par exemple, n'importe quel flux de travail peut utiliser l'activité `say` pour imprimer une chaîne dans la console. Les flux de travail peuvent également plusieurs implémentations d'activité, chacune exécutant un ensemble différent de tâches.

HelloWorld Travailleur du workflow

Pour imprimer « Hello World ! » sur la console, les tâches d'activité doivent être exécutées en séquence dans le bon ordre avec les données correctes. Le travailleur du HelloWorld flux de travail orchestre l'exécution des activités sur la base d'une topologie de flux de travail linéaire simple, illustrée dans la figure suivante.

Les trois activités s'exécutent séquentiellement, et les flux de données depuis une activité vers la suivante.

Le HelloWorld gestionnaire de flux de travail utilise une seule méthode, le point d'entrée du flux de travail, qui est définie dans l'`GreeterWorkflow`interface, comme suit :

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

La classe `GreeterWorkflowImpl` implémente cette interface, comme suit :

```
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);
   }
}
```

La `greet` méthode implémente la HelloWorld topologie en créant une instance de`GreeterActivitiesImpl`, en appelant chaque méthode d'activité dans le bon ordre et en transmettant les données appropriées à chaque méthode.

HelloWorld Démarreur de workflow

Un démarreur du flux de travail est une application qui lance une exécution de flux de travail, et peut également communiquer avec le flux de travail pendant qu'il s'exécute. La `GreeterMain` classe implémente le démarreur HelloWorld de flux de travail, comme suit :

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

`GreeterMain` crée une instance de `GreeterWorkflowImpl` et appelle `greet` pour lancer l'exécuteur de flux de travail. Exécutez `GreeterMain` en tant qu'application Java et vous devriez voir « Hello World ! » dans la sortie de la console.

HelloWorldWorkflow Demande

Bien que l'HelloWorldexemple de base soit structuré comme un flux de travail, il diffère d'un flux de travail Amazon SWF à plusieurs égards essentiels :

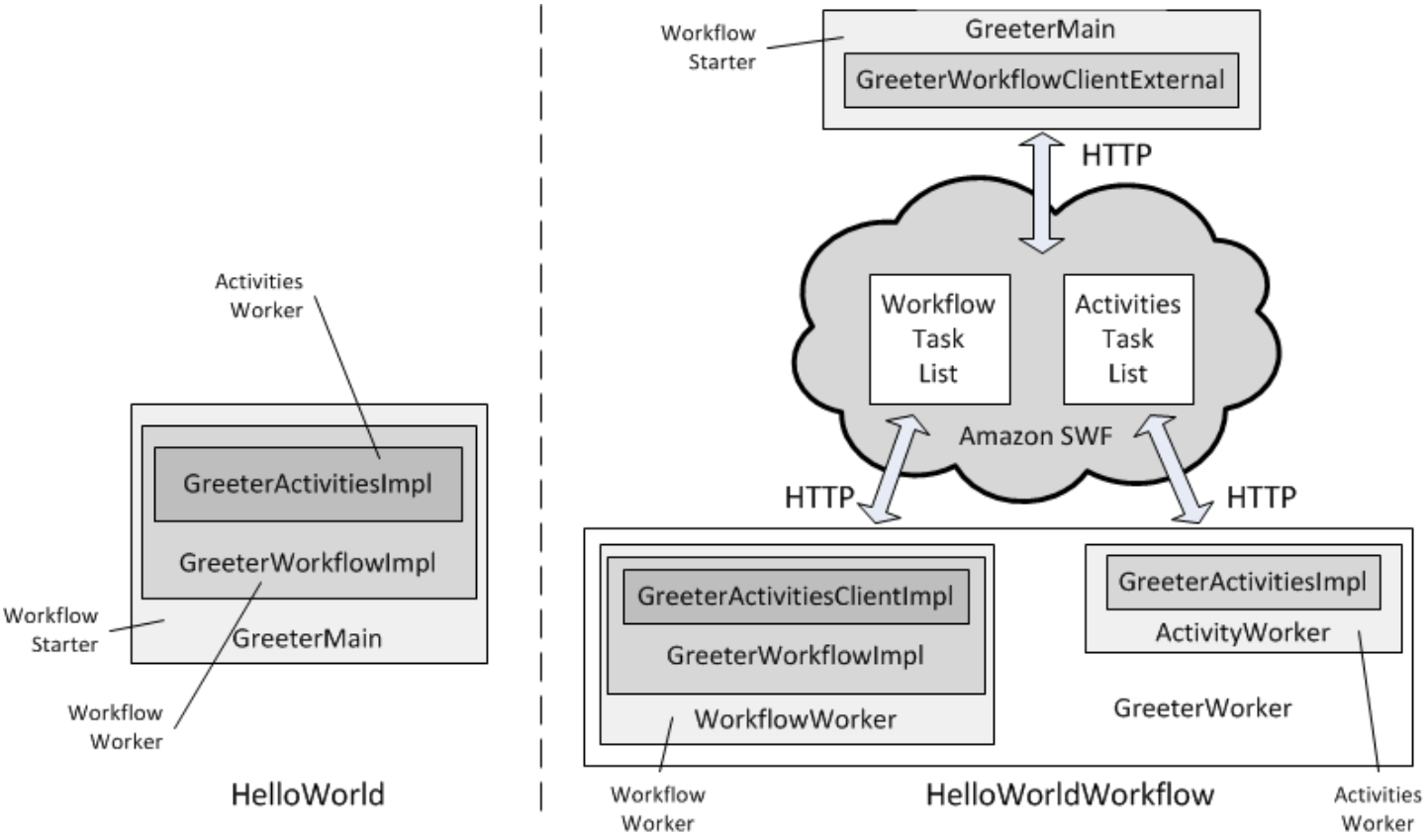
Applications de flux de travail classiques et Amazon SWF

HelloWorld	Flux de travail Amazon SWF
S'exécute localement sous la forme d'un processus unique.	S'exécute sous la forme de plusieurs processus qui peuvent être répartis sur plusieurs systèmes, notamment EC2 des instances Amazon, des centres de données privés, des ordinateurs clients, etc. Ces derniers n'ont pas besoin d'être exécutés sur le même système d'exploitation.
Les activités sont des méthodes synchrones, qui sont bloquées tant qu'elles ne sont pas terminées.	Les activités sont représentées par des méthodes asynchrones, qui renvoient des données immédiatement et qui permettent au flux de travail d'exécuter d'autres tâches en attendant la fin de l'exécution de l'activité.
L'exécuteur de flux de travail interagit avec un exécuteur d'activités en appelant la méthode appropriée.	Les travailleurs du flux de travail interagissent avec les travailleurs des activités à l'aide de requêtes HTTP, Amazon SWF jouant le rôle d'intermédiaire.
Le démarreur de flux de travail interagit avec un objet exécuteur de flux de travail en appelant la méthode appropriée.	Les initiateurs de flux de travail interagissent avec les travailleurs de flux de travail à l'aide de requêtes HTTP, Amazon SWF jouant le rôle d'intermédiaire.

Vous pouvez implémenter une application de flux de travail asynchrone réparti à partir de rien, par exemple, en faisant en sorte que votre objet exécuteur de flux de travail interagisse avec un objet de travail d'activités directement via des appels de services web. Toutefois, vous devez alors implémenter tout le code compliqué nécessaire à la gestion de l'exécution asynchrone de plusieurs

activités, gérer le flux de données, etc. AWS Flow Framework for Java et Amazon SWF s'occupent de tous ces détails, ce qui vous permet de vous concentrer sur la mise en œuvre de la logique métier.

HelloWorldWorkflow est une version modifiée de HelloWorld qui s'exécute en tant que flux de travail Amazon SWF. Le schéma suivant résume le fonctionnement des deux applications.



HelloWorld s'exécute comme un processus unique et le démarreur, le responsable du flux de travail et le responsable des activités interagissent à l'aide d'appels de méthode conventionnels. Avec`HelloWorldWorkflow`, le démarreur, le gestionnaire de flux de travail et le gestionnaire d'activités sont des composants distribués qui interagissent via Amazon SWF à l'aide de requêtes HTTP. Amazon SWF gère l'interaction en tenant à jour des listes de tâches de flux de travail et d'activités, qu'il distribue aux composants respectifs. Cette section décrit le fonctionnement du framework HelloWorldWorkflow.

HelloWorldWorkflow est implémenté à l'aide de l'API AWS Flow Framework for Java, qui gère les détails parfois complexes liés à l'interaction avec Amazon SWF en arrière-plan et simplifie considérablement le processus de développement. Vous pouvez utiliser le même projet que celui pour lequel vous l'avez fait HelloWorld, qui est déjà configuré AWS Flow Framework pour les applications Java. Toutefois, pour exécuter l'application, vous devez configurer un compte Amazon SWF, comme suit :

- Ouvrez un AWS compte, si vous n'en avez pas déjà un, sur Amazon Web Services.
- Attribuez l'ID d'accès et l'ID secret de votre compte aux variables d' AWS_SECRET_KEY environnement AWS_ACCESS_KEY_ID et, respectivement. Il est conseillé de ne pas afficher les valeurs de clé littérale dans votre code. Le stockage de ces clés dans les variables d'environnement vous permet de respecter cette pratique.
- Ouvrez un compte Amazon SWF sur Amazon Simple Workflow Service.
- Connectez-vous au service Amazon SWF AWS Management Console et sélectionnez-le.
- Choisissez Gérer les domaines dans le coin supérieur droit et enregistrez un nouveau domaine Amazon SWF. Un domaine est un conteneur logique pour vos ressources d'application, comme les types de flux de travail et d'activité, et les exécutions de flux de travail. Vous pouvez utiliser n'importe quel nom de domaine pratique, mais les procédures pas à pas utilisent « »helloWorldWalkthrough.

Pour implémenter le HelloWorldWorkflow, créez une copie de HelloWorld. HelloWorld placez le package dans le répertoire de votre projet et nommez-le HelloWorld. HelloWorldWorkflow. Les sections suivantes décrivent comment modifier le HelloWorld code d'origine afin d'utiliser le AWS Flow Framework pour Java et de l'exécuter en tant qu'application de flux de travail Amazon SWF.

HelloWorldWorkflow Travailleur des activités

HelloWorld a mis en œuvre ses activités de travailleur en tant que classe unique. Un outil AWS Flow Framework de travail pour les activités Java comporte trois composants de base :

- Les méthodes d'activité, qui exécutent les tâches réelles, sont définies dans une interface et implémentées dans une classe associée.
- Une ActivityWorkerclasse gère l'interaction entre les méthodes d'activité et Amazon SWF.
- Une application hôte d'activités enregistre et démarre l'exécuteur d'activités, et gère les nettoyages.

Cette section présente les méthodes d'activité ; les deux autres classes seront présentées ultérieurement.

HelloWorldWorkflow définit l'interface des activités dans`GreeterActivities`, comme suit :

```
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);
}
```

Cette interface n'était pas strictement nécessaire pour HelloWorld, mais elle l'est AWS Flow Framework pour une application Java. Notez que la définition de l'interface elle-même n'a pas changé. Cependant, vous devez en appliquer deux AWS Flow Framework pour les annotations Java @ActivityRegistrationOptions et @Activités pour la définition de l'interface. Les annotations fournissent des informations de configuration et indiquent au processeur d'annotations AWS Flow Framework pour Java d'utiliser la définition de l'interface pour générer une classe de clients d'activités, dont il sera question plus loin.

`@ActivityRegistrationOptions`possède plusieurs valeurs nommées qui sont utilisées pour configurer le comportement des activités. HelloWorldWorkflow spécifie deux délais d'expiration :

- `defaultTaskScheduleToStartTimeoutSeconds` indique combien de temps les tâches peuvent rester dans la liste des tâches d'activités ; ce délai est défini sur 300 secondes (5 minutes).
- `defaultTaskStartToCloseTimeoutSeconds` indique le temps maximal dont une activité peut disposer pour exécuter la tâche ; ce délai est défini sur 10 secondes.

Ces délais d'attente permettent de s'assurer que l'activité exécute sa tâche dans un délai raisonnable. Si l'un de ces délais est dépassé, la structure génère une erreur et l'objet exécuteur de flux de travail doit décider de la façon dont le problème sera résolu. Pour obtenir des informations sur la façon de gérer ces erreurs, consultez Gestion des erreurs.

`@Activities` comporte plusieurs valeurs, mais généralement cet élément définit simplement le numéro de version des activités, ce qui vous permet de conserver une trace des différentes générations d'implémentations d'activité. Si vous modifiez une interface d'activité après l'avoir enregistrée auprès d'Amazon SWF, notamment en modifiant les `@ActivityRegistrationOptions` valeurs, vous devez utiliser un nouveau numéro de version.

HelloWorldWorkflow implémente les méthodes `GreeterActivitiesImpl` d'activité comme suit :

```
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);
   }
}
```

Notez que le code est identique à l' HelloWorld implémentation. À la base, une AWS Flow Framework activité n'est qu'une méthode qui exécute du code et renvoie peut-être un résultat. La différence entre une application standard et une application de flux de travail Amazon SWF réside dans la manière dont le flux de travail exécute les activités, dans quel endroit les activités s'exécutent et dans la manière dont les résultats sont renvoyés au gestionnaire du flux de travail.

HelloWorldWorkflow Travailleur du workflow

Un gestionnaire de flux de travail Amazon SWF comporte trois composants de base.

- Une implémentation de flux de travail, qui est une classe exécutant des tâches liés aux flux de travail.
- Une classe client d'activités, qui est à la base un proxy destiné à la classe d'activités et qui est utilisé par une implémentation de flux de travail pour exécuter les méthodes d'activité de façon asynchrone.
- Une WorkflowWorkerclasse qui gère l'interaction entre le flux de travail et Amazon SWF.

Cette section présente l'implémentation de flux de travail et le client d'activités ; la classe `WorkflowWorker` sera présentée ultérieurement.

HelloWorldWorkflow définit l'interface du flux `GreeterWorkflow` de travail comme suit :

```
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();
}
```

Cette interface n'est pas non plus strictement nécessaire HelloWorld mais essentielle AWS Flow Framework pour une application Java. Vous devez en appliquer deux AWS Flow Framework pour les annotations Java @Flux de travail et @WorkflowRegistrationOptions pour la définition de l'interface du flux de travail. Les annotations fournissent des informations de configuration et indiquent également au processeur d'annotations AWS Flow Framework pour Java de générer une classe client de flux de travail basée sur l'interface, comme indiqué plus loin.

`@Workflow`possède un paramètre facultatif, DataConverter, qui est souvent utilisé avec sa valeur par défaut, qui indique qu' NullDataConverteril doit être utilisé. JsonDataConverter

`@WorkflowRegistrationOptions` comporte aussi un certain nombre de paramètres facultatifs qui peuvent être utilisés pour configurer l'objet exécuteur de flux de travail. Ici, nous avons défini la durée pendant `defaultExecutionStartToCloseTimeoutSeconds` laquelle le flux de travail peut s'exécuter à 3 600 secondes (1 heure).

La définition de `GreeterWorkflow` l'interface diffère HelloWorld de l'@Executeannotation sur un point important. Les interfaces de flux de travail définissent les méthodes pouvant être appelées par les applications telles que le démarreur de flux de travail, et sont limitées à une poignée de méthodes , chacune avec un rôle particulier. Le framework ne spécifie pas de nom ou de liste de paramètres pour les méthodes d'interface de flux de travail ; vous utilisez une liste de noms et de paramètres adaptée à votre flux de travail et vous appliquez une annotation AWS Flow Framework pour Java pour identifier le rôle de la méthode.

`@Execute` a deux objectifs :

- Il identifie `greet` comme le point d'entrée du flux de travail, c'est-à-dire la méthode que le démarreur de flux de travail appelle pour démarrer le flux de travail. En général, un point d'entrée peut être associé à un ou plusieurs paramètres, ce qui permet au démarreur d'initialiser le flux de travail, mais cet exemple ne requiert pas d'initialisation.

- Il définit le numéro de version du flux de travail, ce qui vous permet de conserver une trace des différentes générations d'implémentations de flux de travail. Pour modifier une interface de flux de travail après l'avoir enregistrée auprès d'Amazon SWF, notamment pour modifier les valeurs de délai d'expiration, vous devez utiliser un nouveau numéro de version.

Pour obtenir des informations sur les autres méthodes pouvant être incluses dans une interface de flux de travail, consultez Contrats de flux de travail et d'activité.

HelloWorldWorkflow implémente le flux `GreeterWorkflowImpl` de travail comme suit :

```
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);
   }
}
```

Le code est similaire HelloWorld, mais avec deux différences importantes.

- `GreeterWorkflowImpl` crée une instance de `GreeterActivitiesClientImpl`, le client d'activité, au lieu de `GreeterActivitiesImpl`, et exécute les activités en appelant des méthodes sur l'objet client.
- Les activités de nom et de salutation renvoient des objets `Promise<String>` au lieu d'objets `String`.

HelloWorld est une application Java standard qui s'exécute localement en tant que processus unique. Elle `GreeterWorkflowImpl` peut donc implémenter la topologie du flux de travail en créant simplement une instance de`GreeterActivitiesImpl`, en appelant les méthodes dans l'ordre et en transmettant les valeurs de retour d'une activité à l'autre. Avec un flux de travail Amazon SWF, la tâche d'une activité est toujours exécutée par une méthode d'activité provenant de. `GreeterActivitiesImpl` Toutefois, la méthode ne s'exécute pas forcément dans le même processus que le flux de travail (elle peut même ne pas s'exécuter sur le même système) et le flux

de travail a besoin d'exécuter l'activité de façon asynchrone. Ces conditions requises posent les problèmes suivants :

- Comment exécuter une méthode d'activité qui peut être exécutée dans un processus différent, voire même sur un système différent ?
- Comment exécuter une méthode d'activité de façon asynchrone ?
- Comment gérer les valeurs d'entrée et les valeurs de retour des activités ? Par exemple, si la valeur renvoyée par l'activité A est une valeur d'entrée de l'activité B, vous devez vous assurer que l'activité B ne s'exécute pas tant que l'activité A n'est pas terminée.

Vous pouvez implémenter diverses topologies de flux de travail via le flux de contrôle de l'application en utilisant le contrôle de flux Java courant avec le client d'activités et l'élément `Promise<T>`.

Client d'activités

`GreeterActivitiesClientImpl` est à la base un proxy destiné à `GreeterActivitiesImpl` qui permet à une implémentation de flux de travail d'exécuter les méthodes `GreeterActivitiesImpl` de façon asynchrone.

Les classes `GreeterActivitiesClient` et `GreeterActivitiesClientImpl` sont générées automatiquement à partir des informations fournies dans les annotations appliquées à votre classe `GreeterActivities`. Vous n'avez pas besoin de les implémenter vous-même.

> **Note**
>
> Eclipse génère ces classes lorsque vous sauvegardez votre projet. Vous pouvez afficher le code généré dans le sous-répertoire `.apt_generated` de votre répertoire de projet. Afin d'éviter les erreurs de compilation dans votre classe `GreeterWorkflowImpl`, il est conseillé de déplacer le répertoire `.apt_generated` vers le haut de l'onglet Order and Export (Ordonner et exporter) dans la boîte de dialogue Java Build Path (Chemin de génération Java).

Un objet exécuteur de flux de travail exécute une activité en appelant la méthode de client correspondante. Cette méthode est asynchrone et renvoie immédiatement un objet `Promise<T>`, où T représente le type de retour de l'activité. L'objet `Promise<T>` renvoyé est à la base un espace réservé pour la valeur finalement renvoyée par la méthode d'activité.

- Lorsque la méthode du client d'activités renvoie des données, l'objet `Promise<T>` est initialement à l'état non prêt, ce qui indique que l'objet ne représente pas pour l'instant un valeur de retour valide.
- Lorsque la méthode d'activité correspondante a terminé et renvoyé sa tâche, l'infrastructure affecte la valeur de retour à l'objet `Promise<T>` et le place dans l'état prêt.

Promets- <T> Type

Le premier objectif des objets `Promise<T>` est de gérer le flux de données entre les composants asynchrones et le contrôle lorsqu'ils s'exécutent. Il évitent à votre application de devoir gérer de façon explicite la synchronisation ou de dépendre de mécanismes tels que les temporisateurs pour s'assurer que les composants asynchrones ne s'exécutent pas prématurément. Lorsque vous appelez une méthode de client d'activités, elle renvoie immédiatement des données mais l'infrastructure diffère l'exécution de la méthode d'activité correspondante jusqu'à ce qu'un objet d'entrée `Promise<T>` soit prêt et représente des données valides.

À partir de la perspective `GreeterWorkflowImpl`, les trois méthodes de client d'activités renvoient des données immédiatement. À partir de la perspective `GreeterActivitiesImpl`, l'infrastructure n'appelle pas `getGreeting` tan que `name` n'a pas terminé son exécution et n'appelle pas `say` tant que `getGreeting` n'a pas non plus terminé son exécution.

L'utilisation de `Promise<T>` pour transmettre des données d'une activité à la suivante permet à `HelloWorldWorkflow` de s'assurer que les méthodes d'activité ne tentent pas d'utiliser des données non valides, mais aussi de contrôler le moment auquel les activités s'exécutent et, implicitement, de définir la topologie du flux de travail. La transmission de la valeur de retour `Promise<T>` de chaque activité à l'activité suivante nécessite que l'activité suivante s'exécute séquentiellement, en définissant la topologie linéaire présentée préalablement. Avec AWS Flow Framework for Java, vous n'avez pas besoin d'utiliser de code de modélisation spécial pour définir des topologies, même complexes, il suffit d'utiliser un contrôle de flux Java standard et`Promise<T>`. Pour obtenir un exemple d'implémentation d'une topologie parallèle simple, consultez HelloWorldWorkflowParallelTravailleur des activités.

> **Note**
>
> Lorsqu'une méthode d'activité telle que `say` ne renvoie pas de valeur, la méthode du client correspondante renvoie un objet `Promise<Void>`. Cet objet ne représente pas des données, mais il est initialement à l'état non prêt et il devient prêt lorsque l'exécution d'une

activité se termine. Vous pouvez donc transmettre un objet `Promise<Void>` aux autres méthodes de client d'activités afin de vous assurer qu'elles diffèrent l'exécution jusqu'à ce que l'activité initiale soit terminée.

`Promise<T>` permet à une implémentation de flux de travail d'utiliser les méthodes de client d'activités et leurs valeurs de retour de la même manière que les méthodes synchrones. Toutefois, vous devez être prudent lors de l'accès à la valeur d'un objet `Promise<T>`. Contrairement au type Java Future<T>, l'infrastructure gère la synchronisation pour `Promise<T>`, pas l'application. Si vous appelez `Promise<T>.get` et que l'objet n'est pas prêt, `get` émet une exception. Notez que `HelloWorldWorkflow` n'accède jamais directement à un objet `Promise<T>` ; il transmet simplement les objets d'une activité à la suivante. Lorsqu'un objet devient prêt, l'infrastructure extrait la valeur et la transmet à la méthode d'activité sous la forme d'un type standard.

Les objets `Promise<T>` doivent être accessibles uniquement par le code asynchrone, où l'infrastructure garantit que l'objet est prêt et représente une valeur valide. `HelloWorldWorkflow` traite ce problème en transmettant les objets `Promise<T>` uniquement aux activités des méthodes client. Vous pouvez accéder à la valeur d'un objet `Promise<T>` dans l'implémentation du flux de travail en transmettant l'objet à une méthode de flux de travail asynchrone, qui se comporte de façon similaire à une activité. Pour obtenir un exemple, consultez HelloWorldWorkflowAsyncDemande.

HelloWorldWorkflow Mise en œuvre des flux de travail et activités

Les implémentations du flux de travail et des activités ont des classes de travail associées, ActivityWorkeret WorkflowWorker. Ils gèrent la communication entre Amazon SWF et les activités et implémentations de flux de travail en interrogeant la liste de tâches Amazon SWF appropriée pour les tâches, en exécutant la méthode appropriée pour chaque tâche et en gérant le flux de données. Pour plus d'informations, consultez AWS Flow Framework Concepts de base : structure de l'application.

Pour associer les implémentations de flux de travail et d'activités aux objets exécuteurs correspondants, vous implémentez une ou plusieurs applications de travail qui effectuent les tâches suivantes :

- Enregistrez des flux de travail ou des activités avec Amazon SWF.
- Création d'objets exécuteur et association de ces derniers avec les implémentations de travail d'activité ou de flux de travail
- Demandez aux objets de travail de commencer à communiquer avec Amazon SWF.

Si vous souhaitez exécuter le flux de travail et les activités sous forme de processus distincts, vous devez implémenter des hôtes d'exécuteur de flux de travail et d'activités distincts. Pour obtenir un exemple, consultez HelloWorldWorkflowDistributed Demande. Pour des raisons de simplicité, HelloWorldWorkflow implémente un hôte de travail unique qui exécute les activités et les travailleurs du flux de travail dans le même processus, comme suit :

```
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`n'a pas d' HelloWorld équivalent, vous devez donc ajouter une classe Java nommée `GreeterWorker` au projet et copier l'exemple de code dans ce fichier.

La première étape consiste à créer et à configurer un AmazonSimpleWorkflowClientobjet qui invoque les méthodes de service Amazon SWF sous-jacentes. Pour ce faire, `GreeterWorker` :

1. Crée un ClientConfigurationobjet et spécifie un délai d'expiration du socket de 70 secondes. Cette valeur définit combien de temps le système attend que les données soient transférées via une connexion ouverte établie avant de fermer le socket.
2. Crée un AWSCredentials objet Basic pour identifier le AWS compte et transmet les clés du compte au constructeur. Pour plus de commodité et afin d'éviter de les afficher en texte brut dans le code, les clés sont stockées sous forme de variables d'environnement.
3. Crée un AmazonSimpleWorkflowClientobjet pour représenter le flux de travail et transmet les `ClientConfiguration` objets `BasicAWSCredentials` et au constructeur.
4. Définit l'URL du point de terminaison de service de l'objet client. Amazon SWF est actuellement disponible dans toutes les AWS régions.

Pour plus de commodité, `GreeterWorker` définit deux constantes de chaîne :

- `domain`est le nom de domaine Amazon SWF du flux de travail, que vous avez créé lors de la configuration de votre compte Amazon SWF. `HelloWorldWorkflow`suppose que vous exécutez le flux de travail dans le domaine helloWorldWalkthrough « ».
- `taskListToPoll`est le nom des listes de tâches qu'Amazon SWF utilise pour gérer la communication entre les travailleurs du flux de travail et des activités. Vous pouvez attribuer au nom n'importe quelle chaîne appropriée. HelloWorldWorkflow utilise « HelloWorldList » pour les listes de tâches liées aux flux de travail et aux activités. En arrière-plan, les noms se terminent par des espaces de noms différents ; les listes de tâches sont donc distinctes.

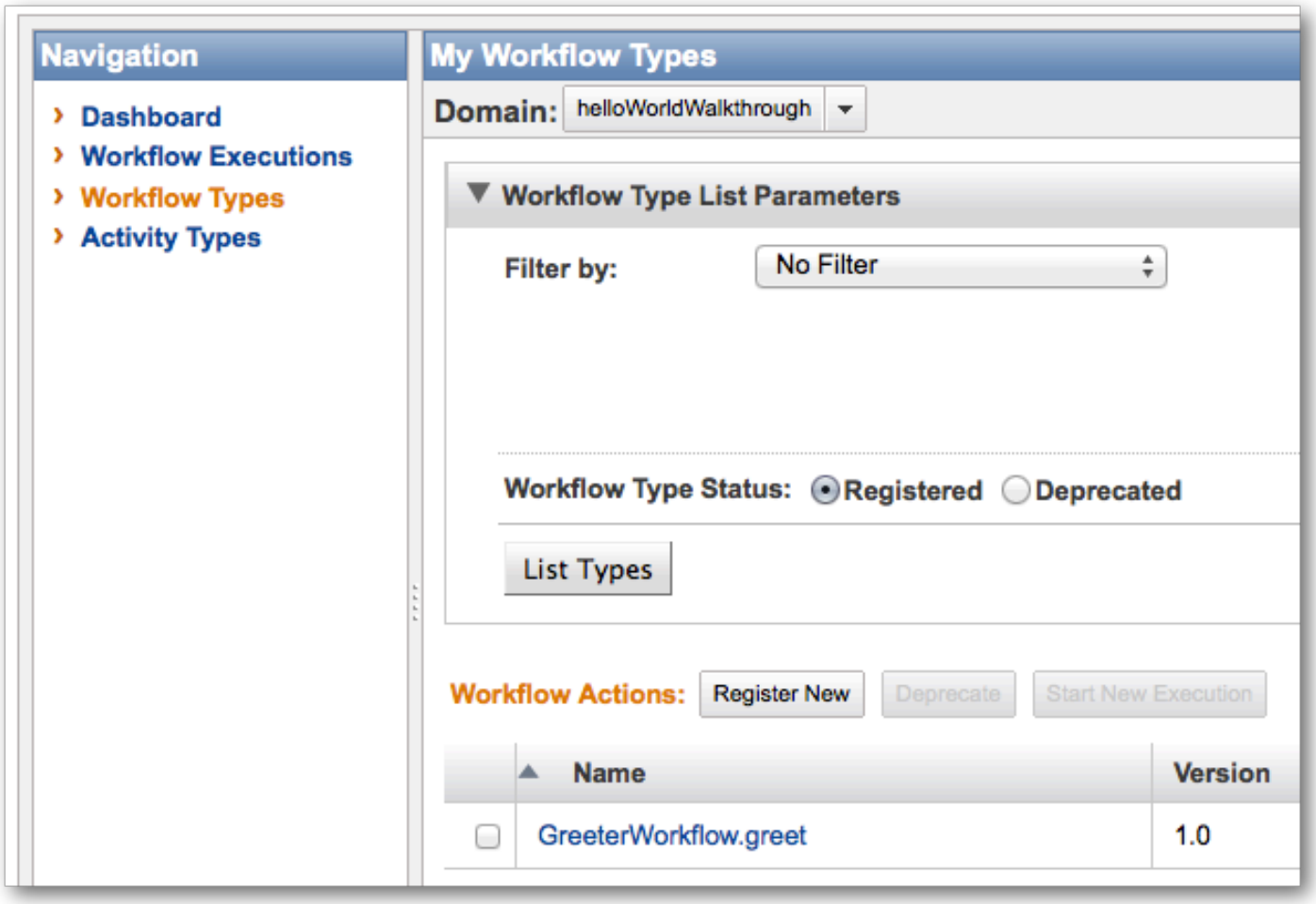
`GreeterWorker`utilise les constantes de chaîne et l'AmazonSimpleWorkflowClientobjet pour créer des objets de travail, qui gèrent l'interaction entre les activités et les implémentations de travail et Amazon SWF. Les objets exécuteur gèrent en particulier la tâche de recherche de tâches dans la liste des tâches appropriée.

`GreeterWorker` crée un objet `ActivityWorker` et le configure pour gérer `GreeterActivitiesImpl` en ajoutant une nouvelle instance de classe. `GreeterWorker` appelle ensuite la méthode `start` de l'objet `ActivityWorker`, qui indique à l'objet de commencer à interroger la liste des tâches des activités spécifiées.

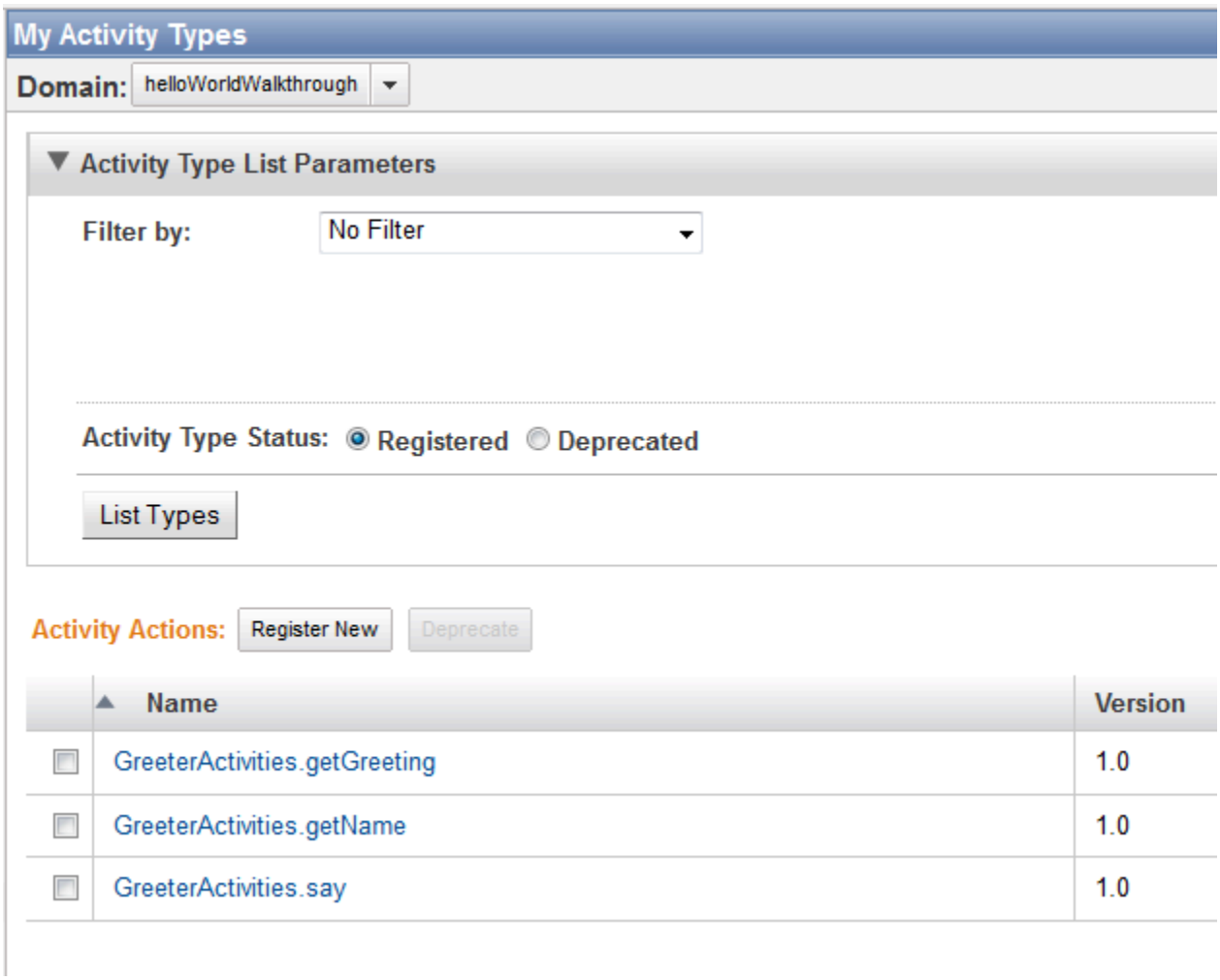
`GreeterWorker` crée un objet `WorkflowWorker` et le configure pour gérer `GreeterWorkflowImpl` en ajoutant le nom de fichier de classe `GreeterWorkflowImpl.class`.

IL appelle ensuite dans l'objet `WorkflowWorker` la méthode `start` qui indique à l'objet qu'il doit lancer la recherche des tâches dans la liste de tâches de flux de travail spécifiée.

Vous pouvez exécuter `GreeterWorker` avec succès à partir de ce moment. Il enregistre le flux de travail et les activités dans Amazon SWF et lance les objets de travail à interroger leurs listes de tâches respectives. Pour vérifier cela, exécutez `GreeterWorker` et accédez à la console Amazon SWF, puis sélectionnez un `helloWorldWalkthrough` domaine dans la liste des domaines. Si vous choisissez Workflow Types (Types de flux de travail) dans le volet Navigation, vous devez voir `GreeterWorkflow.greet`:



Si vous choisissez Activity Types (Types d'activités), les méthodes `GreeterActivities` s'affichent :



Toutefois, si vous choisissez Workflow Executions (Exécutions de flux de travail), vous ne verrez aucune exécution active. Bien que les exécuteurs d'activités et de flux de travail soient en train de rechercher les tâches, nous n'avons pas encore démarré une exécution du flux de travail.

HelloWorldWorkflow Démarreur

La dernière pièce du puzzle consiste à implémenter le démarreur de flux de travail, qui est une application lançant l'exécution du flux de travail. L'état d'exécution est stocké par Amazon SWF, afin que vous puissiez consulter son historique et son état d'exécution. HelloWorldWorkflow implémente un démarreur de flux de travail en modifiant la `GreeterMain` classe, comme suit :

```
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` crée un objet `AmazonSimpleWorkflowClient` en utilisant le même code que `GreeterWorker`. Il crée ensuite un objet `GreeterWorkflowClientExternal` qui agit comme un proxy pour le flux de travail, de la même manière que le client d'activités créé dans `GreeterWorkflowClientImpl` agit comme un proxy pour les méthodes d'activité. Au lieu de créer un objet de client de flux de travail en utilisant `new`, vous devez :

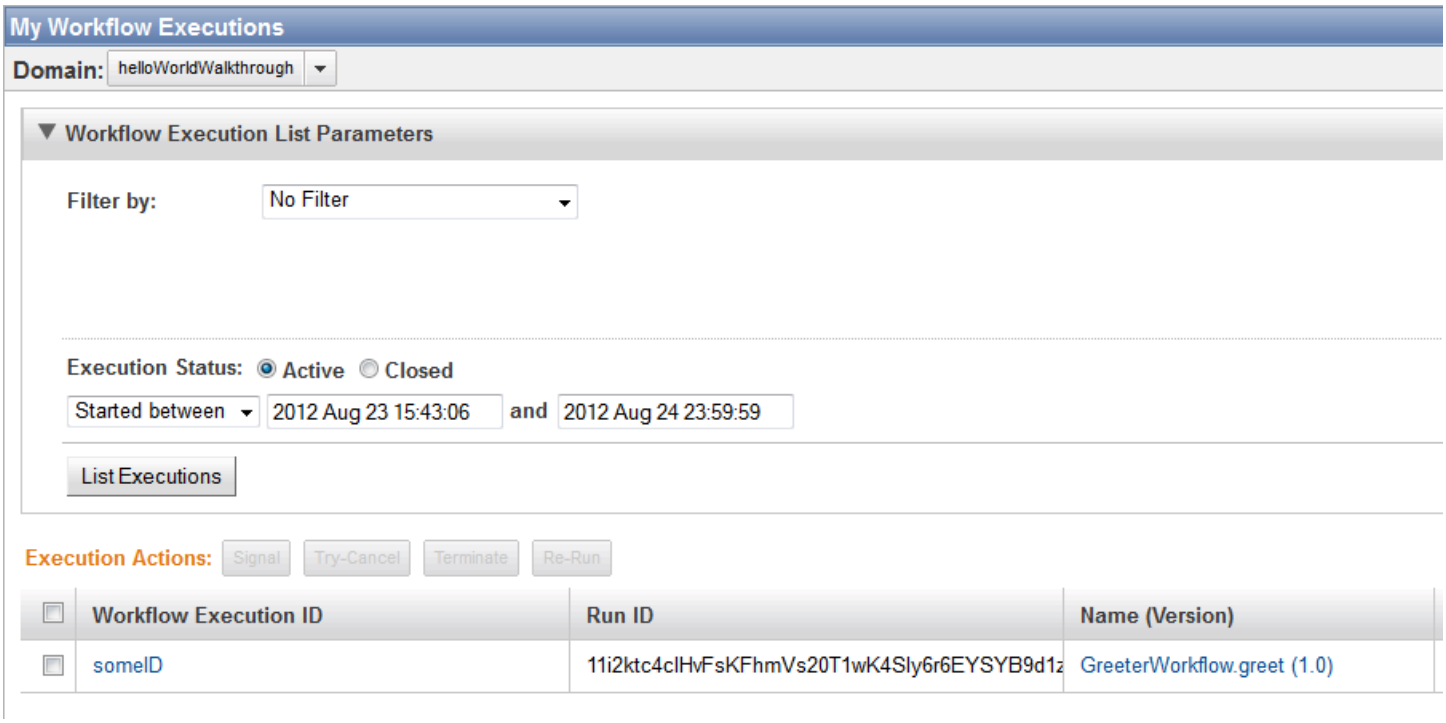
1. Créez un objet d'usine client externe et transmettez l'`AmazonSimpleWorkflowClient`objet et le nom de domaine Amazon SWF au constructeur. L'objet client factory est créé par le processeur d'annotation du framework, qui crée le nom de l'objet en ajoutant simplement « ClientExternalFactoryImpl » au nom de l'interface du flux de travail.
2. Créez un objet client externe en appelant la `getClient` méthode de l'objet d'usine, qui crée le nom de l'objet en ajoutant « ClientExternal » au nom de l'interface du flux de travail. Vous pouvez éventuellement transmettre `getClient` une chaîne qu'Amazon SWF utilisera pour identifier cette instance du flux de travail. Sinon, Amazon SWF représente une instance de flux de travail à l'aide d'un GUID généré.

Le client renvoyé par l'usine créera uniquement des flux de travail nommés avec la chaîne transmise à la méthode GetClient (le client renvoyé par l'usine possède déjà un état dans Amazon SWF). Pour exécuter un flux de travail avec un ID différent, vous devez revenir à la fabrique et créer un nouveau client avec l'ID différent spécifié.

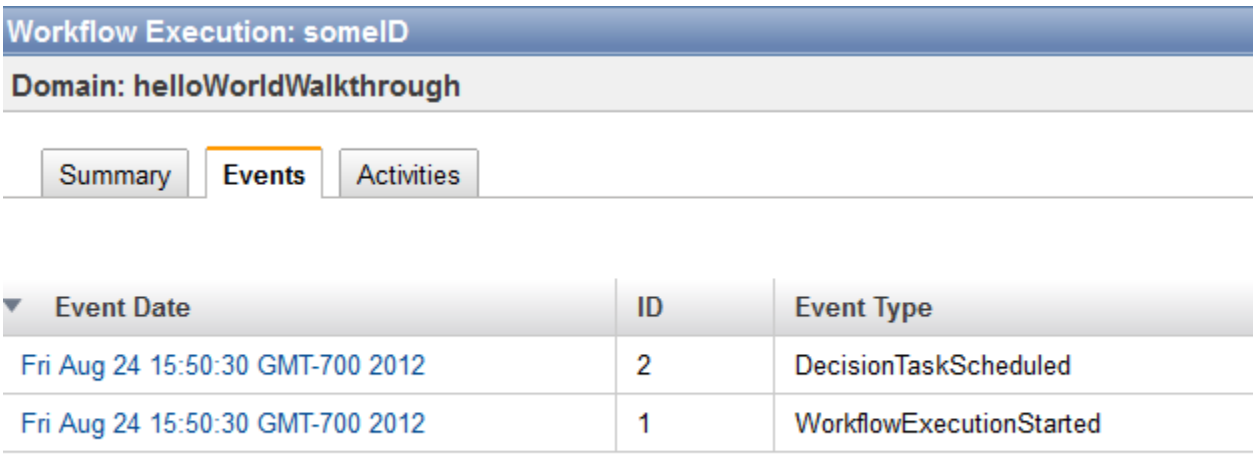
Le client du flux de travail affiche une méthode `greet` qui est appelée par `GreeterMain` pour commencer l'exécution du flux de travail, car `greet()` est la méthode qui a été spécifiée avec l'annotation `@Execute`.

> **Note**
>
> Le processeur d'annotations crée aussi un objet de fabrique de clients interne qui est utilisé pour créer des flux de travail enfants. Pour en savoir plus, consultez Exécutions de flux de travail enfant.

Arrêtez `GreeterWorker` pour le moment s'il est toujours en cours d'exécution, et exécutez `GreeterMain`. Vous devriez maintenant voir SomeID dans la liste des exécutions de flux de travail actives de la console Amazon SWF :.



Si vous choisissez `someID` et que vous choisissez l'onglet Events (Événements), les événements suivants s'affichent :

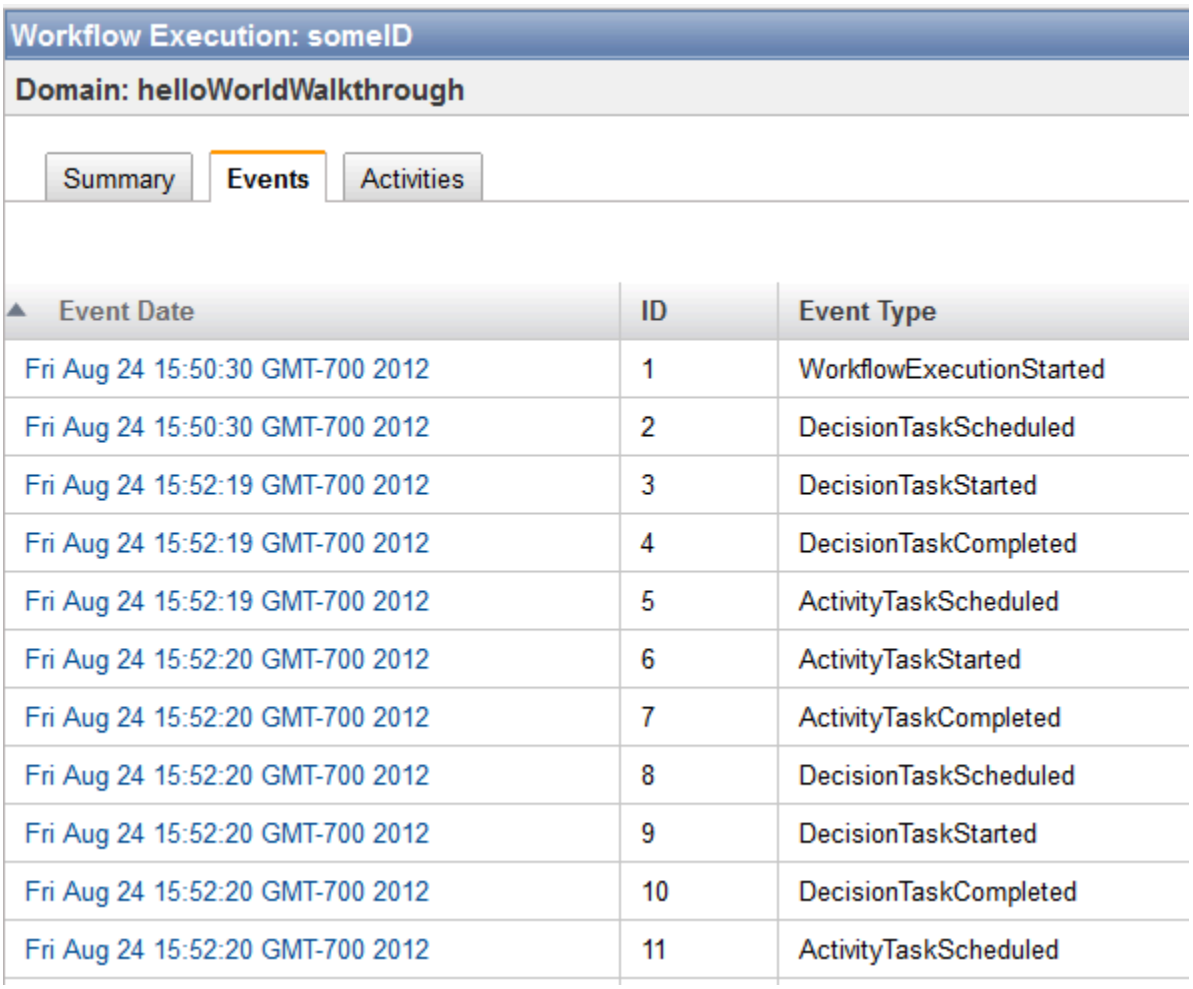


> **Note**
>
> Si vous avez déjà démarré `GreeterWorker` et qu'il est toujours en cours d'exécution, vous verrez une liste d'événements plus longue pour des raisons que nous évoquerons plus loin. Arrêtez `GreeterWorker` et essayez d'exécuter à nouveau `GreaterMain`.

L'onglet Events (Événements) affiche seulement deux événements :

- `WorkflowExecutionStarted` indique que le flux de travail a commencé son exécution.
- `DecisionTaskScheduled`indique qu'Amazon SWF a mis en file d'attente la première tâche de décision.

La raison pour laquelle le flux de travail est bloqué sur la première tâche de décision est que le flux de travail est réparti entre deux applications, `GreeterMain` et `GreeterWorker`. `GreeterMain` a démarré l'exécution du flux de travail, mais `GreeterWorker` n'est pas en cours d'exécution, de telle sorte que les applications de travail n'interrogent pas les listes et n'exécutent pas les tâches. Vous pouvez exécuter l'une ou l'autre des applications de façon indépendante, mais vous avez besoin des deux pour que l'exécution du flux de travail aille au-delà de la première tâche de décision. Si vous exécutez à présent `GreeterWorker`, les objets exécuteur de flux de travail et d'activité vont commencer à interroger les listes et les diverses tâches seront rapidement exécutées. Si vous consultez à présent l'onglet `Events`, le premier lot d'événements s'affiche.



Workflow Execution: someID

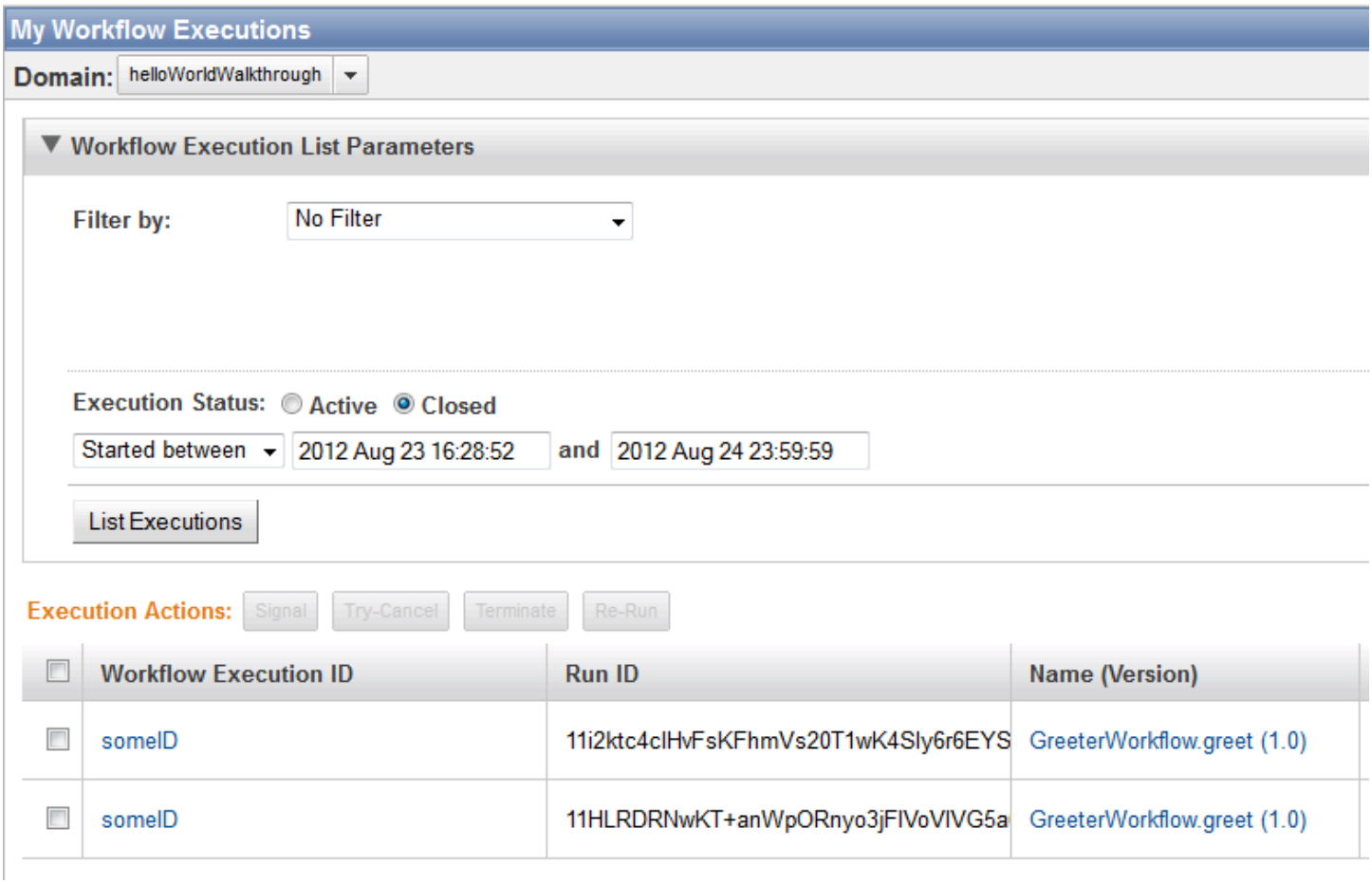
Domain: helloWorldWalkthrough

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

Vous pouvez choisir des événements individuels pour obtenir plus d'informations. Lorsque vous aurez fini de chercher, le flux de travail devrait avoir imprimé « Hello World ! » sur votre console.

Une fois que le flux de travail est terminé, il ne s'affiche plus dans la liste des exécutions actives. Toutefois, si vous souhaitez le passer en revue, choisissez le bouton de statut d'exécution Closed (Fermé), puis choisissez List Executions (Liste des exécutions). Vous affichez ainsi la totalité des instances de flux de travail terminées dans le domaine spécifié (`helloWorldWalkthrough`) et qui n'ont pas dépassé leur durée de conservation maximale (durée que vous avez définie lors de la création du domaine).



Notez que chaque instance de flux de travail possède une valeur Run ID (ID d'exécution) unique. Vous pouvez utiliser le même identifiant de flux de travail pour différentes instances de flux de travail, mais uniquement pour une exécution active à la fois.

HelloWorldWorkflowAsyncDemande

Il est parfois préférable de faire exécuter certaines tâches en local par un flux de travail au lieu d'utiliser une activité. Cependant, les tâches de flux de travail impliquent souvent le traitement des valeurs représentées par des objets `Promise<T>`. Si vous transmettez un objet `Promise<T>` à une méthode de flux de travail synchrone, la méthode s'exécute immédiatement mais elle ne peut pas accéder à la valeur de l'objet `Promise<T>` tant que l'objet n'est pas prêt. Vous pouvez interroger l'objet `Promise<T>.isReady` jusqu'à ce qu'il renvoie `true`, mais cela s'avère inefficace et la méthode peut se bloquer pendant longtemps. Une meilleure approche consiste à utiliser une méthode asynchrone.

Une méthode asynchrone est implémentée de la même manière qu'une méthode standard, souvent en tant que membre de la classe d'implémentation du flux de travail, et s'exécute dans le contexte de

l'implémentation du flux de travail. Vous la désignez en tant que méthode asynchrone en appliquant une annotation `@Asynchronous` qui demande à l'infrastructure de la traiter comme une activité.

- Quand une implémentation de flux de travail appelle une méthode asynchrone, elle renvoie immédiatement un résultat. Les méthodes asynchrones renvoient généralement un objet `Promise<T>` qui devient prêt quand la méthode se termine.
- Si vous transmettez à une méthode asynchrone un ou plusieurs objets `Promise<T>`, cela diffère l'exécution jusqu'à ce que tous les objets d'entrée soient prêts. Une méthode asynchrone peut donc accéder à ses valeurs `Promise<T>` d'entrée sans risquer une exception.

> **Note**
>
> En raison de la façon dont AWS Flow Framework for Java exécute le flux de travail, les méthodes asynchrones s'exécutent généralement plusieurs fois. Vous ne devez donc les utiliser que pour des tâches rapides et peu coûteuses. Il est conseillée d'utiliser des activités pour effectuer des tâches de longue durée comme des calculs volumineux. Pour en savoir plus, consultez AWS Flow Framework Concepts de base : exécution distribuée.

Cette rubrique est une présentation détaillée d' HelloWorldWorkflowAsyncune version modifiée HelloWorldWorkflow qui remplace l'une des activités par une méthode asynchrone. Pour implémenter l'application, créez une copie de HelloWorld. HelloWorldWorkflow placez le package dans le répertoire de votre projet et nommez-le HelloWorld. HelloWorldWorkflowAsync.

> **Note**
>
> Cette rubrique s'appuie sur les concepts et les fichiers présentés dans les rubriques HelloWorld Demande et HelloWorldWorkflow Demande. Familiarisez-vous avec les fichiers et les concepts présentés dans ces rubriques avant de continuer.

Les sections suivantes décrivent comment modifier le HelloWorldWorkflow code d'origine pour utiliser une méthode asynchrone.

HelloWorldWorkflowAsync Mise en œuvre des activités

HelloWorldWorkflowAsync implémente son interface de travail des activités`GreeterActivities`, comme suit :

```
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);
}
```

Cette interface est similaire à celle utilisée par HelloWorldWorkflow, avec les exceptions suivantes :

- Elle omet l'activité `getGreeting` ; cette tâche est maintenant traitée par une méthode asynchrone.
- Le numéro de version est défini sur 2.0. Une fois que vous avez enregistré une interface d'activités auprès d'Amazon SWF, vous ne pouvez pas la modifier à moins de changer le numéro de version.

Les implémentations de méthodes d'activité restantes sont identiques à HelloWorldWorkflow. Supprimez juste `getGreeting` de `GreeterActivitiesImpl`.

HelloWorldWorkflowAsync Mise en œuvre du workflow

HelloWorldWorkflowAsync définit l'interface du flux de travail comme suit :

```
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'interface est identique à l' HelloWorldWorkflow exception d'un nouveau numéro de version. Comme avec les activités, si vous souhaitez modifier un flux de travail enregistré, vous devez changer sa version.

HelloWorldWorkflowAsync implémente le flux de travail comme suit :

```
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 remplace l'`getGreeting`activité par une méthode `getGreeting` asynchrone mais la `greet` méthode fonctionne à peu près de la même manière :

1. Elle exécute l'activité `getName` qui renvoie immédiatement un objet `Promise<String>`, `name`, qui représente le nom.
2. Appelez la méthode asynchrone `getGreeting` et transmettez-lui l'objet `name`. `getGreeting` renvoie immédiatement un objet `Promise<String>`, `greeting`, qui représente la salutation.
3. Elle exécute l'activité `say` et la transmet à l'objet `greeting`.
4. Lorsque `getName` se termine, `name` devient prêt et `getGreeting` utilise sa valeur pour construire la salutation.
5. Lorsque `getGreeting` se termine, `greeting` devient prêt et `say` affiche la chaîne sur la console.

La différence est que, au lieu d'appeler le client d'activités pour exécuter une activité `getGreeting`, la salutation appelle la méthode `getGreeting` asynchrone. le résultat est le même, mais la méthode `getGreeting` fonctionne quelque peu différemment de l'activité `getGreeting`.

- L'exécuteur de flux de travail utilise une sémantique d'appel de fonction standard pour exécuter `getGreeting`. Toutefois, l'exécution asynchrone de l'activité est assurée par Amazon SWF.
- `getGreeting` s'exécute dans le processus d'implémentation de flux de travail.
- `getGreeting` renvoie un objet `Promise<String>` au lieu d'un objet `String`. Pour obtenir la valeur de chaîne détenue par `Promise`, vous appelez sa méthode `get()`. Cependant, étant donné que l'activité est exécutée de manière asynchrone, sa valeur de retour peut ne pas être prête immédiatement ; cela `get()` déclenchera une exception jusqu'à ce que la valeur de retour de la méthode asynchrone soit disponible.

  Pour plus d'informations sur la façon dont `Promise` fonctionne, consultez AWS Flow Framework Concepts de base : échange de données entre les activités et les flux de travail.

`getGreeting` crée une valeur de retour en transmettant la chaîne de salutation à la méthode `Promise.asPromise` statique. Cette méthode crée un objet `Promise<T>` du type approprié, définit la valeur et le met à l'état prêt.

HelloWorldWorkflowAsyncWorkflow et activités Host and Starter

HelloWorldWorkflowAsync implémente en `GreeterWorker` tant que classe hôte pour les implémentations de flux de travail et d'activités. Il est identique à l' HelloWorldWorkflowimplémentation à l'exception du `taskListToPoll` nom, qui est défini sur « `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 implémente le démarreur du flux de travail dans `GreeterMain` ; il est identique à l' HelloWorldWorkflow implémentation.

Pour exécuter le flux de travail, exécutez `GreeterWorker` et`GreeterMain`, comme avec HelloWorldWorkflow.

HelloWorldWorkflowDistributed Demande

Avec HelloWorldWorkflow et HelloWorldWorkflowAsync, Amazon SWF assure l'interaction entre les implémentations du flux de travail et des activités, mais celles-ci s'exécutent localement en tant que processus unique. `GreeterMain`fait l'objet d'un processus distinct, mais il fonctionne toujours sur le même système.

L'une des principales fonctionnalités d'Amazon SWF est qu'il prend en charge les applications distribuées. Par exemple, vous pouvez exécuter le gestionnaire de flux de travail sur une EC2 instance Amazon, le démarreur de flux de travail sur un ordinateur de centre de données et les activités sur un ordinateur de bureau client. Vous pouvez même exécuter différentes activités sur différent systèmes.

L' HelloWorldWorkflowDistributed application s'étend HelloWorldWorkflowAsync pour distribuer l'application sur deux systèmes et trois processus.

- Le flux de travail et le démarreur de flux de travail s'exécutent en tant que processus distincts sur un système.
- Les activités s'exécutent sur un système distinct.

Pour implémenter l'application, créez une copie de HelloWorld. HelloWorldWorkflowAsync placez le package dans le répertoire de votre projet et nommez-le HelloWorld. HelloWorldWorkflowDistributed. Les sections suivantes décrivent comment modifier le HelloWorldWorkflowAsync code d'origine pour distribuer l'application sur deux systèmes et trois processus.

Vous n'avez pas besoin de modifier les implémentations de flux de travail ou d'activités pour les exécuter sur des systèmes distincts, même pas les numéros de version. Vous n'avez pas non plus besoin de modifier `GreeterMain`. Vous devez uniquement modifier l'hôte d'activités et de flux de travail.

Avec HelloWorldWorkflowAsync, une seule application sert d'hôte du flux de travail et de l'activité. Pour exécuter les implémentations de flux de travail et d'activité sur des systèmes distincts, vous devez implémenter des applications distinctes. Supprimer GreeterWorker du projet et ajouter deux nouveaux fichiers de classe, GreeterWorkflowWorker et GreeterActivitiesWorker.

HelloWorldWorkflowDistributed implémente ses activités hébergées dans GreeterActivitiesWorker, comme suit :

```
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 implémente son hôte de flux de travail dans`GreeterWorkflowWorker`, comme suit :

```
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();
    }
}
```

Notez que `GreeterActivitiesWorker` est seulement `GreeterWorker` sans le code `WorkflowWorker` et que `GreeterWorkflowWorker` est seulement `GreeterWorker` sans le code `ActivityWorker`.

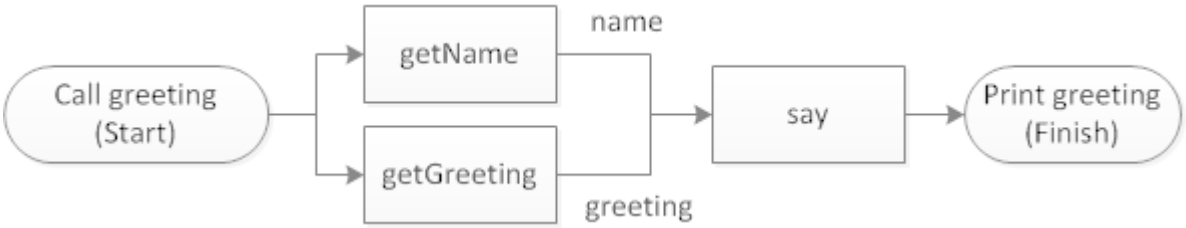
Pour exécuter le flux de travail :

1. Créez un fichier JAR exécutable avec `GreeterActivitiesWorker` comme point d'entrée.
2. Copiez le fichier JAR de l'étape 1 vers un autre système, qui peut exécuter tout système d'exploitation prenant en charge Java.
3. Assurez-vous que les AWS informations d'identification permettant d'accéder au même domaine Amazon SWF sont disponibles sur l'autre système.
4. Exécutez le fichier JAR.
5. Sur votre système de développement, utilisez Eclipse pour exécuter `GreeterWorkflowWorker` et `GreeterMain`.

Hormis le fait que les activités s'exécutent sur un système différent de celui du gestionnaire du flux de travail et du démarreur du flux de travail, le flux de travail fonctionne exactement de la même manière que HelloWorldAsync. Cependant, parce que `println` call that imprime « Hello World ! » si la console est dans l'`say`activité, la sortie apparaîtra sur le système qui exécute le gestionnaire des activités.

HelloWorldWorkflowParallelDemande

Les versions précédentes de Hello World ! toutes utiliser une topologie de workflow linéaire. Amazon SWF ne se limite toutefois pas aux topologies linéaires. L' HelloWorldWorkflowParallel application est une version modifiée HelloWorldWorkflow qui utilise une topologie parallèle, comme le montre la figure suivante.



Avec HelloWorldWorkflowParallel, `getName` et `getGreeting` run in parallel et chacun renvoie une partie du message d'accueil. `say`fusionne ensuite les deux chaînes dans un message d'accueil et l'imprime sur la console.

Pour implémenter l'application, créez une copie de HelloWorld. HelloWorldWorkflow placez le package dans le répertoire de votre projet et nommez-le HelloWorld. HelloWorldWorkflowParallel. Les sections suivantes décrivent comment modifier le HelloWorldWorkflow code d'origine pour qu'il soit exécuté `getName` et `getGreeting` en parallèle.

HelloWorldWorkflowParallelTravailleur des activités

L'interface HelloWorldWorkflowParallel des activités est implémentée dans`GreeterActivities`, comme indiqué dans l'exemple suivant.

```
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'interface est similaire à HelloWorldWorkflow, avec les exceptions suivantes :

- `getGreeting` ne prend aucun paramètre en entrée, mais renvoie simplement une chaîne de message d'accueil.
- `say` prend deux paramètres de type chaîne en entrée, le message d'accueil et le nom.
- L'interface possède un nouveau numéro de version qui est requis chaque fois que vous modifiez une interface enregistrée.

HelloWorldWorkflowParallel met en œuvre les activités dans `GreeterActivitiesImpl` les domaines suivants :

```
public class GreeterActivitiesImpl implements GreeterActivities {

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

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

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

`getName` et `getGreeting` renvoient maintenant la moitié de la chaîne de salutation. `say` concatène les deux parties pour produire la phrase complète et l'affiche sur la console.

HelloWorldWorkflowParallelTravailleur du workflow

L'interface HelloWorldWorkflowParallel de flux de travail est `GreeterWorkflow` implémentée comme suit :

```
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 est identique à la HelloWorldWorkflow version, sauf que le numéro de version a été modifié pour correspondre au travailleur des activités.

Le flux de travail est implémenté dans `GreeterWorkflowImpl`, comme suit :

```
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);
   }
}
```

En un coup d'œil, cette implémentation ressemble beaucoup aux HelloWorldWorkflow trois activités que les méthodes client exécutent en séquence. Pourtant, ce n'est pas le cas des activités.

- HelloWorldWorkflow transmis `name` à`getGreeting`. Étant donné que `name` était un objet `Promise<T>`, `getGreeting` a reporté l'exécution de l'activité jusqu'à ce que `getName` soit terminé, ainsi les deux activités ont été exécutées séquentiellement.
- HelloWorldWorkflowParallel ne transmet aucune entrée `getName` ou`getGreeting`. Aucune des méthodes ne reporte l'exécution et les méthodes d'activité associées s'exécutent immédiatement en parallèle.

L'activité `say` prend `greeting` et `name` en tant que paramètres d'entrée. Puisqu'ils sont des objets `Promise<T>`, `say` reporte l'exécution jusqu'à ce que les deux activités soient terminées, puis construit et imprime le message d'accueil.

Notez qu' HelloWorldWorkflowParallel aucun code de modélisation spécial n'est utilisé pour définir la topologie du flux de travail. Il le fait implicitement en utilisant le contrôle de flux Java standard et en tirant parti des propriétés des `Promise<T>` objets. AWS Flow Framework pour Java, les applications peuvent implémenter des topologies même complexes simplement en utilisant `Promise<T>` des objets en conjonction avec des structures de flux de contrôle Java classiques.

HelloWorldWorkflowParallel Workflow et activités Host and Starter

HelloWorldWorkflowParallel implémente en `GreeterWorker` tant que classe hôte pour les implémentations de flux de travail et d'activités. Il est identique à l' HelloWorldWorkflow implémentation à l'exception du `taskListToPoll` nom, qui est défini sur « HelloWorldParallelList ».

`HelloWorldWorkflowParallel`implémente le démarreur du flux de travail dans`GreeterMain`, et il est identique à l' HelloWorldWorkflow implémentation.

Pour exécuter le flux de travail, lancez `GreeterWorker` et `GreeterMain`, de la même manière qu'avec `HelloWorldWorkflow`.

Comprendre AWS Flow Framework Java

Le AWS Flow Framework for Java fonctionne avec Amazon SWF pour faciliter la création d'applications évolutives et tolérantes aux pannes afin d'effectuer des tâches asynchrones qui peuvent être longues, distantes, ou les deux. Le « Hello World ! » les exemples présentés Qu'est-ce que le AWS Flow Framework pour Java ? ont présenté les bases de l'utilisation du pour AWS Flow Framework implémenter des applications de flux de travail de base. Cette section fournit des informations conceptuelles sur le fonctionnement AWS Flow Framework des applications. La première section résume la structure de base d'une AWS Flow Framework application, et les autres sections fournissent des détails supplémentaires sur le fonctionnement AWS Flow Framework des applications.

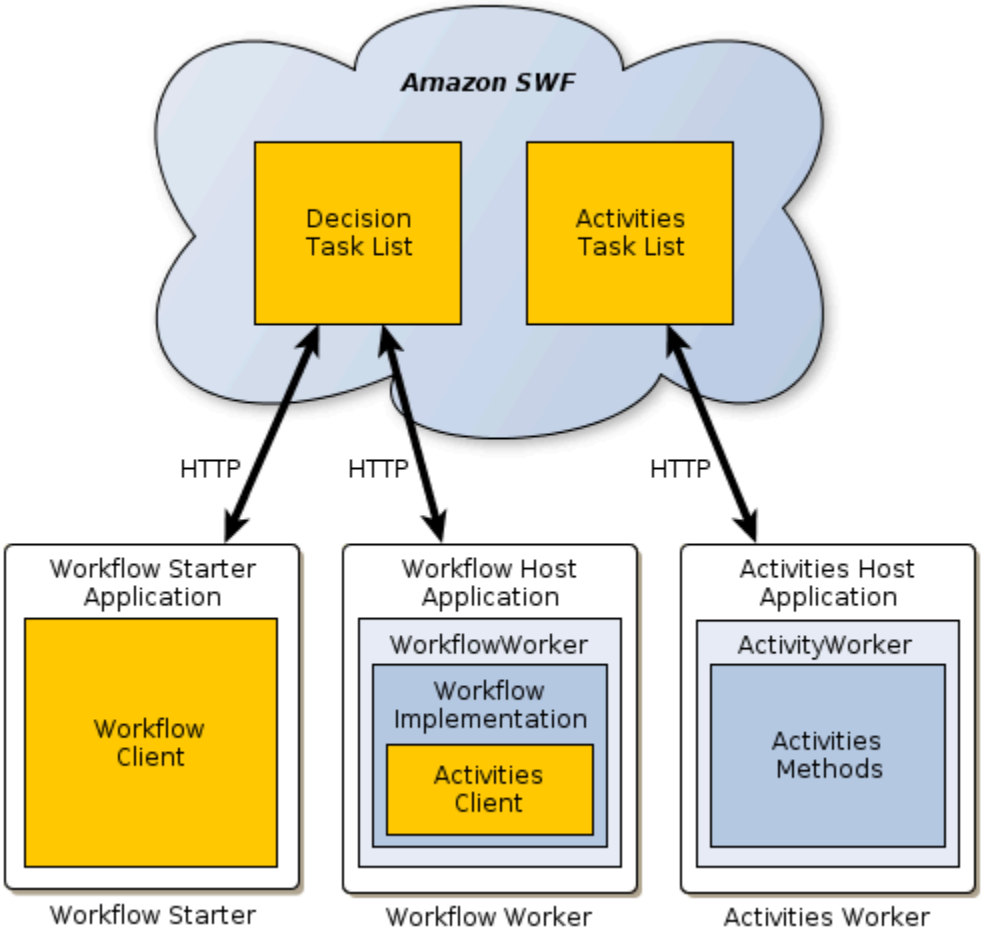
Rubriques

- AWS Flow Framework Concepts de base : structure de l'application
- AWS Flow Framework Concepts de base : exécution fiable
- AWS Flow Framework Concepts de base : exécution distribuée
- AWS Flow Framework Concepts de base : listes de tâches et exécution des tâches
- AWS Flow Framework Concepts de base : applications évolutives
- AWS Flow Framework Concepts de base : échange de données entre les activités et les flux de travail
- AWS Flow Framework Concepts de base : échange de données entre applications et exécutions de flux de travail
- Types de délai d'expiration Amazon SWF

AWS Flow Framework Concepts de base : structure de l'application

Conceptuellement, une AWS Flow Framework application se compose de trois composants de base : les démarreurs de flux de travail, les travailleurs de flux de travail et les travailleurs d'activité. Une ou plusieurs applications hôtes sont chargées d'enregistrer les travailleurs (flux de travail et activité) auprès d'Amazon SWF, de démarrer les travailleurs et de gérer le nettoyage. Les exécuteurs gèrent les mécanismes d'exécution du flux de travail et peuvent être implémentés sur plusieurs hôtes.

Ce schéma représente une AWS Flow Framework application de base :



> **Note**
>
> L'implémentation de ces composants dans trois applications distinctes est pratique d'un point de vue conceptuel, mais vous pouvez créer des applications pour implémenter cette fonctionnalité de différentes façons. Par exemple, vous pouvez utiliser une application hôte unique pour les exécuteurs d'activité et de flux de travail, ou utiliser des hôtes d'activité et de flux de travail distincts. Vous pouvez également avoir plusieurs exécuteurs d'activité, traitant chacun un ensemble d'activités différent sur des hôtes distincts, etc.

Les trois AWS Flow Framework composants interagissent indirectement en envoyant des requêtes HTTP à Amazon SWF, qui gère les demandes. Amazon SWF effectue les opérations suivantes :

- Il gère une ou plusieurs listes de tâches de décision, qui déterminent l'étape suivante à exécuter par un exécuteur de flux de travail.

- Il gère une ou plusieurs listes de tâches d'activité, qui déterminent les tâches qui seront exécutées par un exécuteur d'activité.
- Conserve un step-by-step historique détaillé de l'exécution du flux de travail.

Avec le AWS Flow Framework, le code de votre application n'a pas besoin de traiter directement de nombreux détails illustrés dans la figure, tels que l'envoi de requêtes HTTP à Amazon SWF. Il vous suffit d'appeler AWS Flow Framework des méthodes et le framework gère les détails dans les coulisses.

Rôle de l'exécuteur d'activité

L'exécuteur d'activité exécute les différentes tâches que le flux de travail doit réaliser. Il comprend les éléments suivants :

- L'implémentation des activités, qui comprend un ensemble de méthodes d'activité exécutant des tâches particulières pour le flux de travail.
- Un ActivityWorkerobjet qui utilise de longues requêtes de sondage HTTP pour interroger Amazon SWF afin de déterminer les tâches d'activité à effectuer. Lorsqu'une tâche est nécessaire, Amazon SWF répond à la demande en envoyant les informations nécessaires à l'exécution de la tâche. L'ActivityWorkerobjet appelle ensuite la méthode d'activité appropriée et renvoie les résultats à Amazon SWF.

Rôle de l'exécuteur de flux de travail

L'exécuteur de flux de travail orchestre l'exécution des différentes activités, gère le flux de données et traite les activités ayant échoué. Il comprend les éléments suivants :

- L'implémentation de flux de travail, qui comprend la logique d'orchestration des activités, gère les activités ayant échoué, etc.
- Un client d'activités, qui tient lieu de proxy pour l'exécuteur d'activité et permet au travail de flux de travail de planifier des activités à exécuter de façon asynchrone.
- Un WorkflowWorkerobjet qui utilise de longues requêtes HTTP pour interroger Amazon SWF dans le cadre de tâches décisionnelles. Si des tâches figurent dans la liste des tâches du flux de travail, Amazon SWF répond à la demande en renvoyant les informations nécessaires à l'exécution de la tâche. Le framework exécute ensuite le flux de travail pour effectuer la tâche et renvoie les résultats à Amazon SWF.

Rôle du démarreur de flux de travail

Le démarreur de flux de travail démarre une instance de flux de travail, également appelée exécution de flux de travail, et peut interagir avec une instance lors de l'exécution pour transmettre des données supplémentaires à l'exécuteur de flux de travail ou obtenir l'état actuel du flux de travail.

Le démarreur de flux de travail utilise un client de flux de travail pour lancer l'exécution de flux de travail, interagit avec le flux de travail si nécessaire lors de l'exécution et gère le nettoyage. Le démarreur du flux de travail peut être une application exécutée localement, une application Web, AWS CLI ou même le. AWS Management Console

Comment Amazon SWF interagit avec votre application

Amazon SWF assure l'interaction entre les composants du flux de travail et conserve un historique détaillé du flux de travail. Amazon SWF n'initie pas la communication avec les composants ; il attend les requêtes HTTP provenant des composants et gère les demandes selon les besoins. Par exemple :

- Si la demande provient d'un travailleur qui interroge les tâches disponibles, Amazon SWF répond directement au travailleur si une tâche est disponible. Pour plus d'informations sur le fonctionnement de l'interrogation, consultez Attente active des tâches dans le Manuel du développeur Amazon Simple Workflow Service.
- Si la demande est une notification d'un travailleur d'activité indiquant qu'une tâche est terminée, Amazon SWF enregistre les informations dans l'historique d'exécution et ajoute une tâche à la liste des tâches de décision pour informer le travailleur du flux de travail que la tâche est terminée, lui permettant ainsi de passer à l'étape suivante.
- Si la demande provient du travailleur du flux de travail pour exécuter une activité, Amazon SWF enregistre les informations dans l'historique d'exécution et ajoute une tâche à la liste des tâches des activités pour demander au travailleur d'exécuter la méthode d'activité appropriée.

Cette approche permet aux employés de fonctionner sur n'importe quel système connecté à Internet, y compris les EC2 instances Amazon, les centres de données d'entreprise, les ordinateurs clients, etc. Les travaux n'ont même pas besoin d'être exécutés sur le même système d'exploitation. Comme les demandes HTTP sont initiées avec les exécuteurs, aucun port visible en externe n'est nécessaire ; les exécuteurs peuvent s'exécuter derrière un pare-feu.

Pour en savoir plus

Pour une discussion plus approfondie du fonctionnement d'Amazon SWF, consultez le guide du développeur d'Amazon Simple Workflow Service.

AWS Flow Framework Concepts de base : exécution fiable

Les applications distribuées asynchrones doivent résoudre des problèmes de fiabilité auxquels ne sont pas confrontées les applications conventionnelles, notamment :

- Comment assurer une communication fiable entre des composants distribués asynchrones comme des composants de longue durée sur des systèmes distants.
- Comment s'assurer qu'aucun résultat n'est perdu si un composant échoue ou est déconnecté, en particulier pour les applications de longue durée.
- Comment gérer les composants distribués ayant échoué.

Les applications peuvent compter sur Amazon SWF AWS Flow Framework et Amazon pour gérer ces problèmes. Nous verrons comment Amazon SWF fournit des mécanismes garantissant que vos flux de travail fonctionnent de manière fiable et prévisible, même lorsqu'ils sont de longue durée et dépendent de tâches asynchrones effectuées par calcul et avec interaction humaine.

Assurer une communication fiable

AWS Flow Framework fournit une communication fiable entre un travailleur du flux de travail et ses employés des activités en utilisant Amazon SWF pour répartir les tâches entre les travailleurs des activités distribuées et renvoyer les résultats au travailleur du flux de travail. Amazon SWF utilise les méthodes suivantes pour garantir une communication fiable entre un collaborateur et ses activités :

- Amazon SWF stocke de manière durable les activités planifiées et les tâches de flux de travail et garantit qu'elles seront exécutées au plus une fois.
- Amazon SWF garantit qu'une tâche d'activité s'achèvera correctement et renverra un résultat valide ou qu'elle informera le travailleur du flux de travail que la tâche a échoué.
- Amazon SWF stocke de manière durable le résultat de chaque activité terminée ou, en cas d'échec des activités, stocke les informations d'erreur pertinentes.

Il utilise AWS Flow Framework ensuite les résultats d'activité d'Amazon SWF pour déterminer comment procéder à l'exécution du flux de travail.

S'assurer qu'aucun résultat n'est perdu

Gestion de l'historique du flux de travail

Une activité qui exécute des opérations d'exploration de données sur un pétaoctet de données peut prendre des heures à se terminer et une activité qui exige qu'un être humain effectue une tâche complexe peut même prendre des jours, voire des semaines à s'exécuter !

Pour s'adapter à de tels scénarios, les AWS Flow Framework flux de travail et les activités peuvent prendre un temps arbitrairement long : jusqu'à une limite d'un an pour l'exécution d'un flux de travail. Une exécution fiable des processus de longue durée exige un mécanisme pour stoker durablement l'historique d'exécution du flux de travail de façon continue.

Il AWS Flow Framework gère cela en s'appuyant sur Amazon SWF, qui conserve un historique d'exécution de chaque instance de flux de travail. L'historique du flux de travail fournit un enregistrement complet et fiable de la progression du flux de travail, avec notamment toutes les tâches de flux de travail et d'activité qui ont été planifiées et terminées, et les informations renvoyées par les activités terminées ou ayant échoué.

AWS Flow Framework les applications n'ont généralement pas besoin d'interagir directement avec l'historique du flux de travail, bien qu'elles puissent y accéder si nécessaire. Dans la plupart des cas, les applications peuvent simplement laisser l'infrastructure interagir avec l'historique du flux de travail en arrière-plan. Pour une discussion complète sur l'historique des flux de travail, consultez l'historique des flux de travail dans le guide du développeur Amazon Simple Workflow Service.

Exécution sans état

L'historique d'exécution permet à des exécuteurs de flux de travail d'être sans état. Si vous disposez de plusieurs instances d'un exécuteur de flux de travail ou d'activité, n'importe quel exécuteur peut exécuter toute tâche. Le travailleur reçoit toutes les informations d'état dont il a besoin pour exécuter la tâche depuis Amazon SWF.

Cette approche rend les flux de travail plus fiables. Par exemple, si un exécuteur de flux de travail échoue, vous n'avez pas à redémarrer le flux de travail. Il vous suffit de redémarrer l'exécuteur qui va interroger la liste de tâches et traiter toutes les tâches de la liste, quel que soit le moment où la défaillance a eu lieu. Vous pouvez rendre votre flux de travail global tolérant aux défaillances en utilisant plusieurs exécuteurs de flux de travail et d'activité, peut-être sur des systèmes distincts. Ainsi, si l'un des exécuteurs échoue, les autres continueront de traiter les tâches planifiées sans interruption dans la progression du flux de travail.

Gestion des composants distribués ayant échoué

Les activités échouent souvent pour des raisons éphémères comme une brève déconnexion. Une stratégie courante pour gérer des activités ayant échoué consiste donc à relancer l'activité. Au lieu de traiter le processus de nouvelle tentative en implémentant un message complexe transmettant des stratégies, les applications peuvent s'appuyer sur AWS Flow Framework. Ce dernier fournit plusieurs mécanismes pour relancer les activités ayant échoué et offre un mécanisme de traitement des exceptions intégré qui fonctionne avec l'exécution distribuée asynchrone des tâches d'un flux de travail.

AWS Flow Framework Concepts de base : exécution distribuée

Une instance de flux de travail est essentiellement un fil d'exécution virtuel qui peut couvrir les activités et la logique d'orchestration exécutées sur plusieurs ordinateurs distants. Amazon SWF et la AWS Flow Framework fonction en tant que système d'exploitation qui gère les instances de flux de travail sur un processeur virtuel en :

- Maintenant l'état d'exécution de chaque instance.
- Basculant entre les instances.
- Reprenant l'exécution d'une instance à l'endroit où elle avait été basculée.

Reproduction des flux de travail

Étant donné que les activités peuvent être de longue durée, il n'est pas souhaitable que le flux de travail soit simplement bloqué jusqu'à ce qu'il soit terminé. Il AWS Flow Framework gère plutôt l'exécution du flux de travail à l'aide d'un mécanisme de rediffusion, qui s'appuie sur l'historique du flux de travail conservé par Amazon SWF pour exécuter le flux de travail par épisodes.

Chaque épisode reproduit la logique du flux de travail de façon à exécuter chaque activité une seule fois, et veille à ce que les activités et les méthodes asynchrones ne s'exécutent pas avant que leurs objets Promise soient prêts.

Le démarreur de flux de travail lance le premier épisode de reproduction en même temps que l'exécution du flux de travail. L'infrastructure appelle la méthode de point d'entrée du flux de travail et :

1. Exécute toutes les tâches de flux de travail qui ne dépendent pas de la fin de l'activité, y compris appeler toutes les méthodes client d'activité.

2. Fournit à Amazon SWF une liste d'activités et de tâches dont l'exécution doit être planifiée. Pour le premier épisode, cette liste comporte uniquement les activités qui ne dépendent pas d'un objet Promise et peuvent être exécutées immédiatement.
3. Indique à Amazon SWF que l'épisode est terminé.

Amazon SWF enregistre les tâches d'activité dans l'historique du flux de travail et planifie leur exécution en les plaçant dans la liste des tâches d'activité. Les exécuteurs d'activité interrogent la liste de tâches et exécute ces dernières.

Lorsqu'un travailleur d'activité termine une tâche, il renvoie le résultat à Amazon SWF, qui l'enregistre dans l'historique d'exécution du flux de travail et planifie une nouvelle tâche de flux de travail pour le travailleur de flux de travail en le plaçant sur la liste des tâches du flux de travail. L'exécuteur de flux de travail interroge la liste de tâches et lorsqu'il reçoit la tâche, il exécute le prochain épisode de reproduction, comme suit:

1. L'infrastructure exécute de nouveau la méthode de point d'entrée du flux de travail et :
   - Exécute toutes les tâches de flux de travail qui ne dépendent pas de la fin de l'activité, y compris appeler toutes les méthodes client d'activité. Pourtant, l'infrastructure vérifie l'historique d'exécution et ne planifie aucune tâche d'activité en double.
   - Vérifie l'historique pour consulter les tâches d'activité qui ont été terminées et exécute n'importe quelle méthode de flux de travail asynchrone qui dépend de ces activités.
2. Lorsque toutes les tâches de flux de travail pouvant être exécutées sont terminées, le framework envoie un rapport à Amazon SWF :
   - Il fournit à Amazon SWF une liste de toutes les activités dont les `Promise<T>` objets d'entrée sont prêts depuis le dernier épisode et dont l'exécution peut être planifiée.
   - Si l'épisode n'a généré aucune tâche d'activité supplémentaire mais que des activités sont toujours inachevées, le framework indique à Amazon SWF que l'épisode est terminé. Ensuite, il attend qu'une autre activité soit terminée, en lançant le prochain épisode de reproduction.
   - Si l'épisode n'a généré aucune tâche d'activité supplémentaire et que toutes les activités sont terminées, le framework indique à Amazon SWF que l'exécution du flux de travail est terminée.

Pour obtenir des exemples du comportement de reproduction, consultez AWS Flow Framework pour Java Replay Behavior.

Reproduction et méthodes de flux de travail asynchrones

Les méthodes de flux de travail asynchrones sont souvent utilisées comme les activités, car la méthode diffère l'exécution jusqu'à ce que tous les objets `Promise<T>` d'entrée soient prêts. Pourtant, le mécanisme de reproduction gère les méthodes asynchrones différemment des activités.

- La reproduction ne garantit pas qu'une méthode asynchrone s'exécutera seulement une fois. Elle diffère l'exécution sur une méthode asynchrone jusqu'à ce que ses objets Promise d'entrée soient prêts, mais elle exécute ensuite cette méthode pour tous les épisodes suivants.
- Lorsqu'une méthode asynchrone se termine, elle ne lance pas un nouvel épisode.

Un exemple de reproduction d'un flux de travail asynchrone est proposé dans AWS Flow Framework pour Java Replay Behavior.

Implémentation de reproduction et de flux de travail

Dans la plupart des cas, vous n'avez pas besoin de vous préoccuper des détails du mécanisme de reproduction. Il s'agit essentiellement d'une opération qui se déroule en arrière-plan. Pourtant, la reproduction possède deux implications importantes pour l'implémentation de votre flux de travail.

- N'utilisez pas de méthodes de flux de travail pour exécuter des tâches de longue durée, car la reproduction répétera cette tâche plusieurs fois. Même les méthodes asynchrones s'exécutent généralement plus d'une fois. À la place, utilisez les activités pour les tâches de longue durée ; la reproduction exécute les activités une seule fois.
- Votre logique de flux de travail doit être totalement déterministe ; chaque épisode doit prendre le même chemin de flux de contrôle. Par exemple, le chemin de flux de contrôle ne doit pas dépendre de l'heure actuelle. Pour obtenir une description détaillée de la reproduction et des exigences en matière de déterminisme, consultez Non-déterminisme.

AWS Flow Framework Concepts de base : listes de tâches et exécution des tâches

Amazon SWF gère les tâches liées au flux de travail et aux activités en les publiant dans des listes nominatives. Amazon SWF gère au moins deux listes de tâches, l'une pour les travailleurs du flux de travail et l'autre pour les travailleurs des activités.

> **Note**
>
> Vous pouvez spécifier autant de listes de tâches que nécessaire, différents exécuteurs pouvant être affectés à chaque liste. Le nombre de listes de tâches est illimité. En règle générale, vous spécifiez la liste des tâches d'un exécuteur dans l'application hôte de ce dernier lorsque vous créez l'objet exécuteur.

L'extrait suivant tiré de l'application hôte `HelloWorldWorkflow` crée un nouvel exécuteur d'activité et l'affecte à la liste de tâches des activités `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();
    ...
  }
}
```

Par défaut, Amazon SWF planifie les tâches du travailleur dans la `HelloWorldList` liste. L'exécuteur recherche ensuite les tâches dans cette liste. Vous pouvez affecter n'importe quel nom à une liste de tâches. Vous pouvez même utiliser le même nom pour les listes de flux de travail et d'activités. En interne, Amazon SWF place les noms des flux de travail et des listes de tâches d'activité dans différents espaces de noms, de sorte que les deux listes seront distinctes.

Si vous ne spécifiez pas de liste de tâches, une liste par défaut est spécifiée lorsque le travailleur enregistre le type auprès d'Amazon SWF. AWS Flow Framework Pour de plus amples informations, veuillez consulter Enregistrement des types de flux de travail et d'activité.

Il est parfois utile de faire exécuter certaines tâches par un exécuteur ou un groupe d'exécuteurs. Par exemple, un flux de travail de traitement d'image peut utiliser une activité pour télécharger une image et une autre pour traiter l'image. Il est plus efficace d'exécuter les deux tâches sur le même système et d'éviter la surcharge liée au transfert de fichiers volumineux sur le réseau.

Pour prendre en charge ce type de scénario, vous pouvez spécifier explicitement une liste de tâches lorsque vous appelez une méthode de client d'activité en utilisant une surcharge qui inclut un paramètre `schedulingOptions`. Vous spécifiez la liste des tâches en transmettant à la méthode un `ActivitySchedulingOptions` objet configuré de manière appropriée.

Supposons, par exemple, que l'activité `say` de l'application `HelloWorldWorkflow` soit hébergée par un exécuteur d'activité autre que `getName` et `getGreeting`. L'exemple suivant montre comment s'assurer que `say` utilise la même liste de tâches que `getName` et `getGreeting`, même si elles ont été affectées à l'origine à des listes différentes.

```
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);
  }
}
```

La méthode asynchrone `runSay` obtient la liste de tâches `getGreeting` à partir de son objet client. Ensuite, elle crée et configure un objet `ActivitySchedulingOptions` qui vérifie que `say` interroge la même liste de tâches que `getGreeting`.

> **Note**
>
> Lorsque vous transmettez un paramètre `schedulingOptions` à une méthode de client d'activité, il remplace la liste de tâches d'origine uniquement pour l'exécution de l'activité en question. Si vous appelez à nouveau la méthode client des activités sans spécifier de liste

de tâches, Amazon SWF affecte la tâche à la liste d'origine et le responsable de l'activité interrogera cette liste.

AWS Flow Framework Concepts de base : applications évolutives

Amazon SWF possède deux fonctionnalités clés qui facilitent le dimensionnement d'une application de flux de travail pour gérer la charge actuelle :

- Un historique de flux de travail complet qui vous permet d'implémenter une application sans état.
- La planification des tâches est couplée de façon souple avec l'exécution des tâches, ce qui vous permet de mettre votre application facilement à l'échelle pour répondre aux demandes actuelles.

Amazon SWF planifie les tâches en les publiant dans des listes de tâches allouées dynamiquement, et non en communiquant directement avec les responsables des flux de travail et des activités. Au lieu de cela, les exécuteurs utilisent des demandes HTTP pour rechercher les tâches dans leurs listes respectives. Cette approche associe vaguement la planification des tâches à l'exécution des tâches et permet aux employés de fonctionner sur n'importe quel système approprié, y compris les EC2 instances Amazon, les centres de données d'entreprise, les ordinateurs clients, etc. Comme les requêtes HTTP proviennent des opérateurs, il n'est pas nécessaire de recourir à des ports visibles de l'extérieur, ce qui permet même aux utilisateurs de courir derrière un pare-feu.

Le mécanisme d'attente active de longue durée (interrogation longue) utilisé par les exécuteurs pour rechercher les tâches évite à ceux-ci d'être surchargés. Même en cas de pic dans les tâches planifiées, les exécuteurs extraient les tâches à leur propre rythme. Cependant, comme les exécuteurs sont sans état, vous pouvez mettre une application à l'échelle pour répondre à l'augmentation de la charge en démarrant des instances de travail supplémentaires. Même si les instances s'exécutent sur des systèmes différents, chacune d'entre elles interroge la même liste de tâches et la première instance de travail disponible exécute chaque tâche, quels que soient l'emplacement de l'exécuteur et le moment où il est démarré. Lorsque la charge diminue, vous pouvez réduire le nombre d'exécuteurs en conséquence.

AWS Flow Framework Concepts de base : échange de données entre les activités et les flux de travail

Lorsque vous appelez une méthode client d'activité asynchrone, elle renvoie immédiatement un objet Promise (également appelé Future), qui représente la valeur de renvoi de la méthode d'activité. Au départ, l'objet Promise est à l'état non prêt et la valeur de renvoi est non définie. Lorsque la méthode d'activité a terminé et renvoyé la tâche, l'infrastructure regroupe la valeur de renvoi via le réseau sur l'exécuteur du flux de travail, qui affecte une valeur à Promise et place l'objet à l'état prêt.

Même si une méthode d'activité n'a aucune valeur de renvoi, vous pouvez utiliser l'objet Promise pour gérer l'exécution du flux de travail. Si vous transmettez à une méthode client d'activité ou à une méthode de flux de travail asynchrone un objet Promise renvoyé, l'exécution est différée jusqu'à ce que l'objet soit prêt.

Si vous transmettez à une méthode client d'activité un ou plusieurs objets Promise, l'infrastructure place la tâche en file d'attente mais en diffère la planification jusqu'à ce que tous les objets soient prêts. Elle extrait ensuite les données de chaque objet Promise et les regroupe via Internet sur l'exécuteur d'activité, qui les transmet à la méthode d'activité en tant que type standard.

> **Note**
>
> Si vous avez besoin de transférer de grandes quantités de données entre un flux de travail et des exécuteurs d'activité, la méthode à privilégier consiste à stocker les données dans un emplacement approprié et transmettre uniquement les informations de récupération. Par exemple, vous pouvez stocker les données dans un compartiment Amazon S3 et transmettre l'URL associée.

Le Promesse <T> Type

Le type `Promise<T>` est relativement similaire au type Java `Future<T>`. Ces deux types représentent des valeurs renvoyées par des méthodes asynchrones et sont, au départ, non définis. You accédez à la valeur d'un objet en appelant sa méthode `get`. Au-delà de ça, les deux types ont un comportement assez différent.

- `Future<T>` est une construction de synchronisation qui permet à une application d'attendre la fin d'une méthode asynchrone. Si vous appelez `get` et que l'objet n'est pas prêt, la méthode se bloque jusqu'à ce que l'objet soit prêt.

- Avec `Promise<T>`, la synchronisation est gérée par l'infrastructure. Si vous appelez `get` et que l'objet n'est pas prêt, `get` émet une exception.

L'objectif principal de `Promise<T>` consiste à gérer le flux de données d'une activité à une autre. Cela permet de s'assurer qu'une activité ne s'exécute pas tant que les données d'entrée ne sont pas valides. Dans de nombreux cas, les exécuteurs de flux de travail n'ont pas besoin d'accéder directement aux objets `Promise<T>` ; ils transmettent simplement les objets d'une activité à une autre et laisse l'infrastructure et les travaux d'activité gérer les détails. Pour accéder à la valeur d'un objet `Promise<T>` dans un exécuteur de flux de travail, vous devez être sûr que l'objet est prêt avant d'appeler sa méthode `get`.

- L'approche préférentielle consiste à transmettre l'objet `Promise<T>` à une méthode de flux de travail asynchrone et à traiter les valeurs à ce niveau. Une méthode asynchrone diffère l'exécution jusqu'à ce que tous ses objets `Promise<T>` d'entrée soient prêts, ce qui garantit que vous pouvez accéder en toute sécurité à leurs valeurs.
- `Promise<T>` expose une méthode `isReady` qui renvoie `true` si l'objet est prêt. Il n'est pas recommandé d'utiliser `isReady` pour interroger un objet `Promise<T>`, mais `isReady` est utile dans certaines circonstances.

Le AWS Flow Framework for Java inclut également un `Settable<T>` type dérivé de `Promise<T>` et ayant un comportement similaire. La différence est que le framework définit généralement la valeur d'un `Promise<T>` objet et que le travailleur du flux de travail est chargé de définir la valeur de `Settable<T>` a.

Dans certaines circonstances, un exécuteur de flux de travail doit créer un objet `Promise<T>` et définir sa valeur. Par exemple, une méthode asynchrone qui renvoie un objet `Promise<T>` doit créer une valeur de renvoi.

- Pour créer un objet qui représente une valeur typée, appelez la méthode statique `Promise.asPromise`, qui crée un objet `Promise<T>` du type approprié, définit sa valeur et le place à l'état prêt.
- Pour créer un objet `Promise<Void>`, appelez la méthode statique `Promise.Void`.

> **Note**
>
> `Promise<T>` peut représenter n'importe quel type valide. Toutefois, si les données doivent être regroupées via Internet, le type doit être compatible avec le convertisseur de données. Pour en savoir plus, consultez la section suivante.

Convertisseurs de données et regroupement

Il AWS Flow Framework rassemble les données sur Internet à l'aide d'un convertisseur de données. Par défaut, l'infrastructure utilise un convertisseur de données basé sur le processeur Jackson JSON. Toutefois, ce convertisseur présente certaines limitations. Par exemple, il ne peut pas regrouper les mappages qui n'utilisent pas des chaînes en tant que clés. Si le convertisseur par défaut n'est pas suffisant pour votre application, vous pouvez implémenter un convertisseur de données personnalisé. Pour en savoir plus, consultez DataConverters.

AWS Flow Framework Concepts de base : échange de données entre applications et exécutions de flux de travail

Une méthode de point d'entrée de flux de travail peut comporter un ou plusieurs paramètres, ce qui permet au démarreur de flux de travail de transmettre les données initiales au flux de travail. Elle peut également s'avérer utile pour fournir des données supplémentaires au flux de travail pendant l'exécution. Par exemple, si un client change son adresse de livraison, vous pouvez avertir le flux de travail de traitement des commandes pour qu'il puisse effectuer les modifications appropriées.

Amazon SWF permet aux flux de travail d'implémenter une méthode de signal, qui permet à des applications telles que le démarreur de flux de travail de transmettre des données au flux de travail à tout moment. Une méthode signal peut avoir n'importe quel nom et n'importe quels paramètres utiles. Vous la désignez en tant que méthode signal en l'incluant dans votre définition d'interface de flux de travail et en appliquant une annotation `@Signal` à la déclaration de la méthode.

L'exemple suivant illustre une interface de flux de travail de traitement des commandes qui déclare une méthode signal, `changeOrder`, qui permet au démarreur de flux de travail de modifier la commande initiale après que le flux de travail a démarré.

```
@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 300)
```

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

Le processeur d'annotation de l'infrastructure crée une méthode de client de flux de travail avec le même nom que la méthode signal, et le démarreur de flux de travail appelle la méthode de client pour transmettre des données au flux de travail. Pour un exemple, voir AWS Flow Framework Recettes

Types de délai d'expiration Amazon SWF

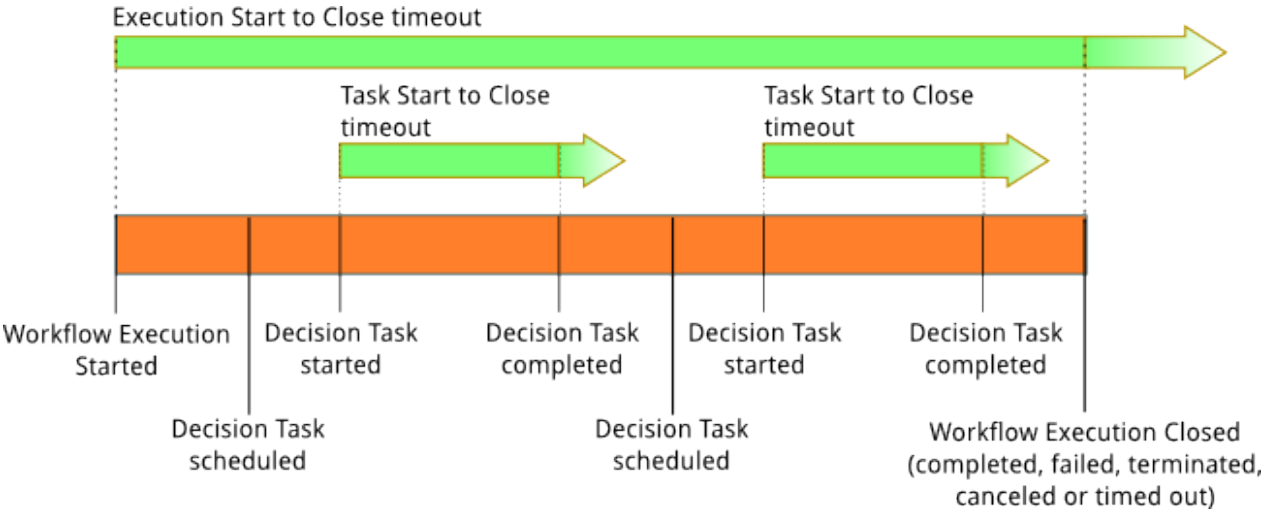
Pour garantir que les exécutions des flux de travail s'exécutent correctement, vous pouvez définir différents types de délais d'expiration avec Amazon SWF. Certains délais spécifient la durée totale d'exécution du flux de travail. D'autres délais définissent combien de temps les tâches d'activité peuvent prendre avant d'être affectées à un outil de traitement, ainsi que combien de temps elles peuvent prendre pour se terminer à compter de l'heure où elles sont planifiées. Tous les délais d'expiration de l'API Amazon SWF sont spécifiés en secondes. Amazon SWF prend également en charge la chaîne en `NONE` tant que valeur de délai d'attente, ce qui indique l'absence de délai d'expiration.

Pour les délais liés aux tâches de décision et aux tâches d'activité, Amazon SWF ajoute un événement à l'historique d'exécution du flux de travail. Les attributs de l'événement fournissent des informations sur le type de délai d'attente qui s'est produit et sur la tâche de décision ou d'activité affectée. Amazon SWF planifie également une tâche de décision. Lorsque le décideur reçoit la nouvelle tâche de décision, il voit l'événement de temporisation dans l'historique et prend l'action appropriée en l'RespondDecisionTaskCompletedappelant.

Une tâche est considérée comme ouverte depuis le moment où elle est planifiée jusqu'à ce qu'elle soit fermée. Par conséquent, une tâche est indiquée comme ouverte lorsqu'un outil de traitement s'en occupe. Une tâche est fermée lorsqu'un outil de traitement la signale comme terminée, comme annulée ou comme ayant échoué. Une tâche peut également être fermée par Amazon SWF en raison d'un délai d'attente.

Délais liés au flux de travail et aux tâches de décision

Le schéma suivant illustre comment les délais de flux de travail et de décision sont liés à la durée de vie d'un flux de travail :

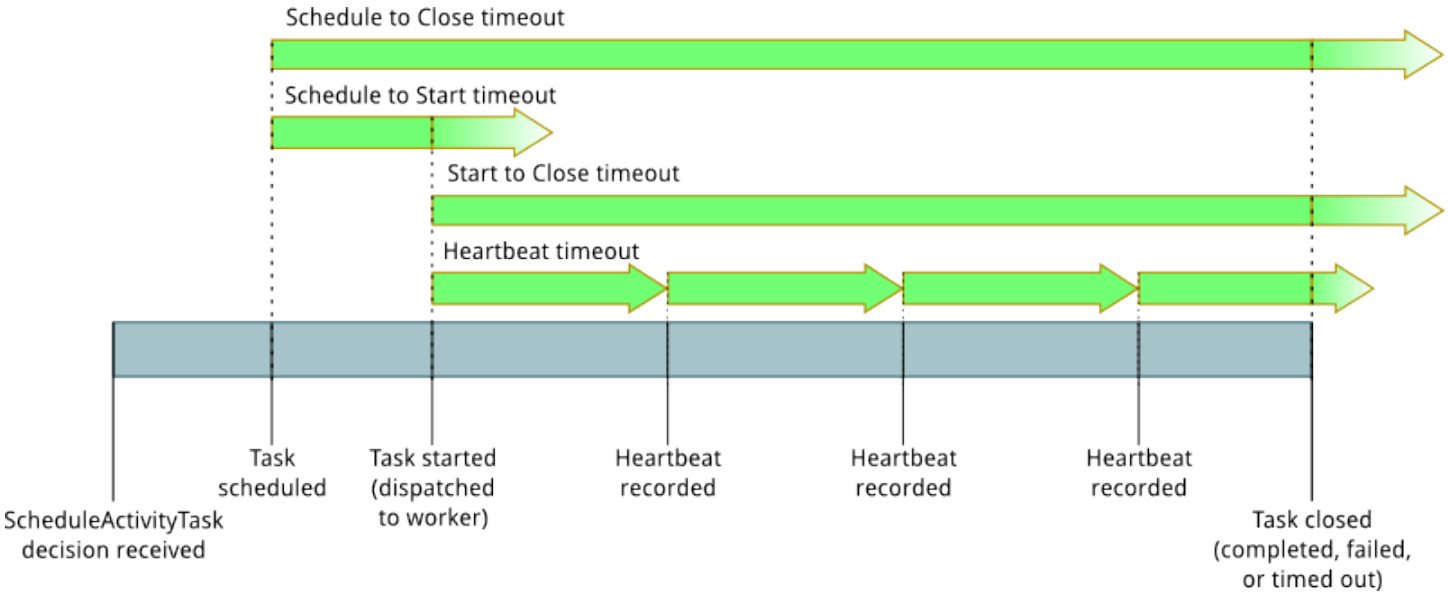


Il existe deux types de délai qui s'appliquent aux tâches du flux de travail et aux tâches de décision :

- Du début à la fermeture du flux de travail (**`timeoutType: START_TO_CLOSE`**) : ce délai indique la durée maximale d'exécution d'un flux de travail. Il est défini comme valeur par défaut lors de l'enregistrement du flux de travail, mais peut être remplacé par une autre valeur lorsque le flux de travail est lancé. Si ce délai est dépassé, Amazon SWF ferme l'exécution du flux de travail et ajoute un événement de WorkflowExecutionTimedOuttype à l'historique d'exécution du flux de travail. Outre `timeoutType`, les attributs de l'événement spécifient la stratégie `childPolicy` qui est en vigueur pour cette exécution de flux de travail. La stratégie enfant définit comment les exécutions de flux de travail enfant sont gérées si l'exécution de flux de travail parent expire ou si elle est arrêtée. Par exemple, si la stratégie `childPolicy` est définie sur TERMINATE, les exécutions de flux de travail enfant sont arrêtées. Une fois qu'une exécution de flux de travail expire, vous ne pouvez plus effectuer que les appels de visibilité.
- Début de la fin de la tâche de décision (**`timeoutType: START_TO_CLOSE`**) : ce délai indique le temps maximum que le décideur correspondant peut prendre pour terminer une tâche de décision. Il est défini lors de l'enregistrement du type de flux de travail. Si ce délai est dépassé, la tâche est marquée comme expirée dans l'historique d'exécution du flux de travail, et Amazon SWF ajoute un événement de DecisionTaskTimedOuttype à l'historique du flux de travail. Les attributs de l'événement incluront IDs les événements correspondant au moment où cette tâche de décision a été planifiée (`scheduledEventId`) et à son lancement (`startedEventId`). Outre l'ajout de l'événement, Amazon SWF planifie également une nouvelle tâche de décision pour avertir le décideur que le délai imparti pour cette tâche de décision a expiré. Après expiration de ce délai, toute tentative de finalisation de la décision avec `RespondDecisionTaskCompleted` échoue.

Délais des tâches d'activité

Le schéma suivant illustre le lien entre les délais d'attente et la durée de vie d'une tâche d'activité :



Quatre types de délai s'appliquent aux tâches d'activité :

- Début de la fin de la tâche d'activité (**`timeoutType: START_TO_CLOSE`**) : ce délai indique le temps maximum qu'un agent d'activité peut prendre pour traiter une tâche une fois qu'il l'a reçue. Les tentatives de clôture d'une tâche d'activité dont le délai imparti a expiré à l'aide de RespondActivityTaskCanceledRespondActivityTaskCompleted, et RespondActivityTaskFailedéchoueront.
- Activity Task Heartbeat (**`timeoutType: HEARTBEAT`**) : ce délai indique la durée maximale pendant laquelle une tâche peut être exécutée avant de fournir sa progression au cours de l'`RecordActivityTaskHeartbeat`action.
- Planification des tâches d'activité jusqu'au début (**`timeoutType: SCHEDULE_TO_START`**) : ce délai indique la durée pendant laquelle Amazon SWF attend avant de mettre fin à la tâche d'activité si aucun collaborateur n'est disponible pour effectuer la tâche. Lorsque la tâche arrive à expiration, elle n'est pas attribuée à un autre outil de traitement.
- Calendrier de clôture des tâches d'activité (**`timeoutType: SCHEDULE_TO_CLOSE`**) : ce délai indique le temps que la tâche peut prendre entre le moment où elle est planifiée et le moment où elle est terminée. Il est recommandé que cette valeur ne soit pas supérieure à la somme du délai d'expiration de la tâche et du schedule-to-start délai d'expiration de la tâche start-to-close.

> **Note**
>
> Chacun des types de délai a une valeur par défaut, généralement définie sur `NONE` (infini). Toutefois, la durée maximale de toute exécution d'activité est limitée à un an.

Vous définissez ces valeurs par défaut lors de l'enregistrement du type d'activité, mais vous pouvez les remplacer par d'autres valeurs lorsque vous planifiez la tâche d'activité. Lorsque l'un de ces délais se produit, Amazon SWF ajoute un événement de ActivityTaskTimedOuttype à l'historique du flux de travail. La valeur d'attribut `timeoutType` de cet événement indique quel délai a expiré. Pour chaque délai, la valeur `timeoutType` est indiquée entre parenthèses. Les attributs d'événement incluront également IDs les événements correspondant au moment où la tâche d'activité a été planifiée (`scheduledEventId`) et à son lancement (`startedEventId`). Outre l'ajout de l'événement, Amazon SWF planifie également une nouvelle tâche de décision pour avertir le décideur que le délai imparti s'est écoulé.

Comprendre une tâche dans AWS Flow Framework for Java

Rubriques

- Tâche
- Ordre d'exécution
- Exécution de flux de travail
- Non-déterminisme

Tâche

La primitive sous-jacente que Java utilise AWS Flow Framework pour gérer l'exécution du code asynchrone est la `Task` classe. Un objet de type `Task` représente le travail qui doit être effectué de manière asynchrone. Lorsque vous appelez une méthode asynchrone, l'infrastructure crée un objet `Task` pour exécuter le code dans cette méthode et le place dans une liste pour une exécution ultérieure. De même, lorsque vous appelez un objet `Activity`, un objet `Task` est créé. L'appel de méthode revient après cela, en revoyant généralement un objet `Promise<T>` comme futur résultat de l'appel.

La classe `Task` est publique et peut être utilisée directement. Par exemple, nous pouvons réécrire l'exemple Hello World pour utiliser un objet `Task` à la place d'une méthode asynchrone.

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

L'infrastructure appelle la méthode `doExecute()` lorsque tous les objets `Promise` transmis au constructeur de l'objet `Task` sont prêts. Pour plus de détails sur la `Task` classe, consultez la AWS SDK pour Java documentation.

L'infrastructure inclut également une classe appelée `Functor` qui représente un objet `Task` qui est également un objet `Promise<T>`. L'objet `Functor` est prêt lorsque l'objet `Task` est terminé. Dans l'exemple suivant, un objet `Functor` est créé pour récupérer le message d'accueil :

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

Ordre d'exécution

Les tâches deviennent éligibles à l'exécution uniquement lorsque tous les paramètres de type `Promise<T>`, transmis à la méthode ou activité asynchrone correspondante, sont prêts. Un objet `Task` prêt pour l'exécution est logiquement déplacé dans une file d'attente des processus prêts. En d'autres termes, elle est planifiée pour l'exécution. La classe de travail exécute la tâche en invoquant le code que vous avez écrit dans le corps de la méthode asynchrone ou en planifiant une tâche d'activité dans Amazon Simple Workflow Service (AWS) dans le cas d'une méthode d'activité.

À mesure que les tâches s'exécutent et produisent des résultats, d'autres tâches deviennent prêtes et l'exécution du programme se poursuit. La manière dont l'infrastructure exécute les tâches est essentielle pour comprendre l'ordre dans lequel votre code asynchrone s'exécute. Un code qui apparaît séquentiellement dans votre programme peut ne pas s'exécuter dans cette ordre.

```
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!");
}
```

Le code dans la liste ci-dessus imprimera les éléments suivants :

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

Cela peut différer de vos attentes mais peut s'expliquer aisément en réfléchissant à la manière dont les tâches ont été exécutées pour les méthodes asynchrones :

1. L'appel à `getUserName` crée un objet `Task`. Appelons-le `Task1`. Parce qu'`getUserName`il ne prend aucun paramètre, `Task1` il est immédiatement placé dans la file d'attente prête.
2. Ensuite, l'appel à `printHelloName` crée un objet `Task` qui doit attendre le résultat de `getUserName`. Appelons-le `Task2`. Comme la valeur requise n'est pas encore prête, elle `Task2` est ajoutée à la liste d'attente.
3. Ensuite, une tâche pour `printHelloWorld` est créée et ajoutée à la file d'attente des processus prêts. Appelons-le `Task3`.
4. La `println` déclaration affiche ensuite « Hello, Amazon ! » à la console.
5. À ce stade, les objets `Task1` et `Task3` sont placés dans la file d'attente des processus prêts et l'objet `Task2` dans la file d'attente.
6. L'application de travail exécute `Task1`, et son résultat rend `Task2` prêt. `Task2` se retrouve ajoutée à la file d'attente derrière `Task3`.
7. Les objets `Task3` et `Task2` sont ensuite exécutés dans cette ordre.

L'exécution des activités suit le même modèle. Lorsque vous appelez une méthode sur le client d'activité, celle-ci crée une méthode `Task` qui, lors de son exécution, planifie une activité dans Amazon SWF.

L'infrastructure s'appuie sur des fonctions comme la génération de code et les proxys dynamiques pour injecter la logique afin de convertir les appels de méthode en appels d'activité et tâches asynchrones dans votre programme.

Exécution de flux de travail

L'exécution de l'implémentation de flux de travail est également gérée par la classe de l'exécuteur. Lorsque vous appelez une méthode sur le client de flux de travail, celui-ci appelle Amazon SWF pour créer une instance de flux de travail. Les tâches d'Amazon SWF ne doivent pas être confondues avec celles du framework. Dans Amazon SWF, une tâche est soit une tâche d'activité, soit une tâche de décision. L'exécution des tâches d'activité est simple. La classe Activity Worker reçoit les tâches d'activité d'Amazon SWF, invoque la méthode d'activité appropriée dans votre implémentation et renvoie le résultat à Amazon SWF.

L'exécution des tâches décisionnelles est plus impliquée. Le gestionnaire de flux de travail reçoit des tâches de décision d'Amazon SWF. Une tâche de décision demande à la logique de flux de travail ce qu'il faut faire ensuite. La première tâche de décision est générée pour une instance de flux de travail lorsqu'elle est lancée via le client de flux de travail. Lors de la réception de cette tâche de décision, l'infrastructure lance l'exécution du code dans la méthode de flux de travail annotée avec `@Execute`. Cette méthode exécute la logique de coordination qui planifie les activités. Lorsque l'état de l'instance de flux de travail change, par exemple lorsqu'une activité se termine, d'autres tâches décisionnelles sont planifiées. À ce stade, la logique de flux de travail peut décider d'entreprendre une action basée sur le résultat de l'activité ; par exemple, elle peut décider de planifier une autre activité.

L'infrastructure cache tous ces détails aux développeurs en traduisant sans heurts les tâches décisionnelles dans la logique de flux de travail. Du point de vue du développeur, le code ressemble à un programme normal. En guise de couverture, le framework l'associe aux appels à Amazon SWF et aux tâches de décision en utilisant l'historique géré par Amazon SWF. Lorsqu'une tâche de décision arrive, l'infrastructure reproduit la connexion de l'exécution du programme dans les résultats des activités terminées jusqu'à présent. Les méthodes et activités asynchrones qui attendaient ces résultats se débloquent, et l'exécution du programme se poursuit.

L'exécution de flux de travail de traitement de l'exemple d'image et de l'historique correspondant est illustrée dans le tableau suivant.

Exécution de flux de travail miniatures

Exécution du programme de flux de travail	Historique géré par Amazon SWF
Exécution initiale	
1. Boucle de distribution 2. getImageUrls	1. Instance de flux de travail lancée, id="1" 2. downloadImage planifié

Exécution du programme de flux de travail	Historique géré par Amazon SWF
3. downloadImage 4. createThumbnail (tâche dans file d'attente) 5. uploadImage (tâche dans file d'attente) 6. <prochaine itération de la boucle>	
Relire	
1. Boucle de distribution 2. getImageUrls 3. Chemin downloadImage image="foo" 4. createThumbnail 5. uploadImage (tâche dans file d'attente) 6. <prochaine itération de la boucle>	1. Instance de flux de travail lancée, id="1" 2. downloadImage planifié 3. downloadImage terminé, retour="foo" 4. createThumbnail planifié
Relire	
1. Boucle de distribution 2. getImageUrls 3. Chemin downloadImage image="foo" 4. Chemin de miniatures createThumbnail="bar" 5. uploadImage 6. <prochaine itération de la boucle>	1. Instance de flux de travail lancée, id="1" 2. downloadImage planifié 3. downloadImage terminé, retour="foo" 4. createThumbnail planifié 5. createThumbnail terminé, retour="bar" 6. uploadImage planifié
Relire	

Exécution du programme de flux de travail	Historique géré par Amazon SWF
1. Boucle de distribution 2. getImageUrls 3. Chemin downloadImage image="foo" 4. Chemin de miniatures createThumbnail="bar" 5. uploadImage 6. <prochaine itération de la boucle>	1. Instance de flux de travail lancée, id="1" 2. downloadImage planifié 3. downloadImage terminé, retour="foo" 4. createThumbnail planifié 5. createThumbnail terminé, retour="bar" 6. uploadImage planifié 7. uploadImage terminé ...

Lorsqu'un appel `processImage` est effectué, le framework crée une nouvelle instance de flux de travail dans Amazon SWF. Il s'agit d'un enregistrement durable de l'instance de flux de travail lancée. Le programme s'exécute jusqu'à l'appel à l'`downloadImage`activité, qui demande à Amazon SWF de planifier une activité. Le flux de travail poursuit son exécution et crée des tâches pour les activités suivantes, mais elles ne peuvent pas être exécutées tant que l'`downloadImage`activité n'est pas terminée ; cet épisode de rediffusion prend donc fin. Amazon SWF répartit la tâche en fonction de l'`downloadImage`activité pour exécution, et une fois qu'elle est terminée, un enregistrement est enregistré dans l'historique avec le résultat. Le flux de travail est maintenant prêt à avancer et une tâche de décision est générée par Amazon SWF. L'infrastructure reçoit la tâche de décision et reproduit la connexion de flux de travail dans le résultat de l'image téléchargée comme enregistré dans l'historique. Cela permet de débloquer la tâche et de `createThumbnail` poursuivre l'exécution du programme en planifiant la tâche `createThumbnail` d'activité dans Amazon SWF. Le même processus se répète pour `uploadImage`. L'exécution du programme continue de cette façon jusqu'à ce que le flux de travail ait traité toutes les images et aucune tâche n'est en attente. Comme aucun état d'exécution n'est stocké localement, chaque tâche de décision peut être exécutée sur une machine différente. Cela vous permet d'écrire facilement des programmes tolérants aux pannes et évolutifs.

Non-déterminisme

Comme le framework repose sur le replay, il est important que le code d'orchestration (tout le code du flux de travail à l'exception des implémentations d'activités) soit déterministe. Par exemple, le flux de contrôle dans votre programme ne doit pas dépendre d'un nombre aléatoire ou de l'heure actuelle.

Comme ces éléments peuvent changer entre les invocations, il est possible que la rediffusion ne suive pas le même chemin dans la logique d'orchestration. Cela entraînera des résultats inattendus ou des erreurs. L'infrastructure fournit un objet `WorkflowClock` que vous pouvez utiliser pour obtenir l'heure actuelle de manière déterministe. Pour en savoir plus, consultez la section sur Contexte d'exécution.

> **Note**
>
> Une connexion Spring incorrecte des objets d'implémentation de flux de travail peut également mener à un non-déterminisme . Les beans d'implémentations de flux de travail ainsi que les beans dont ils dépendent doivent être dans la portée de flux de travail (`WorkflowScope`). Par exemple, la connexion d'un bean d'implémentation de flux de travail à un bean qui conserve un état et se trouve dans un contexte global entraîne un comportement inattendu. Pour en savoir plus, consultez la section Intégration de Spring.

AWS Flow Framework pour le guide de programmation Java

Cette section explique comment utiliser les fonctionnalités de AWS Flow Framework for Java pour implémenter des applications de flux de travail.

Rubriques

- Implémentation d'applications de flux de travail avec AWS Flow Framework
- Contrats de flux de travail et d'activité
- Enregistrement des types de flux de travail et d'activité
- Clients d'activité et de flux de travail
- Implémentation de flux de travail
- Implémentation d'activité
- Mise en œuvre AWS Lambda des tâches
- Exécution de programmes écrits avec le AWS Flow Framework pour Java
- Contexte d'exécution
- Exécutions de flux de travail enfant
- Flux de travail continus
- Définition de la priorité des tâches dans Amazon SWF
- DataConverters
- Transmission des données aux méthodes asynchrones
- Testabilité et injection de dépendances
- Gestion des erreurs
- Relance des activités ayant échoué
- Tâches démon
- AWS Flow Framework pour Java Replay Behavior

Implémentation d'applications de flux de travail avec AWS Flow Framework

Les étapes typiques du développement d'un flux de travail avec le AWS Flow Framework sont les suivantes :

1. Définir les contrats d'activité et de flux de travail. Analysez les exigences de votre application, puis déterminez les activités et la topologie de flux de travail requises. Les activités gèrent les tâches de traitement requises, tandis que la topologie de flux de travail définit la structure de base de flux de travail et la logique métier.

   Prenons l'exemple d'une application de traitement multimédia ayant besoin de télécharger un fichier, de le traiter, puis de charger le fichier traité dans un compartiment Amazon Simple Storage Service (S3). Cette opération peut être divisée en quatre tâches d'activités :

   1. Télécharger le fichier à partir d'un serveur
   2. Traiter le fichier (par exemple, en le transcodant dans un format multimédia différent)
   3. Charger le fichier dans le compartiment S3
   4. Effectuer un nettoyage en supprimant les fichiers locaux

   Ce flux de travail aurait une méthode de point d'entrée et implémenterait une topologie linéaire simple qui exécute les activités séquentiellement, à l'image de la HelloWorldWorkflow Demande.
2. Implémenter les interfaces d'activité et de flux de travail. Les contrats d'activité et de flux de travail sont définis par des interfaces Java, en rendant leurs conventions d'appel prévisibles par SWF, et en vous offrant de la flexibilité lorsque vous implémentez votre logique de flux de travail et vos tâches d'activité. Les différentes parties de votre programme peuvent utiliser les données des autres, mais n'ont pas besoin d'être au courant des détails d'implémentation des autres parties.

   Par exemple, vous pouvez définir une interface `FileProcessingWorkflow` et fournir différentes implémentations de flux de travail pour l'encodage vidéo, la compression, les miniatures, etc. Chacun de ces flux de travail peut avoir différents flux de contrôle et peut appeler différentes méthodes d'activité ; votre démarreur de flux de travail n'a pas besoin de savoir. Grâce à des interfaces, vous pouvez aussi tester simplement vos flux de travail en utilisant des implémentations fictives qui peuvent être remplacées ultérieurement avec du code pratique.
3. Générer des clients d'activité et de flux de travail. Cela vous AWS Flow Framework évite d'avoir à implémenter les détails de la gestion de l'exécution asynchrone, de l'envoi de requêtes HTTP, du marshaling des données, etc. À la place, le démarreur de flux de travail crée une instance de flux de travail en appelant une méthode sur le client de flux de travail, et l'implémentation du flux de travail exécute les activités en appelant des méthodes sur le client d'activité. L'infrastructure gère les détails de ces interactions en arrière-plan.

Si vous utilisez Eclipse et que vous avez configuré votre projet, comme dansConfiguration du AWS Flow Framework pour Java, le processeur AWS Flow Framework d'annotations utilise les définitions d'interface pour générer automatiquement des clients de flux de travail et d'activités qui exposent le même ensemble de méthodes que l'interface correspondante.

4. Implémenter les applications hôtes d'activité et de flux de travail. Vos implémentations de flux de travail et d'activités doivent être intégrées dans les applications hôtes qui interrogent Amazon SWF pour les tâches, rassemblent les données et appellent les méthodes d'implémentation appropriées. AWS Flow Framework pour Java inclut WorkflowWorkeret ActivityWorkerclasse qui simplifient et simplifient la mise en œuvre d'applications hôtes.
5. Testez votre flux de travail. AWS Flow Framework for Java fournit une JUnit intégration que vous pouvez utiliser pour tester vos flux de travail en ligne et localement.
6. Déployer les exécuteurs. Vous pouvez déployer vos employés comme il convient. Par exemple, vous pouvez les déployer sur des EC2 instances Amazon ou sur des ordinateurs de votre centre de données. Une fois le déploiement et le démarrage terminés, les opérateurs commencent à interroger Amazon SWF pour les tâches et à les gérer selon les besoins.
7. Lancer des exécutions. Une application lance une instance de flux de travail à l'aide du client de flux de travail pour appeler le point d'entrée de flux de travail. Vous pouvez également démarrer des flux de travail à l'aide de la console Amazon SWF. Quelle que soit la manière dont vous démarrez une instance de flux de travail, vous pouvez utiliser la console Amazon SWF pour surveiller l'instance de flux de travail en cours d'exécution et examiner l'historique des instances en cours d'exécution, terminées et ayant échoué.

AWS SDK pour JavaIl inclut un ensemble AWS Flow Framework de quatre exemples Java que vous pouvez parcourir et exécuter en suivant les instructions du fichier readme.html situé dans le répertoire racine. Il existe également un ensemble de recettes, des applications simples, qui montrent comment traiter divers problèmes de programmation spécifiques, disponibles sur AWS Flow Framework Recipes.

Contrats de flux de travail et d'activité

Les interfaces Java sont utilisées pour déclarer les signatures des flux de travail et activités. L'interface forme le contrat entre l'implémentation de flux de travail (ou activité) et le client de ce flux de travail (ou activité). Par exemple, un type de flux de travail `MyWorkflow` est défini à l'aide d'une interface annotée avec `@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();
}
```

Le contrat ne possède aucun paramètre propre à l'implémentation. Cette utilisation des contrats à implémentation neutre permet aux clients d'être découplés de l'implémentation et donc fournit la flexibilité pour modifier les détails d'implémentation sans casser le client. À l'inverse, vous pouvez également modifier le client sans modifier le flux de travail ou l'activité en cours d'utilisation. Par exemple, le client peut être modifié pour appeler une activité de manière asynchrone à l'aide d'objets Promises (`Promise<T>`) sans que l'implémentation d'activité ait besoin d'être modifiée. De même, la mise en œuvre de l'activité peut être modifiée afin qu'elle soit réalisée de manière asynchrone, par exemple, par une personne envoyant un e-mail, sans qu'il soit nécessaire de modifier les clients de l'activité.

Dans l'exemple ci-dessus, l'interface de flux de travail `MyWorkflow` contient une méthode, `startMyWF`, pour lancer une nouvelle exécution. Cette méthode est annotée avec `@Execute` et doit posséder un type de retour `void` ou `Promise<>`. Dans une interface de flux de travail donnée, une méthode maximum peut être annotée avec cette annotation. Cette méthode constitue le point d'entrée de la logique de flux de travail, et l'infrastructure appelle cette méthode pour exécuter la logique de flux de travail lorsqu'une tâche de décision est reçue.

L'interface de flux de travail définit également les signaux qui peuvent être envoyés au flux de travail. La méthode de signal est appelée lorsqu'un signal avec un nom correspondant est reçu par l'exécution de flux de travail. Par exemple, l'interface `MyWorkflow` déclare une méthode de signal, `signal1`, annotée avec `@Signal`.

L'annotation `@Signal` est requise sur les méthodes de signal. Le type de retour d'une méthode de signal doit être `void`. Une interface de flux de travail peut comporter zéro ou plusieurs méthodes de

signal définies. Vous pouvez déclarer une interface de flux de travail sans une méthode `@Execute` et certaines méthodes `@Signal` pour générer des clients qui ne peuvent pas lancer leur exécution mais qui peuvent envoyer des signaux pour lancer des exécutions.

Les méthodes annotées avec `@Execute` et `@Signal` peuvent disposer d'un nombre de paramètres de tout type autre que `Promise<T>` ou ses dérivés. Cela vous permet de transmettre des entrées fortement typées à une exécution de flux de travail au lancement et pendant l'exécution. Le type de retour de la méthode `@Execute` doit être `void` ou `Promise<>`.

De plus, vous pouvez également déclarer une méthode dans l'interface de flux de travail pour signaler le dernier état d'une exécution de flux de travail, par exemple, la méthode `getState` de l'exemple précédent. Cet état ne représente pas l'état entier de l'application du flux de travail. L'utilisation souhaitée de cette fonction est de vous permettre de stocker jusqu'à 32 Ko de données pour indiquer le dernier état de l'exécution. Par exemple, dans un flux de travail de traitement des commandes, vous pouvez stocker une chaîne qui indique que la commande a été reçue, traitée ou annulée. Cette méthode est appelée par l'infrastructure chaque fois qu'une tâche de décision est terminée pour obtenir le dernier état. L'état est stocké dans Amazon Simple Workflow Service (Amazon SWF) et peut être récupéré à l'aide du client externe généré. Cela vous permet de vérifier le dernier état d'une exécution de flux de travail. Les méthodes annotées avec `@GetState` ne doivent pas prendre n'importe quel argument et ne doivent pas disposer d'un type de retour `void`. À partir de cette méthode, vous pouvez renvoyer n'importe quel type correspondant à vos besoins. Dans l'exemple ci-dessus, un objet `MyWorkflowState` (consultez la définition ci-dessous) est renvoyé par la méthode utilisée pour stocker une chaîne correspondant à l'état et un pourcentage numérique d'exécution. La méthode est censée effectuer un accès en lecture seule de l'objet d'implémentation de flux de travail et est appelée de manière synchrone, ce qui n'autorise pas l'utilisation de toute opération asynchrone comme l'appel de méthodes annotées avec `@Asynchronous`. Une méthode maximum dans une interface de flux de travail peut être annotée avec `@GetState`.

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

De même, un ensemble d'activités est défini à l'aide d'une interface annotée avec `@Activities`. Chaque méthode de l'interface correspond à une activité, par exemple :

```
@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'interface vous permet de regrouper un ensemble d'activités associées. Vous pouvez définir n'importe quel nombre d'activités dans une interface d'activité, et autant d'interfaces d'activité que vous le souhaitez. De même que pour les méthodes `@Execute` et `@Signal`, les méthodes d'activité peuvent prendre n'importe quel nombre d'arguments de tout type autre que `Promise<T>` ou ses dérivés. Le type de retour d'une activité ne doit pas être `Promise<T>` ou ses dérivés.

Enregistrement des types de flux de travail et d'activité

Amazon SWF exige que les types d'activité et de flux de travail soient enregistrés avant de pouvoir être utilisés. L'infrastructure enregistre automatiquement les flux de travail et les activités dans les implémentations que vous ajoutez à l'exécuteur. Le framework recherche les types qui implémentent des flux de travail et des activités et les enregistre auprès d'Amazon SWF. Par défaut, l'infrastructure utilise les définitions d'interface pour déduire des options d'enregistrement pour les types de flux de travail et d'activité. Toutes les interfaces de flux de travail doivent avoir l'annotation `@WorkflowRegistrationOptions` ou l'annotation `@SkipRegistration`. L'exécuteur de flux de travail enregistre tous les types de flux de travail avec lesquels il est configuré qui ont l'annotation `@WorkflowRegistrationOptions`. De même, chaque méthode d'activité doit être annotée avec l'annotation `@ActivityRegistrationOptions` ou l'annotation `@SkipRegistration`, ou l'une de ces annotations doivent être présente dans l'interface `@Activities`. L'exécuteur d'activité enregistre tous les types d'activité avec lesquelles il est configuré auxquelles une annotation `@ActivityRegistrationOptions` s'applique. L'enregistrement est effectué automatiquement lorsque vous démarrez l'un des exécuteurs. Les flux de travail et types d'activité ayant l'annotation `@SkipRegistration` ne sont pas enregistrés. Les annotations `@ActivityRegistrationOptions` et `@SkipRegistration` remplacent les annotations sémantiques et la plus spécifique est appliquée à un type d'activité.

Notez qu'Amazon SWF ne vous permet pas de réenregistrer ou de modifier le type une fois celui-ci enregistré. L'infrastructure tente d'enregistrer tous les types, mais si le type est déjà enregistré, celui-ci n'est pas réenregistré et aucune erreur n'est signalée.

Si vous devez modifier des paramètres enregistrés, vous devez enregistrer une nouvelle version du type. Vous pouvez également remplacer des paramètres enregistrés quand vous démarrez une nouvelle exécution ou quand vous appelez une activité qui utilise les clients générés.

L'enregistrement nécessite un nom de type et d'autres options d'enregistrement. L'implémentation par défaut détermine ces options comme suit :

Nom et version de type de flux de travail

L'infrastructure détermine le nom du type de flux de travail à partir de l'interface de flux de travail. La forme du nom du type de flux de travail par défaut est {*prefix*} {*name*}. Le {*prefix*} est défini sur le nom de l'`@Workflow`interface suivi d'un «. » et le {*name*} est défini sur le nom de la `@Execute` méthode. Le nom par défaut du type de flux de travail dans l'exemple précédent est `MyWorkflow.startMyWF`. Vous pouvez remplacer le nom par défaut à l'aide du paramètre de nom de la méthode `@Execute`. Le nom par défaut du type de flux de travail dans l'exemple est `startMyWF`. Le nom ne doit pas être une chaîne vide. Notez que lorsque vous remplacez le nom à l'aide de la méthode `@Execute`, l'infrastructure ne le fait pas précéder automatiquement par une préfixe. Vous êtes libre d'utiliser votre propre schéma d'attribution de noms.

La version de flux de travail est spécifiée à l'aide du paramètre `version` de l'annotation `@Execute`. Il n'existe pas de valeur par défaut pour `version` et celle-ci doit être spécifiée explicitement ; `version` est une chaîne de forme libre, et vous pouvez utiliser votre propre schéma de gestion des versions.

Nom du signal

La nom du signal peut être spécifié à l'aide du paramètre de nom de l'annotation `@Signal`. S'il n'est pas spécifié, il prend par défaut le nom de la méthode signal.

Nom et version de type de flux d'activité

L'infrastructure détermine le nom du type d'activité à partir de l'interface d'activités. La forme du nom du type d'activité par défaut est {*prefix*} {*name*}. Le {*prefix*} est défini sur le nom de l'`@Activities`interface suivi d'un «. » et le {*name*} est défini sur le nom de la méthode. La valeur par défaut {*prefix*} peut être remplacée dans l'`@Activities`annotation de l'interface des activités. Vous pouvez également spécifier le nom de type d'activité à l'aide de l'annotation `@Activity` sur la méthode d'activité. Notez que lorsque vous remplacez le nom à l'aide de la méthode `@Activity`,

l'infrastructure ne le fera pas précéder automatiquement par une préfixe. Vous êtes libre d'utiliser votre propre schéma d'attribution de noms.

La version d'activité est spécifiée à l'aide du paramètre de version de l'annotation `@Activities`. Cette version est utilisée comme version par défaut pour toutes les activités définies dans l'interface et peut être remplacée pour une activité individuelle à l'aide de l'annotation `@Activity`.

Default Task List

La liste de tâches par défaut peut être configurée en utilisant les annotations `@WorkflowRegistrationOptions` et `@ActivityRegistrationOptions`, et en définissant le paramètre `defaultTaskList`. Par défaut, l'attribut est défini sur `USE_WORKER_TASK_LIST`. Il s'agit d'une valeur spéciale qui demande à l'infrastructure d'utiliser la liste de tâches configurée sur l'exécuteur qui est utilisé pour enregistrer le type d'activité ou de flux de travail. Vous pouvez également choisir de ne pas enregistrer de liste de tâches par défaut en définissant la liste de tâches par défaut sur `NO_DEFAULT_TASK_LIST` à l'aide de ces annotations. Vous pouvez utiliser cette option si vous souhaitez que la liste de tâches soit spécifiée lors de l'exécution. Si aucune liste de tâches n'a été enregistrée, vous devez spécifier la liste de tâches lorsque vous démarrez le flux de travail ou que vous appelez la méthode d'activité à l'aide des paramètres `StartWorkflowOptions` et `ActivitySchedulingOptions` sur la surcharge des méthodes respectives du client généré.

Autres options d'enregistrement

Toutes les options d'enregistrement des flux de travail et des types d'activités autorisées par l'API Amazon SWF peuvent être spécifiées via le framework.

Pour accéder à une liste complète des options d'enregistrement de flux de travail, consultez les sections suivantes :

- @Flux de travail
- @Execute
- @WorkflowRegistrationOptions
- @Signal

Pour accéder à une liste complète des options d'enregistrement d'activité, consultez les sections suivantes :

- @Activité

- @Activités
- @ActivityRegistrationOptions

Si vous voulez exercer un contrôle complet sur le type d'enregistrement, consultez Extensibilité de l'exécuteur.

Clients d'activité et de flux de travail

Les clients d'activité et de flux de travail sont générés par l'infrastructure sur la base des interfaces `@Workflow` et `@Activities`. Des interfaces client séparées sont générées. Elles contiennent des méthodes et des paramètres qui n'ont de sens que pour le client. Si vous développez à l'aide d'Eclipse, cela est effectué par le plug-in Amazon SWF Eclipse chaque fois que vous enregistrez le fichier contenant l'interface appropriée. Le code généré est placé dans le répertoire des sources générées de votre projet, dans le même package que l'interface.

> **Note**
>
> Notez que le nom de répertoire par défaut utilisé par Eclipse est .apt_generated. Eclipse ne montre pas les répertoires dont le nom commence par un « . » dans Package Explorer. Utilisez un autre nom de répertoire pour afficher les fichiers générés dans Project Explorer. Dans Eclipse, cliquez avec le bouton droit sur le package dans Package Explorer, sélectionnez Properties (Propriétés), Java Compiler (Compilateur Java), Annotation processing (Traitement des annotations), puis modifiez le paramètre Generate source directory (Générer le répertoire des sources).

Clients de flux de travail

Les artefacts générés pour le flux de travail contiennent trois interfaces côté client et les classes qui les implémentent. Les clients générés sont les suivants :

- Un client asynchrone destiné à être consommé à partir d'une implémentation de flux de travail qui fournit des méthodes asynchrones pour démarrer les exécutions de flux de travail et envoyer des signaux
- Un client externe qui peut être utilisé pour démarrer des exécutions, envoyer des signaux et récupérer l'état du flux de travail en dehors du cadre d'une implémentation de flux de travail
- Un client auto-généré qui peut être utilisé pour créer des flux de travail continus

Par exemple, les interfaces client générées pour l'exemple d'interface `MyWorkflow` sont :

```
//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);
```

Les interfaces ont des méthodes surchargées correspondant à chaque méthode dans l'interface `@Workflow` que vous avez déclarée.

Le client externe reflète les méthodes de l'interface `@Workflow` avec une surcharge supplémentaire de la méthode `@Execute` qui prend `StartWorkflowOptions`. Vous pouvez utiliser cette surcharge

pour transmettre des options supplémentaires lors du lancement d'une nouvelle exécution de flux de travail. Ces options vous permettent de remplacer la liste de tâches par défaut, les paramètres de délai d'attente et d'associer des balises à l'exécution du flux de travail.

D'autre part, le client asynchrone dispose de méthodes qui permettent d'effectuer un appel asynchrone de la méthode `@Execute`. Les surcharges de méthode suivantes sont générées dans l'interface client pour la méthode `@Execute` dans l'interface de flux de travail :

1. Une surcharge qui prend les arguments initiaux en l'état. Le type de retour de cette surcharge sera `Promise<Void>` si la méthode initiale a renvoyé `void`. Sinon, ce sera `Promise<>`, tel que déclaré sur la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

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

   Cette surcharge doit être utilisée lorsque tous les arguments du flux de travail sont disponibles et qu'il n'est pas nécessaire de les attendre.
2. Une surcharge qui prend les arguments initiaux en l'état et des arguments variables supplémentaires de type `Promise<?>`. Le type de retour de cette surcharge sera `Promise<Void>` si la méthode initiale a renvoyé `void`. Sinon, ce sera `Promise<>`, tel que déclaré sur la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

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

   Cette surcharge doit être utilisée lorsque tous les arguments du flux de travail sont disponibles et qu'il n'est pas nécessaire de les attendre, mais que vous voulez attendre que d'autres objets Promise soient prêts. L'argument variable peut être utilisé pour transmettre ces objets `Promise<?`

> qui n'ont pas été déclarés comme arguments, mais que vous voulez attendre avant d'exécuter l'appel.

3. Une surcharge qui prend les arguments initiaux en l'état, un argument supplémentaire de type `StartWorkflowOptions` et des arguments variables supplémentaires de type `Promise<?>`. Le type de retour de cette surcharge sera `Promise<Void>` si la méthode initiale a renvoyé `void`. Sinon, ce sera `Promise<>`, tel que déclaré sur la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

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

   Cette surcharge doit être utilisée lorsque tous les arguments du flux de travail sont disponibles et qu'il n'est pas nécessaire de les attendre, lorsque vous voulez remplacer les paramètres par défaut utilisés pour démarrer l'exécution du flux de travail ou lorsque vous voulez attendre que d'autres objets Promise soient prêts. L'argument variable peut être utilisé pour transmettre ces objets `Promise<?>` qui n'ont pas été déclarés comme arguments, mais que vous voulez attendre avant d'exécuter l'appel.

4. Une surcharge dont chaque argument de la méthode initiale est remplacé par un wrapper `Promise<>`. Le type de retour de cette surcharge sera `Promise<Void>` si la méthode initiale a renvoyé `void`. Sinon, ce sera `Promise<>`, tel que déclaré sur la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

   ```
   Promise<Void> startMyWF(
      Promise<Integer> a,
   ```

```
   Promise<String> b);
```

Cette surcharge doit être utilisée lorsque les arguments à transmettre pour l'exécution du flux de travail doivent être évalués de manière asynchrone. Un appel à cette surcharge de méthode sera exécuté uniquement lorsque tous les arguments qui lui auront été transmis seront prêts.

Si certains arguments sont déjà prêts, convertissez-les en un `Promise` qui est déjà prêt à l'aide de la méthode `Promise.asPromise(`*`value`*`)`. Par exemple :

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

5. Une surcharge dont chaque argument de la méthode initiale est remplacé par un wrapper `Promise<>`. La surcharge a aussi des arguments variables supplémentaires de type `Promise<?>`. Le type de retour de cette surcharge sera `Promise<Void>` si la méthode initiale a renvoyé `void`. Sinon, ce sera `Promise<>`, tel que déclaré sur la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

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

   Cette surcharge doit être utilisée lorsque les arguments à transmettre à l'exécution du flux de travail doivent être évalués de manière asynchrone et que vous voulez attendre que d'autres objets Promise soient également prêts. Un appel à cette surcharge de méthode sera exécuté uniquement lorsque tous les arguments qui lui auront été transmis seront prêts.

6. Une surcharge dont chaque argument de la méthode initiale est remplacé par un wrapper `Promise<?>`. La surcharge a aussi un argument supplémentaire de type `StartWorkflowOptions` et des arguments variables de type `Promise<?>`. Le type de retour de cette surcharge sera `Promise<Void>` si la méthode initiale a renvoyé `void`. Sinon, ce sera `Promise<>`, tel que déclaré sur la méthode initiale. Par exemple :

Méthode initiale :

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

Méthode générée :

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

Utilisez cette surcharge lorsque les arguments à transmettre pour l'exécution du flux de travail seront évalués de manière asynchrone et que vous souhaitez remplacer les paramètres par défaut utilisés pour lancer l'exécution du flux de travail. Un appel à cette surcharge de méthode sera exécuté uniquement lorsque tous les arguments qui lui auront été transmis seront prêts.

Une méthode est également générée correspondant à chaque signal de l'interface du flux de travail, par exemple :

Méthode initiale :

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

Méthode générée :

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

Le client asynchrone ne contient pas de méthode correspondant à la méthode annotée avec `@GetState` dans l'interface initiale. Comme la récupération de l'état nécessite un appel de service Web, elle n'est pas adaptée à une utilisation dans un flux de travail. C'est pourquoi elle n'est fournie que par l'intermédiaire du client externe.

Le client auto-généré est destiné à être utilisé depuis un flux de travail pour démarrer une nouvelle exécution à la fin de l'exécution en cours. Les méthodes sur ce client sont similaires à celles qui sont utilisées sur le client asynchrone, mais elles renvoient `void`. Ce client n'a pas de méthodes

correspondant aux méthodes annotées avec `@Signal` et `@GetState`. Pour plus de détails, consultez Flux de travail continus.

Les clients générés dérivent des interfaces de base `WorkflowClient` et `WorkflowClientExternal`, respectivement. Elles fournissent des méthodes que vous pouvez utiliser pour annuler ou interrompre l'exécution du flux de travail. Pour plus de détails sur ces interfaces, consultez la documentation AWS SDK pour Java .

Les clients générés vous permettent d'interagir avec les exécutions de flux de travail d'une manière fortement typée. Une fois créée, une instance d'un client généré est liée à une exécution de flux de travail spécifique et ne peut être utilisée que pour cette exécution. En outre, l'infrastructure fournit également des clients dynamiques qui ne sont pas propres à une exécution ou à un type de flux de travail. Les clients générés s'appuient sur ce client. Vous pouvez également utiliser directement ces clients. Consultez la section Clients dynamiques.

L'infrastructure génère également des fabriques pour la création de clients fortement typés. Les fabriques de clients générées pour l'exemple d'interface `MyWorkflow` sont :

```
//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'interface de base `WorkflowClientFactory` est :

```
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);
}
```

Il est conseillé d'utiliser ces fabriques pour créer des instances du client. La fabrique vous permet de configurer le client générique (celui-ci doit être utilisé pour fournir une implémentation client personnalisée). `DataConverter` doit, pour sa part, être utilisé par le client pour rassembler les données, ainsi que les options utilisées pour démarrer l'exécution du flux de travail. Pour en savoir plus, consultez les sections DataConverters et Exécutions de flux de travail enfant. `StartWorkflowOptions`Il contient des paramètres que vous pouvez utiliser pour remplacer les valeurs par défaut, par exemple les délais d'expiration, spécifiées au moment de l'enregistrement. Pour plus de détails sur la `StartWorkflowOptions` classe, consultez la AWS SDK pour Java documentation.

Le client externe peut être utilisé pour lancer des exécutions de flux de travail en dehors d'un tel flux, tandis que le client asynchrone peut être utilisé pour lancer une exécution de flux de travail à partir du code inscrit dans un tel flux. Pour lancer une exécution, il suffit d'utiliser le client généré pour appeler la méthode qui correspond à la méthode annotée avec `@Execute` dans l'interface de flux de travail.

L'infrastructure génère également des classes d'implémentation pour les interfaces client. Ces clients créent et envoient des demandes à Amazon SWF pour effectuer l'action appropriée. La version client de la `@Execute` méthode lance une nouvelle exécution de flux de travail ou crée une exécution de

flux de travail secondaire à l'aide d'Amazon SWF APIs. De même, la version client de la `@Signal` méthode utilise Amazon SWF APIs pour envoyer un signal.

> **Note**
>
> Le client de flux de travail externe doit être configuré avec le client et le domaine Amazon SWF. Vous pouvez soit utiliser le constructeur client factory qui les prend en tant que paramètres, soit transmettre une implémentation client générique déjà configurée avec le client et le domaine Amazon SWF.
> L'infrastructure suit la hiérarchie des types de l'interface de flux de travail. Elle génère également des interfaces client pour les interfaces de flux de travail parent et en dérive.

Clients d'activité

Comme pour le client de flux de travail, un client est généré pour chaque interface annotée avec `@Activities`. Les artefacts générés incluent une interface côté client et une classe client. L'interface générée pour l'exemple d'interface `@Activities` ci-dessus (`MyActivities`) est la suivante :

```
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'interface contient un ensemble de méthodes surchargées correspondant à chaque méthode d'activité dans l'interface `@Activities`. Ces surcharges sont fournies pour des raisons pratiques et permettent des activités d'appel asynchrones. Pour chaque méthode d'activité de l'interface `@Activities`, les surcharges de méthode suivantes sont générées dans l'interface client :

1. Une surcharge qui prend les arguments initiaux en l'état. Le type de retour de cette surcharge est `Promise<`*T*`>`, où *T* est le type de retour de la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

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

   Cette surcharge doit être utilisée lorsque tous les arguments du flux de travail sont disponibles et qu'il n'est pas nécessaire de les attendre.

2. Une surcharge qui prend les arguments initiaux en l'état, un argument de type `ActivitySchedulingOptions` et des arguments variables supplémentaires de type `Promise<?>`. Le type de retour de cette surcharge est `Promise<`*T*`>`, où *T* est le type de retour de la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

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

   Cette surcharge doit être utilisée lorsque tous les arguments du flux de travail sont disponibles et qu'il n'est pas nécessaire de les attendre, lorsque vous voulez remplacer les paramètres par défaut, ou lorsque vous voulez attendre que des objets `Promise` supplémentaires soient prêts. Les arguments variables peuvent être utilisés pour transmettre des objets `Promise<?>`

supplémentaires qui n'ont pas été déclarés comme arguments, mais que vous voulez attendre avant d'exécuter l'appel.

3. Une surcharge dont chaque argument de la méthode initiale est remplacé par un wrapper `Promise<>`. Le type de retour de cette surcharge est `Promise<`*T*`>`, où *T* est le type de retour de la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

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

   Cette surcharge doit être utilisée lorsque les arguments à transmettre à l'activité seront évalués de manière asynchrone. Un appel à cette surcharge de méthode sera exécuté uniquement lorsque tous les arguments qui lui auront été transmis seront prêts.

4. Une surcharge dont chaque argument de la méthode initiale est remplacé par un wrapper `Promise<>`. La surcharge a aussi un argument supplémentaire de type `ActivitySchedulingOptions` et des arguments variables de type `Promise<?>`. Le type de retour de cette surcharge est `Promise<`*T*`>`, où *T* est le type de retour de la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

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

   Cette surcharge doit être utilisée lorsque les arguments à transmettre à l'activité seront évalués de manière asynchrone, lorsque vous voulez remplacer les paramètres par défaut enregistrés avec le type, ou lorsque vous voulez attendre que des objets `Promise` supplémentaires soient prêts.

Un appel à cette surcharge de méthode sera exécuté uniquement lorsque tous les arguments qui lui auront été transmis seront prêts. La classe du client généré implémente cette interface. L'implémentation de chaque méthode d'interface crée et envoie une demande à Amazon SWF pour planifier une tâche d'activité du type approprié à l'aide d'Amazon SWF. APIs

5. Une surcharge qui prend les arguments initiaux en l'état et des arguments variables supplémentaires de type `Promise<?>`. Le type de retour de cette surcharge est `Promise<`*T*`>`, où *T* est le type de retour de la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

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

   Cette surcharge doit être utilisée lorsque tous les arguments de l'activité sont disponibles et qu'il n'est pas nécessaire de les attendre, mais que vous voulez attendre que d'autres objets `Promise` soient prêts.

6. Une surcharge dont chaque argument de la méthode initiale est remplacé par un wrapper `Promise` et avec des arguments variables supplémentaires de type `Promise<?>`. Le type de retour de cette surcharge est `Promise<`*T*`>`, où *T* est le type de retour de la méthode initiale. Par exemple :

   Méthode initiale :

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

   Méthode générée :

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

   Cette surcharge doit être utilisée lorsque tous les arguments de l'activité font l'objet d'une attente asynchrone et que vous voulez aussi attendre que d'autres `Promise` soient prêts. Un appel à

cette surcharge de méthode s'exécutera de manière asynchrone lorsque tous les objets `Promise` transmis seront prêts.

Le client d'activité généré possède également une méthode protégée correspondant à chaque méthode d'activité, nommée {*activity method name*}`Impl()`, que toutes les surcharges d'activité appellent. Vous pouvez remplacer cette méthode pour créer des implémentations client factices. Cette méthode prend comme arguments tous les arguments de la méthode initiale dans les wrappers `Promise<>`, `ActivitySchedulingOptions`, et les arguments variables de type `Promise<?>`. Par exemple :

Méthode initiale :

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

Méthode générée :

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

Options de planification

Le client d'activité généré vous permet de transmettre `ActivitySchedulingOptions` comme argument. La `ActivitySchedulingOptions` structure contient des paramètres qui déterminent la configuration de la tâche d'activité planifiée par le framework dans Amazon SWF. Ces paramètres remplacent les valeurs par défaut qui sont spécifiées comme options d'enregistrement. Pour spécifier dynamiquement les options de planification, créez un objet `ActivitySchedulingOptions`, configurez-le comme vous le souhaitez et transmettez-le à la méthode d'activité. Dans l'exemple suivant, nous avons spécifié la liste des tâches à utiliser pour la tâche d'activité. Elle remplacera la liste des tâches enregistrées par défaut pour cet appel de l'activité.

```
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);
    }
}
```

Clients dynamiques

Outre les clients générés, le framework fournit également des clients à usage général `DynamicActivityClient` (`DynamicWorkflowClientet`) que vous pouvez utiliser pour démarrer dynamiquement des exécutions de flux de travail, envoyer des signaux, planifier des activités, etc. Par exemple, vous pouvez planifier une activité dont le type n'est pas connu au moment de la conception. Vous pouvez utiliser `DynamicActivityClient` pour planifier une telle tâche d'activité. De même, vous pouvez planifier dynamiquement l'exécution d'un flux de travail enfant en utilisant `DynamicWorkflowClient`. Dans l'exemple suivant, le flux de travail recherche l'activité dans une base de données et utilise le client d'activité dynamique pour la planifier :

```
//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);
}
```

Pour plus de détails, consultez la AWS SDK pour Java documentation.

Signalisation et annulation des exécutions de flux de travail

Le client de flux de travail généré possède des méthodes correspondant à chaque signal qui peut être envoyé au flux. Vous pouvez les utiliser à partir d'un flux de travail pour envoyer des signaux à d'autres exécutions de flux de travail. Ceci fournit un mécanisme typé pour l'envoi de signaux. Cependant, vous devrez parfois déterminer dynamiquement le nom du signal, par exemple lorsque le nom du signal est reçu dans un message. Dans ce cas, vous pouvez utiliser le client de flux de travail dynamique pour envoyer dynamiquement des signaux à n'importe quelle exécution de flux de travail. De même, vous pouvez utiliser le client pour demander l'annulation d'une autre exécution de flux de travail.

Dans l'exemple suivant, le flux de travail recherche l'exécution à laquelle envoyer un signal à partir d'une base de données et envoie le signal dynamiquement en utilisant le client de flux de travail dynamique.

```
//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);
}
```

Implémentation de flux de travail

Afin d'implémenter un flux de travail, vous écrivez une classe qui implémente l'interface `@Workflow` souhaitée. Par exemple, l'exemple d'interface de flux de travail (`MyWorkflow`) peut être implémentée comme suit :

```
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);
  }
}
```

La méthode `@Execute` dans cette classe est le point d'entrée de la logique de flux de travail. Comme le framework utilise le replay pour reconstruire l'état de l'objet lorsqu'une tâche de décision doit être traitée, un nouvel objet est créé pour chaque tâche de décision.

L'utilisation de `Promise<`*T*`>` en tant que paramètre n'est pas autorisée dans la méthode `@Execute` au sein d'une interface `@Workflow`, car la décision d'effectuer un appel asynchrone repose entièrement sur l'appelant. L'implémentation de flux de travail en elle-même ne varie pas selon que l'appel est synchrone ou asynchrone. Par conséquent, l'interface client générée possède des surcharges qui prennent des paramètres `Promise<`*T*`>`, afin que ces méthodes puissent être appelées de manière asynchrone.

Le type de retour d'une méthode `@Execute` peut uniquement être `void` ou `Promise<`*T*`>`. Notez que le type de retour du client externe correspondant est `void` et non pas `Promise<>`. Comme le client

externe n'est pas destiné à être utilisé à partir du code asynchrone, il ne renvoie `Promise` aucun objet. Pour obtenir les résultats des exécutions de flux de travail définies en externe, vous pouvez concevoir le flux de travail de manière à mettre à jour l'état dans un magasin de données externe par le biais d'une activité. La visibilité d'Amazon SWF APIs peut également être utilisée pour récupérer le résultat d'un flux de travail à des fins de diagnostic. Il n'est généralement pas recommandé d'utiliser la visibilité APIs pour récupérer les résultats des exécutions de flux de travail, car ces appels d'API peuvent être limités par Amazon SWF. La visibilité vous APIs oblige à identifier l'exécution du flux de travail à l'aide d'une `WorkflowExecution` structure. Vous pouvez récupérer cette structure auprès du client de flux de travail généré en appelant la méthode `getWorkflowExecution`. Cette méthode renvoie la structure `WorkflowExecution` correspondant à l'exécution de flux de travail à laquelle le client est lié. Consultez le manuel Amazon Simple Workflow Service API Reference pour plus de détails sur la visibilité APIs.

Lorsque vous appelez des activités à partir de l'implémentation de votre flux de travail, vous devez utiliser le client d'activité généré. De même, pour envoyer des signaux, vous utilisez les clients de flux de travail générés.

Contexte décisionnel

L'infrastructure fournit un contexte ambiant chaque fois qu'un code de flux de travail est exécuté par l'infrastructure. Ce contexte fournit une fonctionnalité propre au contexte à laquelle vous pouvez accéder dans l'implémentation de votre flux de travail, comme la création d'un minuteur. Pour plus d'informations, consultez la section Contexte d'exécution.

Exposition de l'état d'exécution

Amazon SWF vous permet d'ajouter un état personnalisé dans l'historique du flux de travail. Le dernier état signalé par l'exécution du flux de travail vous est renvoyé par le biais d'appels de visibilité adressés au service Amazon SWF et dans la console Amazon SWF. Par exemple, dans un flux de travail de traitement des commandes, vous pouvez signaler l'état de la commande à différentes étapes tel que « commande reçue », « commande expédiée », etc. Dans le cas AWS Flow Framework de Java, cela se fait par le biais d'une méthode sur votre interface de flux de travail annotée avec l'`@GetState`annotation. Lorsque le décideur a terminé de traiter une tâche de décision, il appelle cette méthode pour récupérer le dernier état de l'implémentation de flux de travail. Outre les appels de visibilité, l'état peut également être récupéré à l'aide du client externe généré (qui utilise les appels de visibilité en interne).

L'exemple suivant montre comment définir le contexte d'exécution.

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

Le client externe généré peut être utilisé pour récupérer à tout moment le dernier état de l'exécution de flux de travail.

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

Dans l'exemple ci-dessus, l'état d'exécution est signalé à différentes étapes. Lorsque l'instance de flux de travail démarre, `periodicWorkflow` signale l'état initial comme « Just Started » (Vient de démarrer). Chaque appel à `callPeriodicActivity` met à jour l'état de flux de travail. Une fois que `activity1` a été appelé 100 fois, la méthode est renvoyée et l'instance de flux de travail se termine.

Locales de flux de travail

Il arrive que vous ayez besoin d'utiliser des variables statiques dans l'implémentation de votre flux de travail. Par exemple, il se peut que vous souhaitiez stocker un compteur accessible depuis plusieurs emplacements (éventuellement différentes classes) dans l'implémentation de flux de travail. Toutefois, vous ne pouvez pas compter sur les variables statiques de votre flux de travail car ces dernières sont partagées entre les threads, ce qui pose problème car un exécuteur peut traiter différentes tâches décisionnelles sur différents threads au même moment. Vous pouvez également stocker un tel état dans un champ sur l'implémentation de flux de travail, mais vous aurez ensuite besoin de distribuer l'objet d'implémentation. Pour répondre à ce besoin, l'infrastructure fournit une classe `WorkflowExecutionLocal<?>`. N'importe quel état ayant besoin d'une variable statique, comme des sémantiques, doit être conservé en tant que locale d'instance à l'aide de `WorkflowExecutionLocal<?>`. Vous pouvez déclarer et utiliser une variable statique de ce type. Par exemple, dans le code suivant, un objet `WorkflowExecutionLocal<String>` est utilisé pour stocker un nom d'utilisateur.

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

Implémentation d'activité

L'implémentation de l'interface `@Activities` permet d'implémenter les activités. AWS Flow Framework for Java utilise les instances d'implémentation d'activité configurées sur le travailleur pour traiter les tâches d'activité au moment de l'exécution. L'exécuteur recherche automatiquement l'implémentation de l'activité du type approprié.

Vous pouvez utiliser des propriétés et des champs pour transmettre des ressources à des instances d'activité, par exemple à des connexions de base de données. Étant donné que l'objet d'implémentation de l'activité est accessible à partir de plusieurs threads, les ressources partagées doivent être sûres pour les threads.

Notez que l'implémentation de l'activité n'accepte pas les paramètres de type `Promise<>` ou les objets de retour de ce type. En effet, l'implémentation de l'activité ne doit pas dépendre de la manière dont elle a été invoquée (de manière synchrone ou asynchrone).

L'interface des activités ci-dessus peut être implémentée comme suit :

```
public class MyActivitiesImpl implements MyActivities {

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

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

Un contexte local de thread est disponible pour l'implémentation de l'activité utilisable pour récupérer l'objet de tâche, l'objet convertisseur de données utilisé, etc. Le contexte actuel est accessible via

`ActivityExecutionContextProvider.getActivityExecutionContext()`. Pour plus de détails, consultez la AWS SDK pour Java documentation `ActivityExecutionContext` et la sectionContexte d'exécution.

Finalisation manuelle des activités

Dans l'exemple ci-dessus, l'annotation `@ManualActivityCompletion` est facultative. Elle est autorisée uniquement sur les méthodes qui implémentent une activité. Elle est utilisée pour configurer l'activité afin qu'elle ne se termine pas automatiquement lorsque la méthode d'activité est renvoyée. Cela peut être utile lorsque vous souhaitez effectuer l'activité de manière asynchrone, par exemple manuellement une fois qu'une action humaine est terminée.

Par défaut, l'infrastructure considère que l'activité est terminée lorsque votre méthode d'activité est renvoyée. Cela signifie que le responsable de l'activité signale l'achèvement des tâches d'activité à Amazon SWF et lui fournit les résultats (le cas échéant). Dans certains cas d'utilisation, vous ne souhaitez pas que la tâche d'activité soit marquée comme terminée lorsque la méthode d'activité est renvoyée. Cette approche est utile lorsque vous modélisez des tâches manuelles. Par exemple, la méthode d'activité peut envoyer un e-mail à une personne qui doit terminer un travail avant la fin de la tâche d'activité. Dans ces cas, vous pouvez annoter la méthode d'activité avec `@ManualActivityCompletion` pour indiquer à l'exécuteur d'activité qu'il ne doit pas terminer l'activité automatiquement. Pour effectuer l'activité manuellement, vous pouvez soit utiliser la méthode `ManualActivityCompletionClient` fournie dans le framework, soit utiliser la `RespondActivityTaskCompleted` méthode du client Java Amazon SWF fourni dans le SDK Amazon SWF. Pour plus de détails, consultez la AWS SDK pour Java documentation.

Pour mettre fin à la tâche d'activité, vous devez fournir un jeton de tâche. Le jeton de tâche est utilisé par Amazon SWF pour identifier les tâches de manière unique. Vous pouvez accéder à ce jeton à partir de `ActivityExecutionContext` dans l'implémentation de votre activité. Vous devez transmettre ce jeton à la personne chargée de mettre fin à la tâche. Vous pouvez extraire le jeton de `ActivityExecutionContext` en appelant `ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken()`.

L'activité `getName` de l'exemple Hello World peut être implémentée pour envoyer un e-mail demandant à quelqu'un de proposer un message d'accueil :

```
@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";
}
```

L'extrait de code suivant peut être utilisé pour fournir le message d'accueil et fermer la tâche en utilisant `ManualActivityCompletionClient`. Vous pouvez également faire échouer la tâche :

```
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);
    }
}
```

Mise en œuvre AWS Lambda des tâches

Rubriques

- À propos AWS Lambda
- Avantages et limites de l'utilisation des tâches Lambda

- Utilisation de tâches Lambda dans vos flux de travail AWS Flow Framework pour Java
- Voir l' HelloLambda échantillon

À propos AWS Lambda

AWS Lambda est un service de calcul entièrement géré qui exécute votre code en réponse à des événements générés par du code personnalisé ou par divers AWS services tels qu'Amazon S3, DynamoDB, Amazon Kinesis, Amazon SNS et Amazon Cognito. Pour plus d'informations sur Lambda, consultez le guide du développeur AWS Lambda.

Amazon Simple Workflow Service fournit une tâche Lambda qui vous permet d'exécuter des fonctions Lambda à la place ou en parallèle des activités Amazon SWF traditionnelles.

> **Important**
>
> Votre AWS compte sera débité pour les exécutions Lambda (requêtes) exécutées par Amazon SWF en votre nom. Pour plus de détails sur la tarification Lambda, consultez la section tarification/https://aws.amazon.com/lambda/.

Avantages et limites de l'utilisation des tâches Lambda

L'utilisation de tâches Lambda au lieu d'une activité Amazon SWF traditionnelle présente de nombreux avantages :

- Les tâches Lambda n'ont pas besoin d'être enregistrées ou versionnées comme les types d'activité Amazon SWF.
- Vous pouvez utiliser toutes les fonctions Lambda existantes que vous avez déjà définies dans vos flux de travail.
- Les fonctions Lambda sont appelées directement par Amazon SWF ; il n'est pas nécessaire d'implémenter un programme de travail pour les exécuter, comme c'est le cas pour les activités traditionnelles.
- Lambda vous fournit des métriques et des journaux pour suivre et analyser les exécutions de vos fonctions.

Vous devez également connaître les quelques limites qui s'appliquent aux tâches Lambda :

- Les tâches Lambda ne peuvent être exécutées que dans AWS les régions qui prennent en charge Lambda. Consultez la section Régions et points de terminaison Lambda dans le manuel Amazon Web Services General Reference pour en savoir plus sur les régions actuellement prises en charge pour Lambda.
- Les tâches Lambda ne sont actuellement prises en charge que par l'API HTTP SWF de base et pour Java. AWS Flow Framework Les tâches Lambda ne sont actuellement pas prises en charge dans AWS Flow Framework for Ruby.

Utilisation de tâches Lambda dans vos flux de travail AWS Flow Framework pour Java

Trois conditions sont requises pour utiliser les tâches Lambda dans vos flux de travail AWS Flow Framework pour Java :

- Une fonction Lambda à exécuter. Vous pouvez utiliser n'importe quelle fonction Lambda que vous avez définie. Pour plus d'informations sur la création de fonctions Lambda, consultez le manuel du AWS Lambda développeur.
- Rôle IAM qui permet d'exécuter des fonctions Lambda à partir de vos flux de travail Amazon SWF.
- Code permettant de planifier la tâche Lambda depuis votre flux de travail.

Configuration d'un rôle IAM

Avant de pouvoir invoquer des fonctions Lambda depuis Amazon SWF, vous devez fournir un rôle IAM qui permet d'accéder à Lambda depuis Amazon SWF. Vous avez le choix entre les options suivantes :

- choisissez un rôle prédéfini, Role, AWSLambda pour autoriser vos flux de travail à invoquer n'importe quelle fonction Lambda associée à votre compte.
- définissez votre propre politique et le rôle associé pour autoriser les flux de travail à invoquer des fonctions Lambda spécifiques, spécifiées par leur Amazon Resource Names ()ARNs.

Limiter les autorisations sur un rôle IAM

Vous pouvez limiter les autorisations sur un rôle IAM que vous fournissez à Amazon SWF en utilisant les clés de contexte `SourceAccount` et `SourceArn` de votre politique de confiance en matière de ressources. Ces clés limitent l'utilisation d'une politique IAM afin qu'elle ne soit utilisée qu'à partir des

exécutions d'Amazon Simple Workflow Service appartenant à l'ARN du domaine spécifié. Si vous utilisez les deux clés contextuelles de condition globale, la `aws:SourceAccount` valeur et le compte référencés dans la `aws:SourceArn` valeur doivent utiliser le même identifiant de compte lorsqu'ils sont utilisés dans la même déclaration de politique.

Dans l'exemple suivant, la clé de `SourceArn` contexte restreint l'utilisation du rôle de service IAM uniquement dans les exécutions d'Amazon Simple Workflow Service appartenant `someDomain` au compte,. `123456789012`

- Déclaration 1

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

  Action : `sts:AssumeRole`

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

Dans l'exemple suivant, la clé de `SourceAccount` contexte restreint l'utilisation du rôle de service IAM uniquement dans le cadre des exécutions d'Amazon Simple Workflow Service dans le compte. `123456789012`

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

Fournir à Amazon SWF l'accès lui permettant d'invoquer n'importe quel rôle Lambda

Vous pouvez utiliser le rôle prédéfini, Role, AWSLambda pour permettre à vos flux de travail Amazon SWF d'invoquer n'importe quelle fonction Lambda associée à votre compte.

Pour utiliser AWSLambda Role pour autoriser Amazon SWF à invoquer des fonctions Lambda

1. Ouvrez la console Amazon IAM.

2. Choisissez Rôles, puis Créer un rôle.
3. Attribuez un nom à votre rôle, tel que `swf-lambda`, puis choisissez Étape suivante.
4. Sous Rôles de AWS service, sélectionnez Amazon SWF, puis Next Step.
5. Sur l'écran Attach Policy, choisissez Role AWSLambdadans la liste.
6. Choisissez Étape suivante, puis Créer un rôle une fois que vous avez vérifié le rôle.

Définition d'un rôle IAM pour fournir un accès permettant d'invoquer une fonction Lambda spécifique

Si vous souhaitez fournir un accès pour invoquer une fonction Lambda spécifique depuis votre flux de travail, vous devez définir votre propre politique IAM.

Pour créer une politique IAM afin de fournir l'accès à une fonction Lambda particulière

1. Ouvrez la console Amazon IAM.
2. Choisissez Stratégies, puis Créer une stratégie.
3. Choisissez Copier une politique AWS gérée et sélectionnez AWSLambdaRôle dans la liste. Une stratégie sera générée pour vous. Au besoin, modifiez son nom et sa description.
4. Dans le champ Ressource du document de politique, ajoutez l'ARN de vos fonctions Lambda. Par exemple :

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

   > **Note**
   >
   > Pour une description complète de la manière de spécifier les ressources dans un rôle IAM, voir Présentation des politiques IAM dans Using IAM.

5. Choisissez Créer une stratégie afin de finaliser la création de la stratégie.

Vous pouvez ensuite sélectionner cette politique lors de la création d'un nouveau rôle IAM et utiliser ce rôle pour donner un accès d'appel à vos flux de travail Amazon SWF. Cette procédure est très similaire à la création d'un rôle avec la politique des AWSLambdarôles. Choisissez plutôt votre propre politique lors de la création du rôle.

Pour créer un rôle Amazon SWF à l'aide de votre politique Lambda

1. Ouvrez la console Amazon IAM.
2. Choisissez Rôles, puis Créer un rôle.
3. Attribuez un nom à votre rôle, tel que `swf-lambda-function`, puis choisissez Étape suivante.
4. Sous Rôles de AWS service, sélectionnez Amazon SWF, puis Next Step.
5. Sur l'écran Attach Policy, choisissez votre politique spécifique à la fonction Lambda dans la liste.
6. Choisissez Étape suivante, puis Créer un rôle une fois que vous avez vérifié le rôle.

Planifier l'exécution d'une tâche Lambda

Une fois que vous avez défini un rôle IAM qui vous permet d'invoquer des fonctions Lambda, vous pouvez planifier leur exécution dans le cadre de votre flux de travail.

> **Note**
>
> Ce processus est pleinement démontré par l'HelloLambda échantillon du AWS SDK pour Java.

Pour planifier l'exécution d'une tâche Lambda

1. Dans l'implémentation de votre flux de travail, obtenez une instance de `LambdaFunctionClient` en appelant `getLambdaFunctionClient()` sur une instance `DecisionContext`.

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

2. Planifiez la tâche à l'aide de la `scheduleLambdaFunction()` méthode indiquée sur le`LambdaFunctionClient`, en lui transmettant le nom de la fonction Lambda que vous avez créée et toutes les données d'entrée pour la tâche 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. Dans votre outil de démarrage d'exécution de flux de travail, ajoutez le rôle IAM lambda à vos options de flux de travail par défaut en utilisant`StartWorkflowOptions.withLambdaRole()`, puis transmettez les options lors du démarrage du flux de travail.

```
// 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);
```

Voir l' HelloLambda échantillon

Un exemple d'implémentation d'un flux de travail utilisant une tâche Lambda est fourni dans le. AWS SDK pour Java Pour voir and/or Run it, téléchargez le code source.

Une description complète de la façon de créer et d'exécuter l'HelloLambdaexemple est fournie dans le fichier README fourni avec les exemples AWS Flow Framework pour Java.

Exécution de programmes écrits avec le AWS Flow Framework pour Java

Rubriques

- WorkflowWorker
- ActivityWorker
- Modèle de thread d'exécuteur
- Extensibilité de l'exécuteur

Le framework fournit des classes de travail pour initialiser le runtime AWS Flow Framework for Java et communiquer avec Amazon SWF. Pour implémenter un objet exécuteur de flux de travail ou d'activité, vous devez créer et démarrer une instance de classe d'exécuteur. Ces classes de travail sont chargées de gérer les opérations asynchrones en cours, d'invoquer des méthodes asynchrones qui sont débloquées et de communiquer avec Amazon SWF. Elles peuvent être configurées avec des implémentations de flux de travail et d'activité, le nombre de threads, la liste des tâches à interroger, etc.

L'infrastructure est livrée avec deux classes d'exécuteur, une pour les activités et une pour les flux de travail. Pour exécuter la logique de flux de travail, utilisez la classe `WorkflowWorker`. De même, la classe `ActivityWorker` est utilisée pour les activités. Ces classes interrogent automatiquement Amazon SWF pour les tâches d'activité et invoquent les méthodes appropriées dans votre implémentation.

L'exemple suivant montre comment instancier une classe `WorkflowWorker` et lancer l'interrogation pour la recherche de tâches :

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

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

Les étapes de base pour créer une instance de la classe `ActivityWorker` et lancer l'interrogation pour la recherche de tâches sont les suivantes :

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

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

Lorsque vous souhaitez arrêter une activité ou un décideur, votre application doit arrêter les instances des classes de travail utilisées ainsi que l'instance du client Java Amazon SWF. Cela permet de s'assurer que toutes les ressources utilisées par les classes d'exécuteur sont correctement libérées.

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

Pour démarrer une exécution, créez simplement une instance du client externe généré et appelez la méthode `@Execute`.

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

WorkflowWorker

Comme son nom l'indique, cette classe d'exécuteur est destinée à être utilisée par l'implémentation de flux de travail. Elle est configurée avec une liste de tâches et le type d'implémentation du flux de travail. La classe d'exécuteur exécute une boucle pour rechercher les tâches de décision dans la liste des tâches spécifiée. Lors de la réception d'une tâche de décision, elle crée une instance de l'implémentation du flux de travail et appelle la méthode `@Execute` pour traiter la tâche.

ActivityWorker

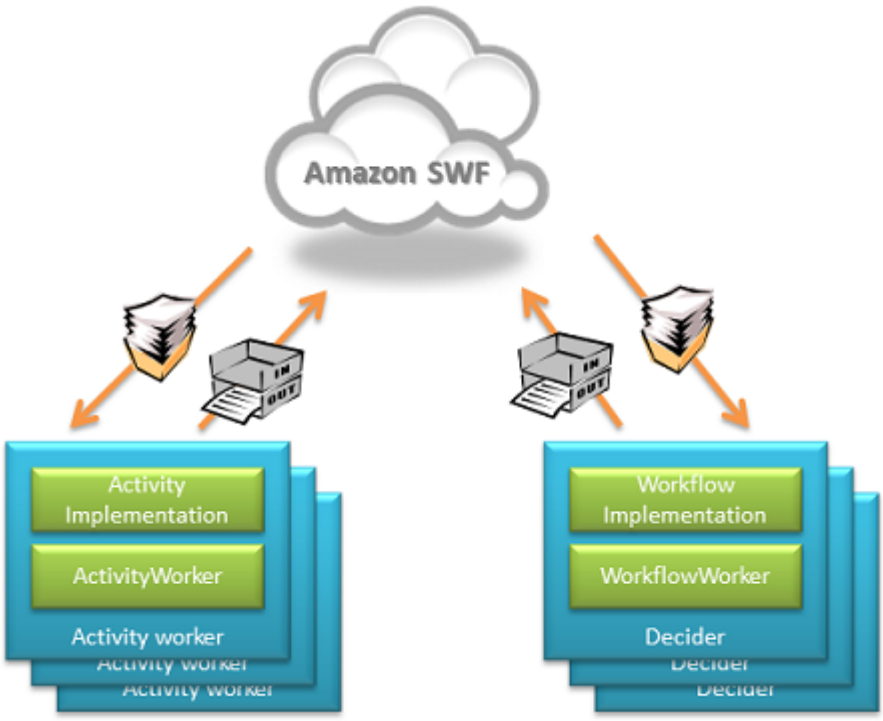
Pour implémenter des exécuteurs d'activité, vous pouvez utiliser la classe `ActivityWorker` pour rechercher facilement des tâches d'activité dans une liste de tâches. Vous configurez l'exécuteur d'activité avec les objets d'implémentation d'activité. Cette classe d'exécuteur exécute une boucle

pour rechercher les tâches d'activité dans la liste des tâches spécifiée. Lors de la réception d'une tâche d'activité, elle recherche l'implémentation appropriée que vous avez fournie et appelle la méthode d'activité pour traiter la tâche. Contrairement à `WorkflowWorker`, qui appelle la fabrique à créer une nouvelle instance pour chaque tâche de décision, `ActivityWorker` utilise simplement l'objet que vous avez fourni.

La `ActivityWorker` classe utilise les annotations AWS Flow Framework for Java pour déterminer les options d'enregistrement et d'exécution.

Modèle de thread d'exécuteur

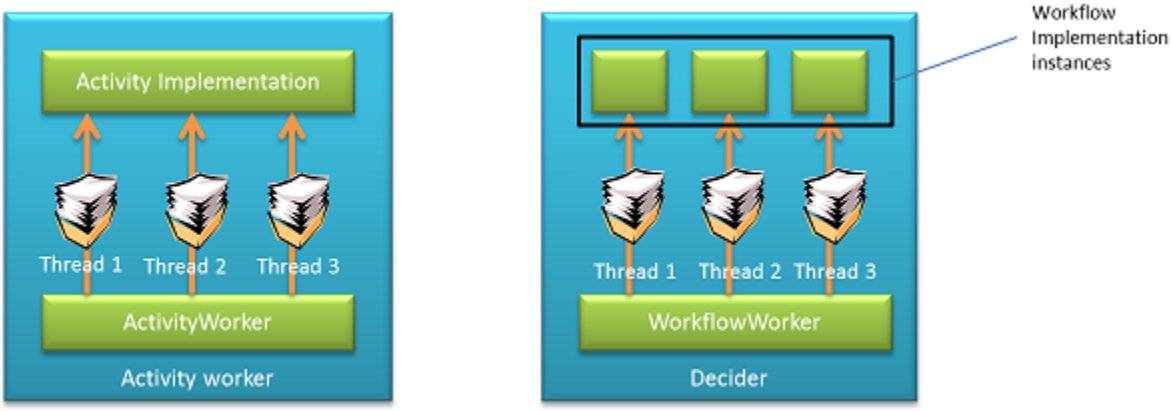
Dans Java, AWS Flow Framework l'incarnation d'une activité ou d'un décideur est une instance de la classe ouvrière. Votre application est responsable de la configuration et de l'instanciation de l'objet exécuteur sur chaque machine, ainsi que du processus agissant en tant qu'exécuteur. L'objet de travail reçoit ensuite automatiquement les tâches d'Amazon SWF, les répartit vers votre activité ou votre implémentation de flux de travail et communique les résultats à Amazon SWF. Une instance de flux de travail peut couvrir de nombreux exécuteurs. Lorsqu'Amazon SWF a une ou plusieurs tâches d'activité en attente, il affecte une tâche au premier travailleur disponible, puis au suivant, etc. Ainsi, des tâches appartenant à la même instance de flux de travail peuvent être traitées simultanément sur des exécuteurs différents.



De plus, chaque exécuteur peut être configuré pour traiter des tâches sur plusieurs threads. Cela signifie que les tâches d'activité d'une instance de flux de travail peuvent être exécutées simultanément même s'il n'y a qu'un exécuteur.

Les tâches de décision se comportent de la même manière, à l'exception du fait qu'Amazon SWF garantit que pour l'exécution d'un flux de travail donné, une seule décision peut être exécutée à la fois. Généralement, une exécution de flux de travail requiert plusieurs tâches de décision ; cela peut donc entraîner l'exécution sur plusieurs processus et threads. Le décideur est configuré avec le type d'implémentation du flux de travail. Lors de la réception d'une tâche de décision par le décideur, une instance (objet) de l'implémentation du flux de travail est créée. L'infrastructure fournit un schéma de fabrique extensible pour la création de ces instances. La fabrique de flux de travail par défaut crée à chaque fois un nouvel objet. Vous pouvez fournir des fabriques personnalisées pour remplacer ce comportement.

Contrairement aux décideurs, qui sont configurés avec des types d'implémentation de flux de travail, les exécuteurs d'activité sont configurés avec des instances (objets) d'implémentation d'activité. Lorsqu'une tâche d'activité est reçue par l'exécuteur d'activité , elle est envoyée vers l'objet d'implémentation d'activité approprié.



Le gestionnaire de flux de travail gère un pool unique de threads et exécute le flux de travail sur le même thread que celui utilisé pour interroger Amazon SWF pour la tâche. Les activités étant longues (du moins si on les compare à la logique du flux de travail), la classe Activity Worker gère deux groupes de threads distincts : l'un pour interroger Amazon SWF pour les tâches d'activité et l'autre pour traiter les tâches en exécutant l'implémentation de l'activité. Cela vous permet de configurer le nombre de threads dans lequel rechercher des tâches indépendamment du nombre de threads pour les exécuter. Par exemple, vous pouvez avoir un petit nombre de threads pour la recherche et un grand nombre de threads pour l'exécution des tâches. La classe Activity Worker interroge Amazon

SWF pour une tâche uniquement lorsqu'elle dispose d'un fil de sondage gratuit ainsi que d'un fil de discussion libre pour traiter la tâche.

Ce comportement de thread et d'instanciation implique que :

1. Les implémentations d'activité doivent être sans état. Vous ne devez pas utiliser des variables d'instance pour stocker un état d'application dans des objets d'activité. Vous pouvez cependant utiliser des champs pour stocker des ressources telles que des connexions de base de données.
2. Les implémentations d'activité doivent être thread-safe. Comme la même instance peut être utilisée pour traiter des tâches provenant de différents threads en même temps, l'accès aux ressources partagées à partir du code d'activité doit être synchronisé.
3. L'implémentation d'un flux de travail peut inclure un état, les variables d'instance pouvant être utilisées pour stocker l'état. Même si une nouvelle instance de l'implémentation du flux de travail est créée pour traiter chaque tâche de décision, l'infrastructure s'assurera ensuite que l'état est recréé correctement. Toutefois, l'implémentation du flux de travail doit être déterministe. Pour en savoir plus, consultez la section Comprendre une tâche dans AWS Flow Framework for Java.
4. Les implémentations de flux de travail ne doivent pas nécessairement être thread-safe lorsque la fabrique par défaut est utilisée. L'implémentation par défaut garantit qu'un seul thread à la fois utilise une instance d'implémentation du flux de travail.

Extensibilité de l'exécuteur

Le AWS Flow Framework for Java contient également quelques classes de travail de bas niveau qui vous offrent un contrôle précis ainsi qu'une extensibilité. Leur utilisation vous permet de personnaliser complètement l'enregistrement du type de flux de travail et d'activité, et de définir des fabriques pour la création d'objets d'implémentation. Ces exécuteurs sont `GenericWorkflowWorker` et `GenericActivityWorker`.

`GenericWorkflowWorker` peut être configuré avec une fabrique pour la création de fabriques de définition de flux de travail. La fabrique de définition de flux de travail est responsable de la création d'instances d'implémentation de flux de travail et de la fourniture des paramètres de configuration tels que les options d'enregistrement. Dans des circonstance normales, vous devez utiliser la classe `WorkflowWorker` directement. Celle-ci crée et configure automatiquement l'implémentation des fabriques fournies dans l'infrastructure, `POJOWorkflowDefinitionFactoryFactory` et `POJOWorkflowDefinitionFactory`. La fabrique nécessite que la classe d'implémentation de flux de travail ait un constructeur sans argument. Ce constructeur est utilisé pour créer des instances de l'objet de flux de travail lors de l'exécution. La fabrique regarde les annotations que vous avez

utilisées sur l'interface et l'implémentation de flux de travail pour créer les options d'enregistrement et d'exécution appropriées.

Vous pouvez fournir votre propre implémentation des fabriques via `WorkflowDefinitionFactory`, `WorkflowDefinitionFactoryFactory` et `WorkflowDefinition`. La classe `WorkflowDefinition` est utilisée par la classe de l'exécuteur pour répartir les tâches et les signaux de décision. En implémentant ces classes de base, vous pouvez personnaliser complètement la fabrique et la répartition des demandes transmises à l'implémentation du flux de travail. Par exemple, vous pouvez utiliser ces points d'extensibilité pour fournir un modèle de programmation personnalisé pour écrire des flux de travail, par exemple, basés sur vos propres annotations, ou en générer un à partir de WSDL au lieu de l'approche de code utilisée par l'infrastructure. Pour utiliser vos fabriques personnalisées, vous devrez utiliser la classe `GenericWorkflowWorker`. Pour plus de détails sur ces classes, consultez la AWS SDK pour Java documentation.

De même, `GenericActivityWorker` vous permet de fournir une fabrique d'implémentation d'activité personnalisée. En implémentant les classes `ActivityImplementationFactory` et `ActivityImplementation`, vous pouvez contrôler complètement l'instanciation d'activité et personnaliser les options d'enregistrement et d'exécution. Pour plus de détails sur ces classes, consultez la AWS SDK pour Java documentation.

Contexte d'exécution

Rubriques

- Contexte décisionnel
- Contexte d'exécution d'une activité

L'infrastructure fournit un contexte ambiant aux implémentations de flux de travail et d'activité. Ce contexte est propre à la tâche en cours de traitement et fournit certains utilitaires que vous pouvez utiliser dans votre implémentation. Un objet de contexte est créé chaque fois qu'une nouvelle tâche est traitée par l'exécuteur.

Contexte décisionnel

Lorsqu'une tâche de décision est exécutée, elle fournit le contexte de la mise en œuvre du flux de travail grâce à la classe `DecisionContext`. `DecisionContext` fournit des informations

contextuelles telles que les ID d'exécution de flux de travail, et les fonctionnalités d'horloge et de minuteur.

Accès lors de DecisionContext l'implémentation du flux de travail

Vous pouvez accéder à la classe `DecisionContext` dans l'implémentation de votre flux de travail à l'aide de la classe `DecisionContextProviderImpl`. Vous pouvez également injecter le contexte dans un champ ou une propriété de l'implémentation de votre flux de travail à l'aide de Spring comme illustré dans la section Injection de la testabilité et de la dépendance.

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

Création d'une horloge et d'un minuteur

La classe `DecisionContext` contient une propriété de type `WorkflowClock` qui fournit une fonction de minuteur et d'horloge. La logique du flux de travail devant être déterministe, vous ne devez pas utiliser directement l'horloge système dans l'implémentation de votre flux de travail. La méthode `currentTimeMills` sur la classe `WorkflowClock` renvoie l'heure de l'événement de début de la décision en cours de traitement. Cela veille à ce que vous obteniez la même valeur temporelle pendant la reproduction, d'où l'importance d'une logique de flux de travail déterministe.

La classe `WorkflowClock` possède également une méthode `createTimer` qui renvoie un objet `Promise` qui sera prêt après l'intervalle spécifié. Vous pouvez utiliser cette valeur en tant que paramètre pour les autres méthodes asynchrones afin de retarder leur exécution de la durée spécifiée. De cette manière, vous pouvez efficacement planifier une méthode ou une activité asynchrone en vue d'une exécution ultérieure.

L'exemple de la liste suivante montre comment utiliser appeler une activité périodiquement.

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

```
}
```

Dans la liste ci-dessus, la méthode asynchrone `callPeriodicActivity` appelle un objet `activity1`, puis crée un minuteur à l'aide de la classe `AsyncDecisionContext` actuelle. Elle transmet l'objet `Promise` renvoyé en tant qu'argument à un appel récursif à elle-même. Cet appel récursif patiente jusqu'à ce que le minuteur se déclenche (1 heure dans cet exemple) avant l'exécution.

Contexte d'exécution d'une activité

De la même manière que la classe `DecisionContext` fournit des informations contextuelles lorsqu'une tâche de décision est traitée, la classe `ActivityExecutionContext` fournit des informations contextuelles similaires lorsqu'une tâche d'activité est en cours de traitement. Ce contexte est disponible depuis votre code d'activité via la classe `ActivityExecutionContextProviderImpl`.

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

À l'aide la classe `ActivityExecutionContext`, vous pouvez exécuter les activités suivantes :

Vérification des pulsations d'une activité de longue durée

Si l'activité dure longtemps, elle doit régulièrement signaler sa progression à Amazon SWF pour lui indiquer que la tâche progresse toujours. En cas d'absence de pulsations, le délai d'attente de la tâche peut être dépassé si un délai de pulsation de la tâche a été défini lors de l'enregistrement du type d'activité ou de la planification de l'activité. Afin d'envoyer une pulsation, vous pouvez utiliser la méthode `recordActivityHeartbeat` sur la classe `ActivityExecutionContext`. Heartbeat fournit également un mécanisme pour annuler les activités en cours. Consultez la section Gestion des erreurs pour en savoir plus et obtenir un exemple.

Obtention des détails de la tâche d'activité

Si vous le souhaitez, vous pouvez obtenir tous les détails de la tâche d'activité transmise par Amazon SWF lorsque l'exécuteur a reçu la tâche. Cela comprend les informations concernant les entrées de la tâche, son type, son jeton, etc. Si vous souhaitez implémenter une activité exécutée manuellement, par exemple par une action humaine, vous devez utiliser le pour récupérer le jeton de tâche et le

transmettre `ActivityExecutionContext` au processus qui finira par terminer la tâche d'activité. Pour en savoir plus, consultez la section sur Finalisation manuelle des activités.

Obtenez l'objet client Amazon SWF utilisé par l'exécuteur

L'objet client Amazon SWF utilisé par l'exécuteur peut être récupéré en appelant `getService` method on. `ActivityExecutionContext` Cela est utile si vous souhaitez appeler directement le service Amazon SWF.

Exécutions de flux de travail enfant

Dans les exemples jusqu'ici, nous avons démarré une exécution de flux de travail directement depuis une application. Cependant, une exécution de flux de travail peut être démarrée depuis un flux de travail en appelant la méthode de point d'entrée du flux de travail sur le client généré. Lorsqu'une exécution de flux de travail est démarrée depuis le contexte de l'exécution d'un autre flux de travail, celle-ci est appelée exécution de flux de travail enfant. Cela vous permet de remanier des flux de travail complexes en unités plus petites et de les partager potentiellement entre différents flux de travail. Par exemple, vous pouvez créer un flux de travail de traitement des paiements et l'appeler à partir d'un flux de travail de traitement des commandes.

Sur le plan sémantique, le flux de travail enfant se comporte comme un flux de travail autonome, à l'exception des différences suivantes :

1. Lorsque le flux de travail parent se termine en raison d'une action explicite de l'utilisateur, par exemple en appelant l'API `TerminateWorkflowExecution` Amazon SWF, ou s'il est interrompu en raison d'un délai imparti, le sort de l'exécution du flux de travail enfant sera déterminé par une politique relative aux enfants. Vous pouvez définir cette stratégie enfant pour arrêter, annuler ou abandonner (continuer d'exécuter) les exécutions de flux de travail enfant.
2. La sortie du flux de travail enfant (valeur de retour de la méthode de point d'entrée) peut être utilisée par l'exécution de flux de travail parent tout comme l'objet `Promise<T>` renvoyé par une méthode asynchrone. Cela est différent des exécutions autonomes où l'application doit obtenir le résultat à l'aide d'Amazon APIs SWF.

Dans l'exemple suivant, le flux de travail `OrderProcessor` crée un flux de travail enfant `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);

}
```

Flux de travail continus

Dans certains cas d'utilisation, vous pouvez avoir besoin d'un flux de travail qui s'exécute pendant longtemps ou indéfiniment, par exemple, un flux de travail qui vérifie l'état d'un parc de serveurs.

> **Note**
>
> Dans la mesure où Amazon SWF conserve l'historique complet de l'exécution d'un flux de travail, celui-ci ne cessera de croître au fil du temps. L'infrastructure récupère cet historique depuis Amazon SWF lorsqu'elle effectue une relecture et cette opération va devenir coûteuse si l'historique est trop volumineux. Dans ces flux de travail de longue durée et continus, vous devez fermer périodiquement l'exécution actuelle et en démarrer une nouvelle pour continuer le traitement.

Il s'agit de la continuation logique de l'exécution de flux de travail. Le client auto-généré peut être utilisé à cette fin. Dans l'implémentation de votre flux de travail, il vous suffit d'appeler la méthode `@Execute` sur le client auto-généré. Une fois que l'exécution actuelle est terminée, l'infrastructure démarre une nouvelle exécution en utilisant le même ID de flux de travail.

Vous pouvez également continuer l'exécution en appelant la méthode `continueAsNewOnCompletion` sur l'objet `GenericWorkflowClient` que vous pouvez récupérer depuis le `DecisionContext` actuel. Par exemple, l'implémentation de flux de travail suivante définit un temporisateur à déclencher après un jour et appelle son propre point d'entrée pour démarrer une nouvelle exécution.

```
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();
    }
}
```

Lorsqu'un flux de travail s'appelle lui-même de façon récursive, l'infrastructure ferme le flux de travail actuel quand toutes les tâches en attente sont terminées et démarre une nouvelle exécution de flux de travail. Notez que tant que des tâches sont en attente, l'exécution de flux de travail actuelle ne se fermera pas. La nouvelle exécution hérite automatique de tout historique ou des données de l'exécution initiale. Si vous souhaitez transmettre un état à la nouvelle exécution, vous devez le transmettre explicitement en tant qu'entrée.

Définition de la priorité des tâches dans Amazon SWF

Par défaut, les tâches d'une liste des tâches dépendent de leur heure d'arrivée : celles qui sont planifiées en premier sont généralement exécutées en premier, aussi loin que possible. En

définissant une priorité de tâche facultative, vous pouvez donner la priorité à certaines tâches : Amazon SWF essaiera de fournir les tâches les plus prioritaires d'une liste de tâches avant celles dont la priorité est inférieure.

Vous pouvez définir des priorités de tâche pour les flux de travail et les activités. La priorité de tâche d'un flux de travail n'a pas d'incidence sur la priorité des tâches d'activité qu'il planifie, ni sur les flux de travail enfants qu'il démarre. La priorité par défaut d'une activité ou d'un flux de travail est définie (par vous ou par Amazon SWF) lors de l'enregistrement, et la priorité de tâche enregistrée est toujours utilisée, sauf si elle est remplacée lors de la planification de l'activité ou du démarrage d'une exécution de flux de travail.

Les valeurs de priorité des tâche peuvent aller de « -2147483648 » à « 2147483647 » (le nombre le plus élevé indique une priorité supérieure). Si vous ne définissez la priorité de tâche d'une activité ou d'un flux de travail, la priorité zéro (« 0 ») lui est attribuée.

Rubriques

- Définition d'une priorité de tâche pour les flux de travail
- Définition d'une priorité de tâche pour les activités

Définition d'une priorité de tâche pour les flux de travail

Vous pouvez définir la priorité de tâche d'un flux de travail lorsque vous l'enregistrez ou lorsque vous le lancez. La priorité de tâche définie lorsque le type de flux de travail est enregistré est utilisée comme valeur par défaut pour toutes les exécutions de flux de travail de ce type, sauf si elle est remplacée lors du lancement de l'exécution de flux de travail.

Pour enregistrer un type de flux de travail avec une priorité de tâche par défaut, définissez l'defaultTaskPriorityoption WorkflowRegistrationOptionslors de sa déclaration :

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

Vous pouvez également définir taskPriority pour un flux de travail lorsque vous la lancez, en remplaçant la priorité de tâche enregistrée par défaut.

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

PriorityWorkflowClientExternalFactory cf
    = new PriorityWorkflowClientExternalFactoryImpl(swfService, domain);

priority_workflow_client = cf.getClient();

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

De plus, vous pouvez définir la priorité de tâche lorsque vous lancez un flux de travail enfant ou poursuivez un flux de travail en tant que nouveau. Par exemple, vous pouvez définir l'option TaskPriority dans ContinueAsNewWorkflowExecutionParametersou dans. StartChildWorkflowExecutionParameters

Définition d'une priorité de tâche pour les activités

Vous pouvez définir la priorité de tâche d'une activité lors de son enregistrement ou de sa planification. La priorité de tâche définie lors de l'enregistrement d'un type d'activité est utilisée comme priorité par défaut lorsque l'activité est exécutée, sauf si elle est remplacée lors de la planification de l'activité.

Pour enregistrer un type d'activité avec une priorité de tâche par défaut, définissez l'defaultTaskPriorityoption ActivityRegistrationOptionslors de sa déclaration :

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

Vous pouvez également définir taskPriority pour une activité lorsque vous la planifiez, en remplaçant la priorité de tâche enregistrée par défaut.

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

ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();

activityClient.doSomethingImportant(activityOptions);
```

DataConverters

Lorsque l'implémentation de votre flux de travail appelle une activité à distance, les entrées la transmettent et le résultat de l'exécution de l'activité doit être sérialisé afin de pouvoir être envoyé sur le réseau. Le framework utilise la DataConverter classe à cette fin. Il s'agit d'une classe abstraite que vous pouvez implémenter pour fournir votre propre sérialiseur. Une implémentation par défaut basée sur le sérialiseur Jackson est fournie dans `JsonDataConverter` le framework. Pour plus d'informations, consultez la documentation AWS SDK pour Java. Reportez-vous à la documentation du processeur Jackson JSON pour obtenir des détails sur la façon dont Jackson effectue la sérialisation ainsi que les annotations qui peuvent être utilisées pour l'influencer. Le format de connexion utilisé est considéré comme faisant partie du contrat. Par conséquent, vous pouvez spécifier un objet `DataConverter` sur vos activités et interfaces de flux de travail en définissant la propriété `DataConverter` des annotations `@Activities` et `@Workflow`.

L'infrastructure crée des objets du type `DataConverter` que vous spécifiez sur l'annotation `@Activities` afin de sérialiser les entrées vers l'activité et de désérialiser son résultat. De même, les objets du type `DataConverter` que vous spécifiez sur l'annotation `@Workflow` sont utilisés pour sérialiser les paramètres transmis au flux de travail, et en cas de flux de travail enfant, pour désérialiser le résultat. Outre les entrées, le framework transmet également des données supplémentaires à Amazon SWF, par exemple les détails des exceptions. Le sérialiseur de flux de travail sera également utilisé pour sérialiser ces données.

Vous pouvez également fournir une instance de la classe `DataConverter` si vous ne souhaitez pas que l'infrastructure la crée automatiquement. Les clients générés disposent de surcharges de constructeur qui prennent un objet `DataConverter`.

Si vous ne spécifiez pas un type `DataConverter` et ne transmettez pas un objet `DataConverter`, la classe `JsonDataConverter` est utilisée par défaut.

Transmission des données aux méthodes asynchrones

Rubriques

- Transmission des collections et des cartes aux méthodes asynchrones
- Définissable <T>
- @NoWait
- Promets- <Vide>
- AndPromise et OrPromise

L'utilisation de `Promise<T>` a été expliquée dans les sections précédentes. Certains cas d'utilisation avancés de `Promise<T>` sont présentés ici.

Transmission des collections et des cartes aux méthodes asynchrones

L'infrastructure prend en charge la transmission des tableaux, collections et cartes comme types `Promise` vers des méthodes asynchrones. Par exemple, une méthode asynchrone peut prendre `Promise<ArrayList<String>>` comme un argument, comme illustré dans la liste suivante.

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

Sur le plan sémantique, cela se comporte comme tout autre paramètre de type `Promise` et la méthode asynchrone patientera jusqu'à ce que la collection soit disponible avant l'exécution. Si les membres d'une collection sont des objets `Promise`, alors vous pouvez faire patienter l'infrastructure jusqu'à ce que tous les membres soient prêts, comme illustré dans le code suivant. Cela fera patienter la méthode asynchrone jusqu'à ce que chaque membre de la collection soit disponible.

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

Notez que l'annotation `@Wait` doit être utilisée sur le paramètre pour indiquer qu'il contient des objets `Promise`.

Notez également que l'activité `printActivity` prend un argument `String`, mais que la méthode correspondante dans le client généré prend le type Promise<String>. Nous appelons la méthode sur le client mais pas directement la méthode d'activité.

Définissable <T>

L'objet `Settable<T>` est un type dérivé d'un objet `Promise<T>` qui fournit une méthode qui vous permet de définir manuellement la valeur d'un objet `Promise`. Par exemple, le flux de travail suivant attend de recevoir un signal en attendant un objet `Settable<?>`, qui est définit dans la méthode de signal :

```
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;
   }
}
```

Un objet `Settable<?>` peut également être lié à un autre objet Promise à la fois. Vous pouvez utiliser les objets `AndPromise` et `OrPromise` pour regrouper les objets Promise. Vous pouvez délier un objet `Settable` lié en appelant la méthode `unchain()`. Une fois liés, les objets `Settable<?>` seront automatiquement prêts lorsque l'objet Promise lié sera prêt. L'action de lier est particulièrement utile lorsque vous souhaitez utiliser un objet Promise renvoyé depuis la portée d'un bloc `doTry()` dans d'autres parties de votre programme. Comme elle `TryCatchFinally`

est utilisée comme classe imbriquée, vous ne pouvez pas déclarer un `Promise<>` dans le champ d'application du parent et le définir. `doTry()` Cela s'explique car Java exige que les variables soient déclarées dans la portée du parent et utilisées dans des classes imbriquées pour être marquées comme finales. Par exemple :

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

    new TryFinally() {

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

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

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

    return result;
}
```

Un objet `Settable` peut également être lié à un seul objet Promise à la fois. Vous pouvez délier un objet `Settable` lié en appelant la méthode `unchain()`.

@NoWait

Lorsque vous transmettez un objet `Promise` à une méthode asynchrone, l'infrastructure attend que les objets `Promise` soient prêts avant d'exécuter la méthode (sauf pour les types de collection). Vous pouvez remplacer ce comportement à l'aide de l'annotation `@NoWait` sur les paramètres dans la déclaration de la méthode asynchrone. Cela est utile si vous transmettez `Settable<T>`, qui sera défini par la méthode asynchrone elle-même.

Promets- <Vide>

Les dépendances dans les méthodes asynchrones sont implémentées en transmettant l'objet `Promise` renvoyé par une méthode en tant qu'argument à une autre méthode. Pourtant, il existe des cas où vous souhaitez renvoyer `void` depuis une méthode, mais souhaitez toujours que les autres méthodes asynchrone s'exécutent après la fin de celle-ci. Dans ces cas-là, vous pouvez utiliser le type `Promise<Void>` en tant que type de retour de la méthode. La classe `Promise` fournit une méthode `Void` statique que vous pouvez utiliser pour créer un objet `Promise<Void>`. Cet objet `Promise` sera prêt lorsque la méthode asynchrone terminera l'exécution. Vous pouvez transmettre cet objet `Promise` à une autre méthode asynchrone comme tout autre objet `Promise`. Si vous utilisez le type `Settable<Void>`, appelez ensuite la méthode de définition avec null pour la préparer.

AndPromise et OrPromise

Les objets `AndPromise` et `OrPromise` vous permettent de regrouper plusieurs objets `Promise<>` dans un objet Promise logique unique. Un objet `AndPromise` sera prêt lorsque tous les objets Promise utilisés pour le construire seront prêts. Un objet `OrPromise` sera prêt lorsque n'importe quel objet Promise de la collection d'objets Promise utilisés pour le construire sera prêt. Vous pouvez appeler `getValues()` sur les objets `AndPromise` et `OrPromise` pour récupérer la liste des valeurs des objets Promise qui les composent.

Testabilité et injection de dépendances

Rubriques

- Intégration de Spring
- JUnit Integration

L'infrastructure est conçue pour être compatible avec l'inversion de contrôle (IoC). Les implémentations d'activité et de flux de travail ainsi que les exécuteurs et les objets de contexte fournis par l'infrastructure peuvent être configurés et instanciés via des conteneurs comme Spring. Par défaut, l'infrastructure permet une intégration avec le framework Spring. En outre, l'intégration JUnit a été fournie pour les implémentations de flux de travail et d'activités de test unitaires.

Intégration de Spring

Le package com.amazonaws.services.simpleworkflow.flow.spring contient des classes qui facilitent l'utilisation du framework Spring dans vos applications. Ces classes incluent une portée (Scope) personnalisée et des exécuteurs d'activité et de flux de travail compatibles avec Spring : `WorkflowScope`, `SpringWorkflowWorker` et `SpringActivityWorker`. Ces classes vous permettent de configurer totalement via Spring vos implémentations d'activité et de flux de travail ainsi que les exécuteurs.

WorkflowScope

`WorkflowScope` est une implémentation de portée (Scope) Spring personnalisée fournie par l'infrastructure. Cette portée vous permet de créer des objets dans le conteneur Spring dont la durée de vie dépend de celle d'une tâche de décision. Les beans de cette portée sont instanciés chaque fois qu'une nouvelle tâche de décision est reçue par l'exécuteur. Vous devez utiliser cette portée pour les beans d'implémentation de flux de travail et pour tous les autres beans dont elle dépend. Les portées singleton et prototype fournies par Spring ne doivent pas être utilisées pour les beans d'implémentation de flux de travail car l'infrastructure requiert qu'un nouveau bean soit crée pour chaque tâche de décision. Si vous ne respectez pas cette règle, vous risquez d'obtenir des comportements inattendus.

L'exemple suivant présente un extrait de configuration Spring qui enregistre la portée `WorkflowScope` puis l'utilise pour la configuration d'un bean d'implémentation de flux de travail et d'un bean de client d'activité.

```
<!-- 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 ligne de configuration `<aop:scoped-proxy proxy-target-class="false" />`, utilisée dans la configuration du bean `workflowImpl`, est obligatoire car la portée `WorkflowScope` ne prend pas en charge la mise en place de proxy avec CGLIB. Vous devez utiliser cette configuration pour tout bean de la portée `WorkflowScope` qui est lié à un autre bean d'une autre portée. Dans ce cas, le bean `workflowImpl` a besoin d'être lié à un bean d'objet exécuteur de flux de travail dans une portée singleton (reportez-vous à l'exemple complet ci-dessous).

Vous trouverez de plus amples informations sur l'utilisation de portées personnalisées dans la documentation du framework Spring.

Exécuteurs compatibles avec Spring

Lorsque vous utilisez Spring, vous devez utiliser les classes d'exécuteur compatibles avec Spring fournies par l'infrastructure : `SpringWorkflowWorker` et `SpringActivityWorker`. Ces exécuteurs peuvent être injectés dans votre application en utilisant Spring comme décrit dans l'exemple suivant. Les exécuteurs compatibles avec Spring implémentent l'interface `SmartLifecycle` de Spring et, par défaut, démarrent automatiquement la recherche des tâches lors de l'initialisation du contexte Spring. Vous pouvez désactiver cette fonctionnalité en définissant la propriété `disableAutoStartup` de l'exécuteur sur `true`.

L'exemple suivant montre comment configurer un décideur. Cet exemple utilise les interfaces `MyActivities` et `MyWorkflow` (non présentées ici) et les implémentations correspondantes, `MyActivitiesImpl` et `MyWorkflowImpl`. Les interfaces et les implémentations de client générées sont `MyWorkflowClient`/`MyWorkflowClientImpl` et `MyActivitiesClient`/`MyActivitiesClientImpl` (également non présentées ici).

Le client des activités est injecté dans l'implémentation du flux de travail via la fonction auto wire (liaison automatique) de Spring :

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

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

La configuration Spring pour le décideur se présente comme suit :

```
<?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>
```

Étant donné que le `SpringWorkflowWorker` est entièrement configuré dans Spring et commence automatiquement à interroger lorsque le contexte Spring est initialisé, le processus hôte pour le décideur est simple :

```
public class WorkflowHost {
   public static void main(String[] args){
      ApplicationContext context
```

```
          = new FileSystemXmlApplicationContext("resources/spring/
WorkflowHostBean.xml");
      System.out.println("Workflow worker started");
   }
}
```

De même, l'exécuteur d'activité peut être configuré comme suit :

```
<?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>
```

Le processus hôte de l'exécuteur d'activité est similaire au décideur :

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

Injection de contexte décisionnel

Si l'implémentation de votre flux de travail dépend des objets de contexte, vous pouvez facilement les injecter via Spring. L'infrastructure enregistre automatiquement les beans liés au contexte dans le conteneur Spring. Par exemple, dans l'extrait suivant, les divers objets de contexte ont été liés

automatiquement via la fonction auto wire. Aucune autre configuration Spring des objets de contexte n'est requise.

```
public class MyWorkflowImpl implements MyWorkflow {
   @Autowired
   public MyActivitiesClient client;
   @Autowired
   public WorkflowClock clock;
   @Autowired
   public DecisionContext dcContext;
   @Autowired
   public GenericActivityClient activityClient;
   @Autowired
   public GenericWorkflowClient workflowClient;
   @Autowired
   public WorkflowContext wfContext;
   @Override
   public void start() {
      client.activity1();
   }
}
```

Si vous souhaitez configurer les objets de contexte dans l'implémentation de flux de travail via la configuration XML Spring, utilisez les noms de beans déclarés dans la classe `WorkflowScopeBeanNames` dans le package com.amazonaws.services.simpleworkflow.flow.spring. Par exemple :

```
<!-- 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>
```

Sinon, vous pouvez injecter un élément `DecisionContextProvider` dans le bean d'implémentation de flux de travail et l'utiliser pour créer le contexte. Cela peut être utile si vous souhaitez fournir des implémentations personnalisées du fournisseur et du contexte.

Injection des ressources dans des activités

Vous pouvez instancier et configurer des implémentations d'activité en utilisant un conteneur d'inversion de contrôle (IoC) et injecter facilement des ressources telles que des connexions de bases de données en les déclarant en tant que propriétés de la classe d'implémentation d'activité. Ces ressources sont généralement définies comme des portées de type singleton. Notez que les implémentations d'activité sont appelées par l'exécuteur d'activité sur plusieurs threads. L'accès aux ressources partagées doit donc être synchronisé.

JUnit Integration

Le framework fournit des JUnit extensions ainsi que des implémentations de test des objets contextuels, telles qu'une horloge de test, que vous pouvez utiliser pour écrire et exécuter des JUnit tests unitaires. Ces extensions vous permettent de tester l'implémentation de votre flux de travail localement en ligne.

Écriture d'un test unitaire simple

Pour écrire des tests pour votre flux de travail, utilisez la classe `WorkflowTest` du package com.amazonaws.services.simpleworkflow.flow.junit. Cette classe est une JUnit `MethodRule` implémentation spécifique au framework et exécute le code de votre flux de travail localement, en appelant les activités en ligne au lieu de passer par Amazon SWF. Cela vous permet d'exécuter vos tests aussi souvent que vous le souhaitez, sans encourir aucun frais.

Pour utiliser cette classe, déclarez simplement un champ de type `WorkflowTest` et annotez-le avec l'annotation `@Rule`. Avant d'exécuter vos tests, créez un nouvel objet `WorkflowTest` et ajoutez-lui vos implémentations d'activité et de flux de travail. Vous pouvez ensuite utiliser la fabrique de clients de flux de travail générée pour créer un client et lancer l'exécution du flux de travail. Le framework fournit également un exécuteur JUnit personnaliséFlowBlockJUnit4ClassRunner, que vous devez utiliser pour vos tests de flux de travail. Par exemple :

```
@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);
    }
}
```

Vous pouvez également spécifier une liste de tâches distincte pour chaque implémentation d'activité que vous ajoutez à `WorkflowTest`. Par exemple, si vous avez une implémentation de flux de travail qui planifie des activités dans des listes de tâches propres à chaque hôte, vous pouvez enregistrer l'activité dans la liste de tâches de chaque hôte :

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

Notez que le code dans `@Test` est asynchrone. Vous devez donc utiliser le client de flux de travail asynchrone pour lancer une exécution. Pour vérifier les résultats de votre test, une classe d'aide `AsyncAssert` est également fournie. Cette classe vous permet d'attendre que les objets attendus

passent à l'état prêt avant de vérifier les résultats. Dans cet exemple, nous attendons que le résultat de l'exécution du flux de travail soit prêt pour vérifier la sortie du test.

Si vous utilisez Spring, la classe `SpringWorkflowTest` peut être utilisé au lieu de la classe `WorkflowTest`. `SpringWorkflowTest` fournit les propriétés que vous pouvez utiliser pour configurer des implémentations d'activité et de flux de travail facilement via la configuration de Spring. Tout comme vous pourriez le faire avec les exécuteurs compatibles Spring, vous devez utiliser la portée `WorkflowScope` pour configurer les beans d'implémentation de flux de travail. Cela permet de s'assurer qu'un nouveau bean d'implémentation de flux de travail est créé pour chaque tâche de décision. Assurez-vous de configurer ces beans avec le proxy-target-class paramètre scoped-proxy défini sur. `false` Pour plus d'informations, consultez la section Intégration de Spring. L'exemple de configuration Spring présenté dans la section Intégration de Spring peut être modifié pour tester le flux de travail à l'aide de `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>
```

Simulation d'implémentations d'activité

Vous pouvez utiliser des implémentations d'activité réelles pendant les tests, mais si vous souhaitez effectuer un test unitaire uniquement sur la logique de flux de travail, vous devez simuler les activités. Vous pouvez le faire en fournissant une implémentation factice de l'interface d'activités à la classe `WorkflowTest`. Par exemple :

```
@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);
    }
}
```

Sinon, vous pouvez aussi fournir une implémentation factice du client d'activités et l'injecter dans l'implémentation de votre flux de travail.

Objets de contexte de test

Si l'implémentation de votre flux de travail dépend des objets contextuels du framework, par exemple, `DecisionContext` vous n'avez rien à faire de spécial pour tester de tels flux de travail. Lorsqu'un test est exécuté via `WorkflowTest`, il injecte automatiquement les objets de contexte de test. Lorsque votre implémentation de flux de travail accède aux objets de contexte, par exemple en utilisant, `DecisionContextProviderImpl` elle obtient l'implémentation de test. Vous pouvez manipuler ces objets de contexte de test dans votre code de test (méthode `@Test`) pour créer des cas de test intéressants. Par exemple, si votre flux de travail crée un temporisateur, vous pouvez faire en sorte qu'il se déclenche en appelant la méthode `clockAdvanceSeconds` sur la classe `WorkflowTest` pour déclencher l'horloge. Vous pouvez également accélérer l'horloge afin que les temporisateurs se déclenchent plus tôt qu'ils ne le feraient normalement en utilisant la propriété `ClockAccelerationCoefficient` sur `WorkflowTest`. Par exemple, si votre flux de travail crée un temporisateur pour un heure, vous pouvez définir le coefficient `ClockAccelerationCoefficient` sur 60 afin que le temporisateur se déclenche au bout d'une minute. Par défaut, `ClockAccelerationCoefficient` est défini sur 1.

Pour plus d'informations sur les packages com.amazonaws.services.simpleworkflow.flow.test et com.amazonaws.services.simpleworkflow.flow.junit, consultez la documentation AWS SDK pour Java .

Gestion des erreurs

Rubriques

- TryCatchFinally Sémantique
- Annulation
- Imbriqué TryCatchFinally

Les blocs `try/catch/finally` intégrés à Java simplifient la gestion des erreurs et sont abondamment utilisés. Ils vous permettent d'associer des gestionnaires d'erreurs à un bloc de code. En interne, cela se concrétise par l'ajout de métadonnées supplémentaires sur les gestionnaires d'erreurs dans la pile d'appel. Lorsqu'une exception est levée, l'environnement d'exécution recherche dans la pile d'appels un gestionnaire d'erreurs associé et l'appelle. S'il ne trouve aucun gestionnaire d'erreurs approprié, il propage l'exception dans la chaîne d'appel.

Cela fonctionne bien pour le code synchrone, mais la gestion des erreurs est asynchrone et les programmes distribués posent des problèmes supplémentaires. Comme un appel asynchrone est renvoyé immédiatement, l'appelant n'est pas dans la pile d'appels lorsque le code asynchrone s'exécute. Cela signifie que les exceptions non gérées dans le code asynchrone ne peuvent pas être gérées par l'appelant de façon classique. Généralement, les exceptions provenant du code asynchrone sont gérées en transmettant l'état d'erreur à un rappel qui est transmis à la méthode asynchrone. Sinon, si un élément `Future<?>` est utilisé, il signale une erreur lorsque vous tentez d'y accéder. Ce processus n'est pas idéal, car le code qui reçoit l'exception (le rappel ou le code qui utilise l'élément `Future<?>`) ne dispose pas du contexte de l'appel initial et peut ne pas être capable de gérer correctement l'exception. En outre, dans un système asynchrone distribué dont les composants s'exécutent simultanément, plusieurs erreurs peuvent se produire simultanément. Ces erreurs peuvent être de différents types et niveaux de gravité ; elles doivent donc être gérées de façon appropriée.

Le nettoyage de la ressource après un appel asynchrone est également difficile. Contrairement au code synchrone, vous ne pouvez pas utiliser try/catch/finally le code d'appel pour nettoyer les ressources, car le travail initié dans le bloc try peut toujours être en cours lorsque le bloc final s'exécute.

Le framework fournit un mécanisme qui rend la gestion des erreurs dans le code asynchrone distribué similaire et presque aussi simple que celle de Java. try/catch/finally

```
ImageProcessingActivitiesClient activitiesClient
     = new ImageProcessingActivitiesClientImpl();

public void createThumbnail(final String webPageUrl) {

  new TryCatchFinally() {

    @Override
    protected void doTry() throws Throwable {
      List<String> images = getImageUrls(webPageUrl);
      for (String image: images) {
        Promise<String> localImage
            = activitiesClient.downloadImage(image);
        Promise<String> thumbnailFile
            = activitiesClient.createThumbnail(localImage);
        activitiesClient.uploadImage(thumbnailFile);
      }
    }

    @Override
    protected void doCatch(Throwable e) throws Throwable {

      // Handle exception and rethrow failures
      LoggingActivitiesClient logClient = new LoggingActivitiesClientImpl();
      logClient.reportError(e);
      throw new RuntimeException("Failed to process images", e);
    }

    @Override
    protected void doFinally() throws Throwable {
      activitiesClient.cleanUp();
    }
  };
}
```

La classe `TryCatchFinally` et ses variantes, `TryFinally` et `TryCatch`, fonctionnent de façon similaire à l'ensemble de blocs Java `try`/`catch`/`finally`. Elle vous permet d'associer des gestionnaires d'exceptions à des blocs de code de flux de travail qui peuvent s'exécuter sous forme de tâches asynchrones et distantes. La méthode `doTry()` est logiquement équivalente au bloc `try`.

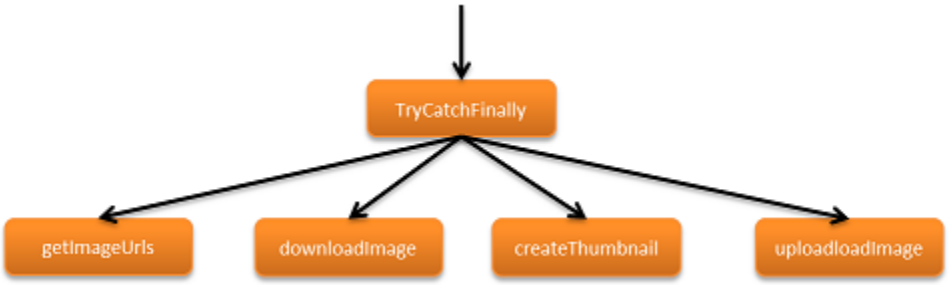
L'infrastructure exécute automatiquement le code dans `doTry()`. Une liste d'objets `Promise` peut être transmise au constructeur de `TryCatchFinally`. La méthode `doTry` est exécutée lorsque tous les objets `Promise` transmis au constructeur sont prêts. Si une exception est levée par le code qui a été appelé de façon asynchrone à partir de `doTry()`, tout travail en attente dans `doTry()` est annulé et `doCatch()` est appelé pour gérer l'exception. Par exemple, dans la liste ci-dessus, si `downloadImage` lève une exception, `createThumbnail` et `uploadImage` sont annulés. Enfin, `doFinally()` est appelé lorsque tous les travaux asynchrones sont terminés (terminés avec succès, en échec ou annulés). Il peut être utilisé pour le nettoyage des ressources. Vous pouvez également imbriquer ces classes en fonction de vos besoins.

Lorsqu'une exception est signalée dans `doCatch()`, l'infrastructure fournit une pile d'appels logique complète qui inclut les appels asynchrones et les appels distants. Cela peut être utile pour le débogage, en particulier si des méthodes asynchrones appellent d'autres méthodes asynchrones. Par exemple, une exception provenant de downloadImage générera une exception similaire à la suivante :

```
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 Sémantique

L'exécution d'un programme AWS Flow Framework pour Java peut être visualisée sous la forme d'un arbre de branches s'exécutant simultanément. Un appel à une méthode asynchrone, une activité et l'élément `TryCatchFinally` lui-même créent une nouvelle branche dans cette arborescence d'exécution. Par exemple, le flux de travail de traitement d'image peut être représenté sous la forme de l'arborescence présentée dans le schéma suivant :



Une erreur dans une branche d'exécution provoque le déroulement de cette branche, tout comme une exception provoque le déroulement de la pile d'appels dans un programme Java. Le déroulement poursuit sa remontée dans la branche d'exécution jusqu'à ce que l'erreur soit résolue ou que la racine de l'arborescence soit atteinte, auquel cas l'exécution du flux de travail est terminée.

L'infrastructure signale les erreurs qui se produisent tout en procédant au traitement des tâches sous la forme d'exceptions. Elle associe les gestionnaires d'exceptions (méthodes `doCatch()`) définis dans `TryCatchFinally` à toutes les tâches qui sont créées par le code dans l'élément `doTry()` correspondant. Si une tâche échoue, par exemple en raison d'un délai d'attente ou d'une exception non gérée, l'exception appropriée sera levée et la correspondante `doCatch()` sera invoquée pour la gérer. Pour ce faire, le framework fonctionne en tandem avec Amazon SWF pour propager les erreurs distantes et les ressusciter sous forme d'exceptions dans le contexte de l'appelant.

Annulation

Lorsqu'une exception se produit dans du code synchrone, le contrôle est directement passé au bloc `catch`, en omettant tout code restant dans le bloc `try`. Par exemple :

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

Dans ce code, si `b()` lève une exception, `c()` n'est jamais appelé. Comparons cela à un flux de travail :

```
new TryCatch() {

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

    @Override
```

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

Dans ce cas, les appels à `activityA`, `activityB` et `activityC` renvoient tous des données avec succès et entraînent la création de trois tâches qui seront exécutées de manière asynchrone. Supposons qu'ultérieurement, la tâche associée à `activityB` engendre une erreur. Cette erreur est enregistrée dans l'historique par Amazon SWF. Pour gérer cela, l'infrastructure tente tout d'abord d'annuler toutes les autres tâches qui ont pour origine le même élément `doTry()` ; dans le cas présent, `activityA` et `activityC`. Lorsque toutes ces tâches sont terminées (annulées, en échec ou exécutées avec succès), la méthode `doCatch()` appropriée est invoquée pour gérer l'erreur.

Contrairement à l'exemple du code synchrone, où `c()` n'a jamais été exécuté, `activityC` a été appelé et une tâche a été programmée pour être exécutée ; l'infrastructure va donc tenter de l'annuler, mais rien ne garantit qu'elle sera annulée. Cette annulation ne peut pas être garantie car l'activité peut être déjà exécutée et terminée, peut ignorer la demande d'annulation ou peut échouer en raison d'une erreur. Toutefois, l'infrastructure garantit que `doCatch()` n'est appelé qu'une fois que toutes les tâches qui ont démarré à partir de l'élément `doTry()` correspondant sont terminées. Elle garantit également que `doFinally()` n'est appelé qu'une fois que toutes les tâches démarrées à partir de `doTry()` et `doCatch()` sont terminées. Si, par exemple, les activités décrites dans l'exemple ci-dessus dépendent les unes des autres, disons `activityB` dépendent de `activityA` et `activityC` de`activityB`, l'annulation `activityC` sera immédiate car elle n'est planifiée dans Amazon SWF qu'`activityB`une fois terminée :

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

Pulsations de l'activité

Le mécanisme d'annulation coopératif de AWS Flow Framework for Java permet d'annuler facilement les tâches liées aux activités en vol. Lorsque l'annulation est déclenchée, les tâches qui sont bloquées ou qui attendent d'être affectées à un exécuteur sont automatiquement annulées. Toutefois, si une tâche est déjà affectée à un exécuteur, l'infrastructure demandera à l'activité de l'annuler. L'implémentation de votre activité doit gérer explicitement ce type de demandes d'annulation. Pour cela, un rapport sur les pulsations de votre activité est émis.

Le fait d'émettre un rapport sur les pulsations permet à l'implémentation d'activité de signaler la progression d'une tâche d'activité en cours, ce qui est utile pour la surveillance et permet à l'activité de détecter les demandes d'annulation. La méthode `recordActivityHeartbeat` lève une exception `CancellationException` si une annulation a été demandée. L'implémentation d'activité peut intercepter cette exception et agir sur la demande d'annulation ou ignorer la demande en digérant l'exception. Pour honorer la demande d'annulation, l'activité doit effectuer le nettoyage souhaité, le cas échéant, puis renvoyer une `CancellationException`. Lorsque cette exception est levée à partir de l'implémentation d'une activité, l'infrastructure enregistre que cette tâche d'activité s'est terminée à l'état annulé.

L'exemple suivant montre une activité qui télécharge et traite des images. Les pulsations varient après le traitement de chaque image et, si l'annulation est demandée, l'activité supprime puis lève à nouveau l'exception pour accuser réception de l'annulation.

```
@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;
        }
    }
}
```

L'émission d'un rapport sur les pulsations de l'activité n'est pas obligatoire, mais elle est recommandée si votre activité s'exécute sur une longue durée ou exécute des opérations onéreuses que vous souhaitez annuler en cas d'erreur. Vous devez appeler `heartbeatActivityTask` périodiquement à partir de l'implémentation de l'activité.

Si l'activité dépasse le délai d'attente qui lui est imparti, l'exception `ActivityTaskTimedOutException` est levée et l'élément `getDetails` lancé sur l'objet d'exception renvoie les données transmises au dernier appel à `heartbeatActivityTask` ayant abouti pour la tâche d'activité correspondante. L'implémentation de flux de travail peut utiliser ces informations pour déterminer le niveau de progression atteint au moment où la tâche d'activité a dépassé le délai qui lui était imparti.

> **Note**
>
> Il n'est pas recommandé de battre trop fréquemment, car Amazon SWF peut ralentir les demandes de pulsation. Consultez le guide du développeur Amazon Simple Workflow Service pour connaître les limites fixées par Amazon SWF.

Annulation explicite d'une tâche

Outre les conditions d'erreur, il existe d'autres cas où vous pouvez être amené à annuler explicitement une tâche. Par exemple, une activité de traitement des règlements à l'aide d'une carte de crédit peut nécessiter une annulation si l'utilisateur annule sa demande. L'infrastructure vous permet d'annuler explicitement des tâches créées dans un bloc `TryCatchFinally`. Dans l'exemple suivant, la tâche de règlement est annulée si un signal est reçu pendant le traitement du règlement.

```
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);
        }
    }
}
```

Réception d'une notification des tâches annulées

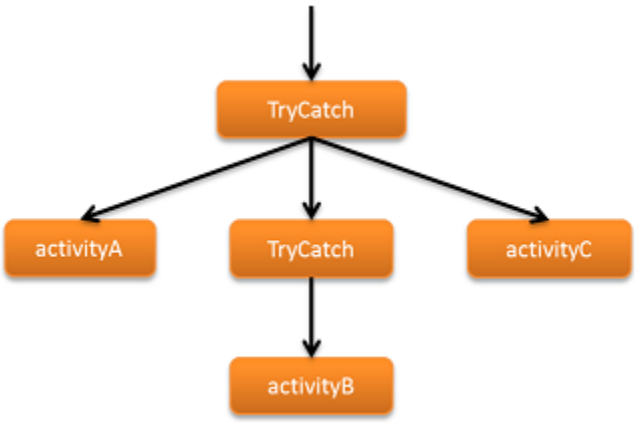
Lorsqu'une tâche se termine à l'état annulé, l'infrastructure informe la logique de flux de travail en levant une exception `CancellationException`. Lorsqu'une activité se termine à l'état annulé, un enregistrement est créé dans l'historique et l'infrastructure appelle la méthode `doCatch()` appropriée avec une exception `CancellationException`. Comme décrit dans l'exemple précédent, lorsque la tâche de traitement du règlement est annulée, le flux de travail reçoit une exception `CancellationException`.

Une exception `CancellationException` non résolue est propagée dans la branche d'exécution, comme c'est le cas pour toute autre exception. Toutefois, la méthode `doCatch()` ne reçoit

l'exception `CancellationException` que s'il n'y a aucune autre exception dans la portée ; les autres exceptions ont une priorité supérieure à celle de l'annulation.

Imbriqué TryCatchFinally

Vous pouvez imbriquer les blocs `TryCatchFinally` en fonction de vos besoins. Comme chacune `TryCatchFinally` crée une nouvelle branche dans l'arbre d'exécution, vous pouvez créer des étendues imbriquées. Les exceptions de la portée parent provoquent des tentatives d'annulation de toutes les tâches initiées par les blocs `TryCatchFinally` imbriqués qu'elle contient. Toutefois, les exceptions présentes dans un bloc `TryCatchFinally` imbriqué ne se propagent pas automatiquement vers le parent. Si vous souhaitez propager une exception d'un bloc `TryCatchFinally` imbriqué vers le bloc `TryCatchFinally` dans lequel il est imbriqué, vous devez lever à nouveau l'exception dans `doCatch()`. En d'autres termes, seules les exceptions non résolues sont remontées, tout comme avec les éléments Java `try/catch`. Si vous annulez un bloc `TryCatchFinally` imbriqué en appelant la méthode cancel, le bloc `TryCatchFinally` imbriqué est annulé, mais le bloc `TryCatchFinally` dans lequel il est imbriqué n'est pas automatiquement annulé.




```
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);
    }
};
```

Relance des activités ayant échoué

Les activités échouent parfois pour des raisons éphémères comme une perte temporaire de connectivité. L'activité peut réussir à un autre moment. La méthode appropriée pour résoudre des échecs d'activité consiste donc souvent à relancer l'activité, peut-être plusieurs fois.

Il existe différentes stratégies pour relancer des activités ; celle qui convient le mieux dépend des détails de votre flux de travail. Les stratégies se répartissent en trois catégories de base :

- La retry-until-success stratégie continue simplement de réessayer l'activité jusqu'à ce qu'elle soit terminée.
- La stratégie de nouvelle tentative exponentielle augmente de façon exponentielle l'intervalle de temps entre les tentatives jusqu'à ce que l'activité se termine ou que le processus atteigne un point d'arrêt spécifié, comme un nombre maximal de tentatives.
- La stratégie de nouvelle tentative personnalisée décide s'il faut relancer l'activité après chaque tentative ayant échoué et de quelle manière.

Les sections suivantes expliquent comment implémenter ces stratégies. Les exemples de exécuteurs de flux de travail utilisent tous une activité unique, `unreliableActivity`, qui exécute de façon aléatoire les actions suivantes :

- Elle se termine immédiatement
- Elle échoue intentionnellement en dépassant la valeur de délai d'expiration
- Elle échoue intentionnellement en déclenchant l'exception `IllegalStateException`

Retry-Until-Success Stratégie

La stratégie de nouvelle tentative la plus simple consiste à relancer chaque fois l'activité jusqu'à ce que celle-ci réussisse. Le modèle de base est le suivant :

1. Implémenter une classe `TryCatch` ou `TryCatchFinally` imbriquée dans la méthode de point d'entrée de votre flux de travail.
2. Exécuter l'activité dans `doTry`.
3. Si l'activité échoue, l'infrastructure appelle `doCatch`, qui exécute à nouveau la méthode de point d'entrée.
4. Répéter les étapes 2 à 3 jusqu'à ce que l'activité se termine correctement.

Le flux de travail suivant met en œuvre la retry-until-success stratégie. L'interface de flux de travail est implémentée dans `RetryActivityRecipeWorkflow` et comporte une méthode, `runUnreliableActivityTillSuccess`, qui est le point d'entrée du flux de travail. L'exécuteur de flux de travail est implémenté dans `RetryActivityRecipeWorkflowImpl`, comme suit :

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

Le flux de travail fonctionne comme suit :

1. `runUnreliableActivityTillSuccess` crée un objet `Settable<Boolean>` nommé `retryActivity` qui est utilisé pour indiquer si l'activité a échoué et doit être réessayée. `Settable<T>` est dérivé de `Promise<T>` et fonctionne de la même manière, mais vous définissez une valeur de l'objet `Settable<T>` manuellement.
2. `runUnreliableActivityTillSuccess` implémente une classe `TryCatch` imbriquée anonyme pour traiter les exceptions qui sont déclenchées par l'activité `unreliableActivity`. Pour en savoir plus sur le traitement des exceptions déclenchées par un code asynchrone, consultez Gestion des erreurs.
3. `doTry` exécute l'activité `unreliableActivity` qui renvoie un objet `Promise<Void>` nommé `activityRanSuccessfully`.
4. `doTry` appelle la méthode `setRetryActivityToFalse` asynchrone et lui transmet deux paramètres :
   - `activityRanSuccessfully` prend l'objet `Promise<Void>` renvoyé par l'activité `unreliableActivity`.
   - `retryActivity` prend l'objet `retryActivity`.

   Si `unreliableActivity` se termine, `activityRanSuccessfully` devient prêt et `setRetryActivityToFalse` définit `retryActivity` sur false. Sinon, `activityRanSuccessfully` ne devient jamais prêt et `setRetryActivityToFalse` ne s'exécute pas.

5. Si `unreliableActivity` déclenche une exception, l'infrastructure appelle `doCatch` et lui transmet l'objet d'exception. `doCatch` définit `retryActivity` avec la valeur true.
6. `runUnreliableActivityTillSuccess` appelle la méthode `restartRunUnreliableActivityTillSuccess` asynchrone et lui transmet l'objet `retryActivity`. Comme `retryActivity` est de type `Promise<T>`, `restartRunUnreliableActivityTillSuccess` diffère l'exécution jusqu'à ce que `retryActivity` soit prêt, ce qui a lieu une fois que `TryCatch` est terminé.
7. Quand `retryActivity` est prêt, `restartRunUnreliableActivityTillSuccess` extrait la valeur.
   - Si la valeur est `false`, la nouvelle tentative a réussi. `restartRunUnreliableActivityTillSuccess` ne fait rien et la séquence de nouvelle tentative est arrêtée.
   - Si la valeur est true, la nouvelle tentative a échoué. `restartRunUnreliableActivityTillSuccess` appelle `runUnreliableActivityTillSuccess` pour exécuter l'activité à nouveau.
8. Le flux de travail répète les étapes 1 à 7 jusqu'à ce que `unreliableActivity` se termine.

> **Note**
>
> `doCatch` ne traite pas l'exception ; il définit simplement l'objet `retryActivity` sur true pour indiquer que l'activité a échoué. La nouvelle tentative est traitée par la méthode `restartRunUnreliableActivityTillSuccess` asynchrone, ce qui diffère l'exécution jusqu'à ce que `TryCatch` se termine. La raison de cette approche est que si vous relancez une activité dans `doCatch`, vous ne pouvez pas l'annuler. La relance de l'activité dans `restartRunUnreliableActivityTillSuccess` vous permet d'exécuter des activités annulables.

Stratégie de nouvelle tentative exponentielle

Avec la stratégie de nouvelle tentative exponentielle, l'infrastructure exécute à nouveau une activité ayant échoué après une période de temps spécifiée, N secondes. Si cette tentative échoue, l'infrastructure exécute à nouveau l'activité après 2N secondes, puis après 4N secondes, et ainsi de suite. Comme le temps d'attente peut devenir très long, vous arrêtez généralement les nouvelles tentatives après un certain temps plutôt que de continuer indéfiniment.

L'infrastructure fournit trois façons d'implémenter une stratégie de nouvelle tentative exponentielle :

- L'annotation `@ExponentialRetry` est l'approche la plus simple, mais vous devez définir les options de configuration de nouvelle tentative lors de la compilation.
- La classe `RetryDecorator` vous permet de définir la configuration de nouvelle tentative lors de l'exécution et de la modifier si nécessaire.
- La classe `AsyncRetryingExecutor` vous permet de définir la configuration de nouvelle tentative lors de l'exécution et de la modifier si nécessaire. En outre, l'infrastructure appelle une méthode `AsyncRunnable.run` implémentée par l'utilisateur pour exécuter chaque nouvelle tentative.

Toutes les approches prennent en charge les options de configuration suivantes, où les valeurs de temps sont exprimées en secondes :

- Le temps d'attente initial avant une nouvelle tentative.
- Le coefficient de recul qui est utilisé pour calculer les intervalles de nouvelle tentative, comme suit :

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

  La valeur par défaut est 2.0.
- Le nombre maximum de nouvelles tentatives autorisées. La valeur par défaut est unlimited (illimité).
- L'intervalle maximum de nouvelle tentative. La valeur par défaut est unlimited (illimité).
- Le délai d'expiration. Les nouvelles tentatives s'arrêtent lorsque la durée totale du processus dépasse cette valeur. La valeur par défaut est unlimited (illimité).
- Les exceptions qui déclenchent le processus de nouvelle tentative. Par défaut, toutes les exceptions déclenchent le processus de nouvelle tentative.
- Les exceptions qui ne déclenchent pas le processus de nouvelle tentative. Par défaut, aucune exception n'est exclue.

Les sections suivantes décrivent les différentes façons d'implémenter une stratégie de nouvelle tentative exponentielle.

Réessayer de façon exponentielle avec @ ExponentialRetry

La façon la plus simple d'implémenter une stratégie de nouvelle tentative exponentielle pour une activité est d'appliquer une annotation `@ExponentialRetry` à l'activité dans la définition d'interface. Si l'activité échoue, l'infrastructure gère automatiquement le processus de nouvelle tentative en fonction des valeurs d'option spécifiées. Le modèle de base est le suivant :

1. Appliquer `@ExponentialRetry` aux activités appropriées et spécifier la configuration de nouvelle tentative.
2. Si une activité annotée échoue, l'infrastructure la relance automatiquement en fonction de la configuration spécifiée par les arguments de l'annotation.

L'exécuteur de flux de travail `ExponentialRetryAnnotationWorkflow` implémente la stratégie de nouvelle tentative exponentielle en utilisant une annotation `@ExponentialRetry`. Il utilise une activité `unreliableActivity` dont la définition d'interface est implémentée dans `ExponentialRetryAnnotationActivities` comme suit :

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

Les options `@ExponentialRetry` spécifient la stratégie suivante :

- Effectuer une nouvelle tentative uniquement si l'activité déclenche `IllegalStateException`.
- Utiliser un temps d'attente initial de 5 secondes.
- Pas plus de 5 nouvelles tentatives.

L'interface de flux de travail est implémentée dans `RetryWorkflow` et comporte une méthode, `process`, qui est le point d'entrée du flux de travail. L'exécuteur de flux de travail est implémenté dans `ExponentialRetryAnnotationWorkflowImpl`, comme suit :

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

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

Le flux de travail fonctionne comme suit :

1. `process` exécute la méthode `handleUnreliableActivity` synchrone.
2. `handleUnreliableActivity` exécute l'activité `unreliableActivity`.

Si l'activité échoue en déclenchant `IllegalStateException`, l'infrastructure exécute automatiquement la stratégie de nouvelle tentative spécifiée dans `ExponentialRetryAnnotationActivities`.

Réessai exponentiel avec la classe RetryDecorator

`@ExponentialRetry` est simple à utiliser. Par contre, la configuration est statique et définie lors de la compilation. L'infrastructure utilise donc la même stratégie de nouvelle tentative chaque fois que l'activité échoue. Vous pouvez implémenter une stratégie de nouvelle tentative exponentielle plus flexible à l'aide de la classe `RetryDecorator`, qui vous permet de spécifier la configuration pendant l'exécution et de la modifier si nécessaire. Le modèle de base est le suivant :

1. Créer et configurer un objet `ExponentialRetryPolicy` qui spécifie la configuration de nouvelle tentative.
2. Créer un objet `RetryDecorator` et transmettre l'objet `ExponentialRetryPolicy` de l'étape 1 au constructeur.
3. Appliquer l'objet décorateur à l'activité en transmettant le nom de classe du client d'activité à la méthode de décoration de l'objet `RetryDecorator`.
4. Exécuter l'activité.

Si l'activité échoue, l'infrastructure la relance automatiquement en fonction de la configuration de l'objet `ExponentialRetryPolicy`. Vous pouvez modifier la configuration de nouvelle tentative si nécessaire en modifiant cet objet.

> **Note**
>
> L'annotation `@ExponentialRetry` et la classe `RetryDecorator` s'excluent mutuellement. Vous ne pouvez pas utiliser `RetryDecorator` pour remplacer dynamiquement une stratégie de nouvelle tentative spécifiée par une annotation `@ExponentialRetry`.

L'implémentation de flux de travail suivante montre comment utiliser la classe `RetryDecorator` pour implémenter une stratégie de nouvelle tentative exponentielle. Elle utilise une activité `unreliableActivity` qui ne comporte pas d'annotation `@ExponentialRetry`. L'interface de flux de travail est implémentée dans `RetryWorkflow` et comporte une méthode, `process`, qui est le point d'entrée du flux de travail. L'exécuteur de flux de travail est implémenté dans `DecoratorRetryWorkflowImpl`, comme suit :

```
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();
  }
}
```

Le flux de travail fonctionne comme suit :

1. `process` crée et configure un objet `ExponentialRetryPolicy` en :
   - Transmettant l'intervalle de nouvelle tentative initial au constructeur.

- Appel de la méthode `withMaximumAttempts` de l'objet pour définir le nombre maximal de tentatives sur 5. `ExponentialRetryPolicy` expose d'autres objets `with` que vous pouvez utiliser pour spécifier d'autres options de configuration.

2. `process` crée un objet `RetryDecorator` nommé `retryDecorator` et transmet l'objet `ExponentialRetryPolicy` de l'étape 1 au constructeur.
3. `process` applique l'objet décorateur à l'activité en appelant la méthode `retryDecorator.decorate` et en lui transmettant le nom de classe du client d'activité.
4. `handleUnreliableActivity` exécute l'activité.

Si une activité échoue, l'infrastructure la relance en fonction de la configuration spécifiée à l'étape 1.

> **Note**
>
> Plusieurs des méthodes `with` de la classe `ExponentialRetryPolicy` ont une méthode `set` correspondante que vous pouvez appeler pour modifier l'option de configuration correspondante à tout moment : `setBackoffCoefficient`, `setMaximumAttempts`, `setMaximumRetryIntervalSeconds` et `setMaximumRetryExpirationIntervalSeconds`.

Réessai exponentiel avec la classe AsyncRetryingExecutor

La classe `RetryDecorator` offre plus de flexibilité pour la configuration du processus de nouvelle tentative que `@ExponentialRetry`, mais l'infrastructure exécute toujours les nouvelles tentatives automatiquement, en fonction de la configuration actuelle de l'objet `ExponentialRetryPolicy`. Une approche plus souple consiste à utiliser la classe `AsyncRetryingExecutor`. En plus de vous permettre de configurer le processus de nouvelle tentative pendant l'exécution, l'infrastructure appelle une méthode `AsyncRunnable.run` implémentée par l'utilisateur pour exécuter chaque nouvelle tentative au lieu de simplement exécuter l'activité.

Le modèle de base est le suivant :

1. Créer et configurer un objet `ExponentialRetryPolicy` pour spécifier la configuration de nouvelle tentative.
2. Créer un objet `AsyncRetryingExecutor`, et lui transmettre l'objet `ExponentialRetryPolicy` et une instance de l'horloge de flux de travail.

3. Implémenter une classe `TryCatch` ou `TryCatchFinally` imbriquée anonyme.
4. Implémenter une classe `AsyncRunnable` anonyme et remplacer la méthode `run` pour implémenter un code personnalisé afin d'exécuter l'activité.
5. Remplacer `doTry` pour appeler la méthode `execute` de l'objet `AsyncRetryingExecutor` et lui transmettre la classe `AsyncRunnable` de l'étape 4. L'objet `AsyncRetryingExecutor` appelle `AsyncRunnable.run` pour exécuter l'activité.
6. Si l'activité échoue, l'objet `AsyncRetryingExecutor` appelle à nouveau la méthode `AsyncRunnable.run`, en fonction de la stratégie de nouvelle tentative spécifiée à l'étape 1.

Le flux de travail suivant montre comment utiliser la classe `AsyncRetryingExecutor` pour implémenter une stratégie de nouvelle tentative exponentielle. Il utilise la même activité `unreliableActivity` que le flux de travail `DecoratorRetryWorkflow` présenté précédemment. L'interface de flux de travail est implémentée dans `RetryWorkflow` et comporte une méthode, `process`, qui est le point d'entrée du flux de travail. L'exécuteur de flux de travail est implémenté dans `AsyncExecutorRetryWorkflowImpl`, comme suit :

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

Le flux de travail fonctionne comme suit :

1. `process` appelle la méthode `handleUnreliableActivity` et lui transmet les paramètres de configuration.
2. `handleUnreliableActivity` utilise les paramètres de configuration de l'étape 1 pour créer un objet `ExponentialRetryPolicy`, `retryPolicy`.
3. `handleUnreliableActivity` crée un objet `AsyncRetryExecutor`, `executor`, et transmet l'objet `ExponentialRetryPolicy` de l'étape 2 et une instance de l'horloge de flux de travail au constructeur.
4. `handleUnreliableActivity` implémente une classe `TryCatch` imbriquée anonyme, et remplace les méthodes `doTry` et `doCatch` pour exécuter les nouvelles tentatives et traiter les exceptions.
5. `doTry` crée une classe `AsyncRunnable` anonyme et remplace la méthode `run` pour implémenter un code personnalisé afin d'exécuter `unreliableActivity`. Pour des raisons de simplicité, `run` exécute seulement l'activité, mais vous pouvez implémenter des approches plus sophistiquées le cas échéant.
6. `doTry` appelle `executor.execute` et transmet l'objet `AsyncRunnable`. `execute` appelle la méthode `run` de l'objet `AsyncRunnable` pour exécuter l'activité.
7. Si l'activité échoue, l'exécuteur appelle à nouveau `run` en fonction de la configuration de l'objet `retryPolicy`.

Pour en savoir plus sur l'utilisation de la classe `TryCatch` pour gérer des erreurs, consultez AWS Flow Framework pour les exceptions Java.

Stratégie de nouvelle tentative personnalisée

L'approche la plus flexible pour réessayer les activités ayant échoué est une stratégie personnalisée, qui appelle de manière récursive une méthode asynchrone qui exécute la nouvelle tentative, un peu comme la stratégie. retry-until-success Par contre, au lieu de relancer simplement l'activité, vous implémentez une logique personnalisée qui décide si chaque nouvelle tentative successive doit être exécutée et de quelle façon. Le modèle de base est le suivant :

1. Créer un objet de statut `Settable<T>` qui est utilisé pour indiquer si l'activité a échoué.
2. Implémenter une classe `TryCatch` ou `TryCatchFinally` imbriquée.
3. `doTry` exécute l'activité.
4. Si l'activité échoue, `doCatch` définit l'objet de statut pour indiquer que l'activité a échoué.
5. Appeler une méthode de gestion des défaillances et lui transmettre l'objet de statut. La méthode diffère l'exécution jusqu'à ce que `TryCatch` ou `TryCatchFinally` soit terminé.
6. La méthode de gestion des défaillances décide s'il faut relancer l'activité, et si oui, quand.

Le flux de travail suivant montre comment implémenter une stratégie de nouvelle tentative personnalisée. Il utilise la même activité `unreliableActivity` que les flux de travail `DecoratorRetryWorkflow` et `AsyncExecutorRetryWorkflow`. L'interface de flux de travail est implémentée dans `RetryWorkflow` et comporte une méthode, `process`, qui est le point d'entrée du flux de travail. L'exécuteur de flux de travail est implémenté dans `CustomLogicRetryWorkflowImpl`, comme suit :

```
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;
    }
}
```

Le flux de travail fonctionne comme suit :

1. `process` appelle la méthode `callActivityWithRetry` asynchrone.
2. `callActivityWithRetry` crée un objet `Settable<Throwable>` nommé failure qui est utilisé pour indiquer si l'activité a échoué. `Settable<T>` est dérivé de `Promise<T>` et fonctionne de la même manière, mais vous définissez une valeur de l'objet `Settable<T>` manuellement.
3. `callActivityWithRetry` implémente une classe `TryCatchFinally` imbriquée anonyme pour traiter les exceptions qui sont déclenchées par `unreliableActivity`. Pour en savoir plus sur le traitement des exceptions déclenchées par un code asynchrone, consultez AWS Flow Framework pour les exceptions Java.
4. `doTry` exécute `unreliableActivity`.
5. Si `unreliableActivity` lève une exception, le framework appelle `doCatch` et transmet l'objet d'exception. `doCatch` définit `failure` sur l'objet d'exception, ce qui indique que l'activité a échoué et place l'objet dans l'état prêt.
6. `doFinally` vérifie si `failure` est prêt, ce qui est vrai seulement si `failure` a été défini par `doCatch`.
   - S'il `failure` est prêt, il `doFinally` ne fait rien.

- Si `failure` n'est pas prêt, l'activité est terminée et `doFinally` définit la défaillance (failure) sur `null`.

7. `callActivityWithRetry` appelle la méthode `retryOnFailure` asynchrone et lui transmet « failure ». Comme « failure » est de type `Settable<T>`, `callActivityWithRetry` diffère l'exécution jusqu'à ce que « failure » soit prêt, ce qui a lieu une fois que `TryCatchFinally` est terminé.
8. `retryOnFailure` extrait la valeur de « failure ».
   - Si l'objet failure est défini avec la valeur null, la nouvelle tentative est réussie. `retryOnFailure` ne fait rien, ce qui arrête le processus de nouvelle tentative.
   - Si « failure » est défini sur un objet d'exception et que `shouldRetry` renvoie true, `retryOnFailure` appelle `callActivityWithRetry` pour relancer l'activité.

     `shouldRetry` implémente une logique personnalisée qui décide s'il faut relancer une activité ayant échoué. Pour des raisons de simplicité, `shouldRetry` renvoie toujours `true` et `retryOnFailure` exécute immédiatement l'activité, mais vous pouvez implémenter une logique plus sophistiquée le cas échéant.
9. Les étapes 2 à 8 se répètent jusqu'à ce que `unreliableActivity` le processus soit terminé ou qu'il soit `shouldRetry` décidé d'arrêter le processus.

> **Note**
>
> `doCatch` ne traite pas le processus de nouvelle tentative ; il définit simplement « failure » pour indiquer que l'activité a échoué. Le processus de nouvelle tentative est géré par la méthode `retryOnFailure` asynchrone, qui diffère l'exécution jusqu'à ce que `TryCatch` se termine. La raison de cette approche est que si vous relancez une activité dans `doCatch`, vous ne pouvez pas l'annuler. La relance de l'activité dans `retryOnFailure` vous permet d'exécuter des activités annulables.

Tâches démon

Le AWS Flow Framework for Java permet de marquer certaines tâches comme`daemon`. Cela permet de créer des tâches pour effectuer du travail en arrière-plan qui doit être annulé lorsque tout le reste du travail est terminé. Par exemple, une tâche de vérification de l'état doit être annulée lorsque le reste du flux de travail est terminé. Vous pouvez accomplir cela en définissant le drapeau `daemon` sur

une méthode asynchrone ou une instance `TryCatchFinally`. Dans l'exemple suivant, la méthode asynchrone `monitorHealth()` est marquée en tant que `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();
  }
}
```

Dans l'exemple ci-dessus, lorsque `doUsefulWorkActivity` se termine, la méthode `monitoringHealth` est automatiquement annulée. Cela entraîne l'annulation de la branche d'exécution entière issue de cette méthode asynchrone. Les sémantiques de l'annulation sont les mêmes que dans `TryCatchFinally`. De même, vous pouvez marquer un démon `TryCatchFinally` en passant un drapeau booléen au constructeur.

```
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
            }
```

```
        };
    }
}
```

Une tâche daemon démarrée dans un `TryCatchFinally` est limitée au contexte dans lequel elle a été créée, c'est-à-dire qu'elle sera limitée aux méthodes, ou. `doTry()` `doCatch()` `doFinally()` Par exemple, dans l'exemple suivant, la méthode asynchrone startMonitoring est marquée en tant que démon et appelée à partir de `doTry()`. La tâche ainsi créée est annulée dès que les autres tâches (`doUsefulWorkActivity` dans ce cas) lancées dans `doTry()` sont terminées.

```
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 pour Java Replay Behavior

Cette rubrique présente des exemples de comportements de reproduction, grâce aux exemples de la section Qu'est-ce que le AWS Flow Framework pour Java ?. Les scénarios synchrones et asynchrones sont présentés.

Exemple 1 : Reproduction synchrone

Pour un exemple du fonctionnement du replay dans un flux de travail synchrone, modifiez les implémentations du HelloWorldWorkflowflux de travail et des activités en ajoutant des `println` appels dans leurs implémentations respectives, comme suit :

```
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);
   }
}
```

Pour plus de détails sur le code, consultez HelloWorldWorkflow Demande. Voici une version modifiée du résultat, avec des commentaires qui indiquent le début de chaque épisode de reproduction.

```
//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
```

Le processus de reproduction pour cet exemple fonctionne comme suit :

- Le premier épisode planifie la tâche d'activité `getName` qui ne possède aucune dépendance.
- Le deuxième épisode planifie la tâche d'activité `getGreeting` qui dépend de `getName`.
- Le troisième épisode planifie la tâche d'activité `say` qui dépend de `getGreeting`.
- Le dernier épisode ne planifie aucune tâche supplémentaire et ne trouve aucune activité inachevée, ce qui termine l'exécution de flux de travail.

> **Note**
>
> Les trois méthodes de client d'activité sont appelées une fois pour chaque épisode. Pourtant, seul un de ces appels se traduit par une tâche d'activité, ainsi chaque tâche n'est exécutée qu'une fois.

Exemple 2 : Reproduction asynchrone

De même que pour l'exemple de reproduction synchrone, vous pouvez modifier la HelloWorldWorkflowAsyncDemande pour observer le fonctionnement d'une reproduction asynchrone. Cela produit le résultat suivant :

```
//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 utilise trois épisodes en replay car il n'y a que deux activités. L'activité `getGreeting` a été remplacée par la méthode de flux de travail asynchrone getGreeting qui ne lance pas un épisode de reproduction lorsqu'elle est terminée.

Le premier épisode n'appelle pas `getGreeting`, car il dépend de la fin de l'activité name. Pourtant, après la fin de getName, la reproduction appelle getGreeting une fois pour chaque épisode suivant.

consultez aussi

- AWS Flow Framework Concepts de base : exécution distribuée

Bonnes pratiques

Utilisez ces bonnes pratiques pour tirer le meilleur parti AWS Flow Framework de Java.

Rubriques

- Modifications du code décideur : Gestion des versions et indicateurs de fonction

Modifications du code décideur : Gestion des versions et indicateurs de fonction

Cette section explique comment éviter les modifications irréversibles apportées à un décideur à l'aide des deux méthodes suivantes :

- La gestion des versions offre une solution basique.
- La gestion des versions avec indicateurs de fonction s'appuie sur la solution de gestion des versions : Aucune nouvelle version du flux de travail n'est présentée, et il n'y a aucun besoin de renvoyer un nouveau code pour mettre à jour la version.

Avant de tester ces solutions, familiarisez-vous avec la section Exemple de scénario qui explique les causes et les effets des modifications irréversibles apportées à un décideur.

Le processus de reproduction et les modifications de code

Lorsqu'un AWS Flow Framework outil de décision pour Java exécute une tâche de décision, il doit d'abord reconstruire l'état actuel de l'exécution avant de pouvoir y ajouter des étapes. Pour ce faire, le décideur utilise un processus appelé reproduction.

Le processus de reproduction exécute de nouveau le code décideur depuis le début, tout en parcourant simultanément l'historique des événements passés. Le fait de parcourir l'historique des événements permet à l'infrastructure de réagir aux signaux ou aux fins de tâches et de débloquer des objets `Promise` dans le code.

Lorsque le framework exécute le code du décideur, il attribue un identifiant à chaque tâche planifiée (une activité, une fonction Lambda, un minuteur, un flux de travail enfant ou un signal sortant) en incrémentant un compteur. Le framework communique cet identifiant à Amazon SWF et l'ajoute aux événements historiques, tels que. `ActivityTaskCompleted`

Pour que le processus de reproduction fonctionne, il est essentiel que le code décideur soit déterministe, et qu'il planifie les mêmes tâches dans le même ordre pour chaque décision dans chaque exécution de flux de travail. Si vous ne respectez pas cette exigence, l'infrastructure pourrait, par exemple, entraîner l'échec de correspondance de l'ID dans un événement `ActivityTaskCompleted` à un objet `Promise` existant.

Exemple de scénario

Il existe une classe de modifications de code considérée comme irréversibles. Ces modifications incluent des mises à jour qui modifient le nombre, le type ou l'ordre des tâches planifiées. Prenez l'exemple suivant :

Vous écrivez du code décideur pour planifier deux tâches de minuteur. Vous commencez une exécution et exécutez une décision. Par conséquent, deux tâches chronométrées sont planifiées, avec IDs 1 et2.

Si vous mettez à jour le code décideur pour planifier uniquement un minuteur avant l'exécution de la prochaine décision, lors de la prochaine tâche de décision l'infrastructure ne pourra pas reproduire le deuxième événement `TimerFired`, car l'ID 2 ne correspond à aucune tâche de minuteur produite par le code.

Aperçu du scénario

L'aperçu suivant décrit les étapes de ce scénario. L'objectif final du scénario est de migrer vers un système qui ne planifie qu'un minuteur, mais n'entraîne aucun échec des exécutions lancées avant la migration.

1. La version initiale du décideur
   a. Écrivez le décideur.
   b. Lancez le décideur.
   c. Le décideur planifie deux minuteurs.
   d. Le décideur lance cinq exécutions.
   e. Arrêtez le décideur.
2. Une modification irréversible du décideur
   a. Modifiez le décideur.
   b. Lancez le décideur.
   c. Le décideur planifie un minuteur.

d. Le décideur lance cinq exécutions.

Les sections suivantes incluent des exemples de code Java qui montrent comment implémenter ce scénario. Les exemples de code dans la section Solutions montrent différents moyens de corriger des modifications irréversibles.

> **Note**
>
> Vous pouvez utiliser la dernière version de AWS SDK pour Java pour exécuter ce code.

Code commun

Le code Java suivant ne change pas entre les exemples de ce scénario.

`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;
    }

}
```

Écriture du code décideur initial

Voici le code Java initial du décideur. Il est enregistré en tant que version 1 et planifie deux tâches de minuteur de cinq secondes.

`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);
        }

    }
}
```

Simulation d'une modification irréversible

Le code Java modifié suivant du décideur est un bon exemple de modification irréversible. Le code est toujours enregistré en tant que version 1, mais il ne planifie qu'un minuteur.

`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);
        }

    }
}
```

Le code Java suivant vous permet de simuler le problème des modifications irréversibles en exécutant le décideur modifié.

`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();
    }

}
```

Lorsque vous exécutez le programme, les trois exécutions qui échouent sont celles lancées sous la version initiale du décideur et poursuivies après la migration.

Solutions

Vous pouvez utiliser les solutions suivantes pour éviter les modifications irréversibles. Pour plus d'informations, consultez Modifications du code décideur et Exemple de scénario.

Utilisation de la gestion des versions

Dans cette solution, vous copiez le décideur dans une nouvelle classe, vous le modifiez, puis vous l'enregistrez sous une nouvelle version de flux de travail.

`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);
        }

    }

}
```

Dans le code Java mis à jour, le deuxième exécuteur de décision exécute les deux versions du flux de travail, permettant de poursuivre les exécutions à la volée indépendamment des modifications apportées à la version 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();
    }

}
```

Lorsque vous lancez le programme, toutes les exécutions se terminent avec succès.

Utilisation des indicateurs de fonction

L'autre solution pour éviter les modifications irréversibles est de créer des branches de code qui prennent en charge deux implémentations de la même classe basées sur des données d'entrée au lieu des versions de flux de travail.

Lorsque vous choisissez cette approche, vous ajoutez des champs à vos objets d'entrée (ou en modifiez des champs existants) chaque fois que vous apportez de petites modifications. Pour les exécutions qui démarrent avant la migration, l'objet d'entrée ne disposera pas du champ (ou possédera une valeur différente). Ainsi, vous n'avez pas à augmenter le numéro de version.

> **Note**
>
> Si vous ajoutez de nouveaux champs, veillez à ce que le processus de désérialisation JSON soit irréversible. Les objets sérialisés avant la présentation du champ doivent toujours être désérialisés avec succès après la migration. Étant donné que JSON définit une valeur `null` dès qu'un champ est manquant, utilisez toujours des types enveloppes (`Boolean` au lieu de `boolean`) et occupez-vous des cas dans lesquels la valeur est `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);
            }
        }

    }
}
```

Dans le code Java mis à jour, le code des deux versions du flux de travail est toujours enregistré pour la version 1. Pourtant, après la migration, de nouvelles exécutions démarrent avec le champ `skipSecondTimer` des données d'entrée définies sur `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();
    }

}
```

Lorsque vous lancez le programme, toutes les exécutions se terminent avec succès.

Conseils de dépannage et de débogage AWS Flow Framework pour Java

Rubriques

- Erreurs de compilation
- Défaillance de ressource inconnue
- Exceptions lors de l'appel à get () sur une promesse
- Workflows non déterministes
- Problèmes liés à la gestion des versions
- Résolution des problèmes et débogage de l'exécution d'un flux de travail
- Tâches perdues
- Échec de validation dû à des contraintes de longueur des paramètres de l'API

Cette section décrit certains écueils courants que vous pourriez rencontrer lors du développement de flux de travail à l'aide AWS Flow Framework de Java. Il fournit également des conseils pour vous aider à diagnostiquer et déboguer des problèmes.

Erreurs de compilation

Si vous utilisez l'option de tissage de compilation d'AspectJ, vous risquez de rencontrer des erreurs de compilation dans lesquelles le compilateur ne parvient pas à trouver les classes client générées pour votre flux de travail et vos activités. La cause la plus probable de ces erreurs de compilation est que le générateur AspectJ a ignoré les clients générés lors de la compilation. Vous pouvez résoudre cette erreur en supprimant AspectJ du projet et en le réactivant. Notez que vous devrez procéder de la sorte à chaque fois que vos interfaces de flux de travail ou d'activité sont modifiées. En raison de ce problème, nous vous recommandons d'utiliser plutôt l'option de tissage de temps de chargement. Pour en savoir plus, consultez la section Configuration du AWS Flow Framework pour Java.

Défaillance de ressource inconnue

Amazon SWF renvoie une erreur de ressource inconnue lorsque vous essayez d'effectuer une opération sur une ressource qui n'est pas disponible. Les causes courantes de cette anomalie sont :

- Vous configurez un exécuteur avec un domaine qui n'existe pas. Pour résoudre ce problème, enregistrez d'abord le domaine à l'aide de la console Amazon SWF ou de l'API du service Amazon SWF.
- Vous essayez de créer une exécution de flux de travail ou des tâches d'activité dont les types n'ont pas encre été enregistrés. Cela peut se produire si vous essayez de créer l'exécution de flux de travail avant que les exécuteurs soient exécutés. Étant donné que les travailleurs enregistrent leurs types lorsqu'ils sont exécutés pour la première fois, vous devez les exécuter au moins une fois avant de tenter de démarrer les exécutions (ou enregistrer manuellement les types à l'aide de la console ou de l'API du service). Notez qu'une fois que les types ont été enregistrés, vous pouvez créer des exécutions même si aucun exécuteur n'est en cours d'exécution.
- Un travail exécuteur de terminer une tâche dont le délai d'attente est déjà dépassé. Par exemple, si un collaborateur met trop de temps à traiter une tâche et dépasse le délai imparti, il sera victime d'une UnknownResource erreur s'il tente de terminer ou d'échouer la tâche. Les AWS Flow Framework travailleurs continueront à interroger Amazon SWF et à effectuer des tâches supplémentaires. Toutefois, vous devez envisager d'ajuster le délai d'attente. L'ajustement du temps d'attente nécessite l'enregistrement d'une nouvelle version du type d'activité.

Exceptions lors de l'appel à get () sur une promesse

Contrairement à Java Future, `Promise` est une construction sans blocage, et l'appel `get()` sur un argument `Promise` qui n'est pas encore prêt émet une exception au lieu d'un blocage. La bonne façon d'utiliser a `Promise` est de le transmettre à une méthode asynchrone (ou à une tâche) et d'accéder à sa valeur dans la méthode asynchrone. AWS Flow Framework for Java garantit qu'une méthode asynchrone n'est appelée que lorsque tous les `Promise` arguments qui lui sont transmis sont prêts. Si vous pensez que votre code est correct ou si vous le rencontrez lors de l'exécution de l'un des AWS Flow Framework exemples, cela est probablement dû au fait qu'AspectJ n'est pas correctement configuré. Pour en savoir plus, consultez la section Configuration du AWS Flow Framework pour Java.

Workflows non déterministes

Comme décrit dans la section Non-déterminisme, l'implémentation de votre flux de travail doit être déterministe. Certaines erreurs courantes qui peuvent mener au non-déterminisme sont l'utilisation de l'horloge système, l'utilisation de nombres aléatoires et la génération de. GUIDs Étant donné que ces structures peuvent renvoyer des valeurs différentes à différents moments, le flux de contrôle de votre flux de travail peut emprunter des chemins différents à chaque fois qu'il est exécuté (consultez

les sections AWS Flow Framework Concepts de base : exécution distribuée et Comprendre une tâche dans AWS Flow Framework for Java pour plus de détails). Si l'infrastructure détecte un non déterminisme lors de l'exécution du flux de travail, une exception est émise.

Problèmes liés à la gestion des versions

Lorsque vous implémentez une nouvelle version de votre flux de travail ou de votre activité, par exemple, lorsque vous ajoutez une nouvelle fonctionnalité, vous devez augmenter la version du type en utilisant l'annotation appropriée :, ou. `@Workflow` `@Activites` `@Activity` Souvent, lorsque de nouvelles versions d'un flux de travail sont déployées, des exécutions de la version existante sont déjà en cours. Vous devez donc vous assurer que les tâches soient transmises aux exécuteurs avec la version appropriée de votre flux de travail et de vos activités. Pour ce faire, vous pouvez utiliser un ensemble de listes de tâches différent pour chaque version. Par exemple, vous pouvez ajouter le numéro de version au nom de la liste de tâches. Cela permet de vous assurer que les tâches appartenant à des versions différentes du flux de travail et des activités sont affectées aux exécuteurs appropriés.

Résolution des problèmes et débogage de l'exécution d'un flux de travail

La première étape pour résoudre les problèmes liés à l'exécution d'un flux de travail consiste à utiliser la console Amazon SWF pour consulter l'historique du flux de travail. L'historique du flux de travail est un enregistrement complet et fiable de tous les événements qui ont modifié l'état d'exécution de l'exécution du flux de travail. Cet historique est conservé par Amazon SWF et est très utile pour diagnostiquer les problèmes. La console Amazon SWF vous permet de rechercher des exécutions de flux de travail et d'accéder à des événements historiques individuels.

AWS Flow Framework fournit une `WorkflowReplayer` classe que vous pouvez utiliser pour rejouer l'exécution d'un flux de travail localement et le déboguer. À l'aide de cette classe, vous pouvez déboguer les exécutions de flux de travail fermées et en cours d'exécution. `WorkflowReplayers`'appuie sur l'historique stocké dans Amazon SWF pour effectuer la rediffusion. Vous pouvez le rediriger vers une exécution de flux de travail dans votre compte Amazon SWF ou lui fournir l'historique des événements (par exemple, vous pouvez récupérer l'historique depuis Amazon SWF et le sérialiser localement pour une utilisation ultérieure). Lorsque vous relisez une exécution de flux de travail avec `WorkflowReplayer`, cette opération n'a pas d'impact sur l'exécution de flux de travail en cours dans votre compte. La relecture est faite entièrement sur le client. Vous pouvez

déboguer le flux de travail, créer des points d'arrêt et marquer des étapes dans le code à l'aide des outils de débogage habituels. Si vous utilisez Eclipse, pensez à ajouter des filtres d'étape pour filtrer les AWS Flow Framework packages.

Par exemple, l'extrait de code suivant peut être utilisé pour relire une exécution de flux de travail :

```
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 vous permet également d'obtenir un thread dump asynchrone de l'exécution de votre flux de travail. Ce vidage de thread vous donne les piles d'appel de toutes les tâches asynchrones ouvertes. Cette information peut être utile pour déterminer quelles sont les tâches de l'exécution en attente et possiblement bloquées. Par exemple :

```
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);
}
```

Tâches perdues

Il arrive parfois que vous arrêtiez des exécuteurs et en lanciez de nouveaux rapidement uniquement pour détecter que des tâches ont été distribuées aux anciens exécuteurs. Cela peut se produire en raison de conditions de concurrence dans le système, qui est réparti sur plusieurs processus. Le problème peut également se produire lorsque vous exécutez des tests d'unité dans une boucle étroite. L'arrêt d'un test dans Eclipse peut aussi parfois provoquer cela car les gestionnaires d'arrêt peuvent ne pas être appelés.

Afin de vous assurer que le problème est en réalité dû à l'obtention des tâches par les anciens exécuteurs, consultez l'historique du flux de travail pour déterminer quel processus a reçu la tâche que vous attendiez que le nouvel exécuteur reçoive. Par exemple, l'événement `DecisionTaskStarted` de l'historique contient l'identité de l'exécuteur de flux de travail ayant reçu la tâche. L'identifiant utilisé par le Flow Framework est de la forme : {*`processId`*} @ {*`host name`*}. Par exemple, voici les détails de l'`DecisionTaskStarted`événement dans la console Amazon SWF pour un exemple d'exécution :

Horodatage d'événement	Mon Feb 20 11:52:40 GMT-800 2012
Identity	2276 @ip -0A6C1 DF5
ID d'événement planifié	33

Afin d'éviter cette situation, utilisez des listes de tâches différentes pour chaque test. Pensez également à ajouter un délai entre l'arrêt des anciens exécuteurs et le démarrage des nouveaux.

Échec de validation dû à des contraintes de longueur des paramètres de l'API

Amazon SWF applique des contraintes de longueur aux paramètres d'API. Vous recevrez un `HTTP 400` message d'erreur si la mise en œuvre de votre flux de travail ou de votre activité

dépasse les contraintes. Par exemple, lors d'un appel `recordActivityHeartbeat` pour `ActivityExecutionContext` envoyer un battement de cœur pour une activité en cours, la chaîne ne doit pas comporter plus de 2 048 caractères.

Un autre scénario courant est celui où une activité échoue en raison d'une exception. Le framework signale un échec d'activité à Amazon SWF en appelant RespondActivityTaskFailedavec l'exception sérialisée comme détails. L'appel d'API signalera une erreur 400 si l'exception sérialisée a une longueur supérieure à 32 768 octets. Pour remédier à cette situation, vous pouvez tronquer le message d'exception ou les causes afin de respecter la contrainte de longueur.

AWS Flow Framework pour Java Reference

Rubriques

- AWS Flow Framework pour les annotations Java
- AWS Flow Framework pour les exceptions Java
- AWS Flow Framework pour les packages Java

AWS Flow Framework pour les annotations Java

Rubriques

- @Activités
- @Activité
- @ActivityRegistrationOptions
- @Asynchrone
- @Execute
- @ExponentialRetry
- @GetState
- @ManualActivityCompletion
- @Signal
- @SkipRegistration
- @Wait et @ NoWait
- @Flux de travail
- @WorkflowRegistrationOptions

@Activités

Cette annotation peut être utilisée dans une interface pour déclarer un ensemble de types d'activités. Chaque méthode de l'interface comportant cette annotation représente un type d'activité. Une interface ne peut pas avoir à la fois des annotations `@Workflow` et `@Activities`.

Les paramètres suivants peuvent être spécifiés sur cette annotation :

`activityNamePrefix`

Spécifie le préfixe du nom des types d'activité déclarés dans l'interface. S'il est défini sur une chaîne vide (valeur par défaut), le nom de l'interface suivi d'un point (.) est utilisé comme préfixe.

`version`

Spécifie la version par défaut des types d'activité déclarés dans l'interface. La valeur par défaut est `1.0`.

`dataConverter`

Spécifie le type de serializing/deserializing données `DataConverter` à utiliser lors de la création de tâches de ce type d'activité et ses résultats. Défini sur `NullDataConverter` par défaut, ce qui indique que `JsonDataConverter` doit être utilisé.

@Activité

Cette annotation peut être utilisée sur des méthodes au sein d'une interface annotée avec `@Activities`.

Les paramètres suivants peuvent être spécifiés sur cette annotation :

`name`

Spécifie le nom du type d'activité. La valeur par défaut est une chaîne vide, qui indique que le préfixe par défaut et le nom de la méthode d'activité doivent être utilisés pour déterminer le nom du type d'activité (au format {{préfixe}{{nom}). Notez que lorsque vous spécifiez un nom dans une annotation `@Activity`, l'infrastructure ne lui ajoutera pas automatiquement un préfixe. Vous êtes libre d'utiliser votre propre schéma d'attribution de noms.

`version`

Spécifie la version du type d'activité. Remplace la version par défaut spécifiée dans l'annotation `@Activities` sur l'interface qui la contient. La valeur par défaut est une chaîne vide.

@ActivityRegistrationOptions

Spécifie les options d'enregistrement d'un type d'activité. Cette annotation peut être utilisée dans une interface annotée avec `@Activities` ou les méthodes qu'elle contient. Si elle est spécifiée aux deux endroits, l'annotation utilisée sur la méthode prend effet.

Les paramètres suivants peuvent être spécifiés sur cette annotation :

`defaultTasklist`

Spécifie la liste de tâches par défaut à enregistrer auprès d'Amazon SWF pour ce type d'activité. Cette valeur par défaut peut être remplacée lors de l'appel de la méthode d'activité sur le client généré à l'aide du paramètre `ActivitySchedulingOptions`. Définie sur `USE_WORKER_TASK_LIST` par défaut. Il s'agit d'une valeur spéciale qui indique que la liste de tâches utilisée par l'exécuteur, qui effectue l'enregistrement, doit être utilisée.

`defaultTaskScheduleToStartTimeoutSeconds`

Spécifie le defaultTaskSchedule ToStartTimeout fichier enregistré auprès d'Amazon SWF pour ce type d'activité. Il s'agit du temps d'attente maximum autorisé pour une tâche de ce type d'activité avant qu'elle soit affectée à un exécuteur. Consultez le manuel Amazon Simple Workflow Service API Reference pour plus de détails.

`defaultTaskHeartbeatTimeoutSeconds`

Spécifie le `defaultTaskHeartbeatTimeout` fichier enregistré auprès d'Amazon SWF pour ce type d'activité. Les exécuteurs doivent indiquer les pulsations pendant cette durée, faute de quoi la tâche sera interrompue. Défini sur -1 par défaut, qui est une valeur spéciale qui indique que ce délai d'attente doit être désactivé. Consultez le manuel Amazon Simple Workflow Service API Reference pour plus de détails.

`defaultTaskStartToCloseTimeoutSeconds`

Spécifie le defaultTaskStart ToCloseTimeout fichier enregistré auprès d'Amazon SWF pour ce type d'activité. Ce délai d'attente détermine le temps maximum de traitement d'une tâche d'activité de ce type par un exécuteur. Consultez le manuel Amazon Simple Workflow Service API Reference pour plus de détails.

`defaultTaskScheduleToCloseTimeoutSeconds`

Spécifie le `defaultScheduleToCloseTimeout` fichier enregistré auprès d'Amazon SWF pour ce type d'activité. Ce délai détermine la durée totale pendant laquelle la tâche peut rester ouverte. Défini sur -1 par défaut, qui est une valeur spéciale qui indique que ce délai d'attente doit être désactivé. Consultez le manuel Amazon Simple Workflow Service API Reference pour plus de détails.

@Asynchrone

Lorsqu'elle est utilisée sur une méthode dans la logique de coordination du flux de travail, indique que la méthode doit être exécutée de manière asynchrone. Un appel à la méthode renverra immédiatement une valeur, mais l'exécution réelle se fera de manière asynchrone lorsque tous les paramètres Promise<> transmis aux méthodes seront prêts. Les méthodes annotées avec @Asynchronous doivent avoir le type de retour Promise<> ou être vides.

`daemon`

Indique si la tâche créée pour la méthode asynchrone doit être une tâche démon. `False` par défaut.

@Execute

En cas d'utilisation sur une méthode dans une interface annotée avec l'annotation `@Workflow`, identifie le point d'entrée du flux de travail.

> **Important**
>
> Une seule méthode de l'interface peut être décorée avec `@Execute`.

Les paramètres suivants peuvent être spécifiés sur cette annotation :

`name`

Spécifie le nom du type de flux de travail. S'il n'est pas défini, le nom par défaut {préfixe}{nom}, où{préfixe} est le nom de l'interface du flux de travail suivi par un '.' et par {nom} est le nom de la méthode décorée `@Execute` du flux de travail.

`version`

Spécifie la version du type de flux de travail.

@ExponentialRetry

En cas d'utilisation sur une activité ou une méthode asynchrone, définit une stratégie de nouvelle tentative exponentielle si la méthode lève une exception non gérée. Une nouvelle tentative est effectuée après une période d'interruption, qui est calculée en fonction du nombre de tentatives.

Les paramètres suivants peuvent être spécifiés sur cette annotation :

`intialRetryIntervalSeconds`

Spécifie la durée d'attente avant la première tentative de relance. Cette valeur ne doit pas être supérieure à `maximumRetryIntervalSeconds` et `retryExpirationSeconds`.

`maximumRetryIntervalSeconds`

Spécifie la durée maximale entre les tentatives de relance. Une fois le délai atteint, l'intervalle de nouvelle tentative est plafonné à cette valeur. La valeur par défaut est -1, ce qui signifie une durée illimitée.

`retryExpirationSeconds`

Spécifie la durée après laquelle la stratégie de nouvelle tentative exponentielle s'arrêtera. La valeur par défaut est -1, ce qui signifie qu'elle n'expire pas.

`backoffCoefficient`

Spécifie le coefficient utilisé pour calculer l'intervalle de nouvelle tentative. Consultez Stratégie de nouvelle tentative exponentielle.

`maximumAttempts`

Indique le nombre de tentatives après lequel la stratégie de nouvelle tentative exponentielle s'arrêtera. La valeur par défaut est -1, ce qui signifie que le nombre de nouvelles tentatives est illimité.

`exceptionsToRetry`

Spécifie le nombre de fois que le client NFS doit déclencher une nouvelle tentative. L'exception non gérée de ces types ne se propagera plus et la méthode sera retentée après l'intervalle de nouvelle tentative calculé. Par défaut, la liste contient `Throwable`.

`excludeExceptions`

Spécifie la liste des types d'exception qui ne doivent pas déclencher de nouvelle tentative. Les exceptions non gérées de ce type seront autorisées à se propager. Par défaut, la liste est vide.

@GetState

En cas d'utilisation sur une méthode dans une interface annotée avec `@Workflow`, identifie que la méthode est utilisée pour récupérer le dernier état d'exécution du flux de travail. Au maximum, il peut

y avoir une méthode avec cette annotation dans une interface portant l'annotation `@Workflow`. Les méthodes ainsi annotées ne doivent pas prendre n'importe quel paramètre et leur type de retour doit impérativement être `void`.

@ManualActivityCompletion

Cette annotation peut être utilisée sur une méthode d'activité pour indiquer que la tâche d'activité ne doit pas être terminée lorsque la méthode est renvoyée. La tâche d'activité ne sera pas automatiquement terminée et devra être effectuée manuellement directement à l'aide de l'API Amazon SWF. Ceci est utile lorsque la tâche d'activité est déléguée à un système externe qui n'est pas automatisé ou nécessite une intervention humaine pour être terminée.

@Signal

En cas d'utilisation sur une méthode dans une interface annotée avec `@Workflow`, identifie un signal qui peut être reçu par des exécutions du type de flux de travail déclaré par l'interface. L'utilisation de cette annotation est nécessaire pour définir une méthode de signal.

Les paramètres suivants peuvent être spécifiés sur cette annotation :

`name`

Spécifie la partie nom du nom du signal. Si ce paramètre n'est pas défini, le nom de la méthode est utilisé.

@SkipRegistration

Lorsqu'il est utilisé sur une interface annotée avec l'`@Workflow`annotation, cela indique que le type de flux de travail ne doit pas être enregistré auprès d'Amazon SWF. L'une des annotations `@WorkflowRegistrationOptions` et `@SkipRegistrationOptions` doit être utilisée sur une interface annotée avec `@Workflow`, mais pas les deux.

@Wait et @ NoWait

Ces annotations peuvent être utilisées sur un paramètre de type `Promise<>` pour indiquer si le AWS Flow Framework for Java doit attendre qu'il soit prêt avant d'exécuter la méthode. Par défaut, les paramètres `Promise<>` transmis aux méthodes `@Asynchronous` doivent être prêts avant l'exécution de la méthode. Dans certaines situations, il est nécessaire de remplacer ce comportement

par défaut. Les paramètres `Promise<>` passés dans les méthodes `@Asynchronous` et annotée avec `@NoWait` ne sont pas attendus.

Les paramètres des collections (ou sous-classes) qui contiennent des objets Promise comme `List<Promise<Int>>`, doivent être annotés avec `@Wait`. Par défaut, l'infrastructure n'attend pas les membres d'une collection.

@Flux de travail

Cette annotation est utilisée sur une interface pour déclarer un type de flux de travail. Une interface décorée avec cette annotation doit contenir exactement une méthode qui est décorée avec @Execute pour déclarer un point d'entrée pour votre flux de travail.

> **Note**
>
> Dans une interface, les annotations `@Workflow` et `@Activities` ne peuvent pas être déclarées en même temps ; elles sont mutuellement exclusives.

Les paramètres suivants peuvent être spécifiés sur cette annotation :

`dataConverter`

Spécifie quel `DataConverter` utiliser lors de l'envoi de demandes et de la réception de résultats pour les exécutions de flux de travail de ce type de flux de travail.

La valeur par défaut est `NullDataConverter` celle qui, à son tour, revient `JsonDataConverter` à traiter toutes les données de demande et de réponse en tant que notation d' JavaScriptobjet (JSON).

exemple

```
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

Lorsqu'il est utilisé sur une interface annotée avec`@Workflow`, fournit les paramètres par défaut utilisés par Amazon SWF lors de l'enregistrement du type de flux de travail.

> **Note**
>
> Vous devez utiliser `@WorkflowRegistrationOptions` ou `@SkipRegistrationOptions` dans une interface annotée avec `@Workflow`, mais vous ne pouvez pas spécifier les deux.

Les paramètres suivants peuvent être spécifiés sur cette annotation :

Description

Texte descriptif facultatif du type de flux de travail.

`defaultExecutionStartToCloseTimeoutSeconds`

Spécifie le `defaultExecutionStartToCloseTimeout` type de flux de travail enregistré auprès d'Amazon SWF. Durée totale autorisée pour qu'une exécution de flux de travail de ce type se termine.

Pour plus d'informations sur les délais d'expiration des flux de travail, consultez la section Types de délai d'expiration Amazon SWF .

`defaultTaskStartToCloseTimeoutSeconds`

Spécifie le `defaultTaskStartToCloseTimeout` type de flux de travail enregistré auprès d'Amazon SWF. Ce paramètre spécifie le temps maximum autorisé pour qu'une tâche de décision unique d'une exécution de flux de travail de ce type se termine.

Si vous ne spécifiez pas `defaultTaskStartToCloseTimeout`, la valeur par défaut est 30 secondes.

Pour plus d'informations sur les délais d'expiration des flux de travail, consultez la section Types de délai d'expiration Amazon SWF .

`defaultTaskList`

Liste de tâches par défaut utilisée pour les tâches de décision pour les exécutions de ce type de flux de travail. Cette valeur par défaut peut être remplacée en utilisant `StartWorkflowOptions` lors du démarrage d'une exécution de flux de travail.

Si vous ne spécifiez pas `defaultTaskList`, la valeur `USE_WORKER_TASK_LIST` sera utilisée par défaut. Ce paramètre indique que la liste de tâches utilisée par l'exécuteur qui effectue l'enregistrement du flux de travail doit être utilisée.

`defaultChildPolicy`

Spécifie la stratégie à utiliser pour les exécutions de flux de travail enfant si une exécution de ce type est arrêtée. La valeur par défaut est `ABANDON`. Les valeurs possibles sont :

- `ABANDON`— Permettre aux exécutions du flux de travail de l'enfant de continuer
- `TERMINATE`— Résout les exécutions de flux de travail pour enfants
- `REQUEST_CANCEL`— Demande l'annulation des exécutions du flux de travail enfant

AWS Flow Framework pour les exceptions Java

Les exceptions suivantes sont utilisées par AWS Flow Framework for Java. Cette section fournit une présentation de l'exception. Pour plus de détails, consultez la AWS SDK pour Java documentation des exceptions individuelles.

Rubriques

- ActivityFailureException
- ActivityTaskException
- ActivityTaskFailedException
- ActivityTaskTimedOutException
- ChildWorkflowException
- ChildWorkflowFailedException
- ChildWorkflowTerminatedException
- ChildWorkflowTimedOutException
- DataConverterException
- DecisionException
- ScheduleActivityTaskFailedException

- SignalExternalWorkflowException
- StartChildWorkflowFailedException
- StartTimerFailedException
- TimerException
- WorkflowException

ActivityFailureException

Cette exception est utilisée en interne par l'infrastructure pour communiquer un échec de l'activité. Lorsqu'une activité échoue en raison d'une exception non gérée, elle est encapsulée `ActivityFailureException` et signalée à Amazon SWF. Vous devez traiter cette exception uniquement si vous utilisez les points d'extensibilité de l'exécuteur d'activité. Votre code d'application ne devra jamais traiter cette exception.

ActivityTaskException

Il s'agit de la classe de base pour les exceptions d'échec de tâche d'activité : `ScheduleActivityTaskFailedException`, `ActivityTaskFailedException`, `ActivityTaskTimedoutException`. Elle contient l'ID de tâche et le type d'activité de la tâche ayant échoué. Vous pouvez détecter cette exception dans l'implémentation de votre flux de travail pour gérer les échecs d'activité de manière générique.

ActivityTaskFailedException

Les exceptions non gérées dans les activités sont signalées à l'implémentation de flux de travail via l'envoi d'une exception `ActivityTaskFailedException`. L'exception d'origine peut être extraite à partir de la propriété cause de cette exception. L'exception fournit également d'autres informations utiles à des fins de débogage, telles que l'identifiant d'activité unique dans l'historique.

L'infrastructure peut fournir l'exception distante en sérialisant l'exception d'origine à partir de l'exécuteur d'activité.

ActivityTaskTimedOutException

Cette exception est levée si le délai imparti à une activité a été dépassé par Amazon SWF. Cela peut se produire si la tâche d'activité n'a pas pu être affectée à l'exécuteur pendant la période de temps requise ou n'a pas pu être effectuée par l'exécuteur dans le temps requis. Vous pouvez définir ces

délais d'attente sur l'activité par l'intermédiaire de l'annotation `@ActivityRegistrationOptions` ou du paramètre `ActivitySchedulingOptions` lors de l'appel de la méthode d'activité.

ChildWorkflowException

Classe de base pour des exceptions utilisées pour signaler l'échec d'exécution d'un flux de travail enfant. L'exception contient l'ID de l'exécution du flux de travail enfant, ainsi que son type de flux de travail. Vous pouvez détecter cette exception pour gérer les échecs d'exécution de flux de travail enfant de manière générique.

ChildWorkflowFailedException

Les exceptions non gérées dans les flux de travail enfants sont signalées à l'implémentation de flux de travail parent via l'envoi d'une exception `ChildWorkflowFailedException`. L'exception d'origine peut être extraite à partir de la propriété `cause` de cette exception. L'exception fournit également d'autres informations utiles à des fins de débogage, telles que les identifiants uniques de l'exécution enfant.

ChildWorkflowTerminatedException

Cette exception est levée dans l'exécution du flux de travail parent pour signaler la résiliation d'une exécution de flux de travail enfant. Vous devez détecter cette exception si vous souhaitez gérer la résiliation d'un flux de travail enfant, par exemple, pour procéder à un nettoyage ou à une compensation.

ChildWorkflowTimedOutException

Cette exception est émise lors de l'exécution du flux de travail parent pour signaler que l'exécution d'un flux de travail enfant a expiré et a été clôturée par Amazon SWF. Vous devez détecter cette exception si vous souhaitez gérer la fermeture forcée d'un flux de travail enfant, par exemple, pour procéder à un nettoyage ou à une compensation.

DataConverterException

L'infrastructure utilise le composant `DataConverter` pour regrouper ou dégrouper des données envoyées sur le réseau. Cette exception est émise si le composant `DataConverter` ne parvient pas à grouper ou à dégrouper les données. Cela peut se produire pour des raisons différentes, par exemple, à cause d'une incohérence dans les composants `DataConverter` utilisés pour grouper et dégrouper les données.

DecisionException

Il s'agit de la classe de base pour les exceptions qui représentent l'échec de la mise en œuvre d'une décision d'Amazon SWF. Vous pouvez détecter cette exception pour gérer ces exceptions de manière générique.

ScheduleActivityTaskFailedException

Cette exception est levée si Amazon SWF ne parvient pas à planifier une tâche d'activité. Cela peut se produire pour diverses raisons : par exemple, l'activité a été abandonnée ou une limite Amazon SWF a été atteinte sur votre compte. La propriété `failureCause` de l'exception spécifie la raison exacte de l'échec de planification de l'activité.

SignalExternalWorkflowException

Cette exception est levée si Amazon SWF ne parvient pas à traiter une demande par l'exécution du flux de travail pour signaler une autre exécution du flux de travail. Cela se produit si l'exécution du flux de travail cible est introuvable, c'est-à-dire si l'exécution du flux de travail que vous avez spécifiée n'existe pas ou est fermée.

StartChildWorkflowFailedException

Cette exception est levée si Amazon SWF ne parvient pas à démarrer l'exécution d'un flux de travail enfant. Cela peut se produire pour diverses raisons : par exemple, le type de flux de travail enfant spécifié est obsolète ou la limite Amazon SWF de votre compte a été atteinte. La propriété `failureCause` de l'exception spécifie la raison exacte de l'échec du lancement de l'exécution du flux de travail enfant.

StartTimerFailedException

Cette exception est levée si Amazon SWF ne parvient pas à démarrer un temporisateur demandé par l'exécution du flux de travail. Cela peut se produire si l'identifiant du temporisateur spécifié est déjà utilisé ou si une limite Amazon SWF a été atteinte sur votre compte. La propriété `failureCause` de l'exception spécifie la raison exacte de l'échec.

TimerException

Il s'agit de la classe de base pour les exceptions liées aux minuteurs.

WorkflowException

Cette exception est utilisée en interne par l'infrastructure pour signaler des échecs dans l'exécution d'un flux de travail. Vous devez traiter cette exception uniquement si vous utilisez un point d'extensibilité d'un exécuteur de flux de travail.

AWS Flow Framework pour les packages Java

Cette section fournit une vue d'ensemble des packages inclus dans le AWS Flow Framework pour Java. Pour plus d'informations sur chaque package, consultez le fichier com.amazonaws.services.simpleworkflow.flow dans le guide de référence des API.AWS SDK pour Java

com.amazonaws.services.simpleworkflow.flow

Contient des composants qui s'intègrent à Amazon SWF.

com.amazonaws.services.simpleworkflow.flow.annotations

Contient les annotations utilisées par le modèle de programmation AWS Flow Framework pour Java.

com.amazonaws.services.simpleworkflow.flow.aspectj

Contient AWS Flow Framework pour Java les composants requis pour des fonctionnalités telles que @Asynchrone et@ExponentialRetry.

com.amazonaws.services.simpleworkflow.flow.common

Contient des utilitaires communs tels que des constantes définies par l'infrastructure.

com.amazonaws.services.simpleworkflow.flow.core

Contient des fonctions telles que `Task` et `Promise`.

com.amazonaws.services.simpleworkflow.flow.generic

Contient des composants essentiels, tels que des clients génériques, sur lesquels d'autres fonctions s'appuient.

com.amazonaws.services.simpleworkflow.flow.interceptors

Contient des implémentations de décorateurs fournis par l'infrastructure dont `RetryDecorator`.

com.amazonaws.services.simpleworkflow.flow.junit

Contient des composants qui fournissent une intégration Junit.

com.amazonaws.services.simpleworkflow.flow.pojo

Contient des classes qui implémentent des définitions d'activité et de flux de travail pour le modèle de programmation basé sur les annotations.

com.amazonaws.services.simpleworkflow.flow.spring

Contient des composants qui fournissent une intégration Spring.

com.amazonaws.services.simpleworkflow.flow.test

Contient des classes d'assistance, telles que `TestWorkflowClock`, pour les tests unitaires d'implémentations de flux de travail.

com.amazonaws.services.simpleworkflow.flow.worker

Contient les implémentations d'exécuteurs d'activité et de flux de travail

Historique du document

Le tableau suivant décrit les modifications importantes apportées à la documentation depuis la dernière version du guide du développeur AWS Flow Framework pour Java.

- Version de l'API : 2012-01-25
- Dernière mise à jour de la documentation : 25 juin 2018

Modification	Description	Date de modification
Mettre à jour	Correction d'une erreur dans la description de `backoffCoefficient` pour `@ExponentialRetry` . Consultez @ExponentialRetry.	25 juin 2018
Mettre à jour	Les exemples de code ont été nettoyés dans ce guide.	5 juin 2017
Mettre à jour	Simplification et amélioration de l'organisation et du contenu de ce guide.	19 mai 2017
Mettre à jour	Simplification et amélioration de la section Modifications du code décideur : Gestion des versions et indicateurs de fonction.	10 avril 2017
Mettre à jour	Nouvelle section Bonnes pratiques ajoutée avec de nouveaux conseils sur la façon de modifier le code décideur.	3 mars 2017
Nouvelle fonctionnalité	Vous pouvez spécifier des tâches Lambda en plus des tâches d'activité traditionnelles dans vos flux de travail. Pour de plus amples informations, veuillez consulter Mise en œuvre AWS Lambda des tâches.	21 juillet 2015
Nouvelle fonctionnalité	Amazon SWF prend en charge la définition de la priorité des tâches sur une liste de tâches, en essayant de fournir les tâches les plus prioritaires avant les tâches les moins prioritaires. Pour de plus amples informations, veuillez	17 décembre 2014

Modification	Description	Date de modification
	consulter Définition de la priorité des tâches dans Amazon SWF.	
Mettre à jour	Mises à jour et correctifs appliqués.	1er août 2013
Mettre à jour	• Mises à jour et correctifs appliqués, y compris les mises à jour des instructions de configuration pour Eclipse 4.3 et AWS SDK pour Java 1.4.7. • Ajout d'un nouvel ensemble de didacticiels pour l'élaboration de scénarios de démarrage	28 juin 2013
Nouvelle fonctionnalité	La version initiale du AWS Flow Framework pour Java.	27 février 2012

Les traductions sont fournies par des outils de traduction automatique. En cas de conflit entre le contenu d'une traduction et celui de la version originale en anglais, la version anglaise prévaudra.