aws

Guia do Desenvolvedor

AWS Flow Framework para Java

Versão da API 2021-04-28

AWS Flow Framework para Java: Guia do Desenvolvedor

Table of Contents

O que é isso AWS Flow Framework para Java?

Com o AWS Flow Framework, você pode se concentrar na implementação de sua lógica de fluxo de trabalho. Nos bastidores, a estrutura usa os recursos de agendamento, roteamento e gerenciamento de estado do Amazon SWF para gerenciar a execução do seu fluxo de trabalho e torná-lo escalável, confiável e auditável. AWS Flow Framework os fluxos de trabalho baseados são altamente simultâneos. Os fluxos de trabalho podem ser distribuídos em vários componentes, que podem ser executados como processos separados em computadores distintos e escalados de forma independente. A aplicação pode continuar a progredir se qualquer um de seus componentes estiver em execução, o que a torna altamente tolerante a falhas.

O que há neste guia?

Este guia tem informações sobre como instalar, configurar e usar o AWS Flow Framework para criar aplicativos Amazon SWF.

Começando com o AWS Flow Framework for Java

Se você está apenas começando com o AWS Flow Framework para Java, leia a Começando com o AWS Flow Framework for Java seção. Ele o guiará pelo download e instalação do AWS Flow Framework para Java, como configurar seu ambiente de desenvolvimento e guiará você por um exemplo simples de criação de um fluxo de trabalho.

Compreensão AWS Flow Framework do Java

Apresenta conceitos AWS Flow Framework e conceitos básicos do Amazon SWF, descrevendo a estrutura básica de AWS Flow Framework um aplicativo e como os dados são trocados entre partes de um fluxo de trabalho distribuído.

AWS Flow Framework Guia de programação para Java

Este capítulo fornece orientações básicas de programação para o desenvolvimento de aplicações de fluxo de trabalho com o AWS Flow Framework para Java, incluindo como registrar tipos de atividade e fluxo de trabalho, implementar clientes de fluxo de trabalho, criar fluxos de trabalho secundários, tratar erros e muito mais.

Entendendo uma tarefa em AWS Flow Framework Java

Este capítulo fornece uma visão mais aprofundada da forma como o AWS Flow Framework para Java funciona, fornecendo informações adicionais sobre a ordem de execução de fluxos de trabalho assíncronos e um passo a passo lógico da execução de um fluxo de trabalho padrão.

Dicas de solução de problemas e depuração para Java AWS Flow Framework

Este capítulo fornece informações sobre erros comuns, que você pode usar para solucionar problemas de fluxos de trabalho ou para saber como evitar erros comuns.

AWS Flow Framework para referência de Java

Este capítulo é uma referência às anotações, exceções e pacotes que o AWS Flow Framework for Java adiciona ao SDK for Java.

Começando com o AWS Flow Framework for Java

Esta seção apresenta uma série AWS Flow Framework de exemplos simples de aplicativos que apresentam o modelo básico de programação e a API. Os aplicativos de exemplo são baseados no aplicativo padrão Hello World que é usado para introduzir C e as linguagens de programação relacionadas. Veja uma implementação típica em Java do Hello World:

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

A seguir está uma breve descrição dos aplicativos exemplo. Eles incluem o código fonte completo para que você possa implementar e executar os aplicativos. Antes de começar, você deve primeiro configurar seu ambiente de desenvolvimento e criar um projeto AWS Flow Framework para Java, como emConfigurando o AWS Flow Framework para Java.

- HelloWorld Aplicação introduz aplicativos de fluxo de trabalho ao implementar Hello World como um aplicativo Java padrão, mas estruturando-o como um aplicativo de fluxo de trabalho.
- HelloWorldWorkflow Aplicaçãousa o AWS Flow Framework for Java para converter HelloWorld em um fluxo de trabalho Amazon SWF.
- HelloWorldWorkflowAsyncAplicação modifica o `HelloWorldWorkflow` para usar um método de fluxo de trabalho assíncrono.
- HelloWorldWorkflowDistributed Aplicação modifica `HelloWorldWorkflowAsync` para que os operadores de fluxo de trabalho e de atividade possam ser executados em sistemas separados.
- HelloWorldWorkflowParallelAplicação modifica `HelloWorldWorkflow` para executar duas atividades em paralelo.

Configurando o AWS Flow Framework para Java

O AWS Flow Framework for Java está incluído no AWS SDK para Java. Se você ainda não configurou o AWS SDK para Java, acesse Introdução no Guia do AWS SDK para Java Desenvolvedor para obter informações sobre como instalar e configurar o SDK em si.

Adicione a estrutura de fluxo com o Maven

As ferramentas de criação do Amazon SWF são de código aberto. Para visualizar ou baixar o código ou para criar as ferramentas você mesmo, visite o repositório em. https://github.com/aws/aws-swf-build-tools

A Amazon fornece ferramentas de criação do Amazon SWF no repositório central do Maven.

Para configurar a estrutura de fluxo para o Maven, adicione a seguinte dependência a seu arquivo `pom.xml` do projeto:

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

HelloWorld Aplicação

Para apresentar a forma como as aplicações do Amazon SWF são estruturadas, criaremos uma aplicação Java que se comporta como um fluxo de trabalho, mas que é executada localmente em um único processo. Nenhuma conexão com a Amazon Web Services será necessária.

> **Note**
>
> O HelloWorldWorkflowexemplo se baseia nesse exemplo, conectando-se ao Amazon SWF para lidar com o gerenciamento do fluxo de trabalho.

Um aplicativo de fluxo de trabalho consiste em três componentes básicos:

- Um operador de atividades oferece suporte a um conjunto de atividades, cada uma das quais é um método que é executado independentemente para realizar uma tarefa específica.
- Um operador de fluxo de trabalho coordena a execução das atividades e gerencia o fluxo de dados. É uma realização programática de uma topologia de fluxo de trabalho, que é basicamente um fluxograma que define quando as várias atividades são executadas, se são executadas sequencialmente ou simultaneamente etc.
- Um iniciador de fluxo de trabalho inicia uma instância de fluxo de trabalho, chamada de uma execução, e pode interagir com ela durante a execução.

HelloWorld é implementado como três classes e duas interfaces relacionadas, descritas nas seções a seguir. Antes de começar, você deve configurar seu ambiente de desenvolvimento e criar um novo projeto AWS Java conforme descrito emConfigurando o AWS Flow Framework para Java. Todos os pacotes usados para as seguintes demonstrações são denominados `helloWorld.`*`XYZ`*. Para usar esses nomes, defina o atributo `within` no aop.xml seguinte da seguinte forma:

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

Para implementar HelloWorld, crie um novo pacote Java em seu projeto AWS SDK chamado `helloWorld.HelloWorld` e adicione os seguintes arquivos:

- Um arquivo de interface denominado `GreeterActivities.java`
- Um arquivo de classe denominado `GreeterActivitiesImpl.java`, que executa o operador de atividades.
- Um arquivo de interface denominado `GreeterWorkflow.java`.
- Um arquivo de classe denominado `GreeterWorkflowImpl.java`, que implementa o operador de fluxo de trabalho.
- Um arquivo de classe denominado `GreeterMain.java`, que implementa o iniciador do fluxo de trabalho.

Os detalhes são discutidos nas seções a seguir e incluem o código completo de cada componente, que você pode adicionar ao arquivo apropriado.

HelloWorld Implantação de atividades

HelloWorld divide a tarefa geral de imprimir uma `"Hello World!"` saudação no console em três tarefas, cada uma das quais é executada por um método de atividade. Os métodos de atividade são definidos na interface de `GreeterActivities`, da seguinte forma.

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

HelloWorld tem uma implementação de atividade`GreeterActivitiesImpl`,, que fornece os `GreeterActivities` métodos conforme mostrado:

```
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);
   }
}
```

As atividades são independentes uma da outra e podem ser usadas com frequência por fluxos de trabalho diferentes. Por exemplo, qualquer fluxo de trabalho pode usar a atividade `say` para imprimir uma sequência no console. Os fluxos de trabalho também podem ter várias implementações de atividade, cada uma executando um conjunto diferente de tarefas.

HelloWorld Trabalhador de fluxo

Para imprimir “Olá mundo!” para o console, as tarefas de atividade devem ser executadas em sequência, na ordem correta, com os dados corretos. O trabalhador do HelloWorld fluxo de trabalho orquestra a execução das atividades com base em uma topologia de fluxo de trabalho linear simples, mostrada na figura a seguir.

As três atividades são executadas em sequência e os dados fluem de uma atividade para a próxima.

O trabalhador do HelloWorld fluxo de trabalho tem um único método, o ponto de entrada do fluxo de trabalho, que é definido na `GreeterWorkflow` interface, da seguinte forma:

```
public interface GreeterWorkflow {
```

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

A classe `GreeterWorkflowImpl` implementa essa interface, da seguinte forma:

```
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);
   }
}
```

O `greet` método implementa a HelloWorld topologia criando uma instância de`GreeterActivitiesImpl`, chamando cada método de atividade na ordem correta e transmitindo os dados apropriados para cada método.

HelloWorld Iniciador de fluxo de trabalho

Um iniciador de fluxo de trabalho é um aplicativo que inicia uma execução de fluxo de trabalho e também pode se comunicar com o fluxo de trabalho enquanto o executa. A `GreeterMain` classe implementa o iniciador do HelloWorld fluxo de trabalho, da seguinte forma:

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

`GreeterMain` cria uma instância de `GreeterWorkflowImpl` e chama `greet` para executar o operador do fluxo de trabalho. Execute `GreeterMain` como uma aplicação Java, e você verá "Hello World!" na saída do console.

HelloWorldWorkflow Aplicação

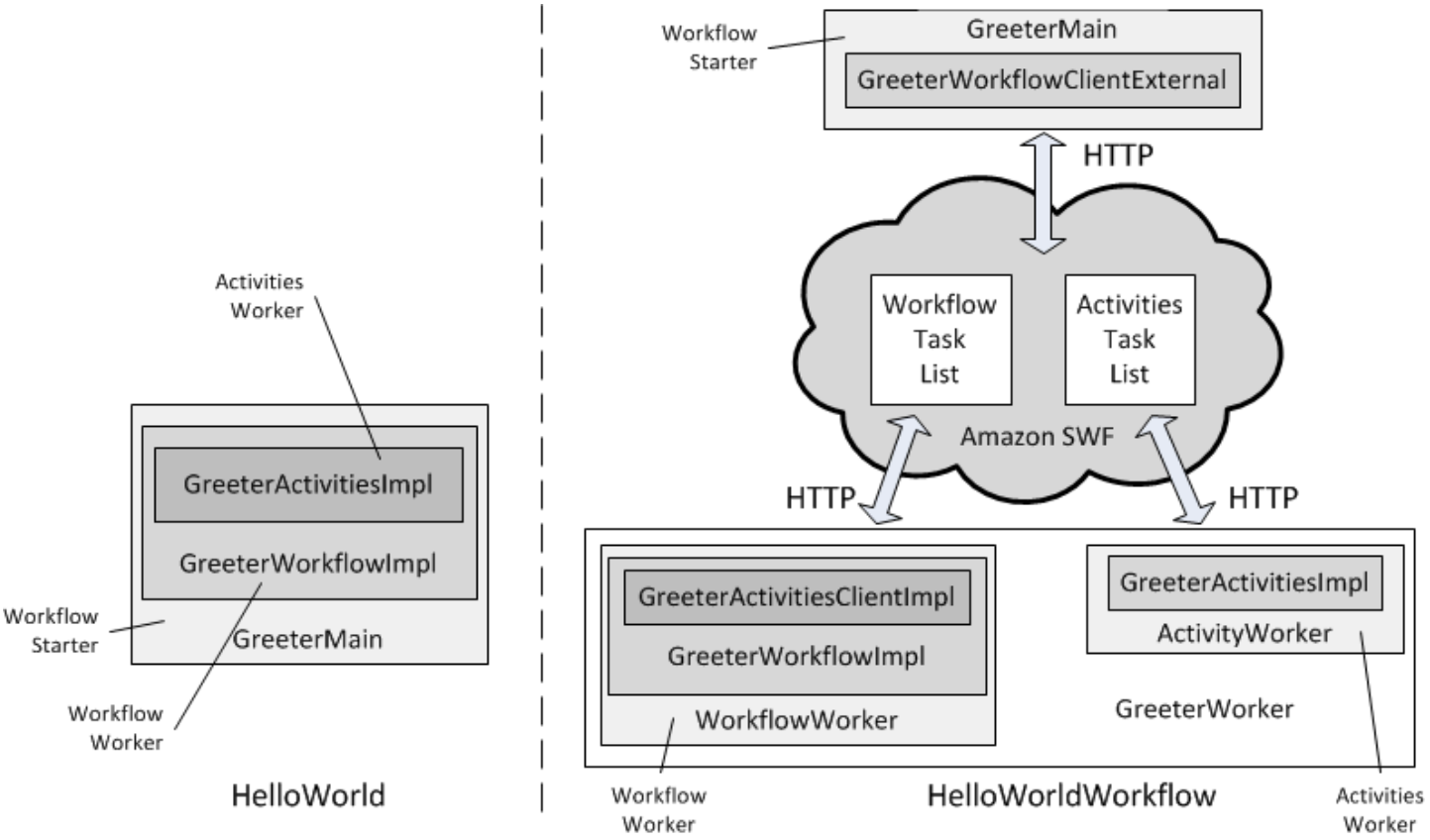
Embora o HelloWorldexemplo básico seja estruturado como um fluxo de trabalho, ele difere de um fluxo de trabalho do Amazon SWF em vários aspectos importantes:

Aplicações de fluxo de trabalho convencionais e do Amazon SWF

HelloWorld	Fluxo de trabalho do Amazon SWF
É executado localmente como um processo único.	É executado como vários processos que podem ser distribuídos em vários sistemas, incluindo EC2 instâncias da Amazon, data centers privados, computadores clientes e assim por diante. Ele nem precisa ser executado no mesmo sistema operacional.
As atividades são métodos síncronos, que são bloqueados até que sejam concluídos.	As atividades são representadas por métodos assíncronos, que retornam imediatamente e permitem que o fluxo de trabalho execute outras tarefas enquanto aguarda que a atividade seja concluída.
O operador do fluxo de trabalho interage com um operador de atividades chamando o método apropriado.	Os operadores do fluxo de trabalho interagem com os operadores das atividades usando solicitações HTTP, com o Amazon SWF atuando como intermediário.
O iniciador do fluxo de trabalho interage com o operador do fluxo de trabalho chamando o método apropriado.	Os iniciadores de fluxo de trabalho interagem com os operadores do fluxo de trabalho usando solicitações HTTP, com o Amazon SWF atuando como um intermediário.

Você pode implementar um aplicativo de fluxo de trabalho assíncrono distribuído a partir do zero, por exemplo, fazendo com que o operador do fluxo de trabalho interaja com um operador de atividades diretamente por meio de chamadas a serviços web. No entanto, você deve implementar todo o código complicado necessário para gerenciar a execução assíncrona de várias atividades, manipular o fluxo de dados etc. O AWS Flow Framework for Java e o Amazon SWF cuidam de todos esses detalhes, o que permite que você se concentre na implementação da lógica de negócios.

HelloWorldWorkflow é uma versão modificada HelloWorld que é executada como um fluxo de trabalho do Amazon SWF. A figura a seguir resume a forma como os dois aplicativos funcionam.



HelloWorld é executado como um único processo e o iniciante, o trabalhador do fluxo de trabalho e o trabalhador das atividades interagem usando chamadas de métodos convencionais. Com `HelloWorldWorkflow`, o iniciador, o operador do fluxo de trabalho e o operador de atividades são componentes distribuídos que interagem por meio do Amazon SWF usando solicitações HTTP. O Amazon SWF gerencia a interação mantendo listas de tarefas de fluxo de trabalho e atividades, que são despachadas para os respectivos componentes. Esta seção descreve como a estrutura funciona para HelloWorldWorkflow.

HelloWorldWorkflow é implementado usando a API AWS Flow Framework for Java, que lida com os detalhes às vezes complicados da interação com o Amazon SWF em segundo plano e simplifica consideravelmente o processo de desenvolvimento. Você pode usar o mesmo projeto que usou HelloWorld, que já está configurado AWS Flow Framework para aplicativos Java. No entanto, para executar a aplicação, você deve configurar uma conta do Amazon SWF, como segue:

- Crie uma AWS conta, se você ainda não tiver uma, na Amazon Web Services.
- Atribua o ID de acesso e o ID secreto da sua conta às variáveis de AWS_SECRET_KEY ambiente AWS_ACCESS_KEY_ID e, respectivamente. É uma boa prática não expor os valores literais

das chaves no código. O armazenamento das chaves em variáveis de ambiente é uma maneira conveniente para lidar com o problema.

- Registre-se para obter uma conta Amazon SWF no Amazon Simple Workflow Service.
- Faça login AWS Management Console e selecione o serviço Amazon SWF.
- Escolha Gerenciar domínios no canto superior direito e registre um novo domínio Amazon SWF. Um domínio é um contêiner lógico dos recursos de seus aplicativos, como os tipos de fluxo de trabalho e de atividade e as execuções de fluxo de trabalho. Você pode usar qualquer nome de domínio conveniente, mas as orientações usam "". helloWorldWalkthrough

Para implementar o HelloWorldWorkflow, crie uma cópia do HelloWorld. HelloWorld pacote no diretório do seu projeto e chame-o de HelloWorld. HelloWorldWorkflow. As seções a seguir descrevem como modificar o HelloWorld código original para usar o AWS Flow Framework para Java e executá-lo como um aplicativo de fluxo de trabalho do Amazon SWF.

HelloWorldWorkflow Trabalhador de atividades

HelloWorld implementou suas atividades de trabalhador como uma única classe. Um trabalhador de atividades AWS Flow Framework para Java tem três componentes básicos:

- Os métodos de atividade, que executam as tarefas reais, são definidos em uma interface e implementados em uma classe relacionada.
- Uma ActivityWorkerclasse gerencia a interação entre os métodos de atividade e o Amazon SWF.
- Um aplicativo host de atividades registra e inicia o operador de atividades e processa a limpeza.

Esta seção discute os métodos de atividade. As outras duas classes são discutidas posteriormente.

HelloWorldWorkflow define a interface de atividades em`GreeterActivities`, da seguinte forma:

```
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);
}
```

Essa interface não era estritamente necessária para HelloWorld, mas é AWS Flow Framework para um aplicativo Java. Observe que a própria definição da interface não foi alterada. No entanto, você deve aplicar dois AWS Flow Framework para anotações Java @ActivityRegistrationOptions e para a @Atividades definição da interface. As anotações fornecem informações de configuração e orientam o processador de anotações AWS Flow Framework para Java a usar a definição da interface para gerar uma classe cliente de atividades, que será discutida posteriormente.

`@ActivityRegistrationOptions`tem vários valores nomeados que são usados para configurar o comportamento das atividades. HelloWorldWorkflow especifica dois tempos limite:

- `defaultTaskScheduleToStartTimeoutSeconds` especifica por quanto tempo as tarefas podem ser enfileiradas na lista de tarefas de atividades, e é definido como 300 segundos (5 minutos).
- `defaultTaskStartToCloseTimeoutSeconds` especifica o tempo máximo que a atividade pode levar para executar a tarefa e é definido como 10 segundos.

Esses tempos limite garantem que a atividade conclua sua tarefa em uma quantidade de tempo razoável. Se um dos tempos limite for excedido, a estrutura gerará um erro e o operador do fluxo de trabalho deverá decidir como lidar com o problema. Para obter uma discussão de como tratar esses erros, consulte Tratamento de erros.

`@Activities` tem vários valores, mas geralmente apenas especifica o número da versão das atividades, permitindo que você controle as diferentes gerações de implementações de atividades. Se você alterar uma interface de atividade depois de registrá-la no Amazon SWF, incluindo a alteração dos valores `@ActivityRegistrationOptions`, deverá usar um novo número de versão.

HelloWorldWorkflow implementa os métodos de atividade em`GreeterActivitiesImpl`, da seguinte forma:

```
public class GreeterActivitiesImpl implements GreeterActivities {
    @Override
    public String getName() {
```

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

Observe que o código é idêntico à HelloWorld implementação. Em essência, uma AWS Flow Framework atividade é apenas um método que executa algum código e talvez retorne um resultado. A diferença entre uma aplicação padrão e uma aplicação de fluxo de trabalho do Amazon SWF está em como o fluxo de trabalho executa as atividades, onde as atividades são executadas e como os resultados são retornados ao operador do fluxo de trabalho.

HelloWorldWorkflow Trabalhador de fluxo

Um operador de fluxo de trabalho do Amazon SWF tem três componentes básicos.

- Uma implementação de fluxo de trabalho que é uma classe que executa tarefas relacionadas ao fluxo de trabalho.
- Uma classe de cliente de atividades que é basicamente um proxy da classe de atividades e é usada por uma implementação de fluxo de trabalho para executar métodos de atividade de forma assíncrona.
- Uma WorkflowWorkerclasse que gerencia a interação entre o fluxo de trabalho e o Amazon SWF.

Esta seção discute a implementação do fluxo de trabalho e do cliente de atividades. A classe `WorkflowWorker` é discutida posteriormente.

HelloWorldWorkflow define a interface do fluxo de trabalho em`GreeterWorkflow`, da seguinte forma:

```
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();
}
```

Essa interface também não é estritamente necessária HelloWorld , mas é essencial AWS Flow Framework para um aplicativo Java. Você deve aplicar dois AWS Flow Framework para anotações Java @Fluxo de trabalho e @WorkflowRegistrationOptions para a definição da interface do fluxo de trabalho. As anotações fornecem informações de configuração e também orientam o processador de anotações AWS Flow Framework para Java a gerar uma classe cliente de fluxo de trabalho com base na interface, conforme discutido posteriormente.

`@Workflow`tem um parâmetro opcional, dataConverter, que geralmente é usado com seu valor padrão NullDataConverter, o que indica JsonDataConverter que deve ser usado.

`@WorkflowRegistrationOptions` também tem vários parâmetros opcionais que podem ser usados para configurar o operador de fluxo de trabalho. Aqui, definimos `defaultExecutionStartToCloseTimeoutSeconds`, que especifica por quanto tempo o fluxo de trabalho pode ser executado, como 3.600 segundos (1 hora).

A definição da `GreeterWorkflow` interface difere de uma HelloWorld forma importante, a @Execute anotação. As interfaces de fluxo de trabalho especificam os métodos que podem ser chamados por aplicativos, como o iniciador do fluxo de trabalho, e são limitadas a alguns dos métodos, cada um com uma função específica. O framework não especifica um nome ou uma lista de parâmetros para métodos de interface de fluxo de trabalho; você usa um nome e uma lista de parâmetros adequados ao seu fluxo de trabalho e aplica uma anotação do AWS Flow Framework para Java para identificar a função do método.

`@Execute` tem dois objetivos:

- Identifica `greet` como o ponto de entrada do fluxo de trabalho — o método que o iniciador do fluxo de trabalho chama para iniciar o fluxo de trabalho. Geralmente, um ponto de entrada pode ter um ou mais parâmetros, o que permite que o iniciador inicialize o fluxo de trabalho, mas este exemplo não requer inicialização.
- Ele especifica o número da versão do fluxo de trabalho permitindo que você controle as diferentes gerações de implementações de fluxo de trabalho. Para alterar uma interface de fluxo de trabalho

depois de registrá-la no Amazon SWF, incluindo a alteração dos valores de tempo limite, você deve usar um novo número de versão.

Para obter informações sobre os outros métodos que podem ser incluídos em uma interface de fluxo de trabalho, consulte Contratos de atividades e de fluxo de trabalho.

HelloWorldWorkflow implementa o fluxo de trabalho em`GreeterWorkflowImpl`, da seguinte forma:

```
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);
   }
}
```

O código é semelhante HelloWorld, mas com duas diferenças importantes.

- O `GreeterWorkflowImpl` cria uma instância de `GreeterActivitiesClientImpl`, o cliente de atividades, em vez de `GreeterActivitiesImpl`, e executa as atividades chamando os métodos no objeto do cliente.
- Atividades de nome e de saudação retornam objetos `Promise<String>` em vez de objetos `String`.

HelloWorld é um aplicativo Java padrão executado localmente como um único processo, portanto, `GreeterWorkflowImpl` pode implementar a topologia do fluxo de trabalho simplesmente criando uma instância de`GreeterActivitiesImpl`, chamando os métodos em ordem e passando os valores de retorno de uma atividade para a próxima. Com um fluxo de trabalho do Amazon SWF, a tarefa de uma atividade ainda é executada por um método de atividade de `GreeterActivitiesImpl`. No entanto, o método não é necessariamente executado no mesmo processo que o fluxo de trabalho, talvez ele nem execute no mesmo sistema, e o fluxo de trabalho precisa executar a atividade de maneira assíncrona. Esses requisitos geram os seguintes problemas:

- Como executar um método de atividade que possa ser executado em outro processo, talvez em outro sistema.

- Como executar um método de atividade de forma assíncrona.
- Como gerenciar entradas e valores de retorno de atividades. Por exemplo, se o valor de retorno da atividade A for uma entrada para a atividade B, você deverá garantir que a atividade B não seja executada até que a atividade A esteja concluída.

Você pode implementar várias topologias de fluxo de trabalho por meio do fluxo de controle do aplicativo usando o controle de fluxo familiar do Java combinado com o cliente de atividades e o `Promise<T>`.

Cliente de atividades

O `GreeterActivitiesClientImpl` é basicamente um proxy para `GreeterActivitiesImpl` que permite que uma implementação de fluxo de trabalho execute os métodos `GreeterActivitiesImpl` de forma assíncrona.

As classes `GreeterActivitiesClient` e `GreeterActivitiesClientImpl` são geradas automaticamente para você usando as informações fornecidas nas anotações aplicadas a sua classe `GreeterActivities`. Não é necessário implementá-las você mesmo.

> **Note**
>
> O Eclipse gera essas classes quando você salva o projeto. Você pode visualizar o código gerado no subdiretório `.apt_generated` do diretório do projeto.
> Para evitar erros de compilação na classe `GreeterWorkflowImpl`, é uma boa prática mover o diretório `.apt_generated` para a parte superior da guia Order and Export (Pedir e exportar) da caixa de diálogo Java Build Path (Caminho de compilação do Java).

Um operador de fluxo de trabalho executa uma atividade chamando o método do cliente correspondente. O método é assíncrono e retorna imediatamente um objeto `Promise<T>`, em que T é o tipo de retorno da atividade. O objeto `Promise<T>` retornado é basicamente um espaço reservado para o valor que o método da atividade retornará eventualmente.

- Quando o método do cliente de atividades retorna, o objeto `Promise<T>` inicialmente está em um estado não pronto, o que indica que o objeto ainda não representa um valor de retorno válido.
- Quando o método da atividade correspondente conclui suas tarefas e retornos, a estrutura atribui o valor de retorno ao objeto `Promise<T>` e o coloca no estado pronto.

Promessa <T> Type

A finalidade principal de objetos `Promise<T>` é gerenciar o fluxo de dados entre componentes assíncronos e controlar quando eles são executados. Ele libera o aplicativo da necessidade de gerenciar explicitamente a sincronização ou de depender de mecanismos como temporizadores para garantir que os componentes assíncronos não sejam executados prematuramente. Quando você chama um método de cliente de atividades, ele retorna imediatamente, mas a estrutura adia a execução do método de atividade correspondente até que todos os objetos `Promise<T>` de entrada estejam prontos e representem dados válidos.

Da perspectiva do `GreeterWorkflowImpl`, todos os três métodos de cliente de atividades retornam imediatamente. Da perspectiva do `GreeterActivitiesImpl`, a estrutura não chama `getGreeting` até que `name` seja concluído e não chama `say` até que `getGreeting` seja concluído.

O uso de `Promise<T>` para passar dados de uma atividade para a próxima, `HelloWorldWorkflow` não só garante que os métodos de atividade não tentem usar dados inválidos, ela também controla quando as atividades são executadas e define implicitamente a topologia do fluxo de trabalho. A passagem do valor de retorno `Promise<T>` de cada atividade para a atividade seguinte requer que as atividades sejam executadas em sequência definindo a topologia linear discutida anteriormente. Com AWS Flow Framework o for Java, você não precisa usar nenhum código de modelagem especial para definir até mesmo topologias complexas, apenas controle de fluxo Java padrão e. `Promise<T>` Para obter um exemplo de como implementar uma topologia paralela simples, consulte HelloWorldWorkflowParallelTrabalhador de atividades.

> **Note**
>
> Quando um método de atividade como `say` não retorna um valor, o método do cliente correspondente retorna um objeto `Promise<Void>`. O objeto não representa dados, mas inicialmente não está pronto e se torna pronto quando a atividade é concluída. Portanto, você pode passar um objeto `Promise<Void>` para outros métodos de cliente de atividades para garantir que eles adiem a execução até que a atividade original seja concluída.

O `Promise<T>` permite que uma implementação de fluxo de trabalho use métodos de cliente de atividades e seus valores de retorno da mesma forma como os métodos sincronizados. No entanto, você deve ter cuidado ao acessar o valor de um objeto `Promise<T>`. Ao contrário do tipo Future<T> do Java, a estrutura controla a sincronização de `Promise<T>`, não o aplicativo. Se você

chamar `Promise<T>.get` e o objeto não estiver pronto, `get` gerará uma exceção. Observe que o `HelloWorldWorkflow` nunca acessa um objeto `Promise<T>` diretamente. Ele simplesmente passa os objetos de uma atividade para a próxima. Quando um objeto torna-se pronto, a estrutura extrai o valor e passa-o para o método de atividade como um tipo padrão.

Os objetos `Promise<T>` devem ser acessados apenas pelo código assíncrono, em que a estrutura garante que o objeto está pronto e representa um valor válido. O `HelloWorldWorkflow` trata esse problema passando os objetos `Promise<T>` somente para métodos de cliente de atividades. Você pode acessar o valor de um objeto `Promise<T>` em sua implementação de fluxo de trabalho passando o objeto para um método de fluxo de trabalho assíncrono, que se comporta de forma semelhante à forma de uma atividade. Para obter um exemplo, consulte HelloWorldWorkflowAsyncAplicação.

HelloWorldWorkflow Fluxo de trabalho e implementação de atividades

As implementações de fluxo de trabalho e atividades têm classes de trabalhadores associadas ActivityWorkere. WorkflowWorker Eles lidam com a comunicação entre o Amazon SWF e as atividades e implementações de fluxo de trabalho, pesquisando a lista de tarefas apropriada do Amazon SWF para tarefas, executando o método apropriado para cada tarefa e gerenciando o fluxo de dados. Para obter detalhes, consulte AWS Flow Framework Conceitos básicos: Estrutura do aplicativo

Para associar as implementações das atividades e do fluxo de trabalho aos objetos correspondentes do operador, você implementa um ou mais aplicativos de operador que:

- Registre fluxos de trabalho ou atividades com o Amazon SWF.
- Cria objetos de operador e os associa às implementações operador de fluxo de trabalho ou de atividades.
- Direcione os objetos de trabalho para iniciar a comunicação com o Amazon SWF.

Para executar o fluxo de trabalho e as atividades como processos separados, você deve implementar hosts separados de operador de fluxo de trabalho e de atividades. Para obter um exemplo, consulte HelloWorldWorkflowDistributed Aplicação. Para simplificar, HelloWorldWorkflow implementa um único host de trabalho que executa atividades e trabalhadores do fluxo de trabalho no mesmo processo, da seguinte forma:

```
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`não tem HelloWorld contrapartida, então você deve adicionar uma classe Java nomeada `GreeterWorker` ao projeto e copiar o código de exemplo para esse arquivo.

A primeira etapa é criar e configurar um AmazonSimpleWorkflowClientobjeto, que invoca os métodos de serviço subjacentes do Amazon SWF. Para fazer isso, o `GreeterWorker`:

1. Cria um ClientConfigurationobjeto e especifica um tempo limite de soquete de 70 segundos. Esse valor especifica por quanto tempo esperar para que os dados sejam transferidos por uma conexão aberta estabelecida antes de fechar o soquete.

2. Cria um AWSCredentials objeto básico para identificar a AWS conta e passa as chaves da conta para o construtor. Por conveniência, e para evitar sua exposição como texto simples no código, as chaves são armazenadas como variáveis de ambiente.
3. Cria um AmazonSimpleWorkflowClientobjeto para representar o fluxo de trabalho e passa os `ClientConfiguration` objetos `BasicAWSCredentials` e para o construtor.
4. Define o URL do endpoint do serviço do objeto cliente. Atualmente, o Amazon SWF está disponível em todas as AWS regiões.

Por conveniência, o `GreeterWorker` define duas constantes de sequências de caracteres.

- `domain`é o nome de domínio Amazon SWF do fluxo de trabalho, que você criou ao configurar sua conta Amazon SWF. `HelloWorldWorkflow`presume que você esteja executando o fluxo de trabalho no domínio helloWorldWalkthrough "".
- `taskListToPoll` é o nome das listas de tarefas que o Amazon SWF usa para gerenciar a comunicação entre o fluxo de trabalho e os operadores das atividades. Você pode definir o nome para qualquer sequência de caracteres conveniente. HelloWorldWorkflow usa "HelloWorldList" para listas de tarefas de fluxo de trabalho e atividades. Nos bastidores, os nomes ficam em namespaces diferentes e, portanto, as duas listas de tarefas são distintas.

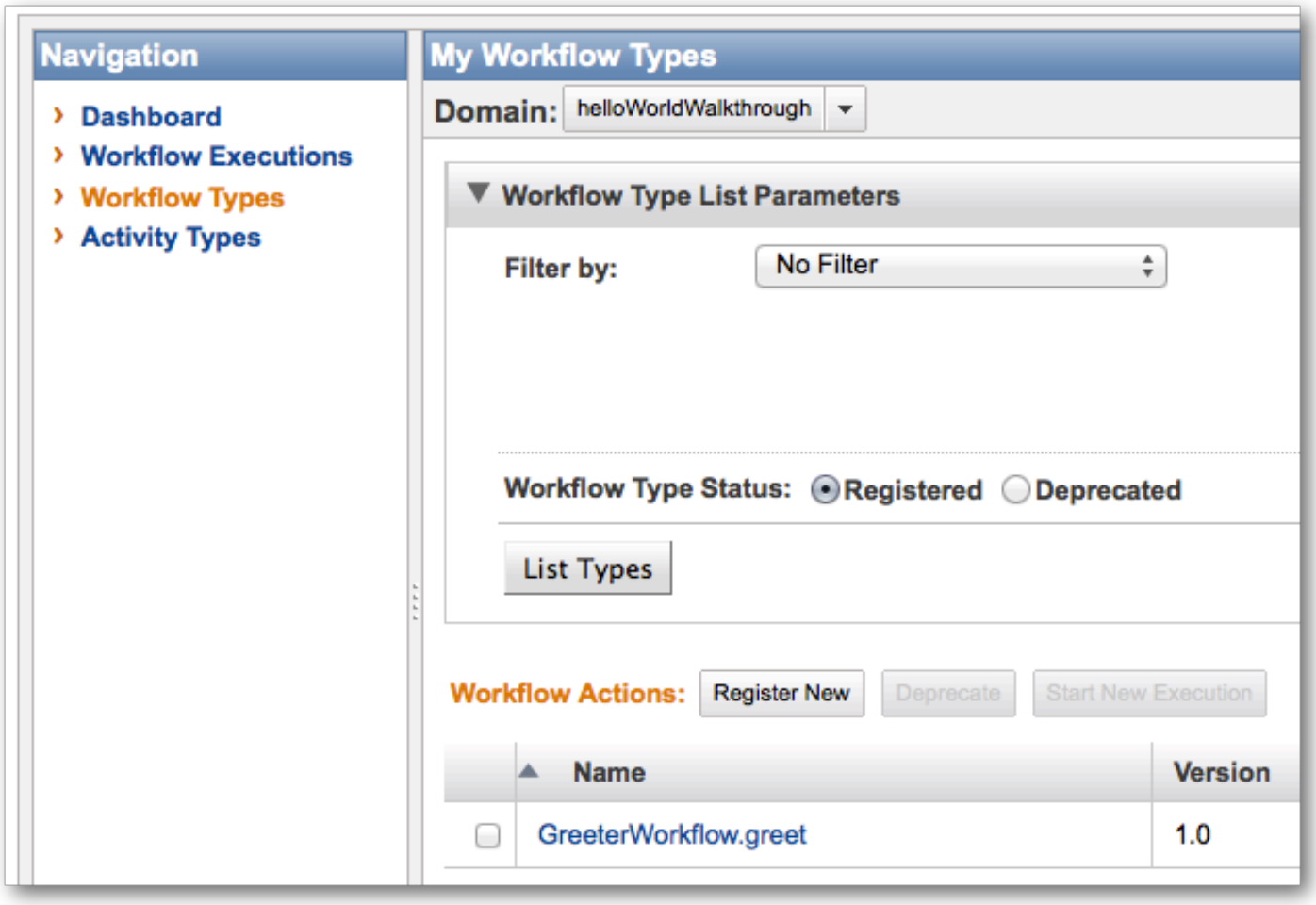
`GreeterWorker`usa as constantes de string e o AmazonSimpleWorkflowClientobjeto para criar objetos de trabalho, que gerenciam a interação entre as atividades e as implementações do trabalhador e o Amazon SWF. Especificamente, os objetos de operador processam a tarefa de pesquisa de tarefas na lista de tarefas adequada.

O `GreeterWorker` cria um objeto `ActivityWorker` e o configura para processar o `GreeterActivitiesImpl` adicionando uma nova instância de classe. Em seguida, o `GreeterWorker` chama o método `ActivityWorker` do objeto `start`, que direciona o objeto para iniciar a pesquisa na lista de tarefas de atividades especificada.

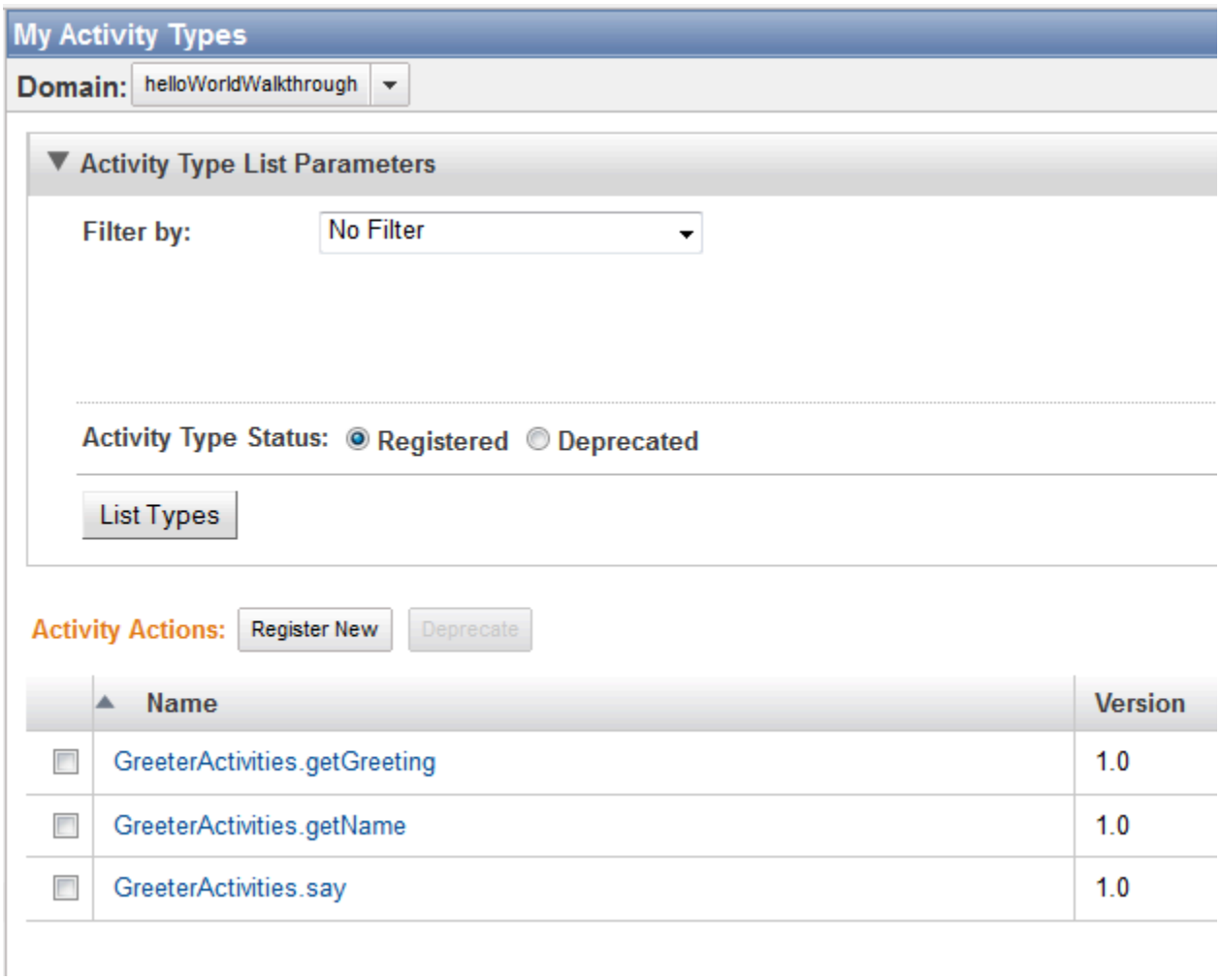
O `GreeterWorker` cria um objeto `WorkflowWorker` e o configura para processar o `GreeterWorkflowImpl` adicionando o nome do arquivo de classe, `GreeterWorkflowImpl.class`. Em seguida, ele chama o método do objeto `WorkflowWorker`, `start`, que direciona o objeto para iniciar a pesquisa na lista de tarefas de fluxo de trabalho especificada.

Você pode executar o `GreeterWorker` com êxito neste ponto. Ele registra o fluxo de trabalho e as atividades com o Amazon SWF e inicia os objetos de trabalho pesquisando suas respectivas listas

de tarefas. Para verificar isso, execute `GreeterWorker` e vá para o console do Amazon SWF e selecione `helloWorldWalkthrough` na lista de domínios. Se você escolher Workflow Types (Tipos de fluxo de trabalho) no painel Navigation (Navegação), deverá ver o `GreeterWorkflow.greet`:



Se você escolher Activity Types (Tipos de atividade), os métodos de `GreeterActivities` serão exibidos:



No entanto, se você escolher Workflow Executions (Execuções de fluxo de trabalho), você não verá nenhuma execução ativa. Embora os operadores de fluxo de trabalho e de atividades estejam pesquisando tarefas, ainda não iniciamos uma execução de fluxo de trabalho.

HelloWorldWorkflow Iniciante

A parte final do quebra-cabeça é implementar um iniciador de fluxo de trabalho que seja um aplicativo que inicia a execução do fluxo de trabalho. O estado de execução é armazenado pelo Amazon SWF, para que você possa visualizar seu histórico e status de execução. HelloWorldWorkflow implementa um iniciador de fluxo de trabalho modificando a `GreeterMain` classe, da seguinte forma:

```
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();
   }
}
```

O `GreeterMain` cria um objeto `AmazonSimpleWorkflowClient` usando o mesmo código que o `GreeterWorker` usa. Em seguida, ele cria um objeto `GreeterWorkflowClientExternal` que atua como um proxy para o fluxo de trabalho da mesma forma como o cliente de atividades criado em `GreeterWorkflowClientImpl` atua como um proxy para os métodos de atividade. Em vez de criar um objeto de cliente de fluxo de trabalho usando `new`, você deve:

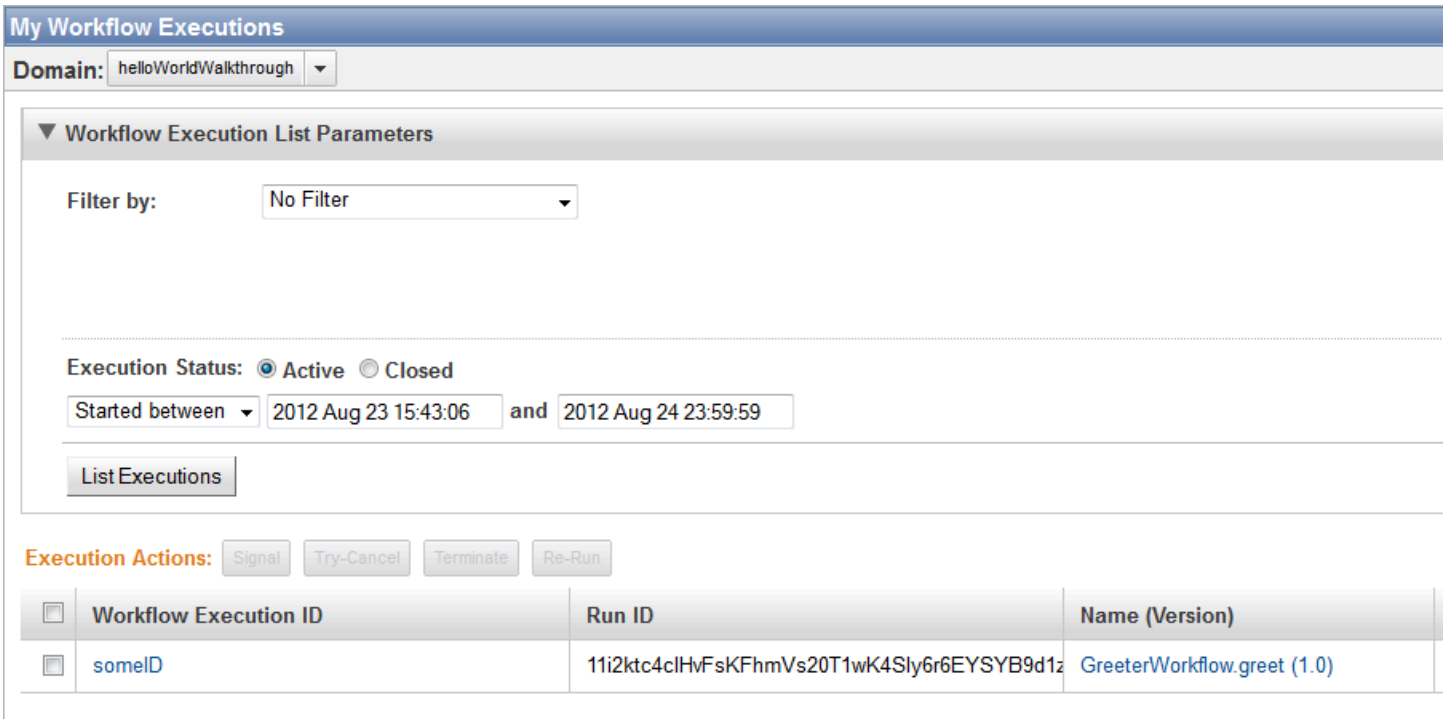
1. Crie um objeto de fábrica de cliente externo e passe o objeto `AmazonSimpleWorkflowClient` e o nome de domínio do Amazon SWF para o construtor. O objeto de fábrica do cliente é criado pelo processador de anotações da estrutura, que cria o nome do objeto simplesmente anexando "ClientExternalFactoryImpl" ao nome da interface do fluxo de trabalho.
2. Crie um objeto cliente externo chamando o `getClient` método do objeto de fábrica, que cria o nome do objeto anexando "ClientExternal" ao nome da interface do fluxo de trabalho. Opcionalmente, você pode passar `getClient`, uma string que o Amazon SWF usará para identificar essa instância do fluxo de trabalho. Caso contrário, o Amazon SWF representa uma instância de fluxo de trabalho usando um GUID gerado.

O cliente retornado da fábrica criará apenas fluxos de trabalho nomeados com a string passada no método getClient (o cliente retornado da fábrica já tem estado no Amazon SWF). Para executar um fluxo de trabalho com outro id, você precisará voltar à fábrica e criar um cliente novo com o outro id especificado.

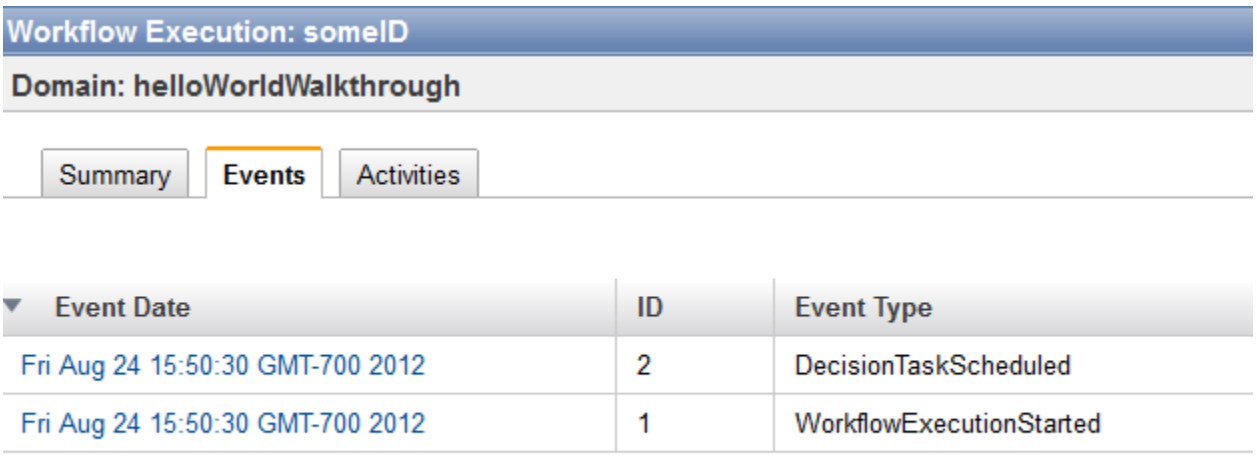
O cliente de fluxo de trabalho expõe um método `greet` que o `GreeterMain` chama para iniciar o fluxo de trabalho, porque `greet()` foi o método especificado com a anotação `@Execute`.

> **Note**
>
> O processador de anotação também cria um objeto interno da fábrica de cliente que é usado para criar fluxos de trabalho filhos. Para obter detalhes, consulte Execuções do fluxo de trabalho filho.

Desligue o `GreeterWorker` por enquanto, se ele ainda estiver em execução e executar `GreeterMain`. Agora você deve ver someID na lista de execuções de fluxo de trabalho ativas do console do Amazon SWF:



Se você escolher `someID` e escolher a guia Events (Eventos), os eventos serão exibidos:

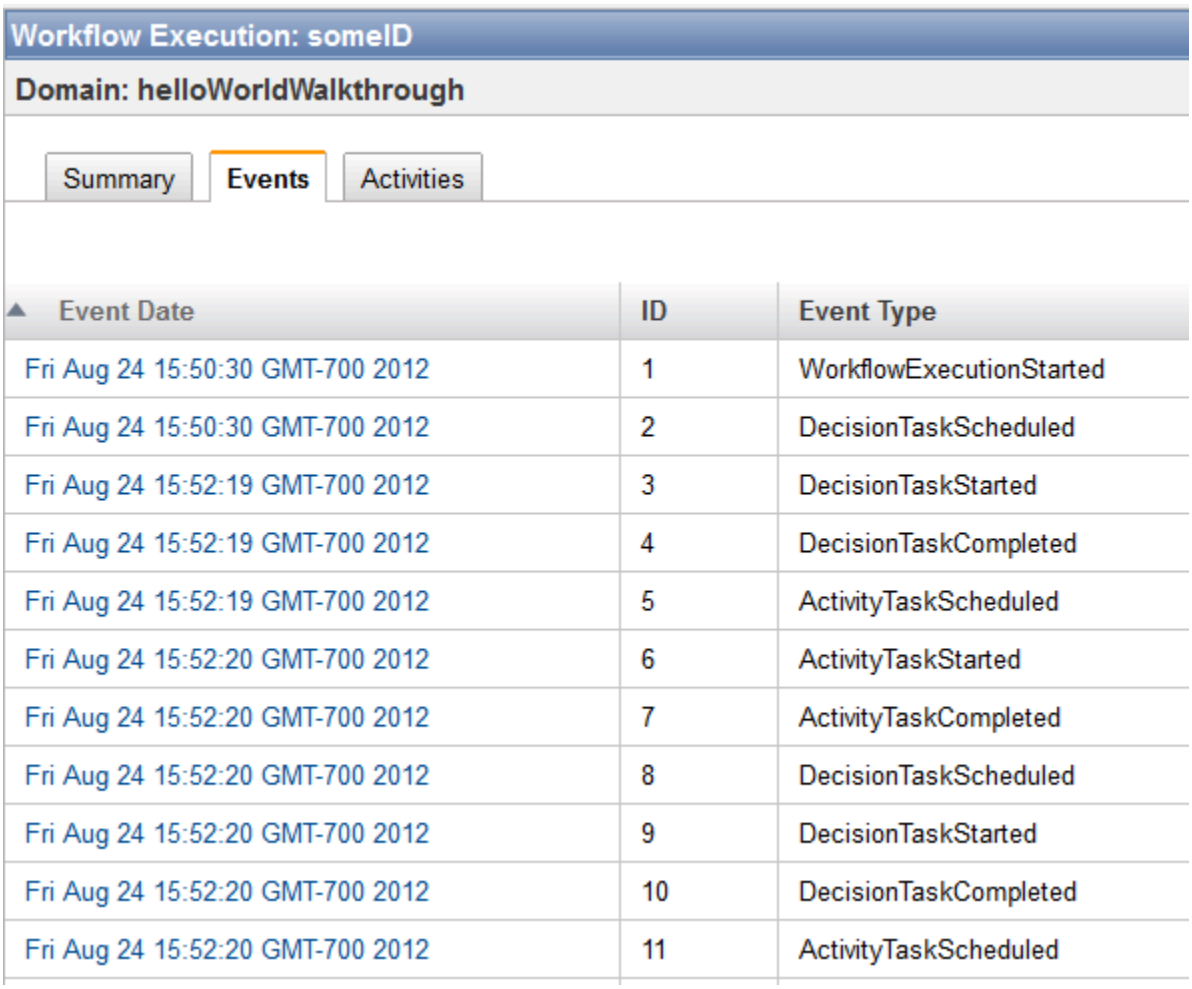


> **Note**
>
> Se você tiver iniciado o `GreeterWorker` anteriormente, e ele inda estiver em execução, verá uma lista de eventos mais longa pelos motivos discutidos brevemente. Interrompa o `GreeterWorker` e tente executar `GreaterMain` novamente.

A guia Events (Eventos) mostra apenas dois eventos:

- O `WorkflowExecutionStarted` indica que o fluxo de trabalho começou a execução.
- `DecisionTaskScheduled` indica que o Amazon SWF enfileirou a primeira tarefa de decisão.

O motivo pelo qual o fluxo de trabalho é bloqueado na primeira tarefa de administração é que o fluxo de trabalho é distribuído entre dois aplicativos, `GreeterMain` e `GreeterWorker`. O `GreeterMain` começou a execução do fluxo de trabalho, mas o `GreeterWorker` não está em execução, portanto os operadores não estão pesquisando as listas e executando as tarefas. Você pode executar qualquer um dos aplicativos de forma independente, mas precisa dos dois para que a execução do fluxo de trabalho continue além da primeira tarefa de decisão. Se você executar o `GreeterWorker` agora, os operadores de fluxo de trabalho e de atividades começarão a pesquisa, e as várias tarefas serão concluídas rapidamente. Se você verificar a guia `Events` agora, o primeiro lote de eventos será exibido.



Workflow Execution: someID

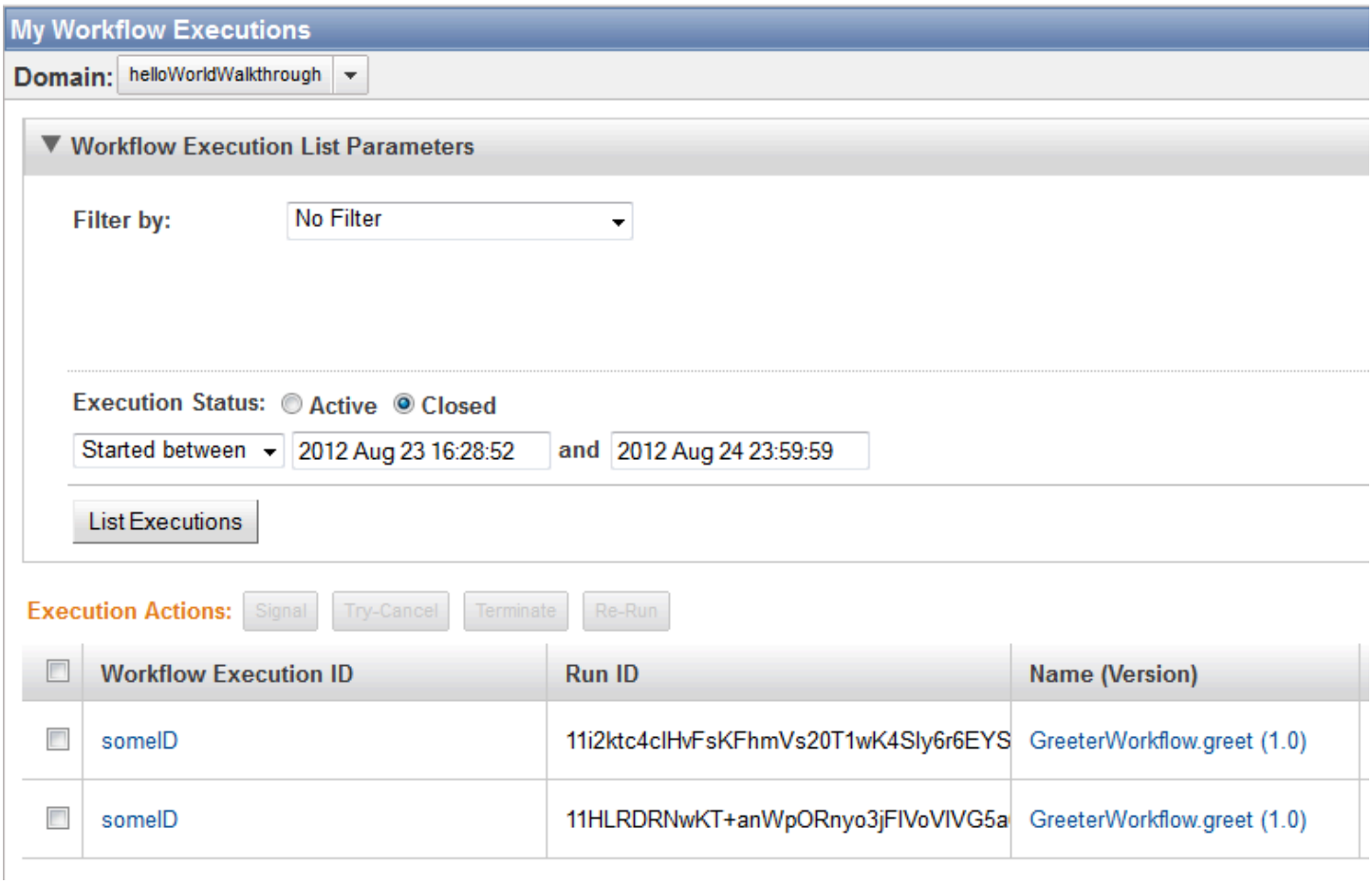
Domain: helloWorldWalkthrough

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

Você pode escolher eventos individuais para obter mais informações. Quando você terminar de olhar, o fluxo de trabalho deverá ter impresso "Hello World!". para o seu console.

Quando o fluxo de trabalho estiver concluído, ele não aparecerá mais na lista de execuções ativas. No entanto, para revê-lo, escolha o botão de status de execução Closed (Fechado) e, em seguida, escolha List Executions (Listar execuções). Isso exibe todas as instâncias de fluxo de trabalho concluídas no domínio especificado (`helloWorldWalkthrough`) que não excederam o período de retenção especificado ao criar o domínio.



Observe que cada instância de fluxo de trabalho tem um valor de Run ID (ID de execução) exclusivo. Você pode usar a mesma ID do fluxo de trabalho para diferentes instâncias do fluxo de trabalho, mas somente para uma execução ativa por vez.

HelloWorldWorkflowAsyncAplicação

Às vezes, é preferível que um fluxo de trabalho execute certas tarefas localmente em vez de usar uma atividade. No entanto, as tarefas do fluxo de trabalho geralmente envolvem o processamento dos valores representados pelos objetos `Promise<T>`. Se enviar um objeto `Promise<T>` para um método do fluxo de trabalho síncrono, o método executa imediatamente mas não é capaz de acessar o valor do objeto `Promise<T>` até que ele esteja pronto. Você pode consultar `Promise<T>.isReady` até ele retornar `true`, mas isso é ineficiente e o método pode ser bloqueado por um longo período. Uma abordagem melhor é usar um método assíncrono.

Um método assíncrono é implementado de forma muito semelhante a um método padrão, geralmente como membro da classe de implementação do fluxo de trabalho, e é executado no

contexto da implementação do fluxo de trabalho. Designe-o como um método assíncrono aplicando uma anotação `@Asynchronous`, que direciona a estrutura para tratá-lo como uma atividade.

- Quando uma implementação do fluxo de trabalho chama um método assíncrono, ele retorna imediatamente. Os métodos assíncronos geralmente retornam um objeto `Promise<T>`, o qual se torna pronto quando o método for concluído.
- Se enviar um ou mais objetos `Promise<T>` para um método assíncrono, ele adia a execução até que todos os objetos de entrada estejam prontos. Um método assíncrono pode, portanto, acessar os valores `Promise<T>` da sua entrada sem risco de uma exceção.

> **Note**
>
> Devido à forma como o AWS Flow Framework for Java executa o fluxo de trabalho, os métodos assíncronos geralmente são executados várias vezes, portanto, você deve usá-los somente para tarefas rápidas e de baixa sobrecarga. Use atividades para executar tarefas longas como computações grandes. Para obter detalhes, consulte AWS Flow Framework Conceitos básicos: execução distribuída.

Este tópico é um passo a passo de HelloWorldWorkflowAsync, uma versão modificada HelloWorldWorkflow que substitui uma das atividades por um método assíncrono. Para implementar o aplicativo, crie uma cópia do HelloWorld. HelloWorldWorkflow pacote no diretório do seu projeto e chame-o de HelloWorld. HelloWorldWorkflowAsync.

> **Note**
>
> Este tópico aprofunda os conceitos e arquivos apresentados nós tópicos HelloWorld Aplicação e HelloWorldWorkflow Aplicação. Familiarize-se com os arquivos e os conceitos apresentados nesses tópicos antes de prosseguir.

As seções a seguir descrevem como modificar o HelloWorldWorkflow código original para usar um método assíncrono.

HelloWorldWorkflowAsync Implantação de atividades

HelloWorldWorkflowAsync implementa sua interface de trabalho de `GreeterActivities` atividades da seguinte forma:

```
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);
}
```

Essa interface é semelhante à usada por HelloWorldWorkflow, com as seguintes exceções:

- Ela omite a atividade `getGreeting`. Essa tarefa agora é realizada por um método assíncrono.
- O número da versão está definido como 2.0. Depois de registrar uma interface de atividades com o Amazon SWF, você não poderá modificá-la, a menos que altere o número da versão.

As demais implementações do método de atividade são idênticas a. HelloWorldWorkflow Apenas exclua `getGreeting` de `GreeterActivitiesImpl`.

HelloWorldWorkflowAsync implementação do fluxo de trabalho

HelloWorldWorkflowAsync define a interface do fluxo de trabalho da seguinte forma:

```
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();
}
```

A interface é idêntica, HelloWorldWorkflow exceto por um novo número de versão. Como com as atividades, se desejar alterar um fluxo de trabalho registrado, é necessário alterar a sua versão.

HelloWorldWorkflowAsync implementa o fluxo de trabalho da seguinte forma:

```
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 substitui a `getGreeting` atividade por um método `getGreeting` assíncrono, mas o `greet` método funciona da mesma forma:

1. Execute a atividade `getName`, que retorna imediatamente um objeto `Promise<String>`, `name`, que representa o nome.
2. Chame o método assíncrono `getGreeting` e envie-lhe o objeto `name`. `getGreeting` retorna imediatamente um objeto `Promise<String>`, `greeting`, que representa a saudação.
3. Execute a atividade `say` e envie-lhe o objeto `greeting`.
4. Quando `getName` concluir, `name` se torna pronto e `getGreeting` usa o seu valor para construir a saudação.
5. Quando `getGreeting` concluir, `greeting` se torna pronto e `say` imprime a string no console.

A diferença é que, em vez de chamar o cliente de atividades para executar uma atividade `getGreeting`, a saudação chama o método assíncrono `getGreeting`. O resultado final é o mesmo, mas o método `getGreeting` funciona de forma um tanto diferente em relação à atividade `getGreeting`.

- O operador do fluxo de trabalho utiliza a semântica de chamada de função padrão para executar o `getGreeting`. No entanto, a execução assíncrona da atividade é mediada pelo Amazon SWF.
- `getGreeting` é executado no processo de implementação do fluxo de trabalho.
- `getGreeting` retorna um objeto `Promise<String>` em vez de um objeto `String`. Para obter o valor da String mantida pelo `Promise`, chame o seu método `get()`. No entanto, como a atividade está sendo executada de forma assíncrona, seu valor de retorno pode não estar pronto imediatamente; `get()` gerará uma exceção até que o valor de retorno do método assíncrono esteja disponível.

  Para obter mais informações sobre como o `Promise` funciona, consulte AWS Flow Framework Conceitos básicos: troca de dados entre atividades e fluxos de trabalho.

O `getGreeting` cria um valor de retorno enviando a string de saudação para o método `Promise.asPromise` estático. Esse método cria um objeto `Promise<T>` do tipo apropriado, define o valor e coloca-o no estado pronto.

HelloWorldWorkflowAsyncAnfitrião e iniciador de fluxo de trabalho e atividades

HelloWorldWorkflowAsync implementa `GreeterWorker` como classe hospedeira para as implementações de fluxo de trabalho e atividades. É idêntico à HelloWorldWorkflow implementação, exceto pelo `taskListToPoll` nome, que está definido como "`HelloWorldAsyncList`".

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

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

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

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

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

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

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

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

HelloWorldWorkflowAsync implementa o iniciador do fluxo de trabalho em`GreeterMain`; é idêntico à HelloWorldWorkflow implementação.

Para executar o fluxo de trabalho, execute `GreeterWorker` e. `GreeterMain` da mesma forma que com HelloWorldWorkflow.

HelloWorldWorkflowDistributed Aplicação

Com HelloWorldWorkflow e HelloWorldWorkflowAsync, o Amazon SWF medeia a interação entre o fluxo de trabalho e as implementações de atividades, mas elas são executadas localmente como um único processo. `GreeterMain`está em um processo separado, mas ainda é executado no mesmo sistema.

Um recurso importante do Amazon SWF é que ele oferece suporte a aplicações distribuídas. Por exemplo, você pode executar o trabalhador do fluxo de trabalho em uma EC2 instância da Amazon, o iniciador do fluxo de trabalho em um computador de data center e as atividades em um computador desktop cliente. Você pode ainda executar atividades diferentes em sistemas diferentes.

O HelloWorldWorkflowDistributed aplicativo se estende HelloWorldWorkflowAsync para distribuir o aplicativo em dois sistemas e três processos.

- O fluxo de trabalho e o acionador do fluxo de trabalho são executados como processos separados em um sistema.
- As atividades são executadas em um sistema separado.

Para implementar o aplicativo, crie uma cópia do HelloWorld. HelloWorldWorkflowAsync pacote no diretório do seu projeto e chame-o de HelloWorld. HelloWorldWorkflowDistributed. As seções a seguir descrevem como modificar o HelloWorldWorkflowAsync código original para distribuir o aplicativo em dois sistemas e três processos.

Não é necessário alterar as implementações do fluxo de trabalho ou das atividades para executá-los em sistemas separados, nem mesmo os números de versão. Também não é necessário modificar o `GreeterMain`. Mude apenas o host das atividades e do fluxo de trabalho.

Com HelloWorldWorkflowAsync, um único aplicativo serve como host do fluxo de trabalho e da atividade. Para executar as implementações do fluxo de trabalho e de atividade em sistemas separados, é necessário implementar aplicativos separados. Exclua GreeterWorker do projeto e adicione dois novos arquivos de classe GreeterWorkflowWorker GreeterActivitiesWorker e.

HelloWorldWorkflowDistributed implementa suas atividades hospedadas em GreeterActivitiesWorker, da seguinte forma:

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

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

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

      AmazonSimpleWorkflow service = new AmazonSimpleWorkflowClient(awsCredentials,
 config);
```

```
      service.setEndpoint("https://swf.us-east-1.amazonaws.com");

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

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

HelloWorldWorkflowDistributed implementa seu host de fluxo de trabalho em`GreeterWorkflowWorker`, da seguinte forma:

```
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();
   }
}
```

Observe que o `GreeterActivitiesWorker` é apenas `GreeterWorker` sem o código `WorkflowWorker` e o `GreeterWorkflowWorker` é apenas `GreeterWorker` sem o código `ActivityWorker`.

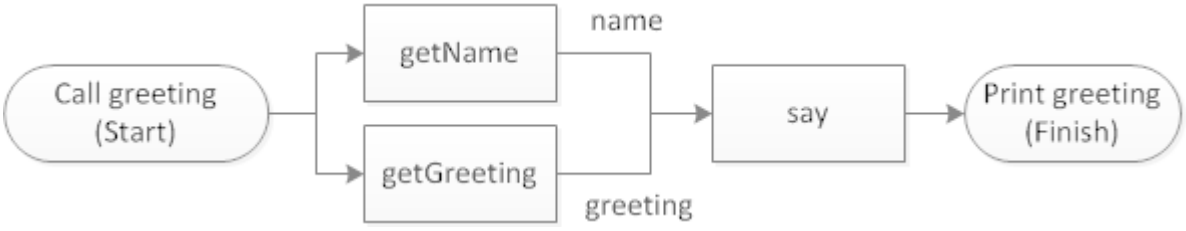
Para executar o fluxo de trabalho:

1. Crie um arquivo executável JAR com `GreeterActivitiesWorker` como ponto de entrada.
2. Copie o arquivo JAR da Etapa 1 para outro sistema, que pode estar executando qualquer sistema operacional que suporta Java.
3. Certifique-se de que AWS as credenciais com acesso ao mesmo domínio Amazon SWF sejam disponibilizadas no outro sistema.
4. Execute o arquivo JAR.
5. No sistema de desenvolvimento, use o Eclipse para executar o `GreeterWorkflowWorker` e o `GreeterMain`.

Além do fato de que as atividades estão sendo executadas em um sistema diferente do trabalhador do fluxo de trabalho e do iniciador do fluxo de trabalho, o fluxo de trabalho funciona exatamente da mesma maneira que HelloWorldAsync. No entanto, pelo fato de a chamada `println` que imprime “Olá, mundo!” para o console estar na atividade `say`, a saída aparecerá no sistema que está executando o operador de atividade.

HelloWorldWorkflowParallelAplicação

As versões anteriores do Hello World! usam uma topologia de fluxo linear. Contudo, o Amazon SWF não se limita a topologias lineares. O HelloWorldWorkflowParallel aplicativo é uma versão modificada HelloWorldWorkflow que usa uma topologia paralela, conforme mostrado na figura a seguir.



Com HelloWorldWorkflowParallel, `getName` e `getGreeting` corra paralelamente e cada um retorne parte da saudação. `say`em seguida, mescla as duas sequências em uma saudação e a imprime no console.

Para implementar o aplicativo, crie uma cópia do HelloWorld. HelloWorldWorkflow pacote no diretório do seu projeto e chame-o de HelloWorld. HelloWorldWorkflowParallel. As seções a seguir descrevem como modificar o HelloWorldWorkflow código original para ser executado `getName` e `getGreeting` em paralelo.

HelloWorldWorkflowParallelTrabalhador de atividades

A interface de HelloWorldWorkflowParallel atividades é implementada em`GreeterActivities`, conforme mostrado no exemplo a seguir.

```
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);
}
```

A interface é semelhante à HelloWorldWorkflow, com as seguintes exceções:

- `getGreeting` não usa nenhuma entrada, simplesmente retorna uma sequência da saudação.
- `say` usa duas sequências de entrada, a saudação e o nome.
- A interface tem um novo número de versão que é necessário sempre que você altera uma interface registrada.

HelloWorldWorkflowParallel implementa as atividades em`GreeterActivitiesImpl`, da seguinte forma:

```
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` e `getGreeting` agora simplesmente retornam metade da string da saudação. `say` concatena as duas partes para produzir a frase completa e a imprime no console.

HelloWorldWorkflowParallelTrabalhador de fluxo

A interface do HelloWorldWorkflowParallel fluxo de trabalho é implementada em`GreeterWorkflow`, da seguinte forma:

```
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();
}
```

A classe é idêntica à HelloWorldWorkflow versão, exceto que o número da versão foi alterado para corresponder ao trabalhador das atividades.

O fluxo de trabalho é implementado em `GreeterWorkflowImpl`, da seguinte maneira:

```
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);
    }
}
```

À primeira vista, essa implementação parece muito semelhante às HelloWorldWorkflow três atividades que os métodos do cliente executam em sequência. No entanto, as atividades não são.

- HelloWorldWorkflow passado `name` para`getGreeting`. Como `name` era um objeto `Promise<T>`, `getGreeting` adiou a execução da atividade até a conclusão de `getName`, portanto, as duas atividades foram executadas em sequência.
- HelloWorldWorkflowParallel não passa nenhuma entrada `getName` ou`getGreeting`. Nenhum dos métodos adia a execução e os métodos de atividade associados são executados imediatamente em paralelo.

A atividade `say` usa `greeting` e `name` como parâmetros de entrada. Como eles são objetos `Promise<T>`, `say` adia a execução até que as duas atividades sejam concluídas e, em seguida, constrói e imprime a saudação.

Observe que HelloWorldWorkflowParallel não usa nenhum código de modelagem especial para definir a topologia do fluxo de trabalho. Ele faz isso implicitamente usando o controle de fluxo Java padrão e aproveitando as propriedades dos `Promise<T>` objetos. AWS Flow Framework para aplicativos Java, é possível implementar até mesmo topologias complexas simplesmente usando `Promise<T>` objetos em conjunto com construções convencionais de fluxo de controle Java.

HelloWorldWorkflowParallel Anfitrião e iniciador de fluxo de trabalho e atividades

HelloWorldWorkflowParallel implementa `GreeterWorker` como classe hospedeira para as implementações de fluxo de trabalho e atividades. É idêntico à HelloWorldWorkflow implementação, exceto pelo `taskListToPoll` nome, que está definido como "HelloWorldParallelList”.

`HelloWorldWorkflowParallel`implementa o iniciador do fluxo de trabalho em `GreeterMain` e é idêntico à HelloWorldWorkflow implementação.

Para executar o fluxo de trabalho, execute `GreeterWorker` e `GreeterMain`, da mesma forma como no `HelloWorldWorkflow`.

Compreensão AWS Flow Framework do Java

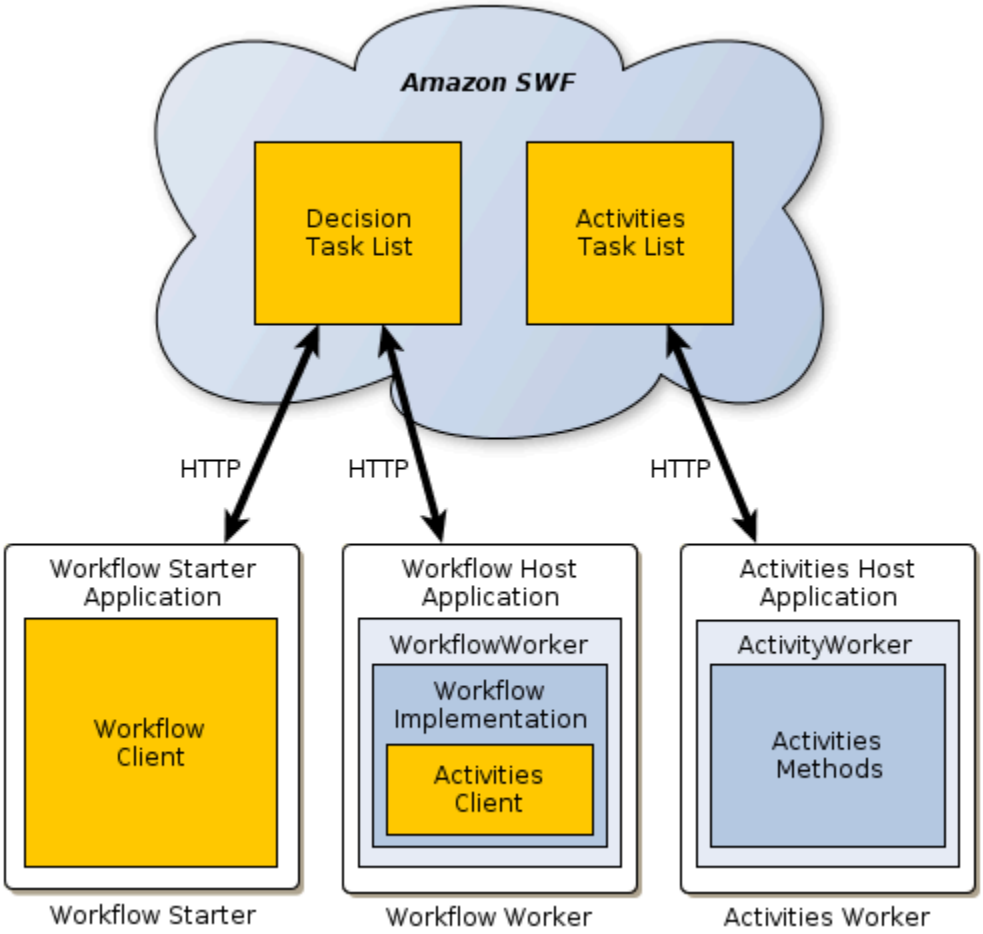
O AWS Flow Framework for Java trabalha com o Amazon SWF para facilitar a criação de aplicativos escaláveis e tolerantes a falhas para realizar tarefas assíncronas que podem ser de longa execução, remotas ou ambas. Os exemplos de “Olá, mundo!” exemplos O que é isso AWS Flow Framework para Java? introduziram os conceitos básicos de como usar o AWS Flow Framework para implementar aplicativos básicos de fluxo de trabalho. Esta seção fornece informações conceituais sobre como os AWS Flow Framework aplicativos funcionam. A primeira seção resume a estrutura básica de um AWS Flow Framework aplicativo, e as seções restantes fornecem mais detalhes sobre como os AWS Flow Framework aplicativos funcionam.

Tópicos

AWS Flow Framework Conceitos básicos: Estrutura do aplicativo

Conceitualmente, um AWS Flow Framework aplicativo consiste em três componentes básicos: iniciadores de fluxo de trabalho, trabalhadores de fluxo de trabalho e trabalhadores de atividades. Uma ou mais aplicações host são responsáveis por registrar os operadores (fluxo de trabalho e atividade) no Amazon SWF, iniciar os operadores e lidar com a limpeza. Os operadores gerenciam a mecânica de execução do fluxo de trabalho e podem ser implementados em diversos hosts.

Este diagrama representa uma AWS Flow Framework aplicação básica:



> **Note**
>
> Implementar esses componentes em três aplicativos separados é conveniente do ponto de vista conceitual, mas você pode criar aplicativos para implementar essa funcionalidade de diversas maneiras. Por exemplo, use um aplicativo de host único para os operadores de atividade e fluxo de trabalho ou use hosts separados para atividade e fluxo de trabalho. Você também pode ter vários operadores de atividade, cada um lidando com um conjunto diferente de atividades em hosts separados e assim por diante.

Os três AWS Flow Framework componentes interagem indiretamente enviando solicitações HTTP para o Amazon SWF, que gerencia as solicitações. O Amazon SWF faz o seguinte:

- Mantém uma ou mais lista de tarefas de decisão, que determinam a próxima etapa a ser realizada por um operador de fluxo de trabalho.

- Mantém uma ou mais listas de tarefas de atividades, que determinam quais tarefas serão realizadas por um operador de atividade.
- Mantém um step-by-step histórico detalhado da execução do fluxo de trabalho.

Com o AWS Flow Framework, o código do seu aplicativo não precisa lidar diretamente com muitos dos detalhes mostrados na figura, como o envio de solicitações HTTP para o Amazon SWF. Você simplesmente chama AWS Flow Framework os métodos e a estrutura trata dos detalhes nos bastidores.

Função do operador de atividade

O operador de atividade executa as diversas tarefas que o fluxo de trabalho precisa realizar. Consistem de:

- A implementação de atividades, que inclui um conjunto de métodos de atividade que realizam tarefas específicas para o fluxo de trabalho.
- Um ActivityWorkerobjeto que usa solicitações de sondagem longa HTTP para sondar o Amazon SWF para que as tarefas de atividade sejam executadas. Quando uma tarefa é necessária, o Amazon SWF responde à solicitação enviando as informações necessárias para executar a tarefa. Em seguida, o ActivityWorkerobjeto chama o método de atividade apropriado e retorna os resultados para o Amazon SWF.

Função do operador de fluxo de trabalho

O operador de fluxo de trabalho orquestra a execução das diversas atividades gerencia o fluxo de dados e lida com as atividades com falha. Consistem de:

- A implementação do fluxo de trabalho, que inclui a lógica de orquestração da atividade, lida com as atividades com falha e assim por diante.
- Um cliente de atividades, que serve como um proxy para o operador de atividade e permite que o operador de fluxo de trabalho agende atividades para execução de forma assíncrona.
- Um WorkflowWorkerobjeto que usa solicitações de sondagem longa HTTP para sondar o Amazon SWF para tarefas de decisão. Se houver tarefas na lista de tarefas do fluxo de trabalho, o Amazon SWF responderá à solicitação retornando as informações necessárias para executar a tarefa. Em seguida, a estrutura executa o fluxo de trabalho para realizar a tarefa e retorna os resultados para o Amazon SWF.

Função do acionador de fluxo de trabalho

O acionador de fluxo de trabalho inicia uma instância de fluxo de trabalho, também chamada de execução de fluxo de trabalho, e pode interagir com uma instância durante a execução para enviar dados adicionais ao operador do fluxo de trabalho ou obter o estado atual do fluxo de trabalho.

O acionador de fluxo de trabalho utiliza um cliente de fluxo de trabalho para iniciar a execução do fluxo de trabalho, interage com o fluxo de trabalho conforme necessário durante a execução e gerencia a limpeza. O iniciador do fluxo de trabalho pode ser um aplicativo executado localmente, um aplicativo da web ou até mesmo o. AWS CLI AWS Management Console

Como o Amazon SWF interage com a aplicação

O Amazon SWF medeia a interação entre os componentes do fluxo de trabalho e mantém um histórico detalhado do fluxo de trabalho. O Amazon SWF não inicia a comunicação com os componentes; ele espera pelas solicitações HTTP dos componentes e gerencia as solicitações conforme necessário. Por exemplo:

- Se a solicitação for de um operador, pesquisando as tarefas disponíveis, o Amazon SWF responderá diretamente ao operador se houver uma tarefa disponível. Para obter mais informações sobre como a sondagem funciona, consulte Sondar tarefas no Guia do desenvolvedor do Amazon Simple Workflow Service.
- Se a solicitação for uma notificação de um operador de atividade de que uma tarefa está concluída, o Amazon SWF registra as informações no histórico de execução e adiciona uma tarefa à lista de tarefas de decisão para informar ao operador do fluxo de trabalho que a tarefa está concluída, permitindo que ele prossiga para a próxima etapa.
- Se a solicitação for do operador do fluxo de trabalho para executar uma atividade, o Amazon SWF registra as informações no histórico de execução e adiciona uma tarefa à lista de tarefas de atividades para direcionar um operador de atividade para executar o método de atividade apropriado.

Essa abordagem permite que os trabalhadores executem em qualquer sistema com conexão à Internet, incluindo EC2 instâncias da Amazon, data centers corporativos, computadores clientes e assim por diante. Eles nem precisam utilizar o mesmo sistema operacional. Como as solicitações HTTP são originadas com os operadores, não há necessidade de portas externamente visíveis. Os operadores podem ser executados atrás de um firewall.

Para obter mais informações

Para uma discussão mais completa sobre como o Amazon SWF funciona, consulte o Guia do desenvolvedor do operadores de atividade.

AWS Flow Framework Conceitos básicos: execução confiável

Os aplicativos distribuídos assíncronos devem lidar com problemas de confiabilidade que não são encontrados por aplicativos convencionais, incluindo:

- Como fornecer comunicação confiável entre componentes distribuídos assíncronos, como componentes de execução prolongada em sistemas remotos.
- Como garantir que os resultados não sejam perdidos se um componente falhar ou for desconectado, especialmente para aplicativos de execução prolongada.
- Como lidar com componentes distribuídos com falha.

Os aplicativos podem contar com o Amazon SWF AWS Flow Framework e o Amazon SWF para gerenciar esses problemas. Exploraremos como o Amazon SWF fornece mecanismos para garantir que seus fluxos de trabalho operem de forma confiável e previsível, mesmo quando são de longa duração e dependem de tarefas assíncronas executadas computacionalmente e com interação humana.

Fornecimento de comunicação confiável

AWS Flow Framework fornece comunicação confiável entre um trabalhador de fluxo de trabalho e seus funcionários de atividades usando o Amazon SWF para enviar tarefas para funcionários de atividades distribuídas e retornar os resultados para o funcionário do fluxo de trabalho. O Amazon SWF usa os seguintes métodos para garantir uma comunicação confiável entre um operador e suas atividades:

- O Amazon SWF armazena de forma duradoura as tarefas agendadas de atividade e fluxo de trabalho e garante que elas serão executadas no máximo uma vez.
- O Amazon SWF garante que uma tarefa de atividade será concluída com êxito e retornará um resultado válido ou notificará o operador do fluxo de trabalho de que a tarefa falhou.
- O Amazon SWF armazena de forma duradoura o resultado de cada atividade concluída ou, no caso de atividades com falha, armazena informações de erro relevantes.

AWS Flow Framework Em seguida, usa os resultados da atividade do Amazon SWF para determinar como prosseguir com a execução do fluxo de trabalho.

Garantia de que os resultados não sejam perdidos

Manutenção do histórico do fluxo de trabalho

Uma atividade que realiza uma operação de mineração de dados em um petabyte de dados pode levar horas para ser concluída, e uma atividade que direciona um operador humano para realizar uma tarefa complexa pode levar dias ou até mesmo semanas para ser concluída!

Para acomodar cenários como esses, AWS Flow Framework fluxos de trabalho e atividades podem levar um tempo arbitrariamente longo para serem concluídos: até o limite de um ano para a execução do fluxo de trabalho. Executar processos prolongados de forma confiável requer um mecanismo para armazenar com durabilidade o histórico de execução do fluxo de trabalho regularmente.

O AWS Flow Framework lida com isso dependendo do Amazon SWF, que mantém um histórico de execução de cada instância do fluxo de trabalho. O histórico do fluxo de trabalho fornece um registro completo e competente sobre o progresso do fluxo de trabalho, incluindo todas as tarefas do fluxo de trabalho e da atividade agendadas e concluídas, e as informações retornadas pelas atividades concluídas ou com falha.

AWS Flow Framework os aplicativos geralmente não precisam interagir diretamente com o histórico do fluxo de trabalho, embora possam acessá-lo se necessário. Para a maioria das finalidades, os aplicativos podem simplesmente deixar as estrutura interagir com o histórico do fluxo de trabalho em segundo plano. Para obter uma discussão completa sobre o histórico do fluxo de trabalho, consulte Histórico do fluxo de trabalho no Guia do desenvolvedor do Amazon Simple Workflow Service.

Execução stateless

O histórico de execução permite que os operadores do fluxo de trabalho sejam stateless. Se tiver várias instâncias de um operador do fluxo de trabalho ou da atividade, qualquer operador pode executar qualquer tarefa. O operador recebe do Amazon SWF todas as informações de estado de que precisa para executar a tarefa.

Essa abordagem torna os fluxos de trabalho mais confiáveis. Por exemplo, se um operador de atividade falha, não é necessário reiniciar o fluxo de trabalho. Apenas reinicie o operador e ele começará a consultar a lista de tarefas e processar as tarefas que estão na lista, independentemente de quando a falha ocorreu. Você pode tornar o fluxo de trabalho geral tolerante a falhas usando dois

ou mais operadores de fluxo de trabalho e de atividade, talvez em sistemas separados. Então, se um dos operadores falhar, os outros continuarão a processar tarefas agendadas sem interrupção no progresso do fluxo de trabalho.

Tratamento de componentes distribuídos com falha

As atividades geralmente falham por motivos efêmeros, como uma breve desconexão, portanto uma estratégia comum para tratar atividades com falha é reiniciar a atividade. Em vez de gerenciar o processo de repetição implementando estratégias de envio de mensagens complexas, os aplicativos podem utilizar o AWS Flow Framework. Ele fornece diversos mecanismos para repetir atividades com falha e oferece um mecanismo de gerenciamento de exceção integrado que funciona com a execução assíncrona distribuída de tarefas em um fluxo de trabalho.

AWS Flow Framework Conceitos básicos: execução distribuída

Uma instância de fluxo de trabalho é essencialmente um thread virtual de execução que pode abranger atividades e lógica de orquestração executadas em vários computadores remotos. O Amazon SWF e a AWS Flow Framework função como um sistema operacional que gerencia instâncias de fluxo de trabalho em uma CPU virtual por meio de:

- Manutenção do estado de execução de cada instância.
- Alternância entre instâncias.
- Retomada da execução de uma instância no ponto em que ela foi alternada.

Reprodução de fluxos de trabalho

Como as atividades podem ser de longa duração, não é desejável ter o fluxo de trabalho simplesmente bloqueado até que seja concluído. Em vez disso, AWS Flow Framework gerencia a execução do fluxo de trabalho usando um mecanismo de repetição, que depende do histórico do fluxo de trabalho mantido pelo Amazon SWF para executar o fluxo de trabalho em episódios.

Cada episódio reproduz a lógica do fluxo de trabalho de uma forma que executa cada atividade apenas uma vez e garante que as atividades e os métodos assíncronos não sejam executados até que seus objetos da Promessa estejam prontos.

O iniciador do fluxo de trabalho inicia o primeiro episódio de reprodução quando inicia a execução do fluxo de trabalho. A estrutura chama o método de ponto de entrada do fluxo de trabalho e:

1. Executa todas as tarefas do fluxo de trabalho que não dependem da conclusão da atividade, inclusive chamando todos os métodos clientes da atividade.
2. Fornece ao Amazon SWF uma lista de tarefas de atividades a serem agendadas para execução. Para o primeiro episódio, essa lista consiste apenas em atividades que não dependem de uma promessa e que podem ser executadas imediatamente.
3. Notifica o Amazon SWF de que o episódio foi concluído.

O Amazon SWF armazena as tarefas de atividade no histórico do fluxo de trabalho e as agenda para execução, colocando-as na lista de tarefas de atividade. Os operadores de atividades pesquisam a lista de tarefas e executam as tarefas.

Quando um operador de atividade conclui uma tarefa, ele retorna o resultado ao Amazon SWF, que o registra no histórico de execução do fluxo de trabalho e agenda uma nova tarefa de fluxo de trabalho para o operador de fluxo de trabalho, colocando-o na lista de tarefas do fluxo de trabalho. O operador do fluxo de trabalho pesquisa a lista de tarefas e, quando recebe a tarefa, executa o próximo episódio de reprodução, da seguinte forma:

1. A estrutura executa o método de ponto de entrada do fluxo de trabalho novamente e:
   - Executa todas as tarefas do fluxo de trabalho que não dependem da conclusão da atividade, inclusive chamando todos os métodos clientes da atividade. Porém, a estrutura verifica o histórico de execução e não programa tarefas de atividades duplicadas.
   - Verifica o histórico para ver quais tarefas de atividades foram concluídas e executa todos os métodos assíncronos de fluxo de trabalho que dependem dessas atividades.
2. Quando todas as tarefas de fluxo de trabalho que podem ser executadas tiverem sido concluídas, a estrutura se reportará ao Amazon SWF:
   - Ele fornece ao Amazon SWF uma lista de todas as atividades cujos objetos de entrada `Promise<T>` ficaram prontos desde o último episódio e podem ser programados para execução.
   - Se o episódio não tiver gerado nenhuma tarefa de atividade adicional, mas ainda houver atividades não concluídas, a estrutura notificará o Amazon SWF de que o episódio está concluído. Em seguida, ela aguarda a conclusão de outra atividade, iniciando o próximo episódio de reprodução.
   - Se o episódio não tiver gerado tarefas de atividade adicionais e todas as atividades tiverem sido concluídas, a estrutura notificará o Amazon SWF de que a execução do fluxo de trabalho foi concluída.

Para obter exemplos do comportamento de reprodução, consulte AWS Flow Framework para o comportamento do Java Replay.

Reprodução e métodos assíncronos de fluxo de trabalho

Os métodos assíncronos de fluxo de trabalho geralmente são usados da mesma forma como atividades, porque o método adia a execução até que todos os objetos `Promise<T>` de entrada estejam prontos. No entanto, o mecanismo de reprodução trata métodos assíncronos de forma diferente de atividades.

- A reprodução não garante que um método assíncrono será executado apenas uma vez. Ele adia a execução em um método assíncrono até que seus objetos Promise estejam prontos, mas executa nesse método para todos os episódios subsequentes.
- Quando é concluído, um método assíncrono não inicia um novo episódio.

Um exemplo de reprodução de um fluxo de trabalho assíncrono é fornecido em AWS Flow Framework para o comportamento do Java Replay.

Reprodução e implementação de fluxo de trabalho

Na maioria das vezes, você não precisa se preocupar com os detalhes do mecanismo de reprodução. Ele é basicamente algo que acontece nos bastidores. Contudo, a reprodução tem duas implicações importantes para a implementação do fluxo de trabalho.

- Não use métodos de fluxo de trabalho para executar tarefas de longa duração, pois a reprodução repetirá essa tarefa várias vezes. Até os métodos assíncronos de fluxo de trabalho normalmente são executados mais de uma vez. Em vez disso, use atividades para tarefas de longa duração. A reprodução executa atividades apenas uma vez.
- Sua lógica de fluxo de trabalho deve ser completamente determinista. Cada episódio deve usar o mesmo caminho do fluxo de controle. Por exemplo, o caminho do fluxo de controle não deve depender da hora atual. Para obter uma descrição detalhada da reprodução e do requisito de determinismo, consulte Não determinismo.

AWS Flow Framework Conceitos básicos: listas de tarefas e execução de tarefas

O Amazon SWF gerencia o fluxo de trabalho e as tarefas de atividade postando-as em listas nomeadas. O Amazon SWF mantém pelo menos duas listas de tarefas, uma para operadores de fluxo de trabalho e outra para operadores de atividade.

> **Note**
>
> Você pode especificar quantas listas de tarefas forem necessárias, com diferentes operadores atribuídos a cada lista. Não há um limite para o número de listas de tarefas. Normalmente, você especifica uma lista de tarefas do operador no aplicativo de host de operador ao criar o objeto do operador.

O trecho do aplicativo de host `HelloWorldWorkflow` a seguir cria um novo operador de atividades e o atribui à lista de tarefas de atividades `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();
    ...
  }
}
```

Por padrão, o Amazon SWF agenda as tarefas do operador na lista `HelloWorldList`. Em seguida, o operador pesquisa tarefas nessa lista. Você pode atribuir qualquer nome a uma lista de tarefas. Você pode até usar o mesmo nome para listas de fluxos de trabalho e de atividades. Internamente, o Amazon SWF coloca os nomes do fluxo de trabalho e da lista de tarefas de atividades em namespaces diferentes, de modo que as duas listas serão distintas.

Se você não especificar uma lista de tarefas, ela AWS Flow Framework especifica uma lista padrão quando o trabalhador registra o tipo no Amazon SWF. Para obter mais informações, consulte Registro do tipo de fluxo de trabalho e atividade.

Às vezes é útil fazer com que um operador ou um grupo de operadores específico execute determinadas tarefas. Por exemplo, um fluxo de trabalho de processamento de imagem pode usar uma atividade para fazer download de uma imagem e outra atividade para processar a imagem. É mais eficiente executar as duas tarefas no mesmo sistema e evitar a sobrecarga de transferir grandes arquivos pela rede.

Para oferecer suporte a esses cenários, você pode especificar explicitamente uma lista de tarefas ao chamar um método de cliente de atividades usando uma sobrecarga que inclua um parâmetro `schedulingOptions`. Você especifica a lista de tarefas passando ao método um `ActivitySchedulingOptions` objeto configurado adequadamente.

Por exemplo, suponha que a atividade `say` do aplicativo `HelloWorldWorkflow` seja hospedada por um operador de atividades diferente de `getName` e de `getGreeting`. O exemplo a seguir mostra como garantir que `say` use a mesma lista de tarefas que `getName` e `getGreeting`, mesmo que tenham sido atribuídos originalmente a listas diferentes.

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

O método assíncrono `runSay` obtém a lista de tarefas de `getGreeting` de seu objeto de cliente. Em seguida, ele cria e configura um objeto `ActivitySchedulingOptions` que garante que `say` pesquise a mesma lista de tarefas que `getGreeting`.

> **Note**
>
> Quando você passa um parâmetro `schedulingOptions` para um método de cliente de atividades, ele substitui a lista de tarefas original apenas para essa execução de atividade. Se você chamar o método do cliente activities novamente sem especificar uma lista de tarefas, o Amazon SWF atribuirá a tarefa à lista original e o operador de atividade pesquisará essa lista.

AWS Flow Framework Conceitos básicos: aplicativos escaláveis

O Amazon SWF tem dois recursos principais que permitem escalar facilmente uma aplicação de fluxo de trabalho para lidar com a carga atual:

- Um histórico completo de execução do fluxo de trabalho, que permite implementar um aplicativo stateless.
- Agendamento de tarefas que livremente acoplado à execução da tarefa, o que facilita escalar o aplicativo para atender às demandas atuais.

O Amazon SWF agenda as tarefas postando-as em listas de tarefas alocadas dinamicamente, e não se comunicando diretamente com os operadores do fluxo de trabalho e das atividades. Em vez disso, os operadores utilizam solicitações HTTP para consultar suas respectivas listas em busca das tarefas. Essa abordagem associa vagamente o agendamento de tarefas à execução de tarefas e permite que os funcionários executem qualquer sistema adequado, incluindo EC2 instâncias da Amazon, data centers corporativos, computadores clientes e assim por diante. Como as solicitações HTTP são originadas pelos trabalhadores, não há necessidade de portas visíveis externamente, o que permite que os trabalhadores funcionem até mesmo atrás de um firewall.

O mecanismo de consulta prolongada utilizado pelos operadores para consultar tarefas garante que eles não se sobrecarreguem. Mesmo que ocorra um pico de tarefas agendadas, os operadores puxam as tarefas em seus próprios ritmos. No entanto, como os operadores são stateless, é possível escalar dinamicamente um aplicativo para atender à carga aumentada iniciando instâncias de operador adicionais. Mesmo que sejam executadas em sistemas diferentes, cada instância

consulta a mesma lista de tarefas e a primeira instância de operador disponível executa cada tarefa, independentemente de onde o operador está localizado ou de quando ele foi iniciado. Quando a carga diminuir, você pode reduzir o número de operadores em conformidade.

AWS Flow Framework Conceitos básicos: troca de dados entre atividades e fluxos de trabalho

Ao chamar um método de cliente de atividade assíncrono, ele retorna imediatamente um objeto Promessa (também conhecido como um Future), que representa o valor de retorno do método de atividade. Inicialmente, a Promessa está em um estado não pronto e o valor de retorno está indefinido. Após a conclusão da tarefa do método de atividade e seu retorno, a estrutura ordena o valor de retorno em toda a rede para o operador do fluxo de trabalho, o que atribuí um valor para a Promessa e coloca o objeto em um estado pronto.

Mesmo que um método de atividade não possua valor de retorno, ainda é possível usar a Promessa para executar a execução do fluxo de trabalho. Se você repassar uma promessa retornada para um método de cliente de atividades ou para um método de fluxo de trabalho assíncrono, ela adia a execução até que o objeto esteja pronto.

Se você enviar uma ou mais Promessas para um método de cliente de atividade, a estrutura enfileira a tarefa mas adia o agendamento até que os objetos estejam prontos. Em seguida ela extrai os dados de cada Promessa e ordena-os pela internet para o operador de atividade, que envia-os para o método de atividade como um tipo padrão.

> **Note**
>
> Se você precisar transferir grandes quantidades de dados entre os operadores de fluxo de trabalho e de atividade, a abordagem preferencial é armazenar os dados em um local conveniente e enviar somente as informações de recuperação. Por exemplo, você pode armazenar os dados em um bucket do Amazon S3 e passar o URL associado.

A Promessa <T> Type

O tipo `Promise<T>` é de certa forma semelhante ao tipo `Future<T>` em Java. Ambos os tipos representam os valores retornados por métodos assíncronos e estão inicialmente indefinidos. Acesse o valor de um objeto ao chamar o seu método `get`. Além disso, os dois tipos comportam-se de forma diferente.

- `Future<T>` é uma construção de sincronização que permite que um aplicativo espere a conclusão de um método assíncrono. Se você chamar `get` e o objeto não estiver pronto, ele é bloqueia até que o objeto esteja pronto.
- Com `Promise<T>`, a sincronização é gerenciada pela estrutura. Se você chamar `get` e o objeto não estiver pronto, `get` gerará uma exceção.

A finalidade principal da `Promise<T>` é gerenciar o fluxo de dados entre as atividades. Isso garante que uma atividade não seja executada até que os dados de entrada sejam válidos. Em muitos casos, os operadores de fluxo de trabalho não precisam acessar os objetos `Promise<T>` diretamente, eles apenas enviam os objetos de uma atividade para outra e deixam que a estrutura e os operadores de atividade cuidem dos detalhes. Para acessar o valor de um objeto `Promise<T>` em um operador de fluxo de trabalho, você deve ter certeza que o objeto está pronto antes de chamar seu método `get`.

- A abordagem preferencial é enviar o objeto `Promise<T>` para um método de fluxo de trabalho assíncrono e processar o valor ali. Um método assíncrono adia a execução até que todos os objetos `Promise<T>` de entrada estejam prontos, o que garante que os valores possam ser acessados com segurança.
- `Promise<T>` expõe um método `isReady` que retorna `true` se o objeto estiver pronto. Usar `isReady` para consultar um objeto `Promise<T>` não é recomendado, mas `isReady` é útil em certas ocasiões.

O AWS Flow Framework for Java também inclui um `Settable<T>` tipo, que é derivado `Promise<T>` e tem um comportamento semelhante. A diferença é que a estrutura geralmente define o valor de um `Promise<T>` objeto e o trabalhador do fluxo de trabalho é responsável por definir o valor de `Settable<T>` a.

Existem ocasiões onde um operador de fluxo de trabalho precisa criar um objeto `Promise<T>` e definir o seu valor. Por exemplo, um método assíncrono que retorna um objeto `Promise<T>` precisa criar um valor de retorno.

- Para criar um objeto que representa um valor digitado, chame o método `Promise.asPromise` estático, que cria um objeto `Promise<T>` do tipo adequado, define o seu valor e coloca-o em estado pronto.
- Para criar um objeto `Promise<Void>`, chame o método `Promise.Void` estático.

> **Note**
>
> `Promise<T>` pode representar qualquer tipo válido. No entanto, se os dados devem ser ordenados na internet, o tipo deve ser compatível com o conversor de dados. Consulte a próxima seção para obter detalhes.

Conversores de dados e ordenação

Eles AWS Flow Framework organizam dados pela Internet usando um conversor de dados. Por padrão, a estrutura usa um conversor de dados que é baseado no processador de Jackson JSON. Contudo, esse conversor tem algumas limitações. Por exemplo, ele não pode ordenar mapas que não utilizam strings como chaves. Se o conversor padrão não for suficiente para o seu aplicativo, você pode implementar um conversor de dados personalizado. Para obter mais detalhes, consulte DataConverters.

AWS Flow Framework Conceitos básicos: troca de dados entre aplicativos e execuções de fluxo de trabalho

Um método de ponto de entrada de fluxo de trabalho pode ter um ou mais parâmetros, o que permite que o iniciador do fluxo de trabalho passe os dados iniciais para o fluxo de trabalho. Ele também pode ser útil para fornecer dados adicionais ao fluxo de trabalho durante a execução. Por exemplo, se um cliente alterar seu endereço para entrega, você poderá notificar o fluxo de trabalho do processamento do pedido para que ele possa fazer as alterações adequadas.

O Amazon SWF permite que os fluxos de trabalho implementem um método de sinal, que permite que aplicações, como o iniciador do fluxo de trabalho, passem dados para o fluxo de trabalho a qualquer momento. Um método de sinal pode ter qualquer nome e parâmetros convenientes. Você o atribui como um método de sinal incluindo-o na definição da interface do fluxo de trabalho e aplicando uma anotação `@Signal` na declaração do método.

O exemplo a seguir mostra a interface de um fluxo de trabalho de processamento de pedido que declara um método de sinal, `changeOrder`, que permite que o iniciador do fluxo de trabalho altere o pedido original depois que o fluxo de trabalho foi iniciado.

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

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

O processador de anotação da estrutura cria um método cliente do fluxo de trabalho com o mesmo nome do método de sinal, e o iniciador do fluxo de trabalho chama o método cliente para passar os dados para o fluxo de trabalho. Para ver um exemplo, consulte Fórmulas do AWS Flow Framework

Tipos de tempo limite do Amazon SWF

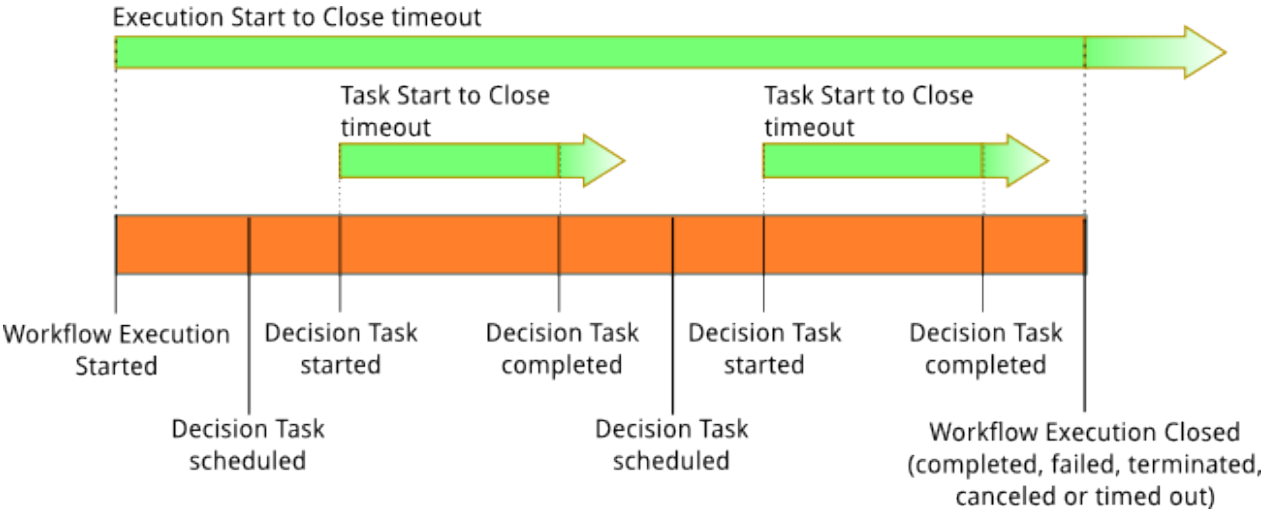
Para garantir que as execuções do fluxo de trabalho sejam executadas corretamente, você pode definir diferentes tipos de tempos limite com o Amazon SWF. Alguns tempos limite especificam por quanto tempo o fluxo de trabalho pode ser executado na sua totalidade. Outros tempos limites especificam quanto tempo atividades de tarefa podem demorar antes de serem atribuídas a um operador e quanto tempo elas podem levar para serem concluídas a partir do momento em que são agendadas. Todos os tempos limite na API do Amazon SWF são especificados em segundos. O Amazon SWF também aceita a string `NONE` como um valor de tempo limite, o que indica que não há tempo limite.

Para tempos limite relacionados a tarefas de decisão e tarefas de atividade, o Amazon SWF adiciona um evento ao histórico de execução do fluxo de trabalho. Os atributos do evento fornecem informações sobre o tipo de tempo limite ocorrido e qual tarefa de decisão ou tarefa de atividade foi afetada. O Amazon SWF também agenda uma tarefa de decisão. Quando o tomador de decisão receber a nova tarefa de decisão, ele verá o evento de tempo limite no histórico e tomará uma ação apropriada chamando a RespondDecisionTaskCompletedação.

Uma tarefa é considerada aberta a partir do momento em que ela é agendada até ser encerrada. Portanto, uma tarefa é informada como aberta enquanto um operador a processa. Uma tarefa é encerrada quando um operador a informa como concluída, cancelada ou falha. Uma tarefa também pode ser fechada pelo Amazon SWF como resultado de um tempo limite.

Tempos limites em tarefas de fluxo de trabalho e decisão

O diagrama a seguir mostra como tempos limites de fluxo de trabalho e decisão estão relacionado ao ciclo de vida de um fluxo de trabalho:



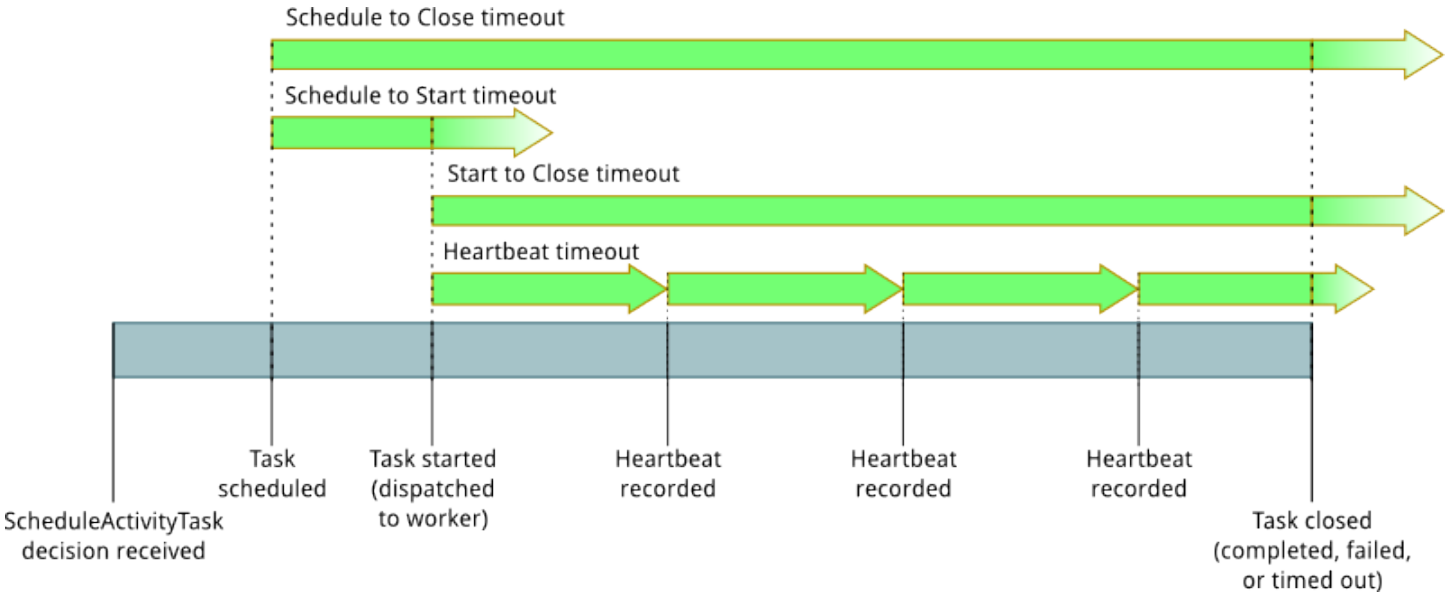
Existem dois tipos de tempo limite que são relevantes para tarefas de decisão e fluxo de trabalho:

- Início do fluxo de trabalho até o encerramento (**`timeoutType: START_TO_CLOSE`**): esse tempo limite especifica o tempo máximo que uma execução de fluxo de trabalho pode levar para ser concluída. Ele é definido como um padrão durante o registro do fluxo de trabalho, mas pode ser substituído por um valor diferente quando o fluxo de trabalho é iniciado. Se esse tempo limite for excedido, o Amazon SWF fecha a execução do fluxo de trabalho e adiciona um evento do WorkflowExecutionTimedOuttipo ao histórico de execução do fluxo de trabalho. Além do `timeoutType`, os atributos de evento especificam o `childPolicy` que está em vigor para essa execução de fluxo de trabalho. A política de elementos secundários especifica como as execuções de fluxo de trabalho secundárias serão tratadas se a execução de fluxo de trabalho principal atingir o tempo limite ou for encerrada de outro modo. Por exemplo `childPolicy`, se ela estiver definida como TERMINATE, as execuções de fluxo de trabalho secundárias serão finalizadas. Quando uma execução de fluxo de trabalho tiver atingido o tempo limite, não será possível realizar nenhuma ação nela além de chamadas de visibilidade.
- Início da tarefa de decisão até o encerramento (**`timeoutType: START_TO_CLOSE`**): esse tempo limite especifica o tempo máximo que o agente de decisão correspondente pode levar para concluir uma tarefa de decisão. Ele é definido durante o registro do tipo de fluxo de trabalho. Se esse tempo limite for excedido, a tarefa será marcada como expirada no histórico de execução do fluxo de trabalho e o Amazon SWF adicionará um evento do tipo DecisionTaskTimedOutao histórico do fluxo de trabalho. Os atributos do evento incluirão IDs os eventos que correspondem a quando essa tarefa de decisão foi agendada (`scheduledEventId`) e quando foi iniciada (`startedEventId`). Além de adicionar o evento, o Amazon SWF também agenda uma nova tarefa de decisão para alertar o agente de decisão de que essa tarefa de decisão atingiu o tempo

limite. Após esse tempo limite, uma tentativa de concluir a tarefa de decisão expirada usando `RespondDecisionTaskCompleted` falhará.

Tempos limites em tarefas de atividade

O diagrama a seguir mostra como tempos limite estão relacionados ao ciclo de vida de uma tarefa de atividade:



Existem quatro tipos de tempo limite que são relevantes para tarefas de atividade:

- Início da tarefa de atividade até o encerramento (**`timeoutType: START_TO_CLOSE`**): esse tempo limite especifica o tempo máximo que um operador de atividade pode levar para processar uma tarefa depois que ele recebeu a tarefa. As tentativas de fechar uma tarefa de atividade com tempo limite expirado usando RespondActivityTaskCanceled, RespondActivityTaskCompleted, RespondActivityTaskFailedfalharão.
- Heartbeat de tarefa de atividade (**`timeoutType: HEARTBEAT`**): esse tempo limite especifica o tempo máximo que uma tarefa pode ser executada antes de fornecer seu andamento por meio da ação `RecordActivityTaskHeartbeat`.
- Programação de início da tarefa de atividade(**`timeoutType: SCHEDULE_TO_START`**): esse tempo limite especifica quanto tempo o Amazon SWF aguarda antes de encerrar a tarefa de atividade se não houver operadores disponíveis para executar a tarefa. Após o tempo limite, a tarefa expirada não será atribuída a outro operador.
- Programação de encerramento da tarefa de atividade (**`timeoutType: SCHEDULE_TO_CLOSE`**): esse tempo limite especifica quanto tempo a tarefa pode levar desde o momento em que é

agendada até o momento em que é concluída. Como prática recomendada, esse valor não deve ser maior que a soma do tempo limite da tarefa e do schedule-to-start tempo limite da tarefa start-to-close.

> **Note**
>
> Cada um dos tipos de tempo limite tem um valor padrão, que geralmente está definido como `NONE` (infinito). No entanto, o tempo máximo para qualquer execução de atividade é limitado a um ano.

Você define valores padrão para eles durante o registro do tipo de atividade, mas pode substituí-los por novos valores ao agendar a tarefa de atividade. Quando um desses tempos limite ocorrer, o Amazon SWF adicionará um evento do ActivityTaskTimedOuttipo ao histórico do fluxo de trabalho. O atributo de valor `timeoutType` desse evento especificará quais desses tempos limites ocorreu. Para cada um dos tempos limites, o valor de `timeoutType` é mostrado entre parênteses. Os atributos do evento também incluirão IDs os eventos que correspondem a quando a tarefa da atividade foi agendada (`scheduledEventId`) e quando foi iniciada (`startedEventId`). Além de adicionar o evento, o Amazon SWF também agenda uma nova tarefa de decisão para alertar o agente de decisão de que o tempo limite ocorreu.

Entendendo uma tarefa em AWS Flow Framework Java

Tópicos

Tarefa

A primitiva subjacente que o AWS Flow Framework for Java usa para gerenciar a execução do código assíncrono é a classe. `Task` Um objeto do tipo `Task` representa trabalho que precisa ser executado de forma assíncrona. Quando você chama um método assíncrono, a estrutura cria uma `Task` para executar o código nesse método e coloca-a em uma lista para execução em um momento posterior. De modo semelhante, quando você invoca uma `Activity`, uma `Task` é criada para ela. A chamada do método retorna depois disso, geralmente retornando uma `Promise<T>` como o resultado futuro da chamada.

A classe `Task` é pública e pode ser usada diretamente. Por exemplo, podemos reescrever o exemplo de Hello World para usar uma `Task` em vez de um método assíncrono.

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

A estrutura chama o método `doExecute()` quando todas as `Promises` passadas para o construtor da `Task` ficam prontas. Para obter mais detalhes sobre a `Task` classe, consulte a AWS SDK para Java documentação.

A estrutura também inclui uma classe chamada `Functor` que representa uma `Task` que também é uma `Promise<T>`. O objeto `Functor` fica pronto quando a `Task` é concluída. No exemplo a seguir, um `Functor` é criado para obter a mensagem de saudação:

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

Ordem de execução

As tarefas se tornam qualificadas para execução somente quando todos os parâmetros tipados de `Promise<T>`, passados para a atividade ou o método assíncrono correspondente estiverem prontos. Uma `Task` pronta para execução é movida logicamente para uma fila pronta. Ou seja, ela é programada para execução. A classe de operador executa a tarefa invocando o código que você escreveu no corpo do método assíncrono ou agendando uma tarefa de atividade no Amazon Simple Workflow Service (AWS) no caso de um método de atividade.

Conforme as tarefas executam e produzem resultados, elas fazem que outras tarefas fiquem prontas e a execução do programa continua. A forma como a estrutura executa tarefas é importante para compreender a ordem em que o código assíncrono é executado. O código que aparece sequencialmente em seu programa pode não ser realmente executado naquela ordem.

```
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!");
}
```

O código na lista acima imprimirá o seguinte:

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

Isso pode não ser o que você esperava, mas pode ser facilmente explicado avaliando a maneira como as tarefas dos métodos assíncronos foram executadas:

1. A chamada para `getUserName` cria uma `Task`. Vamos chamá-la de `Task1`. Como `getUserName` não aceita nenhum parâmetro, `Task1` é imediatamente colocado na fila pronta.
2. Em seguida, a chamada para `printHelloName` cria uma `Task` que precisa esperar o resultado de `getUserName`. Vamos chamá-la de `Task2`. Como o valor exigido ainda não está pronto, `Task2` é colocado na lista de espera.
3. Em seguida, uma tarefa para `printHelloWorld` é criada e adicionada à fila pronta. Vamos chamá-la de `Task3`.
4. A instrução `println` imprime "Hello, Amazon!". para o console.
5. Nesse ponto, `Task1` e `Task3` estão na fila pronta e `Task2` está na lista de espera.
6. O operador executa `Task1` e seu resultado faz com que `Task2` fique pronto. O `Task2` é adicionado à fila pronta atrás de `Task3`.
7. A `Task3` e a `Task2` são então executadas nessa ordem.

A execução de atividades segue o mesmo padrão. Quando você chama um método no cliente de atividade, ele cria um `Task` que, após a execução, agenda uma atividade no Amazon SWF.

A estrutura depende de recursos, como a geração de código e os proxies dinâmicos, para injetar a lógica para converter chamadas de métodos em invocações de atividades e tarefas assíncronas em seu programa.

Execução de fluxo de trabalho

A execução da implementação do fluxo de trabalho também é gerenciada pela classe do operador. Quando você chama um método no cliente de fluxo de trabalho, ele chama o Amazon SWF para criar uma instância de fluxo de trabalho. As tarefas no Amazon SWF não devem ser confundidas com as tarefas na estrutura. Uma tarefa no Amazon SWF é uma tarefa de atividade ou uma tarefa de decisão. A execução de tarefas de atividades é simples. A classe de operador de atividade recebe tarefas de atividade do Amazon SWF, invoca o método de atividade apropriado em sua implementação e retorna o resultado ao Amazon SWF.

A execução de tarefas de decisão é mais envolvida. O operador do fluxo de trabalho recebe tarefas de decisão do Amazon SWF. Uma tarefa de decisão é efetivamente uma solicitação que pergunta à lógica do fluxo de trabalho o que fazer em seguida. A primeira tarefa de decisão é gerada para uma instância de fluxo de trabalho quando é iniciada no cliente de fluxo de trabalho. Ao receber essa tarefa de decisão, a estrutura começa a executar o código no método de fluxo de trabalho anotado com `@Execute`. Esse método executa a lógica de coordenação que programa as atividades. Quando o estado da instância do fluxo de trabalho muda, por exemplo, quando uma atividade é concluída, outras tarefas de decisão são agendadas. Nesse ponto, a lógica do fluxo de trabalho pode decidir realizar uma ação com base no resultado da atividade; por exemplo, pode decidir programar outra atividade.

A estrutura oculta todos esses detalhes do desenvolvedor convertendo perfeitamente as tarefas de decisão na lógica do fluxo de trabalho. Do ponto de vista do desenvolvedor, o código parece ser exatamente um programa normal. Por baixo dos panos, a estrutura mapeia as chamadas para o Amazon SWF e as tarefas de decisão usando o histórico mantido pelo Amazon SWF. Quando uma tarefa chega, a estrutura reproduz a execução do programa conectando os resultados das atividades concluídas até o momento. Os métodos e as atividades assíncronos que estavam esperando esses resultados são desbloqueados, e a execução do programa continua.

A execução do fluxo de trabalho de processamento de imagem de exemplo e o histórico correspondente são mostrados na tabela a seguir.

Execução de fluxo de trabalho de miniaturas

Execução do programa do fluxo de trabalho	Histórico mantido pelo Amazon SWF
Execução inicial	

Execução do programa do fluxo de trabalho	Histórico mantido pelo Amazon SWF
1. Loop de expedição 2. getImageUrls 3. downloadImage 4. createThumbnail (tarefa na fila de espera) 5. uploadImage (tarefa na fila de espera) 6. <próxima iteração do loop>	1. Instância de fluxo de trabalho iniciada, id= "1" 2. downloadImage programado
Reproduzir	
1. Loop de expedição 2. getImageUrls 3. downloadImage image path="foo" 4. createThumbnail 5. uploadImage (tarefa na fila de espera) 6. <próxima iteração do loop>	1. Instância de fluxo de trabalho iniciada, id= "1" 2. downloadImage programado 3. downloadImage concluído, retorno = "foo” 4. createThumbnail programado
Reproduzir	
1. Loop de expedição 2. getImageUrls 3. downloadImage image path="foo" 4. createThumbnail thumbnail path="bar" 5. uploadImage 6. <próxima iteração do loop>	1. Instância de fluxo de trabalho iniciada, id= "1" 2. downloadImage programado 3. downloadImage concluído, retorno = "foo” 4. createThumbnail programado 5. createThumbnail concluído, retorno = "bar" 6. uploadImage programado
Reproduzir	

Execução do programa do fluxo de trabalho	Histórico mantido pelo Amazon SWF
1. Loop de expedição 2. getImageUrls 3. downloadImage image path="foo" 4. createThumbnail thumbnail path="bar" 5. uploadImage 6. <próxima iteração do loop>	1. Instância de fluxo de trabalho iniciada, id= "1" 2. downloadImage programado 3. downloadImage concluído, retorno = "foo” 4. createThumbnail programado 5. createThumbnail concluído, retorno = "bar" 6. uploadImage programado 7. uploadImage concluído ...

Quando uma chamada para `processImage` é feita, a estrutura cria uma nova instância de fluxo de trabalho no Amazon SWF. Este é um registro durável da instância de fluxo de trabalho que está sendo iniciada. O programa é executado até a chamada para a atividade `downloadImage`, que pede ao Amazon SWF para agendar uma atividade. O fluxo de trabalho continua a ser executado e cria tarefas para as atividades subsequentes, mas elas não podem ser executadas até que a atividade `downloadImage` seja concluída; portanto, esse episódio de repetição termina. O Amazon SWF despacha a tarefa para a atividade `downloadImage` para execução e, uma vez concluída, um registro é feito no histórico junto com o resultado. O fluxo de trabalho agora está pronto para avançar e uma tarefa de decisão é gerada pelo Amazon SWF. A estrutura recebe a tarefa de decisão e reproduz a conexão do fluxo de trabalho no resultado da imagem obtida por download conforme registrado no histórico. Isso desbloqueia a tarefa para `createThumbnail`, e a execução do programa continua mais adiante, agendando a tarefa de atividade `createThumbnail` na Amazon. O mesmo processo se repete para `uploadImage`. A execução do programa continua dessa maneira até que o fluxo de trabalho tenha processado todas as imagens e não houver tarefas pendentes. Como nenhum estado de execução é armazenado localmente, cada tarefa de decisão pode ser potencialmente executada em uma máquina diferente. Isso permite que você escreva facilmente programas que são tolerantes a falhas e facilmente escaláveis.

Não determinismo

Como a estrutura depende da repetição, é importante que o código de orquestração (todo o código do fluxo de trabalho, com exceção das implementações de atividades) seja determinístico. Por

exemplo, o fluxo de controle em seu programa não deve depender de um número aleatório ou da hora atual. Como essas coisas mudarão entre as invocações, a repetição pode não seguir o mesmo caminho na lógica de orquestração. Isso levará a resultados ou a erros inesperados. A estrutura fornece um `WorkflowClock` que você pode usar para obter a hora atual de forma determinista. Consulte a seção em Contexto de execução para obter mais detalhes.

> **Note**
>
> A conexão incorreta de objetos da implementação de fluxo de trabalho do Spring também pode levar ao não determinismo. Os beans de implementação de fluxo de trabalho bem como os beans dos quais dependem devem estar no escopo do fluxo de trabalho (`WorkflowScope`). Por exemplo, a conexão de um bean de implementação de fluxo de trabalho a um bean que mantém o estado e esteja no contexto global resultará em comportamento inesperado. Consulte a seção Integração com o Spring para obter mais detalhes.

AWS Flow Framework Guia de programação para Java

Esta seção fornece detalhes sobre como usar os recursos do AWS Flow Framework for Java para implementar aplicativos de fluxo de trabalho.

Tópicos

Implementando aplicativos de fluxo de trabalho com o AWS Flow Framework

As etapas típicas envolvidas no desenvolvimento de um fluxo de trabalho com o AWS Flow Framework são:

1. Definir contratos de atividade e de fluxo de trabalho. Analisar os requisitos do aplicativo, determinar as atividades necessárias e a topologia do fluxo de trabalho. As atividades tratam as tarefas de processamento necessárias, enquanto a topologia do fluxo de trabalho define a estrutura básica e a lógica de negócios do fluxo de trabalho.

   Por exemplo, um aplicativo de processamento de mídia pode precisar fazer download de um arquivo, processá-lo e, em seguida, fazer upload do arquivo processado em um bucket do Amazon Simple Storage Service. (S3). Isso pode ser dividido em quatro tarefas de atividades:

   1. fazer download do arquivo de um servidor
   2. processar o arquivo (por exemplo, transcodificando-o para um formato diferente de mídia)
   3. fazer upload do arquivo no bucket do S3
   4. executar a limpeza excluindo os arquivos locais

   Este fluxo de trabalho tem um método de ponto de entrada e implementa uma topologia linear simples que executa as atividades em sequência, de forma muito semelhante ao HelloWorldWorkflow Aplicação.
2. Implementar interfaces de atividade e de fluxo de trabalho. Os contratos de fluxo de trabalho e de atividade são definidos por interfaces Java, tornando suas convenções de chamada previsíveis pelo SWF e fornecendo flexibilidade para implementar sua lógica de fluxo de trabalho e tarefas de atividades. As várias partes de seu programa podem atuar como consumidores de dados umas das outras, porém não precisam estar cientes de muitos dos detalhes da implementação de qualquer uma das outras partes.

   Por exemplo, você pode definir uma interface `FileProcessingWorkflow` e fornecer implementações diferentes de fluxo de trabalho para codificação, compactação, miniaturas de vídeo etc. Cada um desses fluxos de trabalho pode ter fluxos de controle diferentes e pode chamar métodos diferentes de atividades. Seu iniciador de fluxo de trabalho não precisa saber. Com o uso de interfaces, também é simples testar os fluxos de trabalho usando implementações fictícias que, posteriormente, podem ser substituídas por código de trabalho.
3. Gerar clientes de atividade e de fluxo de trabalho. AWS Flow Framework Isso elimina a necessidade de implementar os detalhes de gerenciamento da execução assíncrona, envio de solicitações HTTP, organização de dados e assim por diante. Em vez disso, o iniciador de fluxo de trabalho executa uma instância de fluxo de trabalho chamando um método no cliente do fluxo de trabalho, e a implementação do fluxo de trabalho executa atividades chamando métodos no cliente de atividades. A estrutura trata os detalhes dessas interações em segundo plano.

Se você estiver usando o Eclipse e tiver configurado seu projeto, como emConfigurando o AWS Flow Framework para Java, o processador de AWS Flow Framework anotações usa as definições da interface para gerar automaticamente clientes de fluxo de trabalho e atividades que expõem o mesmo conjunto de métodos da interface correspondente.

4. Implementar aplicativos de host de atividade e de fluxo de trabalho. Suas implementações de fluxo de trabalho e atividades devem ser incorporadas em aplicativos hospedados que pesquisam o Amazon SWF para tarefas, organizam quaisquer dados e chamam os métodos de implementação apropriados. AWS Flow Framework para Java: WorkflowWorkerinclusões e ActivityWorkerclasses que tornam a implementação de aplicativos hospedeiros simples e fácil de fazer.
5. Teste seu fluxo de trabalho. AWS Flow Framework para Java fornece JUnit integração que você pode usar para testar seus fluxos de trabalho em linha e localmente.
6. Implantar os operadores. Você pode implantar seus trabalhadores conforme apropriado — por exemplo, você pode implantá-los em EC2 instâncias da Amazon ou em computadores em seu data center. Uma vez implantados e iniciados, os operadores começam a sondar o Amazon SWF em busca de tarefas e as tratam conforme necessário.
7. Iniciar execuções. Um aplicativo inicia uma instância de fluxo de trabalho usando o cliente de fluxo de trabalho para chamar o ponto de entrada do fluxo de trabalho. Você também pode iniciar fluxos de trabalho usando o console do Amazon SWF. Independentemente de como você inicia uma instância de fluxo de trabalho, é possível usar o console do Amazon SWF para monitorar a instância de fluxo de trabalho em execução e examinar o histórico do fluxo de trabalho para instâncias em execução, concluídas e com falha.

AWS SDK para JavaInclui um conjunto de AWS Flow Framework quatro exemplos de Java que você pode procurar e executar seguindo as instruções no arquivo readme.html no diretório raiz. Há também um conjunto de fórmulas (aplicações simples) que mostram como lidar com uma variedade de problemas específicos de programação, que estão disponíveis em Fórmulas do AWS Flow Framework.

Contratos de atividades e de fluxo de trabalho

As interfaces Java são usadas para declarar as assinaturas de fluxos de trabalho e de atividades. A interface forma o contrato entre a implementação do fluxo de trabalho (ou da atividade) e o cliente do fluxo de trabalho (ou da atividade). Por exemplo, um tipo de fluxo de trabalho `MyWorkflow` é definido usando uma interface anotada com a anotação `@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();
}
```

O contrato não tem nenhuma configuração específica à implementação. O uso de contratos com neutralidade de implementação permite que os clientes sejam separados da implementação e, portanto, fornece a flexibilidade de alterar os detalhes da implementação sem interromper o cliente. Por outro lado, você também pode alterar o cliente sem precisar alterar o fluxo de trabalho ou a atividade que está sendo consumida. Por exemplo, o cliente pode ser modificado para chamar uma atividade de forma assíncrona usando promessas (`Promise<T>`) sem exigir uma alteração na implementação da atividade. Da mesma forma, a implementação da atividade pode ser alterada para que seja concluída de forma assíncrona, por exemplo, por uma pessoa que envia um e-mail, sem exigir que os clientes da atividade sejam alterados.

No exemplo acima, a interface do fluxo de trabalho `MyWorkflow` contém um método, `startMyWF`, para iniciar uma nova execução. O método é anotado com a anotação `@Execute` e deve ter um tipo de retorno de `void` ou `Promise<>`. Em uma determinada interface de fluxo de trabalho, no máximo um método poder ser anotado com essa anotação. O método é o ponto de entrada da lógica do fluxo de trabalho, e a estrutura chama esse método para executar a lógica do fluxo de trabalho quando uma tarefa de decisão é recebida.

A interface de fluxo de trabalho também define os sinais que podem ser enviados ao fluxo de trabalho. O método de sinal é invocado quando um sinal com um nome correspondente é recebido pela execução do fluxo de trabalho. Por exemplo, a interface de `MyWorkflow` declara um método de sinal, `signal1`, anotado com a anotação `@Signal`.

A anotação `@Signal` é necessária em métodos de sinal. O tipo de retorno de um método de sinal deve ser `void`. Uma interface de fluxo de trabalho pode ter zero ou mais métodos de sinal definidos

nela. Você pode declarar uma interface de fluxo de trabalho sem um método `@Execute` e alguns métodos `@Signal` para gerar clientes que não podem iniciar sua execução, mas podem enviar sinais a execuções em execução.

Os métodos anotados com as anotações `@Execute` e `@Signal` podem ter qualquer número de parâmetros de qualquer tipo além de `Promise<T>` ou de seus derivados. Isso permite que você passe entradas fortemente tipadas para uma execução de fluxo de trabalho no início e durante sua execução. O tipo de retorno do método `@Execute` deve ser `void` ou `Promise<>`.

Além disso, também é possível declarar um método na interface de fluxo de trabalho para relatar o estado mais recente de uma execução de fluxo de trabalho, por exemplo, o método `getState` do exemplo anterior. Esse estado não é o estado inteiro do aplicativo de fluxo de trabalho. O uso pretendido desse recurso é permitir que você armazene até 32 KB de dados para indicar o status mais recente da execução. Por exemplo, em um fluxo de trabalho de processamento de pedido, você pode armazenar uma sequência que indique que o pedido foi recebido, processado ou cancelado. O método é chamado pela estrutura sempre que uma tarefa de decisão é concluída para obter o estado mais recente. O estado é armazenado no Amazon Simple Workflow Service (Amazon SWF) e pode ser recuperado usando o cliente externo gerado. Isso permite que você verifique o estado mais recente de uma execução de fluxo de trabalho. Os métodos anotados com `@GetState` não devem usar nenhum argumento e não devem ter um tipo de retorno `void`. Você pode retornar qualquer tipo que atenda às suas necessidades desse método. No exemplo acima, um objeto de `MyWorkflowState` (consulte a definição a seguir) é retornado pelo método que é usado para armazenar um estado de sequência e uma porcentagem numérica completos. Espera-se que o método execute o acesso de somente leitura do objeto de implementação de fluxo de trabalho e seja invocado de forma síncrona, o que não permite o uso de qualquer operação assíncrona, como a chamada de métodos anotados com `@Asynchronous`. No máximo um método em uma interface de fluxo de trabalho pode ser anotado com a anotação `@GetState`.

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

Da mesma forma, um conjunto de atividades é definido usando uma interface anotada com a anotação `@Activities`. Cada método na interface corresponde a uma atividade; por exemplo:

```
@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);
}
```

A interface permite que você agrupe um conjunto de atividades relacionadas. Você pode definir qualquer número de atividades em uma interface de atividades e definir quantas interfaces de atividades desejar. Semelhante aos métodos `@Execute` e `@Signal`, os métodos de atividades podem usar qualquer número de argumentos de qualquer tipo diferente de `Promise<T>` ou de seus derivados. O tipo de retorno de uma atividade não deve ser `Promise<T>` ou seus derivados.

Registro do tipo de fluxo de trabalho e atividade

O Amazon SWF exige que os tipos de atividade e fluxo de trabalho sejam registrados antes de poderem ser usados. A estrutura registra automaticamente os fluxos de trabalho e as atividades nas implementações adicionadas ao operador. A estrutura procura por tipos que implementam fluxos de trabalho e atividades e os registra no Amazon SWF. Por padrão, a estrutura usa as definições de interface para inferir as opções de registro para os tipos de fluxo de trabalho e de atividade. Todas as interfaces de fluxo de trabalho precisam ter a anotação `@WorkflowRegistrationOptions` ou a anotação `@SkipRegistration`. O operador do fluxo de trabalho registra todos os tipos de fluxos de trabalho com os quais está configurado que possuem a anotação `@WorkflowRegistrationOptions`. Do mesmo modo, cada método de atividade precisa estar anotado com a anotação `@ActivityRegistrationOptions` ou a anotação `@SkipRegistration`, ou uma dessas anotações deve estar presente na interface `@Activities`. O operador da atividade registra todos os tipos de atividades com as quais está configurado e que possuem uma anotação `@ActivityRegistrationOptions`. O registro é realizado automaticamente ao iniciar um dos operadores. Tipos de fluxos de trabalho e de atividades que têm a anotação `@SkipRegistration` não são registrados. As anotações `@ActivityRegistrationOptions` e `@SkipRegistration` têm semântica de substituição e o mais específico é aplicado a um tipo de atividade.

Observe que o Amazon SWF não permite que você registre novamente ou modifique o tipo depois que ele foi registrado. A estrutura tentará registrar todos os tipos, mas se o tipo já está registrado ele não será registrado novamente e nenhum erro será relatado.

Se precisar modificar as configurações registadas, é necessário registrar uma nova versão do tipo. Você também pode substituir as configurações registradas ao iniciar uma nova execução ou ao chamar uma atividade que usa os clientes gerados.

O registro exige um nome do tipo e outras opções de registro. A implementação padrão determina as seguintes:

Nome e versão do tipo de fluxo de trabalho

A estrutura determina o nome do tipo de fluxo de trabalho a partir da interface do fluxo de trabalho. A forma do nome do tipo de fluxo de trabalho padrão é {*prefix*} {*name*}. O {*prefix*} é definido com o nome da `@Workflow` interface seguido por um '.' e o {*name*} é definido com o nome do `@Execute` método. O nome padrão do tipo de fluxo de trabalho no exemplo anterior é `MyWorkflow.startMyWF`. Você pode substituir o nome padrão usando o parâmetro nome do método `@Execute`. O nome padrão do tipo de fluxo de trabalho no exemplo é `startMyWF`. O nome não pode ser uma string vazia. Observe que ao substituir o nome usando `@Execute`, a estrutura não adiciona um prefixo automaticamente. Você está livre para usar o seu próprio esquema de nomenclatura.

A versão do fluxo de trabalho é especificada usando o parâmetro `version` da anotação `@Execute`. Não há um padrão para `version` e ele deve ser explicitamente especificado. O `version` é uma string de formato livre e você pode usar o seu próprio esquema de versionamento.

Nome do sinal

O nome do sinal pode ser especificado usando o parâmetro de nome da anotação `@Signal`. Se não especificado, ele assume o nome do método do sinal como padrão.

Nome e versão do tipo de atividade

A estrutura determina o nome do tipo de atividade a partir da interface de atividades. A forma do nome do tipo de atividade padrão é {*prefix*} {*name*}. O {*prefix*} é definido com o nome da `@Activities` interface seguido por um '.' e o {*name*} é definido com o nome do método. O padrão {*prefix*} pode ser substituído na `@Activities` anotação na interface de atividades. Também é possível especificar o nome do tipo de atividade usando a anotação `@Activity` no método da

atividade. Observe que ao substituir o nome usando `@Activity`, a estrutura não adicionará um prefixo automaticamente. Você está livre para usar o seu próprio esquema de nomenclatura.

A versão da atividade é especificada usando o parâmetro de versão da anotação `@Activities`. Essa versão é usada como padrão para todas as atividades definidas na interface e pode ser substituída para cada atividade usando a anotação `@Activity`.

Lista de tarefas padrão

A lista de tarefas padrão pode ser configurada usando as anotações `@WorkflowRegistrationOptions` e `@ActivityRegistrationOptions` e definindo o parâmetro `defaultTaskList`. Por padrão, ele é definido como `USE_WORKER_TASK_LIST`. Este é um valor especial que instrui a estrutura a usar a lista de tarefas que está configurada no objeto do operador usado para registrar o tipo de atividade ou de fluxo de trabalho. Você pode optar por não registrar uma lista de tarefas padrão configurando a lista de tarefas padrão como `NO_DEFAULT_TASK_LIST` usando essas anotações. Isso pode ser usada em casos onde deseja exigir que a lista de tarefas seja especificada durante o tempo de execução. Se nenhuma lista de tarefas padrão foi registrada, é necessário especificar a lista de tarefas ao iniciar o fluxo de trabalho ou ao chamar o método da atividade usando os parâmetros `StartWorkflowOptions` e `ActivitySchedulingOptions` na respectiva sobrecarga de método do cliente gerado.

Outras opções de registro

Todas as opções de registro de fluxo de trabalho e tipo de atividade permitidas pela API do Amazon SWF podem ser especificadas por meio da estrutura.

Para obter uma lista completa das opções de registro de fluxo de trabalho, consulte os seguintes:

- @Fluxo de trabalho
- @Execute
- @WorkflowRegistrationOptions
- @Signal

Para obter uma lista completa das opções de registro de atividade, consulte os seguintes:

- @Atividades
- @Atividades

- @ActivityRegistrationOptions

Se desejar ter controle total sobre o registro de tipo, consulte Extensibilidade de operadores.

Clientes de atividades e de fluxo de trabalho

Os clientes de fluxo de trabalho e de atividades são gerados pela estrutura com base nas interfaces `@Workflow` e `@Activities`. Interfaces separadas de clientes são geradas contendo métodos e configurações que fazem sentido somente no cliente. Se você estiver desenvolvendo usando o Eclipse, isso será feito pelo plug-in Amazon SWF Eclipse sempre que você salvar o arquivo que contém a interface apropriada. O código gerado é colocado no diretório de origens gerado em seu projeto no mesmo pacote que a interface.

> **Note**
>
> Observe que o nome do diretório padrão usado pelo Eclipse é .apt_generated. O Eclipse não mostra os diretórios cujos nomes começam com um '.' em Package Explorer. Use um nome de diretório diferente se desejar visualizar os arquivos gerados no Project Explorer (Explorador de pacote). No Eclipse, clique com o botão direito do mouse no Package Explorer (Explorador de pacote) e escolha Properties (Propriedades), Java Compiler (Compilador Java), Annotation processing (Processamento de anotação) e modifique a configuração Generate source directory (Gerar diretório de origem).

Clientes de fluxo de trabalho

Os artefatos gerados para o fluxo de trabalho contêm três interfaces do lado cliente e as classes que os implementam. Os clientes gerados incluem:

- Um cliente assíncrono com o objetivo de ser consumido em uma implementação de fluxo de trabalho que fornece métodos assíncronos para iniciar execuções de fluxo de trabalho e enviar sinais
- Um cliente externo que pode ser usado para execuções de início e envio de sinais e para recuperar o estado do fluxo de trabalho de fora do escopo de uma implementação de fluxo de trabalho
- Um cliente interno que pode ser usado para criar fluxos de trabalho contínuos

Por exemplo, as interfaces de clientes geradas para a interface `MyWorkflow` de exemplo são:

```
//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);
```

As interfaces têm métodos sobrecarregados que correspondem a cada método na interface `@Workflow` que você declarou.

O cliente externo espelha os métodos na interface `@Workflow` com uma sobrecarga adicional do método `@Execute` que usa `StartWorkflowOptions`. Você pode usar essa sobrecarga

para passar opções adicionais ao iniciar uma nova execução de fluxo de trabalho. Essas opções permitem que você substitua a lista de tarefas padrão, as configurações de tempo limite e as tags associadas à execução do fluxo de trabalho.

Por outro lado, o cliente assíncrono tem métodos que permitem invocações assíncronas do método `@Execute`. As sobrecargas do método a seguir são geradas na interface do cliente para o método `@Execute` na interface do fluxo de trabalho:

1. Uma sobrecarga que usa os argumentos originais como estão. O tipo de retorno dessa sobrecarga será `Promise<Void>` se o método original tiver retornado `void`. Caso contrário, será `Promise<>`, conforme declarado no método original. Por exemplo:

   Método original:

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

   Método gerado:

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

   Essa sobrecarga deve ser usada quando todos os argumentos do fluxo de trabalho estiverem disponíveis e não precisarem ser aguardados.
2. Uma sobrecarga que usa os argumentos originais como estão e argumentos variáveis adicionais do tipo `Promise<?>`. O tipo de retorno dessa sobrecarga será `Promise<Void>` se o método original tiver retornado `void`. Caso contrário, será `Promise<>`, conforme declarado no método original. Por exemplo:

   Método original:

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

   Método gerado:

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

   Essa sobrecarga deve ser usada quando todos os argumentos do fluxo de trabalho estiverem disponíveis e não precisarem ser aguardados, mas você desejar aguardar que algumas outras promessas fiquem prontas. O argumento da variável pode ser usado para passar esses objetos

`Promise<?>` que não foram declarados como argumentos, mas pelos quais você deseja aguardar antes de executar a chamada.

3. Uma sobrecarga que usa os argumentos originais como estão, um argumento adicional do tipo `StartWorkflowOptions` e argumentos variáveis adicionais do tipo `Promise<?>`. O tipo de retorno dessa sobrecarga será `Promise<Void>` se o método original tiver retornado `void`. Caso contrário, será `Promise<>`, conforme declarado no método original. Por exemplo:

   Método original:

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

   Método gerado:

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

   Essa sobrecarga deve ser usada quando todos os argumentos do fluxo de trabalho estiverem disponíveis e não precisarem ser aguardados, quando você desejar substituir as configurações padrão usadas para iniciar a execução do fluxo de trabalho ou quando desejar aguardar que algumas outras promessas fiquem prontas. O argumento da variável pode ser usado para passar esses objetos `Promise<?>` que não foram declarados como argumentos, mas pelos quais você deseja aguardar antes de executar a chamada.

4. Uma sobrecarga com cada argumento no método original substituído por um wrapper de `Promise<>`. O tipo de retorno dessa sobrecarga será `Promise<Void>` se o método original tiver retornado `void`. Caso contrário, será `Promise<>`, conforme declarado no método original. Por exemplo:

   Método original:

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

   Método gerado:

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

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

Essa sobrecarga deve ser usada quando os argumentos a serem passados para a execução do fluxo de trabalho deverem ser avaliados de forma assíncrona. Uma chamada para essa sobrecarga do método não será executada até que todos os argumentos passados para ela estejam prontos.

Se alguns dos argumentos já estiverem prontos, converta-os em uma `Promise` que já esteja em estado pronto por meio do método `Promise.asPromise(`*`value`*`)`. Por exemplo:

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

5. Uma sobrecarga com cada argumento no método original é substituída por um wrapper de `Promise<>`. A sobrecarga também tem argumentos variáveis adicionais do tipo `Promise<?>`. O tipo de retorno dessa sobrecarga será `Promise<Void>` se o método original tiver retornado `void`. Caso contrário, será `Promise<>`, conforme declarado no método original. Por exemplo:

   Método original:

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

   Método gerado:

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

   Essa sobrecarga deve ser usada quando os argumentos a serem passados para a execução do fluxo de trabalho deverem ser avaliados de forma assíncrona, e você desejar aguardar que algumas outras promessas também fiquem prontas. Uma chamada para essa sobrecarga do método não será executada até que todos os argumentos passados para ela estejam prontos.

6. Uma sobrecarga com cada argumento no método original substituído por um wrapper de `Promise<?>`. A sobrecarga também tem um argumento adicional do tipo `StartWorkflowOptions` e argumentos variáveis do tipo `Promise<?>`. O tipo de retorno dessa

sobrecarga será `Promise<Void>` se o método original tiver retornado `void`. Caso contrário, será `Promise<>`, conforme declarado no método original. Por exemplo:

Método original:

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

Método gerado:

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

Use esta sobrecarga quando os argumentos a serem passados para a execução do fluxo de trabalho serão avaliados de forma assíncrona, e você desejar substituir as configurações padrão usadas para iniciar a execução do fluxo de trabalho. Uma chamada para essa sobrecarga do método não será executada até que todos os argumentos passados para ela estejam prontos.

Também é gerado um método correspondente a cada sinal na interface do fluxo de trabalho, por exemplo:

Método original:

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

Método gerado:

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

O cliente assíncrono não contém um método correspondente ao método anotado com `@GetState` na interface original. Como a recuperação do estado requer uma chamada de serviço web, ela não é adequada para uso em um fluxo de trabalho. Portanto, ela é fornecida somente por meio do cliente externo.

O cliente interno tem o objetivo de ser usado dentro de um fluxo de trabalho para iniciar uma nova execução na conclusão da execução atual. Os métodos nesse cliente são semelhantes aos do

cliente assíncrono, mas retornam `void`. Esse cliente não tem métodos correspondentes a métodos anotados com `@Signal` e `@GetState`. Para obter mais detalhes, consulte Fluxos de trabalho contínuos.

Os clientes gerados derivam das interfaces base: `WorkflowClient` e `WorkflowClientExternal`, respectivamente, que fornecem os métodos que você pode usar para cancelar ou encerrar uma execução do fluxo de trabalho. Para obter mais detalhes sobre essas interfaces, consulte a documentação do AWS SDK para Java .

Os clientes gerados permitem interagir com as execuções de fluxo de trabalho de uma forma fortemente tipada. Depois de criada, uma instância de um cliente gerado é unida a uma execução de fluxo de trabalho específica e pode ser usada somente para essa execução. Além disso, a estrutura também fornece clientes dinâmicos que não são específicos a um tipo de fluxo de trabalho ou execução. Os clientes gerados dependem desse cliente nos bastidores. Você também pode usar esses clientes diretamente. Consulte a seção sobre Clientes dinâmicos.

A estrutura também gera fábricas para criação de clientes fortemente tipados. As fábricas de clientes geradas para a interface de `MyWorkflow` de exemplo são:

```
//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);
```

```
}
```

A interface base de `WorkflowClientFactory` é:

```
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);
}
```

Você deve usar essas fábricas para criar instâncias do cliente. A fábrica permite configurar o cliente genérico (o cliente genérico deve ser usado para fornecer a implementação personalizada do cliente) e o `DataConverter` usado pelo cliente para marshal de dados, bem como as opções usadas para iniciar a execução do fluxo de trabalho. Para obter mais detalhes, consulte as seções DataConverters e Execuções do fluxo de trabalho filho. O `StartWorkflowOptions` contém configurações que podem ser usadas para substituir os padrões (por exemplo, tempos limite) especificados no momento do registro. Para obter mais detalhes sobre a `StartWorkflowOptions` classe, consulte a AWS SDK para Java documentação.

O cliente externo pode ser usado para iniciar execuções de fluxo de trabalho de fora do escopo de um fluxo de trabalho enquanto o cliente assíncrono pode ser usado para iniciar uma execução de fluxo de trabalho no código dentro de um fluxo de trabalho. Para iniciar uma execução, use simplesmente o cliente gerado para chamar o método correspondente ao método anotado com `@Execute` na interface do fluxo de trabalho.

A estrutura também gera classes de implementação para as interfaces do cliente. Esses clientes criam e enviam solicitações ao Amazon SWF para executar a ação apropriada. A versão cliente do `@Execute` método inicia uma nova execução de fluxo de trabalho ou cria uma execução de fluxo de

trabalho secundária usando o Amazon SWF APIs. Da mesma forma, a versão cliente do `@Signal` método usa o Amazon SWF APIs para enviar um sinal.

> **Note**
>
> O cliente de fluxo de trabalho externo deve ser configurado com o cliente e o domínio do Amazon SWF. Você pode usar o construtor de fábrica do cliente que recebe esses parâmetros ou passar uma implementação de cliente genérica que já esteja configurada com o cliente e o domínio do Amazon SWF.
>
> A estrutura percorre a hierarquia de tipos da interface do fluxo de trabalho e também gera interfaces clientes para interfaces de fluxo de trabalho pai e deriva delas.

Clientes de atividades

De forma semelhante ao cliente de fluxo de trabalho, um cliente é gerado para cada interface anotada com `@Activities`. Os artefatos gerados incluem uma interface do lado do cliente e uma classe de cliente. A interface gerada para a interface `@Activities` de exemplo acima (`MyActivities`) é a seguinte:

```
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);
}
```

A interface contém um conjunto de métodos sobrecarregados que correspondem a cada método de atividade na interface `@Activities`. Essas sobrecargas são fornecidas por conveniência e permitem chamar atividades de forma assíncrona. Para cada método de atividade na interface `@Activities`, as seguintes sobrecargas do método são geradas na interface do cliente:

1. Uma sobrecarga que usa os argumentos originais como estão. O tipo de retorno dessa sobrecarga é `Promise<`*T*`>`, em que *T* é o tipo de retorno do método original. Por exemplo:

   Método original:

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

   Método gerado:

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

   Essa sobrecarga deve ser usada quando todos os argumentos do fluxo de trabalho estiverem disponíveis e não precisarem ser aguardados.

2. Uma sobrecarga que usa os argumentos originais como estão, um argumento do tipo `ActivitySchedulingOptions` e argumentos variáveis adicionais do tipo `Promise<?>`. O tipo de retorno dessa sobrecarga é `Promise<`*T*`>`, em que *T* é o tipo de retorno do método original. Por exemplo:

   Método original:

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

   Método gerado:

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

   Essa sobrecarga deve ser usada quando todos os argumentos do fluxo de trabalho estiverem disponíveis e não precisarem ser aguardados, quando você desejar substituir as configurações padrão ou quando desejar aguardar que `Promises` adicionais fiquem prontas. Os argumentos variáveis podem ser usados para passar esses objetos `Promise<?>` adicionais que não foram

declarados como argumentos, mas pelos quais você deseja aguardar antes de executar a chamada.

3. Uma sobrecarga com cada argumento no método original substituído por um wrapper de `Promise<>`. O tipo de retorno dessa sobrecarga é `Promise<`*T*`>`, em que *T* é o tipo de retorno do método original. Por exemplo:

   Método original:

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

   Método gerado:

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

   Essa sobrecarga deve ser usada quando os argumentos a serem passados para a atividade serão avaliados de forma assíncrona. Uma chamada para essa sobrecarga do método não será executada até que todos os argumentos passados para ela estejam prontos.

4. Uma sobrecarga com cada argumento no método original substituído por um wrapper de `Promise<>`. A sobrecarga também tem um argumento adicional do tipo `ActivitySchedulingOptions` e argumentos variáveis do tipo `Promise<?>`. O tipo de retorno dessa sobrecarga é `Promise<`*T*`>`, em que *T* é o tipo de retorno do método original. Por exemplo:

   Método original:

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

   Método gerado:

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

   Essa sobrecarga deve ser usada quando todos os argumentos a serem passados para a atividade serão avaliados de forma assíncrona, quando você desejar substituir as configurações padrão registradas com o tipo ou quando desejar aguardar que `Promises` adicionais fiquem prontas. Uma chamada para essa sobrecarga do método não será executada até que todos os argumentos

passados para ela estejam prontos. A classe de cliente gerada implementa essa interface. A implementação de cada método de interface cria e envia uma solicitação ao Amazon SWF para programar uma tarefa de atividade do tipo apropriado usando o Amazon SWF. APIs

5. Uma sobrecarga que usa os argumentos originais como estão e argumentos variáveis adicionais do tipo `Promise<?>`. O tipo de retorno dessa sobrecarga é `Promise<`*T*`>`, em que *T* é o tipo de retorno do método original. Por exemplo:

   Método original:

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

   Método gerado:

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

   Essa sobrecarga deve ser usada quando todos os argumentos da atividade estiverem disponíveis e não precisarem ser aguardados, mas você desejar aguardar que outros objetos `Promise` fiquem prontos.

6. Uma sobrecarga com cada argumento no método original substituído pelo wrapper de `Promise` e argumentos variáveis adicionais do tipo `Promise<?>`. O tipo de retorno dessa sobrecarga é `Promise<`*T*`>`, em que *T* é o tipo de retorno do método original. Por exemplo:

   Método original:

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

   Método gerado:

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

   Essa sobrecarga deve ser usada quando todos os argumentos da atividade serão aguardados de forma assíncrona e você também desejar aguardar que algumas outras `Promises` fiquem prontas. Uma chamada para essa sobrecarga do método será executada de forma assíncrona quando todos os objetos `Promise` passados estiverem prontos.

O cliente de atividade gerado tem um método protegido correspondente a cada método de atividade, chamado `{activity method name}Impl()`, no qual todas as sobrecargas de atividade chamam. Você pode substituir esse método para criar implementações fictícias de cliente. Esse método usa como argumentos: todos os argumentos do método original em wrappers de `Promise<>`, `ActivitySchedulingOptions` e argumentos variáveis do tipo `Promise<?>`. Por exemplo:

Método original:

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

Método gerado:

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

Opções de programação

O cliente de atividade gerado permite passar `ActivitySchedulingOptions` como um argumento. A estrutura `ActivitySchedulingOptions` contém definições que determinam a configuração da tarefa de atividade que a estrutura agenda no Amazon SWF. Essas configurações substituem os padrões especificados como opções de registro. Para especificar opções de programação de forma dinâmica, crie um objeto `ActivitySchedulingOptions`, configure-o conforme desejado e passe-o para o método da atividade. No exemplo a seguir, especificamos a lista de tarefas que deve ser usada para a tarefa de atividade. Isso substituirá a lista padrão de tarefas registradas para esta invocação da atividade.

```
public class OrderProcessingWorkflowImpl implements OrderProcessingWorkflow {

    OrderProcessingActivitiesClient activitiesClient
            = new OrderProcessingActivitiesClientImpl();

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

```
            schedulingOptions.setTaskList("TasklistAsia");
        } else {
            schedulingOptions.setTaskList("TasklistNorthAmerica");
        }

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

Clientes dinâmicos

Além dos clientes gerados, o framework também fornece clientes de uso geral (`DynamicWorkflowClient` e `DynamicActivityClient`), os quais você pode usar para iniciar dinamicamente execuções de fluxo de trabalho, enviar sinais, agendar atividades etc. Por exemplo, você pode desejar programar uma atividade cujo tipo não é conhecido no momento do design. Você pode usar o `DynamicActivityClient` para programar essa tarefa de atividade. Da mesma forma, você pode programar dinamicamente uma execução de fluxo de trabalho filho usando o `DynamicWorkflowClient`. No exemplo a seguir, o fluxo de trabalho procura a atividade em um banco de dados e usa o cliente de atividade dinâmico para programá-la:

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

```
                                    Void.class);
}
```

Para obter mais detalhes, consulte a AWS SDK para Java documentação.

Sinalização e cancelamento de execuções de fluxo de trabalho

O cliente de fluxo de trabalho gerado tem métodos correspondentes a cada sinal que pode ser enviado para o fluxo de trabalho. Você pode usá-los em um fluxo de trabalho para enviar sinais a outras execuções de fluxo de trabalho. Isso fornece um mecanismo tipado para envio de sinais. Entretanto, às vezes você pode precisar determinar dinamicamente o nome do sinal, por exemplo, quando o nome do sinal é recebido em uma mensagem. Você pode usar o cliente de fluxo de trabalho dinâmico para enviar sinais dinamicamente a qualquer execução de fluxo de trabalho. Da mesma forma, você pode usar o cliente para solicitar o cancelamento de outra execução de fluxo de trabalho.

No exemplo a seguir, o fluxo de trabalho procura a execução para enviar um sinal em um banco de dados e envia o sinal dinamicamente usando o cliente de fluxo de trabalho dinâmico.

```
//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);
}
```

Implementação do fluxo de trabalho

Para implementar um fluxo de trabalho, você escreve uma classe que implementa a interface `@Workflow` desejada. Por exemplo, a interface do fluxo de trabalho de exemplo (`MyWorkflow`) pode ser implementada da seguinte forma:

```
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);
  }
}
```

O método `@Execute` nessa classe é o ponto de entrada da lógica do fluxo de trabalho. Como a estrutura usa a repetição para reconstruir o estado do objeto quando uma tarefa de decisão deve ser processada, um novo objeto é criado para cada tarefa de decisão.

O uso de `Promise<`*T*`>` como um parâmetro não é permitido no método `@Execute` na interface de `@Workflow`. Isso é feito porque a execução de uma chamada assíncrona é puramente uma decisão do chamador. A própria implementação do fluxo de trabalho não depende de a invocação ser síncrona ou assíncrona. Portanto, a interface cliente gerada tem sobrecargas que usam parâmetros `Promise<`*T*`>` para que esses métodos possam ser chamados de forma assíncrona.

O tipo de retorno de um método `@Execute` só pode ser `void` ou `Promise<`*T*`>`. Observe que um tipo de retorno do cliente externo correspondente é `void` e não `Promise<>`. Como o cliente externo não deve ser usado a partir do código assíncrono, o cliente externo não retorna objetos. `Promise` Para obter resultados de execuções de fluxo de trabalho declaradas externamente, você pode projetar o fluxo de trabalho para atualizar o estado em um armazenamento de dados externo por meio de uma atividade. A visibilidade do Amazon SWF também APIs pode ser usada para recuperar o resultado de um fluxo de trabalho para fins de diagnóstico. Não é recomendável usar a visibilidade APIs para recuperar resultados de execuções de fluxo de trabalho como prática geral, pois essas chamadas de API podem ser limitadas pelo Amazon SWF. A visibilidade APIs exige que você identifique a

execução do fluxo de trabalho usando uma `WorkflowExecution` estrutura. Você pode obter essa estrutura do cliente do fluxo de trabalho gerado chamando o método `getWorkflowExecution`. Esse método retornará a estrutura `WorkflowExecution` correspondente à execução do fluxo de trabalho à qual o cliente está associado. Consulte a referência da API do Amazon Simple Workflow Service para obter mais detalhes sobre a visibilidade APIs.

Ao chamar atividades em sua implementação do fluxo de trabalho, você deve usar o cliente de atividades gerado. Da mesma forma, para enviar sinais, use os clientes de fluxo de trabalho gerados.

Contexto de decisão

A estrutura fornece um contexto de ambiente sempre que o código do fluxo de trabalho é executado pela estrutura. Esse contexto fornece funcionalidade específica ao contexto que você pode acessar em sua implementação de fluxo de trabalho, como a criação de um temporizador. Consulte a seção Contexto de execução para obter mais informações.

Exposição do estado da execução

O Amazon SWF permite que você adicione um estado personalizado no histórico do fluxo de trabalho. O estado mais recente relatado pela execução do fluxo de trabalho é retornado a você por meio de chamadas de visibilidade para o serviço Amazon SWF e no console do Amazon SWF. Por exemplo, em um fluxo de trabalho de processamento de pedidos, você pode relatar o status dos pedidos em diferentes estágios, como “pedido recebido “, “pedido enviado” etc. No AWS Flow Framework para Java, isso é feito por meio de um método em sua interface de fluxo de trabalho que é anotado com a anotação `@GetState`. Quando o agente de decisão conclui o processamento da tarefa de decisão, ele chama esse método para obter o estado mais recente da implementação do fluxo de trabalho. Além de chamadas de visibilidade, o estado também pode ser recuperado usando o cliente externo gerado (que usa chamadas de visibilidade da API internamente).

O exemplo a seguir demonstra como definir o contexto de execução.

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

O cliente externo gerado pode ser usado para recuperar o estado mais recente da execução do fluxo de trabalho a qualquer momento.

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

No exemplo acima, o estado da execução é relatado em vários estágios. Quando a instância do fluxo de trabalho é iniciada, `periodicWorkflow` relata o estado inicial como 'Just Started' (Acabou de iniciar). Cada chamada à `callPeriodicActivity` atualiza o estado do fluxo de trabalho. Depois que a `activity1` tiver sido chamada 100 vezes, o método retorna, e a instância de fluxo de trabalho é concluída.

Locais de fluxo de trabalho

Às vezes, pode ser necessário usar variáveis estáticas em sua implementação de fluxo de trabalho. Por exemplo, você pode desejar armazenar um contador que deve ser acessado de vários lugares (possivelmente de classes diferentes) na implementação do fluxo de trabalho. No entanto, você não pode depender de variáveis estáticas em seus fluxos de trabalho porque as variáveis estáticas são compartilhadas entre threads, o que é problemático porque um operador pode processar tarefas de decisão diferentes em threads diferentes ao mesmo tempo. Como alternativa, você pode

armazenar esse estado em um campo na implementação do fluxo de trabalho, mas por outro lado você precisará distribuir o objeto de implementação. Para atender a essa necessidade, a estrutura fornece uma classe `WorkflowExecutionLocal<?>`. Todos os estados que precisarem ter uma variável estática, como a semântica, devem ser mantidos como um local de instância usando `WorkflowExecutionLocal<?>`. Você pode declarar e usar uma variável estática deste tipo. Por exemplo, no trecho de código a seguir, um `WorkflowExecutionLocal<String>` é usado para armazenar um nome de usuário.

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

Implementação de atividades

As atividades são implementadas fornecendo uma implementação da interface `@Activities`. O AWS Flow Framework for Java usa as instâncias de implementação de atividades configuradas no trabalhador para processar tarefas de atividade em tempo de execução. O operador procura automaticamente a implementação de atividade do tipo apropriado.

Você pode usar propriedades e campos para passar recursos para instâncias de atividades, como conexões de banco de dados. Como o objeto de implementação da atividade pode ser acessado de vários encadeamentos, os recursos compartilhados devem ser seguros para encadeamentos.

Observe que a implementação de atividade não usa parâmetros do tipo `Promise<>` ou retorna objetos desse tipo. Porque a implementação da atividade não deve depender de como foi invocada (de forma síncrona ou assíncrona).

A interface de atividades mostrada antes pode ser implementada da seguinte forma:

```
public class MyActivitiesImpl implements MyActivities {

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

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

Um contexto local de thread está disponível para a implementação de atividade que pode ser usada para recuperar o objeto da tarefa, o objeto do conversor de dados que está sendo usado etc. O contexto atual pode ser acessado por meio do `ActivityExecutionContextProvider.getActivityExecutionContext()`. Para obter mais detalhes, consulte a AWS SDK para Java documentação `ActivityExecutionContext` e a seçãoContexto de execução.

Conclusão manual de atividades

A anotação `@ManualActivityCompletion` no exemplo acima é uma anotação opcional. É permitida somente em métodos que implementam uma atividade e é usada para configurar a atividade para não ser concluída automaticamente quando o método de atividade é retornado. Isso pode ser útil quando você deseja concluir a atividade de forma assíncrona; por exemplo, manualmente após a conclusão de uma ação humana.

Por padrão, a estrutura considera a atividade concluída quando o método da atividade é retornado. Isso significa que o operador de atividade informa a conclusão da tarefa da atividade ao Amazon SWF e fornece os resultados (se houver). Contudo, há casos de uso em que você não deseja que a tarefa de atividade seja marcada como concluída quando o método da atividade é retornado. Isso é especialmente útil quando você está modelando tarefas humanas. Por exemplo, o método de atividade pode enviar um e-mail para uma pessoa que deve concluir um trabalho antes que a tarefa de atividade seja concluída. Nesses casos, você pode anotar o método de atividade com a anotação `@ManualActivityCompletion` para informar ao operador de atividades de que não deve concluir a atividade automaticamente. Para concluir a atividade manualmente, você pode usar o `ManualActivityCompletionClient` fornecido na estrutura ou usar o método `RespondActivityTaskCompleted` no cliente Java do Amazon SWF fornecido no SDK do Amazon SWF. Para obter mais detalhes, consulte a AWS SDK para Java documentação.

Para concluir a tarefa de atividade, você precisa fornecer um token de tarefa. O token de tarefa é usado pelo Amazon SWF para identificar tarefas de forma exclusiva. Você pode acessar esse token no `ActivityExecutionContext` na implementação da atividade. Você deve passar esse token para a parte responsável por concluir a tarefa. Esse token pode ser recuperado do `ActivityExecutionContext` chamando `ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken()`.

A atividade `getName` do exemplo Hello World pode ser implementada para enviar um e-mail solicitando que alguém forneça uma mensagem de saudação:

```
@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";
}
```

O trecho de código a seguir pode ser usado para fornecer a saudação e fechar a tarefa usando o `ManualActivityCompletionClient`. Como alternativa, você também pode reprovar a tarefa:

```
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);
    }
}
```

Implementando AWS Lambda tarefas

Tópicos

- Sobre AWS Lambda
- Benefícios e limitações do uso de tarefas Lambda
- Usando tarefas Lambda em seus fluxos de trabalho AWS Flow Framework para Java
- Veja a HelloLambda amostra

Sobre AWS Lambda

AWS Lambda é um serviço computacional totalmente gerenciado que executa seu código em resposta a eventos gerados por código personalizado ou de vários AWS serviços, como Amazon S3, DynamoDB, Amazon Kinesis, Amazon SNS e Amazon Cognito. Para obter mais informações sobre o Lambda, consulte o Manual do desenvolvedor do AWS Lambda.

O Amazon Simple Workflow Service fornece uma tarefa Lambda para que você possa executar funções do Lambda no lugar ou ao lado das atividades tradicionais do Amazon SWF.

> **Important**
>
> Sua AWS conta será cobrada pelas execuções (solicitações) do Lambda executadas pelo Amazon SWF em seu nome. Para obter detalhes sobre os preços do Lambda, consulte https://aws.amazon.com/lambda/ preços/.

Benefícios e limitações do uso de tarefas Lambda

Há vários benefícios no uso de tarefas Lambda em vez de uma atividade tradicional do Amazon SWF:

- As tarefas Lambda não precisam ser registradas ou versionadas como os tipos de atividade do Amazon SWF.
- Você pode usar quaisquer funções do Lambda existentes que já tenha definido em seus fluxos de trabalho.
- As funções do Lambda são chamadas diretamente pelo Amazon SWF; não há necessidade de implementar um programa de trabalho para executá-las, como deve ser feito com as atividades tradicionais.
- O Lambda fornece métricas e logs para rastrear e analisar as execuções das funções.

Há também uma série de limitações em relação a tarefas Lambda sobre as quais você deve estar ciente:

- As tarefas do Lambda só podem ser executadas em AWS regiões que fornecem suporte para o Lambda. Consulte Regiões e endpoints do Lambda na Referência geral da Amazon Web Services para obter detalhes sobre as regiões atualmente compatíveis com o Lambda.

- Atualmente, as tarefas Lambda são suportadas somente pela API HTTP SWF básica e pelo AWS Flow Framework for Java. Atualmente, não há suporte para tarefas Lambda no AWS Flow Framework for Ruby.

Usando tarefas Lambda em seus fluxos de trabalho AWS Flow Framework para Java

Há três requisitos para usar tarefas Lambda em seus fluxos de trabalho AWS Flow Framework para Java:

- Uma função do Lambda a ser executada. Você pode usar qualquer função do Lambda que tenha definido. Para obter mais informações sobre como criar funções do Lambda, consulte o Guia do desenvolvedor do AWS Lambda.
- Um perfil do IAM que fornece acesso para executar funções do Lambda a partir de seus fluxos de trabalho do Amazon SWF.
- Código para agendar a tarefa Lambda de dentro do seu fluxo de trabalho.

Configurar uma função do IAM

Antes de poder invocar as funções do Lambda a partir do Amazon SWF, você deve fornecer um perfil do IAM que forneça acesso ao Lambda a partir do Amazon SWF. Você também pode:

- escolha uma função predefinida, Role, AWSLambda para dar permissão aos seus fluxos de trabalho para invocar qualquer função Lambda associada à sua conta.
- defina sua própria política e função associada para dar permissão aos fluxos de trabalho para invocar funções específicas do Lambda, especificadas por seus Amazon Resource Names (). ARNs

Limite de permissões de um perfil do IAM

Você pode limitar as permissões em um perfil do IAM fornecido ao Amazon SWF usando as chaves de contexto `SourceArn` e `SourceAccount` em sua política de confiança de recursos. Essas chaves limitam o uso de uma política do IAM para que ela seja usada somente em execuções do Amazon Simple Workflow Service que pertençam ao domínio ARN especificado. Se você usar ambas as chaves de contexto de condição global, o valor `aws:SourceAccount` e a conta referenciada no

valor `aws:SourceArn` deverão usar o mesmo ID de conta quando usados na mesma declaração de política.

No exemplo de política de confiança a seguir, usamos a chave de contexto `SourceArn` para restringir o perfil de serviço do IAM a ser usado somente nas execuções do Amazon Simple Workflow Service que pertençam a `someDomain` na conta `123456789012`.

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

No exemplo de política de confiança a seguir, usamos a chave de contexto `SourceAccount` para restringir o perfil de serviço do IAM a ser usado somente em execuções do Amazon Simple Workflow Service na conta `123456789012`.

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

```
            "aws:SourceAccount": "123456789012"
          }
        }
      }
    ]
}
```

Fornecer ao Amazon SWF acesso para invocar qualquer função do Lambda

Você pode usar a função predefinida, Role, para dar aos fluxos de trabalho do Amazon SWF a capacidade de invocar qualquer AWSLambdafunção Lambda associada à sua conta.

Para usar a AWSLambda função para dar ao Amazon SWF acesso para invocar funções do Lambda

1. Abra o console do Amazon IAM.
2. Escolha Roles e depois Create New Role.
3. Dê um nome para a sua função, como `swf-lambda` e escolha Next Step.
4. Em Perfis de serviço da AWS , escolha Amazon SWF e selecione Próxima etapa.
5. Na tela Anexar política, escolha AWSLambdaFunção na lista.
6. Escolha Next Step e depois Create Role depois de analisar a função.

Definir um perfil do IAM para fornecer acesso para invocar uma função do Lambda específica

Se quiser fornecer acesso para invocar uma função do Lambda específica do seu fluxo de trabalho, você precisará definir sua própria política do IAM.

Para criar uma política do IAM para fornecer acesso a uma função do Lambda específica

1. Abra o console do Amazon IAM.
2. Escolha Policies e depois Create Policy.
3. Escolha Copiar uma política AWS gerenciada e selecione AWSLambdaFunção na lista. Uma política será gerada para você. Opcionalmente, edite seu nome e descrição para atender às suas necessidades.
4. No campo Recurso do Documento de política, adicione o ARN de uma ou mais funções do Lambda. Por exemplo:

```
{
  "Version": "2012-10-17",
```

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

> **Note**
>
> Para obter uma descrição completa de como especificar recursos em um perfil do IAM, consulte Visão geral das políticas do IAM em Uso do IAM.

5. Escolha Create Policy para concluir a criação da sua política.

Em seguida, você pode selecionar essa política ao criar um perfil do IAM e usá-lo para conceder acesso de invocação aos seus fluxos de trabalho do Amazon SWF. Esse procedimento é muito semelhante à criação de uma função com a política de AWSLambdafunção. Em vez disso, escolha sua própria política ao criar a função.

Para criar uma função do Amazon SWF usando sua política Lambda

1. Abra o console do Amazon IAM.
2. Escolha Roles e depois Create New Role.
3. Dê um nome para a sua função, como `swf-lambda-function` e escolha Next Step.
4. Em Perfis de serviço da AWS , escolha Amazon SWF e selecione Próxima etapa.
5. Em Anexar política, escolha a política específica da função do Lambda na lista.
6. Escolha Next Step e depois Create Role depois de analisar a função.

Agendar uma tarefa Lambda para execução

Depois de definir um perfil do IAM que permite invocar funções do Lambda, você pode programá-las para execução como parte do seu fluxo de trabalho.

> **Note**
>
> Esse processo é totalmente demonstrado pela HelloLambda amostra no AWS SDK para Java.

Como agendar uma tarefa Lambda para execução

1. Na implementação do fluxo de trabalho, obtenha uma instância do `LambdaFunctionClient` chamando `getLambdaFunctionClient()` em uma instância `DecisionContext`.

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

2. Programe a tarefa usando o método `scheduleLambdaFunction()` no `LambdaFunctionClient`, passando a ele o nome da função do Lambda que você criou e quaisquer dados de entrada para a tarefa 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. Em seu iniciador de execução de fluxo de trabalho, adicione a função do Lambda do IAM às opções padrão do fluxo de trabalho usando `StartWorkflowOptions.withLambdaRole()` e, em seguida, passe as opções ao iniciar o fluxo de trabalho.

```
// 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);
```

Veja a HelloLambda amostra

Uma amostra que fornece uma implementação de um fluxo de trabalho que usa uma tarefa Lambda é fornecida no AWS SDK para Java. Para visualizar e/ou executá-lo, faça download da fonte.

Uma descrição completa de como criar e executar a HelloLambdaamostra é fornecida no arquivo README fornecido com as amostras AWS Flow Framework de Java.

Executando programas escritos com o AWS Flow Framework para Java

Tópicos

- WorkflowWorker
- ActivityWorker
- Modelo de threading de operador
- Extensibilidade de operadores

A estrutura fornece classes de trabalho para inicializar o AWS Flow Framework para o Ambiente de Execução Java e se comunicar com o Amazon SWF. Para implementar um fluxo de trabalho ou um operador de atividade, você deve criar e iniciar uma instância de uma classe de operador. Essas classes de trabalho são responsáveis pelo gerenciamento de operações assíncronas em andamento, pela invocação de métodos assíncronos que se tornam desbloqueados e pela comunicação com o Amazon SWF. Elas podem ser configuradas com implementações de fluxo de trabalho e de atividades, o número de threads, a lista de tarefas para consulta etc.

A estrutura é fornecida com duas classes de operador, uma para atividades e uma para fluxos de trabalho. Para executar a lógica do fluxo de trabalho, use a classe `WorkflowWorker`. De

modo semelhante, para atividades a classe `ActivityWorker` é usada. Essas classes pesquisam automaticamente o Amazon SWF em busca de tarefas de atividade e invocam os métodos apropriados em sua implementação.

O exemplo a seguir mostra como instanciar um `WorkflowWorker` e iniciar a pesquisa de tarefas:

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

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

As etapas básicas para criar uma instância do `ActivityWorker` e iniciar a pesquisa de tarefas são:

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

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

Quando você quiser encerrar uma atividade ou um agente de decisão, sua aplicação deverá encerrar as instâncias das classes de trabalho que estão sendo usadas, bem como a instância do cliente Java do Amazon SWF. Isso garante que todos os recursos usados pelas classes de operador sejam liberados corretamente.

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

Para iniciar uma execução, simplesmente crie uma instância do cliente externo gerado e chame o método `@Execute`.

```
MyWorkflowClientExternalFactory factory = new MyWorkflowClientExternalFactoryImpl();
```

```
MyWorkflowClientExternal client = factory.getClient();
client.start();
```

WorkflowWorker

Como o nome sugere, essa classe de operador tem o objetivo de ser usada pela implementação do fluxo de trabalho. É configurada com uma lista de tarefas e o tipo de implementação de fluxo de trabalho. A classe de operador executa um loop para pesquisar tarefas de decisão na lista de tarefas especificada. Quando uma tarefa de decisão é recebida, ela cria uma instância da implementação do fluxo de trabalho e chama o método `@Execute` para processar a tarefa.

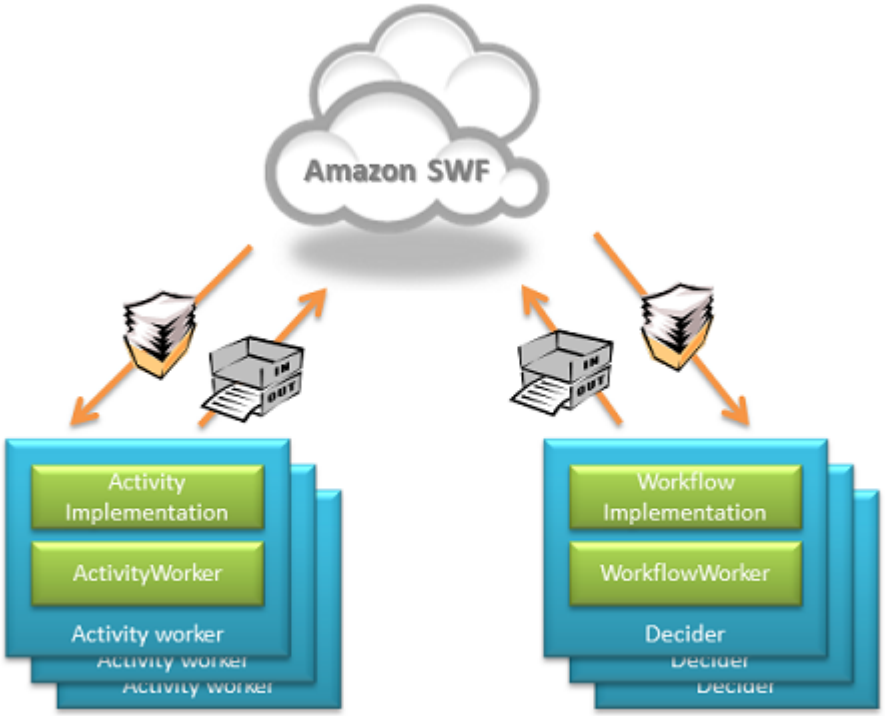
ActivityWorker

Para implementar operadores de atividades, você pode usar a classe `ActivityWorker` para pesquisar tarefas de atividades convenientemente em uma lista de tarefas. Você configura o operador de atividades com objetos da implementação da atividade. Essa classe de operador executa um loop para pesquisar tarefas de atividades na lista de tarefas especificada. Quando uma tarefa de atividade é recebida, ela procura a implementação apropriada que você forneceu e chama o método de atividade para processar a tarefa. Ao contrário de `WorkflowWorker`, que chama uma a fábrica para criar uma nova instância para cada tarefa de decisão, o `ActivityWorker` simplesmente usa o objeto que você forneceu.

A `ActivityWorker` classe usa as anotações AWS Flow Framework for Java para determinar as opções de registro e execução.

Modelo de threading de operador

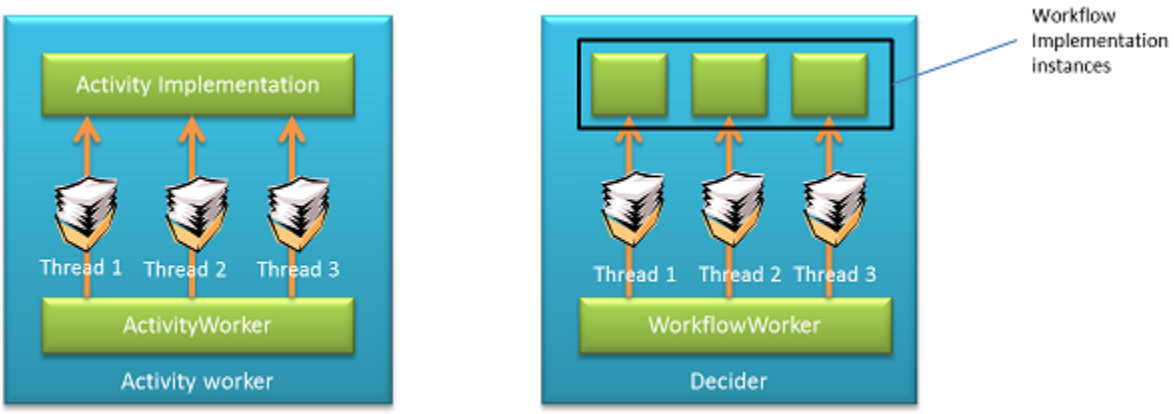
No AWS Flow Framework for Java, a personificação de uma atividade ou decisor é uma instância da classe trabalhadora. O aplicativo é responsável por configurar e instanciar o objeto de operador em cada máquina e processo que deve atuar como um operador. O objeto operador recebe automaticamente as tarefas do Amazon SWF, despacha-as para sua atividade ou implementação de fluxo de trabalho e informa os resultados ao Amazon SWF. É possível que uma única instância de fluxo de trabalho abranja vários operadores. Quando o Amazon SWF tem uma ou mais tarefas de atividade pendentes, ele atribui uma tarefa ao primeiro operador disponível, depois ao próximo, e assim por diante. Isso possibilita que tarefas que pertencem à mesma instância de fluxo de trabalho sejam processadas simultaneamente em diferentes operadores.



Além disso, cada operador pode ser configurado para processar tarefas em vários threads. Isso significa que as tarefas de atividades de uma instância de fluxo de trabalho podem ser executadas simultaneamente mesmo que haja apenas um operador.

As tarefas de decisão se comportam de forma semelhante, com a exceção de que o Amazon SWF garante que, para uma determinada execução de fluxo de trabalho, apenas uma decisão pode ser executada por vez. Normalmente, uma única execução de fluxo de trabalho exige várias tarefas de decisão, portanto, ela pode acabar executando em vários processos e threads também. O agente de decisão é configurado com o tipo da implementação do fluxo de trabalho. Quando uma tarefa de decisão é recebida pelo agente de decisão, ele cria uma instância (objeto) da implementação do fluxo de trabalho. A estrutura fornece um padrão de fábrica extensível para criar essas instâncias. A fábrica de fluxo de trabalho padrão cria um novo objeto todas as vezes. Você pode fornecer fábricas personalizadas para substituir esse comportamento.

Ao contrário dos agentes de decisão, que são pré-configurados com tipos de implementação de fluxo de trabalho, os operadores de atividades são configurados com instâncias (objetos) das implementações de atividade. Quando uma tarefa de atividade é recebida pelo operador de atividades, ela é despachada para o objeto adequado da implementação de atividade.



O operador do fluxo de trabalho mantém um único pool de threads e executa o fluxo de trabalho no mesmo thread que foi usado para pesquisar o Amazon SWF para a tarefa. Como as atividades são de longa duração (pelo menos quando comparadas à lógica do fluxo de trabalho), a classe de trabalhador de atividades mantém dois grupos separados de encadeamentos; um para pesquisar o Amazon SWF para tarefas de atividade e outro para processar tarefas executando a implementação da atividade. Isso permite que você configure o número de threads para pesquisa de tarefas separado do número de threads para execução de tarefas. Por exemplo, você pode ter um pequeno número de threads para pesquisar um grande número de threads para executar as tarefas. A classe de operador de atividade pesquisa o Amazon SWF para uma tarefa somente quando tiver um thread de pesquisa livre, bem como um thread livre para processar a tarefa.

Esse comportamento de thread e de instância sugere que:

1. As implementações de atividades devem ser stateless. Você não deve usar variáveis de instância para armazenar o estado do aplicativo em objetos de atividade. Você pode, porém, usar campos para armazenar recursos, como conexões de banco de dados.
2. As implementações de atividades devem ser livres de thread. Como a mesma instância pode ser usada para processar tarefas de diferentes segmentos ao mesmo tempo, o acesso aos recursos compartilhados a partir do código da atividade deve ser sincronizado.
3. A implementação de fluxo de trabalho pode ser stateful, e as variáveis de instância podem ser usadas para armazenar o estado. Embora uma nova instância da implementação de fluxo de trabalho seja criada para processar cada tarefa de decisão, a estrutura garantirá que o estado seja recriado adequadamente. No entanto, a implementação do fluxo de trabalho deve ser determinista. Consulte a seção Entendendo uma tarefa em AWS Flow Framework Java para obter mais detalhes.

4. As implementações de fluxo de trabalho não precisam estar livres de thread ao usar a fábrica padrão. A implementação padrão garante que apenas um thread usa uma instância da implementação de fluxo de trabalho de cada vez.

Extensibilidade de operadores

O AWS Flow Framework for Java também contém algumas classes de trabalho de baixo nível que oferecem controle refinado e extensibilidade. Usando-as, você pode personalizar totalmente o registro dos tipos de fluxo de trabalho e de atividades e definir fábricas criando objetos de implementação. Esses operadores são `GenericWorkflowWorker` e `GenericActivityWorker`.

O `GenericWorkflowWorker` pode ser configurado com uma fábrica para criar fábricas de definição de fluxo de trabalho. A fábrica de definição de fluxo de trabalho é responsável por criar instâncias da implementação de fluxo de trabalho e por fornecer definições de configuração, como opções de registro. Em condições normais, você deve usar a classe `WorkflowWorker` diretamente. Ela criará e configurará automaticamente a implementação das fábricas fornecidas na estrutura, `POJOWorkflowDefinitionFactoryFactory` e `POJOWorkflowDefinitionFactory`. A fábrica exige que a classe de implementação de fluxo de trabalho deve ter um construtor sem argumentos. Esse construtor é usado para criar instâncias do objeto de fluxo de trabalho em tempo de execução. A fábrica procura as anotações que você usou na interface e na implementação do fluxo de trabalho para criar opções adequadas de registro e de execução.

Você pode fornecer sua própria implementação das fábricas implementando `WorkflowDefinitionFactoryFactory`, `WorkflowDefinitionFactory` e `WorkflowDefinition`. A classe `WorkflowDefinition` é usada pela classe do operador para despachar tarefas de decisão e sinais. Com a implementação dessas classes base, você pode personalizar totalmente a fábrica e a expedição de solicitações para a implementação de fluxo de trabalho. Por exemplo, você pode usar esses pontos de extensibilidade para fornecer um modelo de programação personalizado para escrever fluxos de trabalho, por exemplo, com base em suas próprias anotações, ou em gerá-los no WSDL em vez da primeira abordagem de código usada pela estrutura. Para usar suas fábricas personalizadas, você precisará usar a classe `GenericWorkflowWorker`. Para obter mais detalhes sobre essas classes, consulte a AWS SDK para Java documentação.

Da mesma forma, o `GenericActivityWorker` permite que você forneça uma fábrica personalizada de implementação de atividades. Com a implementação das classes `ActivityImplementationFactory` e `ActivityImplementation` você pode controlar completamente a instanciação de atividades bem como personalizar as opções de registro e

de execução. Para obter mais detalhes sobre essas classes, consulte a AWS SDK para Java documentação.

Contexto de execução

Tópicos

A estrutura fornece um contexto de ambiente às implementações de fluxo de trabalho e de atividades. Esse contexto é específico à tarefa que está sendo processada e fornece alguns utilitários que você pode usar em sua implementação. Um objeto de contexto é criado sempre que uma tarefa nova é processada pelo operador.

Contexto de decisão

Quando uma tarefa de decisão é executada, a estrutura fornece o contexto para a implementação do fluxo de trabalho por meio da classe `DecisionContext`. `DecisionContext` fornece informações sensíveis ao contexto, como ID de execução de fluxo de trabalho e funcionalidade de relógio e temporizador.

Acesso DecisionContext na implementação do fluxo de trabalho

Você pode acessar o `DecisionContext` em sua implementação de fluxo de trabalho usando a classe `DecisionContextProviderImpl`. Como alternativa, você pode injetar o contexto em um campo ou em uma propriedade de sua implementação de fluxo de trabalho usando Spring conforme mostrado na seção Injeção de capacidade de teste e dependência.

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

Criação de um relógio e de um temporizador

O `DecisionContext` contém uma propriedade do tipo `WorkflowClock` que fornece a funcionalidade de temporizador e de relógio. Como a lógica do fluxo de trabalho precisa ser determinística, você não deve usar diretamente o relógio do sistema na implementação do fluxo de trabalho. O método `currentTimeMills` no `WorkflowClock` retorna a hora de início do evento

da decisão que está sendo processada. Isso garante que você obtenha o mesmo valor de tempo durante a reprodução, portanto, tornando a lógica do seu fluxo de trabalho determinista.

O `WorkflowClock` também tem um método `createTimer` que retorna um objeto de `Promise` que se torna pronto depois do intervalo especificado. Você pode usar esse valor como um parâmetro para outros métodos assíncronos para atrasar sua execução no período especificado. Dessa forma, você pode programar com eficiência uma atividade ou um método assíncrono para ser executado posteriormente.

O exemplo na lista a seguir demonstra como chamar periodicamente uma atividade.

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

Na lista acima, o método assíncrono `callPeriodicActivity` chama `activity1` e cria um temporizador usando o `AsyncDecisionContext` atual. Ele passa o `Promise` retornado como um argumento para uma chamada recursiva a si próprio. Essa chamada recursiva espera até que o temporizador seja disparado (uma hora neste exemplo) antes de executar.

Contexto de execução de atividades

Assim como a `DecisionContext` fornece informações de contexto quando uma tarefa de decisão está sendo processada, a `ActivityExecutionContext` fornece informações de contexto semelhantes quando uma tarefa de atividade está sendo processada. Esse contexto está disponível para o código de atividade por meio da classe `ActivityExecutionContextProviderImpl`.

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

Usando a `ActivityExecutionContext`, você pode executar o seguinte:

Pulsação de uma atividade de longa execução

Se a atividade estiver em execução há muito tempo, ela deverá informar periodicamente seu progresso ao Amazon SWF para que ele saiba que a tarefa ainda está progredindo. Na ausência dessa pulsação, a tarefa poderá esgotar o tempo limite se um tempo limite de pulsação da tarefa tiver sido definido no registro do tipo da atividade ou durante a programação da atividade. Para enviar uma pulsação, você pode usar o método `recordActivityHeartbeat` na `ActivityExecutionContext`. A pulsação também fornece um mecanismo para cancelar atividades em andamento. Consulte a seção Tratamento de erros para obter mais detalhes e um exemplo.

Obtenção de detalhes da tarefa de atividade

Se desejar, você pode obter todos os detalhes da tarefa de atividade que foram passados pelo Amazon SWF quando o executor obteve a tarefa. Isso inclui informações sobre as entradas para a tarefa, o tipo da tarefa, o token da tarefa etc. Se você quiser implementar uma atividade que seja concluída manualmente, por exemplo, por uma ação humana, deverá usar o `ActivityExecutionContext` para recuperar o token da tarefa e passá-lo para o processo que acabará por concluir a tarefa da atividade. Consulte a seção em Conclusão manual de atividades para obter mais detalhes.

Obter o objeto do cliente Amazon SWF que está sendo usado pelo executor

O objeto do cliente Amazon SWF que está sendo usado pelo executor pode ser recuperado chamando o método `getService` em `ActivityExecutionContext`. Isso é útil se você quiser fazer uma chamada direta para o serviço Amazon SWF.

Execuções do fluxo de trabalho filho

Nos exemplos até agora, iniciamos uma execução de fluxo de trabalho diretamente de um aplicativo. Contudo, uma execução de fluxo de trabalho pode ser iniciada de dentro de um fluxo de trabalho chamando o método do ponto de entrada do fluxo de trabalho no cliente gerado. Quando uma execução de fluxo de trabalho é iniciada no contexto de outra execução de fluxo de trabalho, ela é chamada de execução do fluxo de trabalho filho. Isso permite levar em conta novamente fluxos de trabalho complexos em unidades menores e, potencialmente, compartilhá-los em fluxos de trabalho diferentes. Por exemplo, você pode criar um fluxo de trabalho de processamento de pagamento e chamá-lo a partir de um fluxo de trabalho de processamento de pedido.

Semanticamente, a execução do fluxo de trabalho filho se comporta da mesma forma que um fluxo de trabalho autônomo, exceto para as seguintes diferenças:

1. Quando o fluxo de trabalho principal for encerrado devido a uma ação explícita do usuário (por exemplo, chamando a API `TerminateWorkflowExecution` do Amazon SWF ou devido a um tempo limite), o destino da execução do fluxo de trabalho secundário será determinado por uma política secundária. Defina essa política filha para encerrar, cancelar ou abandonar (manter em execução) execuções do fluxo de trabalho filho.
2. A saída do fluxo de trabalho filho (valor de retorno do método do ponto de entrada) pode ser usada pela execução do fluxo de trabalho pai assim como o `Promise<T>` retornado por um método assíncrono. Isso é diferente das execuções autônomas em que o aplicativo deve obter a saída usando o Amazon SWF. APIs

No exemplo a seguir, o fluxo de trabalho `OrderProcessor` cria um fluxo de trabalho filho `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);

}
```

Fluxos de trabalho contínuos

Em alguns casos de uso, talvez seja necessário um fluxo de trabalho que é executado para sempre ou durante um longo período, por exemplo, um fluxo de trabalho que monitora a integridade de um frota de servidor.

> **Note**
>
> Como o Amazon SWF mantém todo o histórico da execução de um fluxo de trabalho, o histórico continuará crescendo com o tempo. A estrutura recupera esse histórico do Amazon SWF ao realizar um replay, e isso será caro se o tamanho do histórico for muito grande. Em tais fluxos de trabalho contínuos ou de longa duração, feche periodicamente a execução atual e inicie uma nova para continuar processando.

Isso é uma continuação lógica da execução do fluxo de trabalho. O cliente interno gerado pode ser usado para esse propósito. Na implementação do fluxo de trabalho, apenas chame o método `@Execute` no cliente interno. Assim que a execução atual for concluída, a estrutura iniciará uma nova execução usando o mesmo ID de fluxo de trabalho.

Você também pode continuar execução chamando o método `continueAsNewOnCompletion` no `GenericWorkflowClient` que você pode recuperar do `DecisionContext` atual. Por exemplo, a implementação de fluxo de trabalho a seguir define um temporizador após um dia e chama o seu próprio ponto de entrada para iniciar uma nova execução.

```
public class ContinueAsNewWorkflowImpl implements ContinueAsNewWorkflow {

    private DecisionContextProvider contextProvider
         = new DecisionContextProviderImpl();

    private ContinueAsNewWorkflowSelfClient selfClient
         = new ContinueAsNewWorkflowSelfClientImpl();

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

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

```
    }

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

Quando um fluxo de trabalho se chama de forma recursiva, a estrutura fechará o fluxo de trabalho atual quando todas as tarefas pendentes forem concluídas e iniciará uma nova execução de fluxo de trabalho. Observe que enquanto houver tarefas pendentes, a execução de fluxo de trabalho atual não fechará. A nova execução não herdará automaticamente nenhum histórico ou dados de execução original. Se desejar carregar algum estado para a nova execução, é necessário enviá-lo explicitamente como entrada.

Definindo a prioridade da tarefa no Amazon SWF

Por padrão, as tarefas em uma lista de tarefas são entregues de acordo com sua hora de chegada: as primeiras tarefas a serem agendadas são geralmente executadas antes, na medida do possível. Ao definir uma prioridade de tarefa opcional, você pode dar prioridade a determinadas tarefas: O Amazon SWF tentará entregar as tarefas de maior prioridade em uma lista de tarefas antes daquelas com menor prioridade.

Você pode definir prioridades de tarefas para fluxos de trabalho e atividades. A prioridade de tarefas de um fluxo de trabalho não afeta a prioridade de nenhuma das tarefas de atividades que ele agenda, nem afeta nenhum dos fluxos de trabalho secundários que ele inicia. A prioridade padrão de uma atividade ou fluxo de trabalho é definida (por você ou pelo Amazon SWF) durante o registro, e a prioridade da tarefa registrada é sempre usada, a menos que seja substituída durante o agendamento da atividade ou o início da execução de um fluxo de trabalho.

Valores de prioridade de tarefas podem variar de "-2147483648" até "2147483647", com números mais altos indicando maior prioridade. Se você não definir a prioridade de tarefas para uma atividade ou um fluxo de trabalho, este(a) último(a) receberá uma prioridade de zero ("0").

Tópicos

- Definindo a prioridade de tarefas para fluxos de trabalho
- Definindo a prioridade de tarefas para atividades

Definindo a prioridade de tarefas para fluxos de trabalho

Você pode definir a prioridade de tarefa para um fluxo de trabalho ao registrá-lo ou iniciá-lo. A prioridade de tarefa definida quando o tipo de fluxo de trabalho é registrado é utilizada como padrão para qualquer execução de fluxo de trabalho desse tipo, a menos que ela seja substituída no momento de iniciar a execução de fluxo de trabalho.

Para registrar um tipo de fluxo de trabalho com uma prioridade de tarefa padrão, defina a defaultTaskPrioritydopção WorkflowRegistrationOptionsao declará-la:

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

Você também pode definir a taskPriority para um fluxo de trabalho ao iniciá-lo, substituindo a prioridade de tarefa registrada (padrão).

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

PriorityWorkflowClientExternalFactory cf
    = new PriorityWorkflowClientExternalFactoryImpl(swfService, domain);

priority_workflow_client = cf.getClient();

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

Além disso, você pode definir a prioridade de tarefa ao iniciar um fluxo de trabalho filho ou ao continuar um fluxo de trabalho como novo. Por exemplo, você pode definir a opção ContinueAsNewWorkflowExecutionParameterstaskPriority em ou em. StartChildWorkflowExecutionParameters

Definindo a prioridade de tarefas para atividades

Você pode definir a prioridade de tarefa para uma atividade ao registrá-la ou agendá-la. A prioridade de tarefa definida ao registrar um tipo de atividade é usada como a prioridade padrão quando essa atividade é executada, a menos que ela seja substituída no momento de agendar a atividade.

Para registrar um tipo de atividade com uma prioridade de tarefa padrão, defina a defaultTaskPriorityopção ActivityRegistrationOptionsao declará-la:

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

Você também pode definir a taskPriority para uma atividade ao programá-la, substituindo a prioridade de tarefa registrada (padrão).

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

ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();

activityClient.doSomethingImportant(activityOptions);
```

DataConverters

Quando sua implementação de fluxo de trabalho chama uma atividade remota, as entradas passadas para ela e o resultado da execução da atividade devem ser serializados para que possam ser enviados pela conexão. A estrutura usa a DataConverter classe para essa finalidade. Essa é uma classe abstrata que você pode implementar para fornecer seu próprio serializador. Uma implementação padrão baseada no serializador Jackson, `JsonDataConverter`, é fornecida na estrutura. Para obter mais detalhes, consulte a documentação do AWS SDK para Java. Consulte a documentação do processador Jackson JSON para obter detalhes sobre como o Jackson executa a serialização, bem como sobre as anotações do Jackson que podem ser usadas para influenciá-la. O formato da conexão usada é considerado como parte do contrato. Portanto, você pode especificar

um `DataConverter` em suas atividades e interfaces de fluxo de trabalho definindo a propriedade `DataConverter` das anotações `@Activities` e `@Workflow`.

A estrutura criará objetos do tipo `DataConverter` que você especificou na anotação `@Activities` para serializar as entradas para a atividade e para desserializar o resultado. Da mesma forma, os objetos do tipo `DataConverter` que você especifica na anotação `@Workflow` serão usados para serializar os parâmetros que você passa para o fluxo de trabalho e, no caso de fluxo de trabalho filho, para desserializar o resultado. Além das entradas, o framework também passa dados adicionais para o Amazon SWF; por exemplo, detalhes de exceções. O serializador de fluxo de trabalho também será usado para serializar esses dados.

Você também pode fornecer uma instância do `DataConverter` se não desejar que a estrutura a crie automaticamente. Os clientes gerados têm as sobrecargas do construtor que usam um `DataConverter`.

Se você não especificar um tipo de `DataConverter` e não passar um objeto `DataConverter`, o `JsonDataConverter` será usado por padrão.

Passagem de dados para métodos assíncronos

Tópicos

O uso da `Promise<T>` foi explicado nas seções anteriores. Alguns casos de uso avançados da `Promise<T>` são discutidos aqui.

Passagem de coleções e mapas para métodos assíncronos

A estrutura oferece suporte à passagem de matrizes, coleções e mapas como tipos `Promise` para métodos assíncronos. Por exemplo, um método assíncrono pode usar `Promise<ArrayList<String>>` como um argumento, conforme mostrado na lista a seguir.

```
@Asynchronous
```

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

Semanticamente, isso se comporta como qualquer outro parâmetro de tipo `Promise`, e o método assíncrono esperará até que a coleção se torne disponível antes de executar. Se os membros de uma coleção forem objetos `Promise`, você poderá fazer com que a estrutura espere que todos os membros se tornem prontos, conforme mostrado no trecho de código a seguir. Isso fará com que o método assíncrono espere que cada membro da coleção se torne disponível.

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

Observe que a anotação `@Wait` deve ser usada no parâmetro para indicar que contém objetos `Promise`.

Observe também que a atividade `printActivity` usa o argumento `String`, mas o método correspondente no cliente gerado usa uma Promise<String>. Estamos chamando o método no cliente e não estamos invocando o método da atividade diretamente.

definíveis <T>

`Settable<T>` é um tipo derivado de `Promise<T>` que fornece um método de definição que permite definir manualmente o valor de uma `Promise`. Por exemplo, o fluxo de trabalho a seguir espera que um sinal seja recebido aguardando em um `Settable<?>`, que é definido no método de sinal:

```
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;
   }
}
```

Uma `Settable<?>` também pode ser encadeada em outra promessa de cada vez. Você pode usar `AndPromise` e `OrPromise` para agrupar promessas. Você pode desencadear uma `Settable` encadeada chamando o método `unchain()` nela. Quando encadeada, a `Settable<?>` se torna pronta automaticamente quando a promessa encadeada se torna pronta. O encadeamento é especialmente útil quando você deseja usar uma promessa retornada do escopo de um `doTry()` em outras partes de seu programa. Como `TryCatchFinally` é usada como uma classe aninhada, você não pode declarar a `Promise<>` no escopo do pai e configurá-la. `doTry()` Isso acontece porque o Java exige que as variáveis sejam declaradas no escopo pai e usadas em classes aninhadas para serem marcadas finais. Por exemplo:

```
@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;
}
```

Uma `Settable` pode ser encadeada para uma promessa de cada vez. Você pode desencadear uma `Settable` encadeada chamando o método `unchain()` nela.

@NoWait

Quando você passa uma `Promise` para um método assíncrono, por padrão, a estrutura aguardará que a(s) `Promise`(s) se torne(m) pronta(s) antes de executar o método (exceto para tipos de coleção). Você pode substituir esse comportamento usando a anotação `@NoWait` em parâmetros na declaração do método assíncrono. Isso será útil se você estiver passando `Settable<T>` que será definido pelo próprio método assíncrono.

Promessa <Anulação>

As dependências de métodos assíncronos são implementadas passando a `Promise` retornada por um método como um argumento para outro. Contudo, pode haver casos em que você deseje retornar `void` de um método, mas ainda deseje que outros métodos assíncronos sejam executados após a conclusão. Nesses casos, você pode usar `Promise<Void>` como o tipo de retorno do método. A classe `Promise` fornece um método `Void` estático que pode ser usado para criar um objeto de `Promise<Void>`. Essa `Promise` se tornará pronta quando o método assíncrono concluir a execução. Você pode passar essa `Promise` para outro método assíncrono como qualquer outro objeto de `Promise`. Se você estiver usando `Settable<Void>`, chame o método set nele com null para torná-lo pronto.

AndPromise and OrPromise

`AndPromise` e `OrPromise` permitem que você agrupe vários objetos `Promise<>` em uma única promessa lógica. Uma `AndPromise` torna-se pronta quando todas as promessas usadas para construí-la se tornarem prontas. Uma `OrPromise` torna-se pronta quando qualquer promessa na coleção de promessas usadas para construí-la se tornarem prontas. Você pode chamar `getValues()` em `AndPromise` e `OrPromise` para recuperar a lista de valores das promessas constituintes.

Injeção de capacidade de teste e dependência

Tópicos

A estrutura é projetada para ser amigável para inversão de controle (IoC). As implementações de atividades e de fluxo de trabalho assim como os operadores fornecidos pela estrutura e os objetos de contexto podem ser configurados e instanciados usando contêineres, como o Spring. Pronta para uso, a estrutura fornece integração com o Spring Framework. Além disso, a integração com JUnit foi fornecida para implementações de atividades e fluxos de trabalho de testes unitários.

Integração com o Spring

O pacote de com.amazonaws.services.simpleworkflow.flow.spring contém classes que facilitam o uso do Spring Framework em seus aplicativos. Esses incluem operadores de atividades e de fluxo de trabalho com reconhecimento de um escopo personalizado do Spring: `WorkflowScope`, `SpringWorkflowWorker` e `SpringActivityWorker`. Essas classes permitem configurar suas implementações de fluxo de trabalho e de atividade bem como os operadores totalmente por meio do Spring.

WorkflowScope

`WorkflowScope` é uma implementação de Escopo do Spring fornecida pela estrutura. O escopo permite criar objetos no contêiner do Spring cujo tempo de vida tem o escopo de uma tarefa de decisão. Os beans nesse escopo são instanciados sempre que uma nova tarefa de decisão é recebida pelo operador. Você deve usar esse escopo para beans de implementação de fluxo de trabalho e todos os outros beans dos quais ele depende. Os escopos singleton e de protótipo fornecidos pelo Spring não devem ser usados para beans de implementação de fluxo de trabalho porque a estrutura exige que um novo bean seja criado para cada tarefa de decisão. Não fazer isso resultará em comportamento inesperado.

O exemplo a seguir mostra um trecho de código de configuração do Spring que registra o `WorkflowScope` e, em seguida, usa-o para configurar um bean de implementação de fluxo de trabalho e um bean de cliente de atividade.

```
<!-- 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>
```

A linha de configuração: `<aop:scoped-proxy proxy-target-class="false" />`, usada na configuração do bean `workflowImpl`, é necessária porque o `WorkflowScope` não oferece suporte a proxy usando CGLIB. Você deve usar essa configuração para qualquer bean no `WorkflowScope` que esteja conectado a outro bean em outro escopo. Nesse caso, o bean `workflowImpl` precisa ser conectado a um bean de operador de fluxo de trabalho em escopo singleton (consulte o exemplo completo a seguir).

Saiba mais sobre o uso de escopos personalizados na documentação do Spring Framework.

Operadores com reconhecimento do Spring

Ao usar o Spring, você deve usar as classes de operador com reconhecimento do Spring fornecidas pela estrutura: `SpringWorkflowWorker` e `SpringActivityWorker`. Esses operadores podem ser injetados em seu aplicativo usando o Spring, conforme mostrado no exemplo a seguir. Os operadores com reconhecimento do Spring implementam a interface `SmartLifecycle` do Spring e, por padrão, começam a pesquisar tarefas automaticamente quando o contexto do Spring é inicializado. Você pode desativar essa funcionalidade definindo a propriedade `disableAutoStartup` do operador como `true`.

O exemplo a seguir mostra como configurar um agente de decisão. O exemplo usa as interfaces `MyActivities` e `MyWorkflow` (não mostradas aqui) e as implementações correspondentes, `MyActivitiesImpl` e `MyWorkflowImpl`. As interfaces e as implementações de cliente geradas são `MyWorkflowClient`/`MyWorkflowClientImpl` e `MyActivitiesClient`/`MyActivitiesClientImpl` (também não mostradas aqui).

O cliente de atividades é injetado na implementação do fluxo de trabalho usando o recurso de conexão automática do Spring:

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

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

A configuração do agente de decisão do Spring é a seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/
spring-aop-2.5.xsd
   http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

```
      </property>
   </bean>
   <context:annotation-config/>

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

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

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

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

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

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

```
      </property>
   </bean>
</beans>
```

Como o `SpringWorkflowWorker` está totalmente configurado no Spring e inicia automaticamente a pesquisa quando o contexto do Spring é inicializado, o processo de hospedagem para o decisor é simples:

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

Da mesma forma, o operador de atividades pode ser configurado da seguinte maneira:

```
<?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>
```

O processo de host do operador de atividades é semelhante ao do agente de decisão:

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

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

Injeção do contexto de decisão

Se a implementação do fluxo de trabalho depender dos objetos do contexto, você também poderá injetá-los facilmente por meio do Spring. A estrutura registra automaticamente os beans relacionados ao contexto no contêiner do Spring. Por exemplo, no trecho de código a seguir, os vários objetos do contexto foram conectados automaticamente. Não é necessária nenhuma outra configuração do Spring para os objetos de contexto.

```
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();
   }
}
```

Para configurar os objetos de contexto na implementação do fluxo de trabalho por meio da configuração XML do Spring, use os nomes de beans declarados na classe `WorkflowScopeBeanNames` no pacote com.amazonaws.services.simpleworkflow.flow.spring. Por exemplo:

```
<!-- 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>
```

Como alternativa, você pode injetar um `DecisionContextProvider` no bean de implementação de fluxo de trabalho e usá-lo para criar o contexto. Isso pode ser útil se você desejar fornecer implementações personalizadas de provedor e do contexto.

Injeção de recursos em atividades

Você pode instanciar e configurar implementações de atividades usando um contêiner de inversão de controle (IoC) e injetar recursos facilmente, como conexões de banco de dados, declarando-os como propriedades da classe de implementação da atividade. Normalmente, esses recursos terão escopos singletons. Observe que as implementações de atividades são chamadas por operadores de atividades em vários threads. Portanto, o acesso aos recursos compartilhados deve ser sincronizado.

JUnit Integração

A estrutura fornece JUnit extensões e implementações de teste dos objetos de contexto, como um relógio de teste, que você pode usar para escrever e executar testes de JUnit unidade. Com essas extensões, você pode testar a implementação do fluxo de trabalho localmente em linha.

Criação de um teste de unidade simples

Para escrever testes para seu fluxo de trabalho, use a classe `WorkflowTest` no pacote com.amazonaws.services.simpleworkflow.flow.junit. Essa classe é uma JUnit `MethodRule` implementação específica da estrutura e executa seu código de fluxo de trabalho localmente, chamando atividades em linha em vez de usar o Amazon SWF. Isso fornece a flexibilidade de executar testes quantas vezes forem desejadas sem incorrer em encargos.

Para usar essa classe, simplesmente declare um campo do tipo `WorkflowTest` e anote-o com a anotação `@Rule`. Antes de executar seus testes, crie um novo objeto `WorkflowTest` e adicione suas implementações de atividades e de fluxo de trabalho a ele. Em seguida, você pode usar a fábrica de cliente de fluxo de trabalho gerada para criar e iniciar uma execução do fluxo de trabalho.

A estrutura também fornece um JUnit executor personalizado,`FlowBlockJUnit4ClassRunner`, que você deve usar para seus testes de fluxo de trabalho. Por exemplo:

```
@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);
    }
}
```

Você também pode especificar uma lista de tarefas separada para cada implementação de atividade adicionada ao `WorkflowTest`. Por exemplo, se tiver uma implementação de fluxo de trabalho que

programa atividades em listas de tarefas específicas ao host, você poderá registrar a atividade na lista de tarefas de cada host:

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

Observe que o código em `@Test` é assíncrono. Portanto, você deve usar o cliente de fluxo de trabalho assíncrono para iniciar uma execução. Para verificar os resultados do teste, uma classe auxiliar `AsyncAssert` também é fornecida. Essa classe permite que você aguarde que as promessas estejam prontas antes de verificar os resultados. Neste exemplo, esperamos que o resultado da execução do fluxo de trabalho esteja pronto antes de verificar a saída do teste.

Se você estiver usando o Spring, a classe `SpringWorkflowTest` pode ser usada em vez da classe `WorkflowTest`. O `SpringWorkflowTest` oferece propriedades que você pode usar para configurar facilmente por meio de implementações de atividade e de fluxo de configuração do Spring. Assim como os operadores com reconhecimento do Spring, você deve usar o `WorkflowScope` para configurar beans de implementação de fluxo de trabalho. Isso garante que um novo bean de implementação de fluxo de trabalho seja criado para cada tarefa de decisão. Certifique-se de configurar esses beans com a proxy-target-class configuração scoped-proxy definida como. `false` Consulte a seção Integração com o Spring para obter mais detalhes. A configuração de exemplo do Spring mostrada na seção Integração com o Spring pode ser alterada para testar o fluxo de trabalho usando o `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>
```

Implementações de atividades fictícias

Você pode usar as implementações de atividades reais durante o teste, mas se desejar executar testes de unidade apenas da lógica do fluxo de trabalho, você deve usar atividades fictícias. Isso pode ser feito fornecendo uma implementação fictícia da interface de atividades para a classe `WorkflowTest`. Por exemplo:

```
@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);
    }
}
```

Como alternativa, você pode fornecer uma implementação fictícia do cliente de atividades e injetá-la em sua implementação de fluxo de trabalho.

Teste de objetos de contexto

Se a implementação do seu fluxo de trabalho depender dos objetos de contexto do framework (por exemplo, o `DecisionContext`), você não precisará fazer nada de especial para testar esses fluxos de trabalho. Quando um teste é executado por meio do `WorkflowTest`, ele injeta automaticamente objetos de contexto de teste. Quando a implementação do seu fluxo de trabalho acessar os objetos de contexto (por exemplo, usando `DecisionContextProviderImpl`), ela obterá a implementação

de teste. Você pode manipular esses objetos de contexto de teste no código de teste (método `@Test`) para criar casos de teste interessantes. Por exemplo, se o fluxo de trabalho criar um temporizador, você poderá fazer com que o temporizador seja acionado chamando o método `clockAdvanceSeconds` na classe `WorkflowTest` para adiantar a hora do relógio. Você também pode acelerar o relógio para fazer com que os temporizadores sejam acionados antes do que seriam normalmente usando a propriedade `ClockAccelerationCoefficient` no `WorkflowTest`. Por exemplo, se o fluxo de trabalho criar um temporizador para uma hora, você poderá definir o `ClockAccelerationCoefficient` como 60 para fazer com que o temporizador acione em um minuto. Por padrão, o `ClockAccelerationCoefficient` é definido como 1.

Para obter mais detalhes sobre os pacotes com.amazonaws.services.simpleworkflow.flow.test and com.amazonaws.services.simpleworkflow.flow.junit, consulte a documentação do AWS SDK para Java .

Tratamento de erros

Tópicos

O constructo `try/catch/finally` no Java simplifica o tratamento de erros e é usado de forma ubíqua. Ele permite associar manipuladores de erros a um bloco de código. Internamente, isso funciona inserindo metadados adicionais sobre os manipuladores de erro na pilha de chamada. Quando uma exceção é gerada, o tempo de execução examina a pilha de chamada para localizar um manipulador de erros associado e o invocar, e se nenhum manipulador de erros apropriado for localizado, ele propagará a exceção para cima na cadeia de chamada.

Isso funciona bem para código síncrono, mas a manipulação de erros em assíncronos e em programas distribuídos impõe desafios adicionais. Como uma chamada assíncrona retorna imediatamente, o chamador não está na pilha de chamadas quando o código assíncrono é executado. Isso significa que as exceções não tratadas no código assíncrono não podem ser tratadas pelo chamador na forma habitual. Normalmente, as exceções originadas em código assíncrono são tratadas passando o estado de erro para um retorno de chamada que é passado para o método assíncrono. Como alternativa, se um `Future<?>` estiver sendo usado, ele relatará um erro quando você tentar acessá-lo. Isso não é o ideal porque o código que recebe a exceção

(o retorno de chamada ou o código que usa o `Future<?>`) não tem o contexto da chamada original e não pode controlar adequadamente a exceção. Além disso, em um sistema assíncrono distribuído, com componentes que são executados simultaneamente, mais de um erro pode ocorrer simultaneamente. Esses erros podem ser de tipos e de severidades diferentes e precisam ser tratados de forma adequada.

A limpeza de um recurso após uma chamada assíncrona também é difícil. Ao contrário do código síncrono, você não pode usá-lo try/catch/finally no código de chamada para limpar recursos porque o trabalho iniciado no bloco try ainda pode estar em andamento quando o bloco final for executado.

A estrutura fornece um mecanismo que torna o tratamento de erros em código assíncrono distribuído semelhante e quase tão simples quanto o do 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();
    }
  };
}
```

A classe `TryCatchFinally` e suas variantes, `TryFinally` e `TryCatch`, funcionam de forma semelhante ao `try`/`catch`/`finally` do Java. Usando-a, você pode associar manipuladores de exceção a blocos de código de fluxo de trabalho que podem executar como tarefas assíncronas e remotas. O método `doTry()` é logicamente equivalente ao bloco `try`. A estrutura executa automaticamente o código em `doTry()`. Uma lista de objetos `Promise` pode ser passada para o construtor de `TryCatchFinally`. O método `doTry` será executado quando todos os objetos `Promise` passados para o construtor estiverem prontos. Se uma exceção for gerada pelo código que foi invocado de forma assíncrona em `doTry()`, qualquer trabalho pendente em `doTry()` será cancelado e `doCatch()` será chamado para tratar a exceção. Por exemplo, na lista acima, se `downloadImage` lançar uma exceção, `createThumbnail` e `uploadImage` serão cancelados. Finalmente, `doFinally()` é chamado quando todo o trabalho assíncrono for feito (concluído, com falha ou cancelado). Ele pode ser usado para limpeza de recursos. Você também pode aninhar essas classes para atender às suas necessidades.

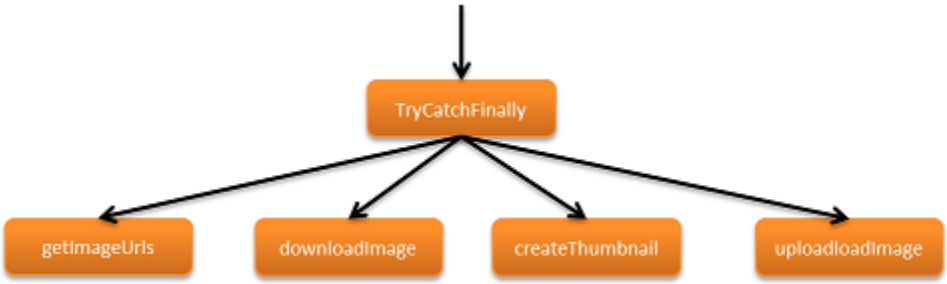
Quando uma exceção é relatada em `doCatch()`, a estrutura fornece uma pilha de chamada lógica completa que inclui chamadas assíncronas e remotas. Isso pode ser útil ao depurar, especialmente se você tiver métodos assíncronos que chamam outros métodos assíncronos. Por exemplo, uma exceção de downloadImage produzirá uma exceção como esta:

```
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 Semântica

A execução de um programa AWS Flow Framework para Java pode ser visualizada como uma árvore de ramificações em execução simultânea. Uma chamada para um método assíncrono, para uma atividade e para o próprio `TryCatchFinally` cria uma nova ramificação nessa árvore de

execução. Por exemplo, o fluxo de trabalho de processamento de imagem pode ser visualizado como na árvore mostrada na figura a seguir.



Um erro em uma ramificação de execução causa o desenrolamento da ramificação, assim como uma exceção causa o desenrolamento da pilha de chamada em um programa Java. O desenrolamento continua movendo a movimentação da ramificação de execução para cima até que o erro seja tratado ou a raiz da árvore seja acessada, nesse caso a execução do fluxo de trabalho é encerrada.

A estrutura relata erros que acontecem durante o processamento de tarefas como exceções. Ela associa os manipuladores de exceção (métodos `doCatch()`) definidos em `TryCatchFinally` com todas as tarefas criadas pelo código no `doTry()` correspondente. Se uma tarefa falhar, por exemplo, devido a um tempo limite ou a uma exceção não tratada, a exceção apropriada será levantada e o `doCatch()` correspondente será invocado para tratá-la. Para conseguir isso, a estrutura trabalha em conjunto com o Amazon SWF para propagar erros remotos e ressuscitá-los como exceções no contexto do chamador.

Cancelamento

Quando ocorre uma exceção em código síncrono, o controle salta diretamente para o bloco `catch`, ignorando qualquer código restante no bloco `try`. Por exemplo:

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

Neste código, se `b()` gerar uma exceção, `c()` nunca será invocado. Compare isso com um fluxo de trabalho:

```
new TryCatch() {

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

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

Nesse caso, todas as chamadas para `activityA`, `activityB` e `activityC` retornam com êxito e resultam na criação de três tarefas que serão executadas assincronamente. Digamos que posteriormente a tarefa para a `activityB` resulte em um erro. Esse erro é registrado no histórico pelo Amazon SWF. Para tratar esse erro, a estrutura primeiro tentará cancelar todas as outras tarefas originadas no escopo do mesmo `doTry()`, nesse caso, `activityA` e `activityC`. Quando todas essas tarefas forem concluídas (canceladas, com falha ou concluídas com êxito), o método `doCatch()` apropriado será invocado para tratar o erro.

Ao contrário do exemplo síncrono, onde `c()` nunca foi executado, `activityC` foi invocada e uma tarefa foi programada para execução. Portanto, a estrutura fará uma tentativa de cancelá-la, mas não há garantia de que ela será cancelada. O cancelamento não pode ser garantido porque a atividade pode já ter sido concluída, pode ignorar a solicitação de cancelamento ou pode falhar devido a um erro. Contudo, a estrutura fornece a garantia de que `doCatch()` é chamado somente depois que todas as tarefas iniciadas no `doTry()` correspondente foram concluídas. Também garante que `doFinally()` seja chamado somente depois que todas as tarefas iniciadas em `doCatch()` e `doTry()` foram concluídas. Se, por exemplo, as atividades no exemplo acima dependerem umas das outras, digamos que `activityB` dependa de `activityA` e `activityC` de `activityB`, o cancelamento de `activityC` será imediato porque não está programado no Amazon SWF até que `activityB` seja concluído:

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

Pulsação de atividade

O mecanismo de cancelamento cooperativo do AWS Flow Framework for Java permite que as tarefas de atividades em voo sejam canceladas normalmente. Quando o cancelamento é acionado, as tarefas que foram bloqueadas ou estão aguardando para serem atribuídas a um operador são canceladas automaticamente. Se, no entanto, uma tarefa já estiver atribuída a um operador, a estrutura solicitará que a atividade seja cancelada. A implementação da atividade deve tratar explicitamente essas solicitações de cancelamento. Isso é feito relatando a pulsação da atividade.

Relatar a pulsação permite que a implementação da atividade relate o progresso de uma tarefa de atividade em andamento, o que é útil para o monitoramento e permite que a atividade verifique se há solicitações de cancelamento. O método `recordActivityHeartbeat` gerará uma `CancellationException` se um cancelamento tiver sido solicitado. A implementação da atividade pode capturar essa exceção e responder à solicitação de cancelamento, ou pode ignorar a solicitação engolindo a exceção. Para honrar a solicitação de cancelamento, a atividade deve executar a limpeza desejada, se houver, e, em seguida gerar a `CancellationException` novamente. Quando essa exceção é gerada em uma implementação de atividade, a estrutura registra que a tarefa de atividade foi concluída em estado cancelado.

O exemplo a seguir mostra uma atividade que faz download e processa imagens. Se houver pulsação depois do processamento de cada imagem e se o cancelamento for solicitado, ele limpará e gerará a exceção novamente para reconhecer o cancelamento.

```
@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;
        }
    }
}
```

Relatar a pulsação da atividade não é necessário, mas é recomendável se a atividade for de longa execução ou estiver executando operações caras que você deseja cancelar em condições de erros. Você deve chamar `heartbeatActivityTask` periodicamente na implementação da atividade.

Se o tempo limite da atividade for esgotado, `ActivityTaskTimedOutException` será gerada, e `getDetails` no objeto de exceção retornará os dados passados para a última chamada bem-sucedida para `heartbeatActivityTask` para a tarefa de atividade correspondente. A implementação do fluxo de trabalho pode usar essas informações para determinar quanto de progresso foi feito antes do tempo limite da tarefa de atividade ter esgotado.

> **Note**
>
> Não é uma boa prática fazer heartbeat com muita frequência, pois o Amazon SWF pode aplicar o controle de utilização às solicitações de heartbeat. Consulte o Guia do desenvolvedor do Amazon Simple Workflow Service para obter informações sobre os limites impostos pelo Amazon SWF.

Cancelamento explícito de uma tarefa

Além das condições de erro, há outros casos em você pode cancelar explicitamente uma tarefa. Por exemplo, uma atividade para processar pagamentos usando um cartão de crédito pode precisar ser cancelada se o usuário cancelar o pedido. A estrutura permite que você cancele explicitamente tarefas criadas no escopo de um `TryCatchFinally`. No exemplo a seguir, a tarefa de pagamento será cancelada se um sinal for recebido enquanto o pagamento estava sendo processado.

```
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);
        }
    }
}
```

Recebimento de notificação de tarefas canceladas

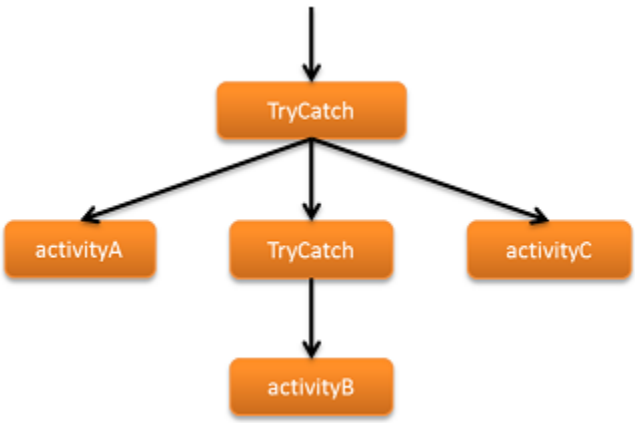
Quando uma tarefa é concluída em estado cancelado, a estrutura informa a lógica do fluxo de trabalho gerando uma `CancellationException`. Quando uma atividade é concluída

em estado cancelado, é feito um registro no histórico, e a estrutura chama o `doCatch()` apropriado com uma `CancellationException`. Conforme mostrado no exemplo anterior, quando a tarefa de processamento de pagamento é cancelada, o fluxo de trabalho recebe uma `CancellationException`

Uma `CancellationException` não tratada é propagada para cima na ramificação de execução como qualquer outra exceção. No entanto, o método `doCatch()` receberá a `CancellationException` apenas se não houver outra exceção no escopo. Outras exceções têm prioridade mais alta que um cancelamento.

Aninhado TryCatchFinally

Você pode aninhar `TryCatchFinally`s para atender às suas necessidades. Como cada uma `TryCatchFinally` cria uma nova ramificação na árvore de execução, você pode criar escopos aninhados. As exceções no escopo pai provocarão tentativas de cancelamento de todas as tarefas iniciadas pelos `TryCatchFinally`s aninhados dentro dele. No entanto, as exceções em `TryCatchFinally` aninhado não são propagadas automaticamente para o pai. Para propagar uma exceção de um `TryCatchFinally` aninhado para o `TryCatchFinally` que o contém, você deve gerar novamente a exceção em `doCatch()`. Ou seja, apenas as exceções não tratadas são movidas para acima, assim como o `try/catch` do Java. Se você cancelar um `TryCatchFinally` aninhado chamando o método de cancelamento, o `TryCatchFinally` aninhado será cancelado, mas o `TryCatchFinally` que o contém não será cancelado automaticamente.




```
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);
    }
};
```

Tentar novamente atividades com falha

As atividades falham às vezes por motivos efêmeros, como uma perda temporária de conectividade. Em outro momento, a atividade pode funcionar, assim a forma apropriada de lidar com falhas de atividade geralmente é repetir a atividade, talvez algumas vezes.

Existe uma variedade de estratégias para repetir atividades. A melhor depende dos detalhes do fluxo de trabalho. As estratégias são divididas em três categorias básicas:

- A retry-until-success estratégia simplesmente continua repetindo a atividade até que ela seja concluída.
- A estratégia de repetição exponencial aumenta o intervalo de tempo entre as tentativas exponencialmente até que a atividade seja concluída ou o processo alcançar um ponto de parada especificado, como um número máximo de tentativas.
- A estratégia de repetição personalizada decide se ou como repetir a atividade após cada tentativa falha.

As seções a seguir descrevem como implementar essas estratégias. Todos os operadores de fluxo de trabalho de exemplo utilizam uma única atividade, `unreliableActivity`, que realiza de forma aleatória um dos seguintes:

- Conclui imediatamente
- Falha intencionalmente ao exceder o valor do tempo limite
- Falha intencionalmente lançando uma `IllegalStateException`

Retry-Until-Success Estratégia

A estratégia mais simples de repetição é continuar repetindo a atividade até que eventualmente seja concluída. O padrão básico é:

1. Implementar uma classe aninhada `TryCatch` ou `TryCatchFinally` no método do ponto de entrada do fluxo de trabalho.
2. Executar a atividade em `doTry`
3. Se a atividade falhar, a estrutura chama `doCatch`, que executa o método do ponto de entrada novamente.
4. Repetir as Etapas 2 e 3 até que a atividade seja concluída com êxito.

O fluxo de trabalho a seguir implementa a retry-until-success estratégia. A interface do fluxo de trabalho é implementada no `RetryActivityRecipeWorkflow` e possui um método, `runUnreliableActivityTillSuccess`, que é o ponto de entrada do fluxo de trabalho. O operador do fluxo de trabalho é implementado em `RetryActivityRecipeWorkflowImpl`, da seguinte maneira:

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

O fluxo de trabalho funciona da seguinte forma:

1. `runUnreliableActivityTillSuccess` cria um objeto `Settable<Boolean>` chamado `retryActivity` que é usado para indicar se a atividade falhou e deve ser executada novamente. `Settable<T>` é derivado do `Promise<T>` e funciona da mesma forma, mas você define o valor de um objeto `Settable<T>` manualmente.
2. `runUnreliableActivityTillSuccess` implementa uma classe aninhada `TryCatch` anônima para processar todas as exceções lançadas pela atividade `unreliableActivity`. Para obter mais discussões sobre como gerenciar as exceções lançadas pelo código assíncrono, consulte Tratamento de erros.
3. `doTry` executa a atividade `unreliableActivity`, que retorna um objeto `Promise<Void>` chamado `activityRanSuccessfully`.
4. `doTry` chama o método assíncrono `setRetryActivityToFalse`, que possui dois parâmetros:

- `activityRanSuccessfully` recebe o objeto `Promise<Void>` retornado pela atividade `unreliableActivity`.
- `retryActivity` recebe o objeto `retryActivity`.

Se a `unreliableActivity` for concluída, `activityRanSuccessfully` se torna pronto e `setRetryActivityToFalse` define `retryActivity` como falso. Caso contrário, `activityRanSuccessfully` nunca se torna pronto e `setRetryActivityToFalse` não é executado.

5. Se a `unreliableActivity` lança uma exceção, a estrutura chama `doCatch` e envia-lhe o objeto de exceção. `doCatch` define `retryActivity` como verdadeiro.
6. `runUnreliableActivityTillSuccess` chama o método assíncrono `restartRunUnreliableActivityTillSuccess` e envia-lhe o objeto `retryActivity`. Como `retryActivity` é do tipo `Promise<T>`, `restartRunUnreliableActivityTillSuccess` adia a execução até que a `retryActivity` esteja pronta, o que ocorre após `TryCatch` concluir.
7. Quando a `retryActivity` estiver pronta, `restartRunUnreliableActivityTillSuccess` extrai o valor.
   - Se o valor for `false`, a nova tentativa foi bem-sucedida. `restartRunUnreliableActivityTillSuccess` não faz nada e a sequência de repetição é encerrada.
   - Se o valor for verdadeiro, a nova tentativa falhou. `restartRunUnreliableActivityTillSuccess` chama `runUnreliableActivityTillSuccess` para executar a atividade novamente.
8. As Etapas 1 a 7 são repetias até que a `unreliableActivity` seja concluída.

> **Note**
>
> `doCatch` não lida com a exceção, ele simplesmente define o objeto `retryActivity` como verdadeiro para indicar que a atividade falhou. A repetição é gerenciada pelo método assíncrono `restartRunUnreliableActivityTillSuccess`, que adia a execução até que `TryCatch` seja concluído. O motivo desta abordagem é que, se você repetir uma atividade no `doCatch`, não é possível cancelá-la. Repetir a atividade em `restartRunUnreliableActivityTillSuccess` permite executar atividades canceláveis.

Estratégia de repetição exponencial

Com a estratégia de repetição exponencial, a estrutura executa uma atividade falha novamente após um período especificado, N segundos. Se essa tentativa falhar, a estrutura executa a atividade novamente após 2N segundos e, em seguida, 4N segundos e assim por diante. Como o tempo de espera pode se tornar grande, normalmente você interrompe as tentativas de repetição em algum momento em vez de continuar indefinidamente.

A estrutura oferece três maneiras para implementar uma estratégia de repetição exponencial:

- A anotação `@ExponentialRetry` é a abordagem mais simples, mas você deve definir as opções de configuração da repetição no momento da compilação.
- A classe `RetryDecorator` permite definir a configuração de repetição durante o tempo de execução e alterá-la conforme necessário.
- A classe `AsyncRetryingExecutor` permite definir a configuração de repetição durante o tempo de execução e alterá-la conforme necessário. Além disso, a estrutura chama um método `AsyncRunnable.run` implementado pelo usuário para executar cada tentativa de repetição.

Todas as abordagens oferecem suporte para as seguintes opções de configuração, onde os valores de tempo estão em segundos:

- O tempo de espera da repetição inicial.
- O coeficiente de recuo, que é usado para computar os intervalos de repetição, da seguinte forma:

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

  O valor padrão é 2.0.
- O número máximo de tentativas de repetição. O valor padrão é ilimitado.
- O intervalo máximo de repetição. O valor padrão é ilimitado.
- O tempo de expiração. As tentativas de repetição param quando a duração total do processo excede esse valor. O valor padrão é ilimitado.
- As exceções que acionarão o processo de repetição. Por padrão, toda exceção aciona o processo de repetição.
- As exceções que não acionarão um processo de repetição. Por padrão, nenhuma exceção está excluída.

As seções a seguir descrevem as diversas formas para implementar uma estratégia de repetição exponencial.

Tentativa exponencial com @ ExponentialRetry

A forma mais simples de implementar uma estratégia de repetição exponencial para uma atividade é aplicar uma anotação `@ExponentialRetry` à atividade na definição da interface. Se a atividade falhar, a estrutura gerencia o processo de repetição automaticamente, com base nos valores da opção especificada. O padrão básico é:

1. Aplique `@ExponentialRetry` às atividades apropriadas e especifique a configuração de repetição.
2. Se uma atividade anotada falhar, a estrutura repete automaticamente a atividade de acordo com a configuração especificada pelos argumentos da anotação.

O operador de fluxo de trabalho `ExponentialRetryAnnotationWorkflow` implementa a estratégia de repetição exponencial usando uma anotação `@ExponentialRetry`. Ele usa uma atividade `unreliableActivity`, cuja definição de interface é implementada nas `ExponentialRetryAnnotationActivities`, da seguinte forma:

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

As opções `@ExponentialRetry` especificam a seguinte estratégia:

- Repetir somente se a atividade lançar uma `IllegalStateException`.
- Usar um tempo de espera inicial de 5 segundos.
- Não mais que cinco tentativas de repetição.

A interface do fluxo de trabalho é implementada no `RetryWorkflow` e possui um método, `process`, que é o ponto de entrada do fluxo de trabalho. O operador do fluxo de trabalho é implementado em `ExponentialRetryAnnotationWorkflowImpl`, da seguinte maneira:

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

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

O fluxo de trabalho funciona da seguinte forma:

1. `process` executa o método síncrono `handleUnreliableActivity`.
2. `handleUnreliableActivity` executa a atividade `unreliableActivity`.

Se a atividade falhar ao lançar uma `IllegalStateException`, a estrutura executa automaticamente a estratégia de repetição especificada nas `ExponentialRetryAnnotationActivities`.

Repetição exponencial com a classe RetryDecorator

O uso do `@ExponentialRetry` é simples. No entanto, a configuração é estática e definida durante a compilação, portanto a estrutura usa a mesma estratégia de repetição sempre que a atividade falhar. É possível implementar uma estratégia de repetição exponencial mais flexível usando a classe `RetryDecorator`, que permite especificar a configuração durante o tempo de execução e alterá-la conforme necessário. O padrão básico é:

1. Crie e configure um objeto `ExponentialRetryPolicy` que especifica a configuração de repetição.
2. Crie um objeto `RetryDecorator` e envie o objeto `ExponentialRetryPolicy` da Etapa 1 para o construtor.
3. Aplique o objeto decorador à atividade enviando o nome de classe do cliente da atividade para o método de decoração do objeto `RetryDecorator`.
4. Execute a atividade.

Se a atividade falhar, a estrutura repete a atividade de acordo com a configuração do objeto `ExponentialRetryPolicy`. Altere a configuração de repetição conforme necessário modificando esse objeto.

> **Note**
>
> A anotação `@ExponentialRetry` e a classe `RetryDecorator` são mutuamente exclusivas. Não é possível usar `RetryDecorator` para substituir dinamicamente uma política de repetição especificada por uma anotação `@ExponentialRetry`.

A implementação de fluxo de trabalho a seguir mostra como usar a classe `RetryDecorator` para implementar uma estratégia de repetição exponencial. Ela usa uma atividade `unreliableActivity` que não possui uma anotação `@ExponentialRetry`. A interface do fluxo de trabalho é implementada no `RetryWorkflow` e possui um método, `process`, que é o ponto de entrada do fluxo de trabalho. O operador do fluxo de trabalho é implementado em `DecoratorRetryWorkflowImpl`, da seguinte maneira:

```
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();
  }
}
```

O fluxo de trabalho funciona da seguinte forma:

1. `process` cria e configura um objeto `ExponentialRetryPolicy` ao:
   - Enviar o intervalo inicial de repetição ao construtor.

- Chamar o método `withMaximumAttempts` do objeto para definir o número máximo de tentativas como cinco. `ExponentialRetryPolicy` expõe outros objetos `with` que você pode usar para especificar outras opções de configuração.

2. `process` cria um objeto `RetryDecorator` chamado `retryDecorator` e envia o objeto `ExponentialRetryPolicy` da Etapa 1 ao construtor.
3. `process` aplica o decorador à atividade chamando o método `retryDecorator.decorate` e enviando-lhe o nome de classe do cliente da atividade.
4. `handleUnreliableActivity` executa a atividade.

Se a atividade falhar, a estrutura repete-a de acordo com a configuração especificada na Etapa 1.

> **Note**
>
> Muitos dos métodos `with` da classe `ExponentialRetryPolicy` possuem um método `set` correspondente que pode ser chamado para modificar a opção de configuração correspondente a qualquer momento: `setBackoffCoefficient`, `setMaximumAttempts`, `setMaximumRetryIntervalSeconds` e `setMaximumRetryExpirationIntervalSeconds`.

Repetição exponencial com a classe AsyncRetryingExecutor

A classe `RetryDecorator` oferece mais flexibilidade ao configurar o processo de repetição que a `@ExponentialRetry`, mas a estrutura ainda executa as tentativas de repetição automaticamente, com base na configuração atual do objeto `ExponentialRetryPolicy`. Uma abordagem mais flexível é usar a classe `AsyncRetryingExecutor`. Além de permitir que você configure o processo de repetição durante o tempo de execução, a estrutura chama um método `AsyncRunnable.run` implementado pelo usuário para executar cada tentativa de repetição em vez de simplesmente executar a atividade.

O padrão básico é:

1. Crie e configure um objeto `ExponentialRetryPolicy` para especificar a configuração de repetição.
2. Crie um objeto `AsyncRetryingExecutor` e envie-lhe o objeto `ExponentialRetryPolicy` e uma instância do clock do fluxo de trabalho.

3. Implemente uma classe aninhada `TryCatch` ou `TryCatchFinally` anônima.
4. Implemente uma classe `AsyncRunnable` anônima e substitua o método `run` para implementar o código personalizado para execução da atividade.
5. Substitua `doTry` para chamar o método `execute` do objeto `AsyncRetryingExecutor` enviar-lhe a classe `AsyncRunnable` da Etapa 4. O objeto `AsyncRetryingExecutor` chama `AsyncRunnable.run` para executar a atividade.
6. Se a atividade falhar, o objeto `AsyncRetryingExecutor` chama o método `AsyncRunnable.run` novamente, de acordo com a política de repetição especificada na Etapa 1.

O fluxo de trabalho a seguir mostra como usar a classe `AsyncRetryingExecutor` para implementar uma estratégia de repetição exponencial. Ele usa a mesma atividade `unreliableActivity` que o fluxo de trabalho `DecoratorRetryWorkflow` discutido anteriormente. A interface do fluxo de trabalho é implementada no `RetryWorkflow` e possui um método, `process`, que é o ponto de entrada do fluxo de trabalho. O operador do fluxo de trabalho é implementado em `AsyncExecutorRetryWorkflowImpl`, da seguinte maneira:

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

O fluxo de trabalho funciona da seguinte forma:

1. `process` chama o método `handleUnreliableActivity` e envia-lhe as definições de configuração.
2. `handleUnreliableActivity` usa as definições de configuração da Etapa 1 para criar um objeto `ExponentialRetryPolicy`, `retryPolicy`.
3. `handleUnreliableActivity` cria um objeto `AsyncRetryExecutor`, `executor`, e envia o objeto `ExponentialRetryPolicy` da Etapa 2 e uma instância do clock do fluxo de trabalho ao construtor
4. `handleUnreliableActivity` implementa uma classe aninhada `TryCatch` anônima e substitui os métodos `doTry` e `doCatch` para executar as tentativas de repetição e gerenciar quaisquer exceções.
5. `doTry` cria uma classe `AsyncRunnable` anônima e substitui o método `run` para implementar o código personalizado para executar a `unreliableActivity`. Por simplicidade, `run` apenas executa a atividade, mas você pode implementar abordagens mais sofisticadas conforme for apropriado.
6. `doTry` chama `executor.execute` e envia-lhe o objeto `AsyncRunnable`. `execute` chama o método `AsyncRunnable` do objeto `run` para executar a atividade.
7. Se a atividade falhar, o executor chama `run` novamente, de acordo com a configuração do objeto `retryPolicy`.

Para obter mais discussões sobre como usar a classe `TryCatch` para gerenciar erros, consulte AWS Flow Framework para exceções de Java.

Estratégia de repetição personalizada

A abordagem mais flexível para repetir atividades com falha é uma estratégia personalizada, que chama recursivamente um método assíncrono que executa a tentativa de repetição, de forma muito parecida com a estratégia. retry-until-success No entanto, em vez de simplesmente executar a atividade novamente, implemente a lógica personalizada que decide se e como executar cada tentativa sucessiva de repetição. O padrão básico é:

1. Crie um objeto de status `Settable<T>`, que é usado para indicar se a atividade falhou.
2. Implemente uma classe aninhada `TryCatch` ou `TryCatchFinally`.
3. `doTry` executa a atividade.
4. Se a atividade falhar, `doCatch` define o objeto de status para indicar que a atividade falhou.
5. Chame um método assíncrono de gerenciamento de falhas e envie-lhe o objeto de status. O método adia a execução até que `TryCatch` ou `TryCatchFinally` conclua.
6. O método de gerenciamento de falhas decide se deve repetir a atividade e, se sim, quando.

O fluxo de trabalho a seguir mostra como implementar uma estratégia de repetição personalizada. Ele usa a mesma atividade `unreliableActivity` que os fluxos de trabalho `DecoratorRetryWorkflow` e `AsyncExecutorRetryWorkflow`. A interface do fluxo de trabalho é implementada no `RetryWorkflow` e possui um método, `process`, que é o ponto de entrada do fluxo de trabalho. O operador do fluxo de trabalho é implementado em `CustomLogicRetryWorkflowImpl`, da seguinte maneira:

```
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;
    }
}
```

O fluxo de trabalho funciona da seguinte forma:

1. `process` chama o método assíncrono `callActivityWithRetry`.
2. `callActivityWithRetry` cria um objeto `Settable<Throwable>` chamado failure que é usado para indicar se a atividade falhou. `Settable<T>` é derivado de `Promise<T>` e funciona da mesma forma, mas você define o valor de um objeto `Settable<T>` manualmente.
3. `callActivityWithRetry` implementa uma classe aninhada `TryCatchFinally` anônima para processar todas as exceções lançadas pela `unreliableActivity`. Para obter mais discussões sobre como gerenciar as exceções lançadas pelo código assíncrono, consulte AWS Flow Framework para exceções de Java.
4. `doTry` executa a `unreliableActivity`.
5. Se `unreliableActivity` lança uma exceção, a estrutura chama `doCatch` e envia-lhe o objeto de exceção. `doCatch` define `failure` para o objeto de exceção, o que indica que a atividade falhou e coloca o objeto em um estado pronto.
6. `doFinally` verifica se `failure` está pronto, que será verdadeiro somente se `failure` foi definido por `doCatch`.
   - Se `failure` estiver pronto, `doFinally` não faz nada.

- Se `failure` não estiver pronto, a atividade foi concluída e `doFinally` define failure como `null`.

7. `callActivityWithRetry` chama o método assíncrono `retryOnFailure` e envia-lhe o failure. Como failure é do tipo `Settable<T>`, `callActivityWithRetry` adia a execução até que failure esteja pronto, o que ocorre após `TryCatchFinally` concluir.
8. `retryOnFailure` obtém o valor de failure.
   - Se failure está definido como null, a tentativa de repetição foi bem-sucedida. `retryOnFailure` não faz nada, o que encerra o processo de repetição.
   - Se failure for definido como um objeto de exceção e `shouldRetry` retornar verdadeiro, `retryOnFailure` chama `callActivityWithRetry` para repetir a atividade.

     `shouldRetry` implementa a lógica personalizada para decidir se deve repetir uma atividade com falha. Por simplicidade, `shouldRetry` sempre retorna `true` e `retryOnFailure` executa a atividade imediatamente, mas você pode implementar uma lógica mais sofisticada conforme necessário.
9. As etapas 2 a 8 se repetem até que `unreliableActivity` seja concluído ou `shouldRetry` decida interromper o processo.

> **Note**
>
> `doCatch` não lida com o processo de repetição, ele simplesmente define failure para indicar que a atividade falhou. A processo de repetição é gerenciado pelo método assíncrono `retryOnFailure`, que adia a execução até que `TryCatch` seja concluído. O motivo desta abordagem é que, se você repetir uma atividade no `doCatch`, não é possível cancelá-la. Repetir a atividade em `retryOnFailure` permite executar atividades canceláveis.

Tarefas de daemon

O AWS Flow Framework for Java permite a marcação de determinadas tarefas como`daemon`. Isso permite criar tarefas que realizam algum trabalho de plano de fundo que deve ser cancelado quando todo o outro trabalho for concluído. Por exemplo, uma tarefa de monitoramento da integridade deve ser cancelada quando o restante do fluxo de trabalho for concluído. Você pode realizar isso configurando o sinalizador `daemon` em um método assíncrono ou em uma instância de

`TryCatchFinally`. No exemplo a seguir, o método assíncrono `monitorHealth()` está marcado como `daemon`.

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

  @Override
  public void startMyWF(int a, String b) {
    activitiesClient.doUsefulWorkActivity();
    monitorHealth();
  }

  @Asynchronous(daemon=true)
  void monitorHealth(Promise<?>... waitFor) {
    activitiesClient.monitoringActivity();
  }
}
```

No exemplo acima, quando `doUsefulWorkActivity` é concluído, `monitoringHealth` será cancelado automaticamente. Isso por sua vez cancelará toda a ramificação da execução enraizada nesse método assíncrono. A semântica do cancelamento é igual à do `TryCatchFinally`. De forma semelhante, você pode marcar um daemon `TryCatchFinally` passando um sinalizador booliano para o construtor.

```
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
            }
```

```
        };
    }
}
```

Uma tarefa daemon iniciada em um `TryCatchFinally` tem como escopo o contexto em que foi criada, ou seja, ela terá como escopo os métodos `doTry()` `doCatch()` ou `doFinally()`. Por exemplo, no exemplo a seguir, o método assíncrono startMonitoring é marcado como daemon e chamado em `doTry()`. A tarefa criada para ele será cancelada assim que as outras tarefas (`doUsefulWorkActivity` nesse caso) iniciadas em `doTry()` forem concluídas.

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

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

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

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

AWS Flow Framework para o comportamento do Java Replay

Este tópico discute exemplos de comportamento de reprodução usando os exemplos da seção O que é isso AWS Flow Framework para Java?. Os cenários síncrono e assíncrono são discutidos.

Exemplo 1: reprodução síncrona

Para obter um exemplo de como a repetição funciona em um fluxo de trabalho síncrono, modifique as implementações do HelloWorldWorkflowfluxo de trabalho e da atividade adicionando `println` chamadas em suas respectivas implementações, da seguinte forma:

```
public class GreeterWorkflowImpl implements GreeterWorkflow {
...
   public void greet() {
      System.out.println("greet executes");
      Promise<String> name = operations.getName();
      System.out.println("client.getName returns");
      Promise<String> greeting = operations.getGreeting(name);
      System.out.println("client.greeting returns");
      operations.say(greeting);
      System.out.println("client.say returns");
   }
}
**************
public class GreeterActivitiesImpl implements GreeterActivities {
   public String getName() {
      System.out.println("activity.getName completes");
      return "World";
   }

   public String getGreeting(String name) {
      System.out.println("activity.getGreeting completes");
      return "Hello " + name + "!";
   }

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

Para obter detalhes sobre o código, consulte HelloWorldWorkflow Aplicação. O seguinte é uma versão editada da saída, com comentários que indicam o início de cada episódio de reprodução.

```
//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
```

O processo de reprodução desse exemplo funciona da seguinte forma:

- O primeiro episódio programa a tarefa da atividade `getName`, que não tem dependências.
- O segundo episódio programa a tarefa da atividade `getGreeting`, que depende de `getName`.
- O terceiro episódio programa a tarefa da atividade `say`, que depende de `getGreeting`.
- O episódio final não programa tarefas adicionais e não localiza atividades não concluídas, o que termina a execução do fluxo de trabalho.

> **Note**
>
> Os três métodos de cliente de atividades são chamados uma vez para cada episódio. Contudo, apenas uma dessas chamadas resulta em uma tarefa de atividade, portanto cada tarefa é executada apenas uma vez.

Exemplo 2: reprodução assíncrona

Assim como no exemplo de reprodução síncrona, você pode modificar o HelloWorldWorkflowAsyncAplicação para ver como uma reprodução assíncrona funciona. Ele produz o seguinte resultado:

```
//Episode 1
greet executes
client.name returns
workflow.getGreeting returns
client.say returns

activity.getName completes
//Episode 2
greet executes
client.name returns
workflow.getGreeting returns
client.say returns
workflow.getGreeting completes

Hello World! //say completes
//Episode 3
greet executes
client.name returns
workflow.getGreeting returns
client.say returns
workflow.getGreeting completes
```

HelloWorldAsync usa três episódios de repetição porque há apenas duas atividades. A atividade `getGreeting` foi substituída pelo método de fluxo de trabalho assíncrono getGreeting, que não inicia um episódio de reprodução quando é concluído.

O primeiro episódio não chama `getGreeting`, porque depende da conclusão da atividade nome. Contudo, após a conclusão de getName, a reprodução chama getGreeting uma vez para cada episódio bem-sucedido.

Consulte também

- AWS Flow Framework Conceitos básicos: execução distribuída

Práticas recomendadas

Use essas melhores práticas para aproveitar ao máximo o AWS Flow Framework Java.

Tópicos

Como fazer alterações no código do agente de decisão: sinalizadores de versionamento e de recursos

Esta seção mostra como evitar alterações incompatíveis com versões anteriores em um agente de decisão usando dois métodos:

- O Versionamento fornece uma solução básica.
- O Versionamento com sinalizadores de recursos se baseia na solução de versionamento: nenhuma nova versão de fluxo de trabalho é introduzida, e não há necessidade de novo código de push para atualizar a versão.

Antes de testar essas soluções, familiarize-se com a seção Cenário de exemplo que explica as causas e os efeitos de alterações incompatíveis com versões anteriores do agente de decisão.

As alterações do processo de reprodução e do código

Quando um decider worker AWS Flow Framework for Java executa uma tarefa de decisão, ele deve primeiro reconstruir o estado atual da execução antes de poder adicionar etapas a ela. O agente de decisão faz isso usando um processo chamado reprodução.

O processo de reprodução executa novamente o código do agente de decisão desde o início e, simultaneamente, passa pelo histórico de eventos que já ocorreram. A passagem pelo histórico de eventos permite que a estrutura reaja a sinais ou a conclusão de tarefas e desbloqueie objetos `Promise` no código.

Quando a estrutura executa o código do agente de decisão, ela atribui um ID a cada tarefa agendada (uma atividade, função do Lambda, cronômetro, fluxo de trabalho filho ou sinal de saída)

incrementando um contador. A estrutura comunica esse ID ao Amazon SWF e adiciona o ID aos eventos do histórico, como `ActivityTaskCompleted`.

Para que o processo de reprodução tenha êxito, é importante que o código do agente de decisão seja determinista e que programe as mesmas tarefas na mesma ordem para cada decisão em cada execução do fluxo de trabalho. Se você não seguir esse requisito, a estrutura poderá, por exemplo, não corresponder o ID em um evento `ActivityTaskCompleted` para um objeto `Promise` existente.

Cenário de exemplo

Há uma classe de alterações de código consideradas como incompatíveis com versões anteriores. Essas alterações incluem atualizações que modificam o número, o tipo ou a ordem das tarefas programadas. Considere o seguinte exemplo:

Você escreve o código do agente de decisão para programar duas tarefas com temporizador. Você inicia uma execução e executa uma decisão. Como resultado, duas tarefas do cronômetro são agendadas, com IDs 1 e. 2

Se você atualizar o código do agente de decisão para programar apenas um temporizador antes da próxima decisão ser executada, durante a próxima tarefa de decisão a estrutura não reproduzirá o segundo evento `TimerFired`, porque o ID 2 não corresponde a nenhuma tarefa com temporizador produzida pelo código.

Esboço do cenário

O esboço a seguir mostra as etapas deste cenário. O objetivo final do cenário é migrar para um sistema que programa apenas um temporizador mas não provoca falhas em execuções iniciadas antes da migração.

1. A versão inicial do agente de decisão
   a. Escrever o agente de decisão.
   b. Iniciar o agente de decisão.
   c. O agente de decisão programa dois temporizadores.
   d. O agente de decisão inicia cinco execuções.
   e. Interromper o agente de decisão.
2. Uma alteração no agente de decisão incompatível com versões anteriores
   a. Modificar o agente de decisão.

b. Iniciar o agente de decisão.

c. O agente de decisão programa um temporizador.

d. O agente de decisão inicia cinco execuções.

As seções a seguir incluem exemplos de código Java que mostram como implementar esse cenário. Os exemplos de código da seção Soluções mostram várias maneiras de como corrigir alterações incompatíveis com versões anteriores.

> **Note**
>
> Você pode usar a versão mais recente do AWS SDK para Java para executar esse código.

Código comum

O código Java a seguir não é alterado entre os exemplos desse cenário.

`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;
    }
```

```
}
```

Como escrever o código inicial do agente de decisão

O seguinte é o código Java inicial do agente de decisão. Ele está registrado como a versão 1 e programa duas tarefas com temporizador de cinco segundos.

`InitialDecider.java`

```
package sample.v1;

import com.amazonaws.services.simpleworkflow.flow.DecisionContext;
import com.amazonaws.services.simpleworkflow.flow.DecisionContextProviderImpl;
import com.amazonaws.services.simpleworkflow.flow.WorkflowClock;
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
 defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {

    @Execute(version = "1")
    public void sample(Input input);

    public static class Impl implements Foo {

        private DecisionContext decisionContext = new
 DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

        @Override
        public void sample(Input input) {
            System.out.println("Decision (V1) WorkflowId: " +
 decisionContext.getWorkflowContext().getWorkflowExecution().getWorkflowId());
            clock.createTimer(5);
            clock.createTimer(5);
        }
```

```
    }
}
```

Simulação de uma alteração incompatível com versões anteriores

O código Java do agente de decisão a seguir é um bom exemplo de uma alteração incompatível com versões anteriores. O código ainda está registrado como a versão 1, mas programa apenas um temporizador.

`ModifiedDecider.java`

```
package sample.v1.modified;

import com.amazonaws.services.simpleworkflow.flow.DecisionContext;
import com.amazonaws.services.simpleworkflow.flow.DecisionContextProviderImpl;
import com.amazonaws.services.simpleworkflow.flow.WorkflowClock;
import com.amazonaws.services.simpleworkflow.flow.annotations.Execute;
import com.amazonaws.services.simpleworkflow.flow.annotations.Workflow;
import
 com.amazonaws.services.simpleworkflow.flow.annotations.WorkflowRegistrationOptions;

import sample.Input;

@Workflow
@WorkflowRegistrationOptions(defaultExecutionStartToCloseTimeoutSeconds = 60,
 defaultTaskStartToCloseTimeoutSeconds = 5)
public interface Foo {

    @Execute(version = "1")
    public void sample(Input input);

    public static class Impl implements Foo {

        private DecisionContext decisionContext = new
 DecisionContextProviderImpl().getDecisionContext();
        private WorkflowClock clock = decisionContext.getWorkflowClock();

        @Override
        public void sample(Input input) {
            System.out.println("Decision (V1 modified) WorkflowId: " +
 decisionContext.getWorkflowContext().getWorkflowExecution().getWorkflowId());
            clock.createTimer(5);
        }
```

```
    }
}
```

O código Java a seguir permite que você simule o problema de alterações incompatíveis com versões anteriores executando o agente de decisão modificado.

RunModifiedDecider.java

```
package sample;

import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class BadChange extends SampleBase {

    public static void main(String[] args) throws Exception {
        new BadChange().run();
    }

    public void run() throws Exception {
        // Start the first version of the decider
        WorkflowWorker before = new WorkflowWorker(service, domain, taskList);
        before.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        before.start();

        // Start a few executions
        startFiveExecutions("Foo.sample", "1", new Input());

        // Stop the first decider worker and wait a few seconds
        // for its pending pollers to match and return
        before.suspendPolling();
        sleep(2000);

        // At this point, three executions are still open, with more decisions to make

        // Start the modified version of the decider
        WorkflowWorker after = new WorkflowWorker(service, domain, taskList);
        after.addWorkflowImplementationType(sample.v1.modified.Foo.Impl.class);
        after.start();

        // Start a few more executions
        startFiveExecutions("Foo.sample", "1", new Input());
```

```
        printExecutionResults();
    }

}
```

Quando você executa o programa, as três execuções que falham são as que começaram na versão inicial do agente de decisão e continuaram depois da migração.

Soluções

Você pode usar as seguintes soluções para evitar alterações incompatíveis com versões anteriores. Para obter mais informações, consulte Como fazer alterações no código do agente de decisão e Cenário de exemplo.

Usar versionamento

Nesta solução, você copia o agente de decisão em uma classe nova, modifica o agente de decisão e o registra em uma versão nova do fluxo de trabalho.

`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);
        }

    }

}
```

No código Java atualizado, o segundo operador de agente de decisão executa as duas versões do fluxo de trabalho permitindo que as execuções em andamento continuem a ser executadas independentemente da alterações na versão 2.

RunVersionedDecider.java

```
package sample;

import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class VersionedChange extends SampleBase {

    public static void main(String[] args) throws Exception {
        new VersionedChange().run();
    }

    public void run() throws Exception {
        // Start the first version of the decider, with workflow version 1
        WorkflowWorker before = new WorkflowWorker(service, domain, taskList);
        before.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        before.start();

        // Start a few executions with version 1
        startFiveExecutions("Foo.sample", "1", new Input());

        // Stop the first decider worker and wait a few seconds
        // for its pending pollers to match and return
```

```
        before.suspendPolling();
        sleep(2000);

        // At this point, three executions are still open, with more decisions to make

        // Start a worker with both the previous version of the decider (workflow
 version 1)
        // and the modified code (workflow version 2)
        WorkflowWorker after = new WorkflowWorker(service, domain, taskList);
        after.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        after.addWorkflowImplementationType(sample.v2.Foo.Impl.class);
        after.start();

        // Start a few more executions with version 2
        startFiveExecutions("Foo.sample", "2", new Input());

        printExecutionResults();
    }

}
```

Quando você executa o programa, todas as execuções são concluídas com êxito.

Uso de sinalizadores de recursos

Outra solução de problemas de compatibilidade com versões anteriores é ramificar o código para oferecer suporte a duas implementações na mesma classe com base nos dados de entrada e não nas versões do fluxo de trabalho.

Ao usar essa abordagem, você adiciona campos (ou modifica os campos existentes) a seus objetos de entrada sempre que introduz alterações importantes. Em execuções iniciadas antes da migração, o objeto de entrada não terá o campo (ou terá um valor diferente). Portanto, você não precisa aumentar o número da versão.

> **Note**
>
> Se você adicionar novos campos, verifique se o processo de desserialização do JSON é compatível com versões anteriores. Os objetos serializados antes da introdução do campo ainda devem ser desserializados com êxito após a migração. Como o JSON define um valor `null` sempre que um campo está ausente, use sempre tipos demarcados (`Boolean` em vez de `boolean`) e trate dos casos em que o valor é `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);
            }
        }

    }
}
```

No código Java atualizado, o código das duas versões do fluxo de trabalho ainda está registrado para a versão 1. No entanto, após a migração, as novas execuções são iniciadas com o campo `skipSecondTimer` dos dados de entrada definido como `true`.

`RunFeatureFlagDecider.java`

```
package sample;

import com.amazonaws.services.simpleworkflow.flow.WorkflowWorker;

public class FeatureFlagChange extends SampleBase {

    public static void main(String[] args) throws Exception {
        new FeatureFlagChange().run();
    }

    public void run() throws Exception {
        // Start the first version of the decider
        WorkflowWorker before = new WorkflowWorker(service, domain, taskList);
        before.addWorkflowImplementationType(sample.v1.Foo.Impl.class);
        before.start();

        // Start a few executions
        startFiveExecutions("Foo.sample", "1", new Input());

        // Stop the first decider worker and wait a few seconds
        // for its pending pollers to match and return
        before.suspendPolling();
        sleep(2000);

        // At this point, three executions are still open, with more decisions to make

        // Start a new version of the decider that introduces a change
        // while preserving backwards compatibility based on input fields
        WorkflowWorker after = new WorkflowWorker(service, domain, taskList);
        after.addWorkflowImplementationType(sample.v1.featureflag.Foo.Impl.class);
        after.start();

        // Start a few more executions and enable the new feature through the input
 data
        startFiveExecutions("Foo.sample", "1", new Input().setSkipSecondTimer(true));

        printExecutionResults();
```

```
    }

}
```

Quando você executa o programa, todas as execuções são concluídas com êxito.

Dicas de solução de problemas e depuração para Java AWS Flow Framework

Tópicos

Esta seção descreve algumas armadilhas comuns que você pode encontrar ao desenvolver fluxos de trabalho usando AWS Flow Framework para Java. Ela também oferece algumas dicas para ajudá-lo a diagnosticar e depurar problemas.

Erros de compilação

Se estiver usando a opção de construção do tempo de compilação AspectJ é possível que você encontre erros de tempo de compilação para os quais o compilador não é capaz de encontrar as classes do cliente gerado para o seu fluxo de trabalho e suas atividades. A causa provável desses erros de compilação é que o construtor AspectJ ignorou os clientes gerados durante a compilação. É possível corrigir esse problema removendo o recurso AspectJ do projeto e reativando-o. Observe que será necessário fazer isso toda vez que a interface do fluxo de trabalho ou das atividades for alterada. Devido a esse problema, recomendamos que você use a opção de construção do tempo de carregamento. Consulte a seção Configurando o AWS Flow Framework para Java para obter mais detalhes.

Falha de recurso desconhecida

O Amazon SWF retorna uma falha de recurso desconhecido quando você tenta executar uma operação em um recurso que não está disponível. As causas comuns para essa falha são:

- Você configura um operador com um domínio que não existe. Para corrigir isso, primeiro registre o domínio usando o console do Amazon SWF ou a API de serviço do Amazon SWF.
- Você tenta criar execução de fluxo de trabalho ou tarefas de atividade dos tipos que não foram registrados. Isso pode acontecer se você tentar criar a execução do fluxo de trabalho antes dos operadores serem executados. Como os trabalhadores registram seus tipos quando são executados pela primeira vez, você deve executá-los pelo menos uma vez antes de tentar iniciar as execuções (ou registrar manualmente os tipos usando o console ou a API de serviço). Observe que assim que os tipos forem registrados, é possível criar execuções mesmo que nenhum operador esteja sendo executado.
- Um operador tenta concluir uma tarefa que já ultrapassou o tempo limite. Por exemplo, se um trabalhador demorar muito para processar uma tarefa e exceder o tempo limite, ele terá uma UnknownResource falha ao tentar concluir ou falhar na tarefa. Os AWS Flow Framework trabalhadores continuarão pesquisando o Amazon SWF e processando tarefas adicionais. No entanto, considere ajustar o tempo limite. Ajustar o limite requer que você registre uma nova versão do tipo de atividade.

Exceções ao chamar get () em uma promessa

Diferente do Java Future, o `Promise` é uma construção sem bloqueio e chamar `get()` em uma `Promise` que ainda não está pronta lançará um exceção em vez de bloquear. A maneira correta de usar um `Promise` é passá-lo para um método assíncrono (ou uma tarefa) e acessar seu valor no método assíncrono. O AWS Flow Framework for Java garante que um método assíncrono seja chamado somente quando todos os argumentos `Promise` passados para ele estiverem prontos. Se você acredita que seu código está correto ou se depara com isso ao executar uma das AWS Flow Framework amostras, provavelmente é porque o AspectJ não está configurado corretamente. Para obter detalhes, consulte a seção Configurando o AWS Flow Framework para Java.

Fluxos de trabalho não determinísticos

Conforme descrito na seção Não determinismo, a implementação do seu fluxo de trabalho deve ser determinística. Alguns erros comuns que podem levar ao não determinismo são o uso do relógio do sistema, o uso de números aleatórios e a geração de. GUIDs Como essas construções podem retornar valores diferentes em momentos diferentes, o fluxo de controle do seu fluxo de trabalho pode seguir caminhos diferentes cada vez que é executado (consulte as seções AWS Flow Framework Conceitos básicos: execução distribuída e Entendendo uma tarefa em AWS Flow

Framework Java para obter detalhes). Se a estrutura detecta não determinismo enquanto executa o fluxo de trabalho, uma exceção será lançada.

Problemas devido ao controle de versão

Ao implementar uma nova versão do seu fluxo de trabalho ou atividade (por exemplo, ao adicionar um novo recurso), você deve aumentar a versão do tipo usando a anotação apropriada: `@Workflow`, `@Activites` ou `@Activity`. Quando as novas versões do fluxo de trabalho são implantadas, geralmente existirão execuções da versão existente que já estão sendo executadas. Portanto, é necessário garantir que os operadores com a versão adequada do fluxo de trabalho e das atividades obtenham as tarefas. Alcance isso usando um conjunto de lista de tarefas diferentes para cada versão. Por exemplo, anexe o número da versão ao nome da lista de tarefas. Isso garante que as tarefas que pertencem a versões diferentes do fluxo de trabalho e das atividades sejam atribuídas aos operadores adequados.

Solução de problemas e depuração da execução de um fluxo de trabalho

A primeira etapa na solução de problemas de execução de um fluxo de trabalho é usar o console do Amazon SWF para examinar o histórico do fluxo de trabalho. O histórico do fluxo de trabalho é um registro completo e confiável de todos os eventos que alteraram o estado de execução da execução do fluxo de trabalho. Esse histórico é mantido pelo Amazon SWF e é de grande valia para o diagnóstico de problemas. O console do Amazon SWF permite pesquisar execuções de fluxo de trabalho e detalhar eventos individuais do histórico.

AWS Flow Framework fornece uma `WorkflowReplayer` classe que você pode usar para reproduzir a execução de um fluxo de trabalho localmente e depurá-la. Usando essa classe, você pode depurar execuções de fluxo de trabalho fechadas e em execução. O `WorkflowReplayer` conta com o histórico armazenado no Amazon SWF para executar a reprodução. Você pode apontá-lo para a execução de um fluxo de trabalho na sua conta do Amazon SWF ou fornecer a ele os eventos do histórico (por exemplo, você pode recuperar o histórico do Amazon SWF e serializá-lo localmente para uso posterior). Ao repetir uma execução do fluxo de trabalho usando o `WorkflowReplayer`, não há impacto sobre a execução do fluxo de trabalho em andamento em sua conta. A repetição é feita totalmente no cliente. Depure o fluxo de trabalho, crie pontos de interrupção e analise o código usando suas ferramentas de depuração como sempre. Se você estiver usando o Eclipse, considere adicionar filtros de etapas aos AWS Flow Framework pacotes de filtro.

Por exemplo, o trecho de código a seguir pode ser usado para repetir uma execução de fluxo de trabalho:

```
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 também permite que você obtenha um despejo assíncrono da execução do seu fluxo de trabalho. Essa thread dump fornece as pilhas de chamada de todas as tarefas assíncronas abertas. Essa informação pode ser útil para determinar quais tarefas na execução estão pendentes e possivelmente travadas. Por exemplo:

```
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);
```

```
}
```

Tarefas perdidas

Às vezes você pode desativar operadores e iniciar novos em uma sucessão rápida apenas para descobrir que as tarefas são entregues aos operadores antigos. Isso pode acontece devido a condições de corrida no sistema, que é distribuído entre diversos processos. O problema também pode aparecer ao executar testes de unidade em um ciclo apertado. Interromper um teste no Eclipse pode causar isso pois os manipuladores do desligamentos podem não ser chamados.

Para garantir que o problema existe devido ao fato dos operadores antigos obterem tarefas, observe o histórico do fluxo de trabalho para determinar qual processo recebeu a tarefa que deveria ser recebida pelo novo operador. Por exemplo, o evento `DecisionTaskStarted` no histórico contém a identidade do operador de fluxo de trabalho que recebeu a tarefa. O id usado pelo Flow Framework tem o formato: {*`processId`*} @ {*`host name`*}. Por exemplo, a seguir estão os detalhes do evento `DecisionTaskStarted` no console do Amazon SWF para uma execução de amostra:

Data do evento	Segunda-feira 20 de fevereiro 11:52:40 GMT-800 2012
Identidade	2276 @ip -0A6C1 DF5
ID do evento programado	33

Para evitar essa situação, use listas de tarefas diferentes para cada teste. Além disso, considere adicionar um atraso entre o desligamento de operadores antigos e o início de novos.

Falha na validação devido a restrições de comprimento dos parâmetros da API

O Amazon SWF impõe restrições de comprimento nos parâmetros da API. Você receberá um `HTTP 400` erro se a implementação do fluxo de trabalho ou da atividade exceder as restrições. Por exemplo, ao `recordActivityHeartbeat` ligar `ActivityExecutionContext` para enviar uma pulsação para uma atividade em execução, a string não deve ter mais de 2048 caracteres.

Outro cenário comum é quando uma atividade falha devido a uma exceção. A estrutura relata uma falha de atividade ao Amazon SWF chamando RespondActivityTaskFailedcom a exceção

serializada como detalhes. A chamada de API reportará um erro 400 se a exceção serializada tiver um comprimento maior que 32.768 bytes. Para mitigar essa situação, você pode truncar a mensagem de exceção ou as causas de acordo com a restrição de comprimento.

AWS Flow Framework para referência de Java

Tópicos

AWS Flow Framework para anotações Java

Tópicos

@Atividades

Essa anotação pode ser usada em uma interface para declarar um conjunto de tipos de atividade. Cada método em uma interface anotada com essa anotação representa um tipo de atividade. Uma interface não pode ter anotações `@Workflow` e `@Activities`

Os seguintes parâmetros podem ser especificados nessa anotação:

`activityNamePrefix`

Especifica o prefixo do nome dos tipos de atividade declarados na interface. Se for definido como uma sequência vazia (padrão), o nome da interface, seguido por '.', será usado como prefixo.

`version`

Especifica a versão padrão dos tipos de atividade declarados na interface. O valor padrão é `1.0`.

`dataConverter`

Especifica o tipo de `DataConverter` a ser usado para serialização/desserialização de dados ao criar tarefas desse tipo de atividade e seus resultados. Definido como `NullDataConverter` por padrão, o que indica que o `JsonDataConverter` deve ser usado.

@Atividades

Essa anotação pode ser usada em métodos em uma interface anotada com `@Activities`.

Os seguintes parâmetros podem ser especificados nessa anotação:

`name`

Especifica o nome do tipo de atividade. O padrão é uma sequência vazia, o que indica que o prefixo padrão e o nome do método de atividades devem ser usados para determinar o nome do tipo de atividade (que tem o formato {prefixo}{nome}). Observe que quando você especifica um nome em uma anotação `@Activity`, a estrutura não precede um prefixo automaticamente a ele. Você está livre para usar o seu próprio esquema de nomenclatura.

`version`

Especifica a versão do tipo da atividade. Substitui a versão padrão especificada na anotação `@Activities` na interface que a contém. O padrão é uma string vazia.

@ActivityRegistrationOptions

Especifica as opções de registro de um tipo de atividade. Essa anotação pode ser usada em uma interface anotada com `@Activities` ou com os métodos dentro dela. Se especificada nos dois lugares, a anotação usada no método entrará em vigor.

Os seguintes parâmetros podem ser especificados nessa anotação:

`defaultTasklist`

Especifica a lista de tarefas padrão a ser registrada no Amazon SWF para esse tipo de atividade. Esse padrão pode ser substituído ao chamar o método de atividade no cliente gerado usando o parâmetro `ActivitySchedulingOptions`. Definida como `USE_WORKER_TASK_LIST` por padrão. Esse é um valor especial que indica que a lista de tarefas usada pelo operador, que está executando o registro, deve ser usada.

`defaultTaskScheduleToStartTimeoutSeconds`

Especifica o defaultTaskSchedule ToStartTimeout registrado no Amazon SWF para esse tipo de atividade. Esse é o tempo máximo que uma tarefa desse tipo de atividade tem permissão para aguardar antes de ser atribuída a um operador. Consulte mais detalhes em Amazon Simple Workflow Service API Reference.

`defaultTaskHeartbeatTimeoutSeconds`

Especifica o `defaultTaskHeartbeatTimeout` registrado no Amazon SWF para esse tipo de atividade. Os operadores de atividades devem fornecer pulsação dentro dessa duração. Caso contrário, o tempo limite da tarefa será esgotado. Definido como -1 por padrão, que é um valor especial que indica que esse tempo limite deve ser desativado. Consulte mais detalhes em Amazon Simple Workflow Service API Reference.

`defaultTaskStartToCloseTimeoutSeconds`

Especifica o defaultTaskStart ToCloseTimeout registrado no Amazon SWF para esse tipo de atividade. Esse tempo limite determina o tempo máximo que um operador pode usar para processar uma tarefa de atividade desse tipo. Consulte mais detalhes em Amazon Simple Workflow Service API Reference.

`defaultTaskScheduleToCloseTimeoutSeconds`

Especifica o `defaultScheduleToCloseTimeout` registrado no Amazon SWF para esse tipo de atividade. Esse tempo limite determina a duração total na qual uma tarefa pode permanecer em estado aberto. Definido como -1 por padrão, que é um valor especial que indica que esse tempo limite deve ser desativado. Consulte mais detalhes em Amazon Simple Workflow Service API Reference.

@Assíncrono

Quando usada em um método na lógica de coordenação do fluxo de trabalho, indica que o método deve ser executado de forma assíncrona. Uma chamada ao método retornará imediatamente, mas

a execução real ocorrerá de forma assíncrona quando todos os parâmetros de Promise<> passados para os métodos estiverem prontos. Os métodos anotados com @Asynchronous devem ter um tipo de retorno de Promise<> ou nulo.

`daemon`

Indica se a tarefa criada para o método assíncrono deve ser uma tarefa de daemon. Por padrão, `False`.

@Execute

Quando usada em um método em uma interface anotada com a anotação `@Workflow`, identifica o ponto de entrada do fluxo de trabalho.

> **Important**
>
> Apenas um método na interface pode ser decorado com `@Execute`.

Os seguintes parâmetros podem ser especificados nessa anotação:

`name`

Especifica o nome do tipo do fluxo de trabalho. Se não estiver definido, o nome padrão será {prefixo}{nome}, em que {prefixo} é o nome da interface do fluxo de trabalho seguido por um '.' e {nome} é o nome do método atribuído a `@Execute` no fluxo de trabalho.

`version`

Especifica a versão do tipo do fluxo de trabalho.

@ExponentialRetry

Quando usada em uma atividade ou método assíncrono, define uma política de repetição exponencial se o método gerar uma exceção não tratada. Uma tentativa de repetição é feita depois de um período de recuo, que é calculado pela potência do número de tentativas.

Os seguintes parâmetros podem ser especificados nessa anotação:

`intialRetryIntervalSeconds`

Especifica a duração da espera antes da primeira tentativa de repetição. Esse valor deve ser maior que ou igual a `maximumRetryIntervalSeconds` e `retryExpirationSeconds`.

`maximumRetryIntervalSeconds`

Especifica a duração máxima entre tentativas de repetição. Após atingido, o intervalo de repetição é limitado a esse valor. Definido como -1 por padrão, o que significa duração ilimitada.

`retryExpirationSeconds`

Especifica a duração depois da qual a repetição exponencial será interrompida. Definido como -1 por padrão, o que significa que não há expiração.

`backoffCoefficient`

Especifica o coeficiente usado para calcular o intervalo de repetição. Consulte Estratégia de repetição exponencial.

`maximumAttempts`

Especifica o número de tentativas depois do qual a repetição exponencial será interrompida. Definido como -1 por padrão, o que significa que não há nenhum limite no número de tentativas de repetição.

`exceptionsToRetry`

Especifica a lista de tipos de exceção que devem acionar uma repetição. Uma exceção não tratada desses tipos não será mais propagada, e o método será repetido depois do intervalo calculado de repetição. Por padrão, a lista contém `Throwable`.

`excludeExceptions`

Especifica a lista de tipos de exceção que não devem acionar uma repetição. As exceções não tratadas desse tipo terão permissão para propagar. A lista está vazia por padrão.

@GetState

Quando usada em um método em uma interface anotada com a anotação `@Workflow`, identifica que o método é usado para recuperar o último estado de execução do fluxo de trabalho. Deve haver no máximo um método com essa anotação em uma interface com a anotação `@Workflow`. Os métodos com essa anotação não devem usar nenhum parâmetro e devem ter um tipo de retorno diferente de `void`.

@ManualActivityCompletion

Essa anotação pode ser usada em um método de atividade para indicar que a tarefa de atividade não deve ser concluída quando o método retornar. A tarefa da atividade não será concluída automaticamente e precisará ser concluída manualmente, usando diretamente a API do Amazon SWF. Isso é útil para casos de uso em que a tarefa de atividade é delegada para um sistema externo que não é automatizado ou requer intervenção humana para ser concluído.

@Signal

Quando usada em um método em uma interface anotada com a anotação `@Workflow`, identifica um sinal que pode ser recebido por execuções do tipo de fluxo de trabalho declarado pela interface. O uso dessa anotação é necessário para definir um método de sinal.

Os seguintes parâmetros podem ser especificados nessa anotação:

`name`

Especifica a parte do nome do sinal. Se não definido, o nome do método será usado.

@SkipRegistration

Quando usado em uma interface anotada com a anotação `@Workflow`, indica que o tipo de fluxo de trabalho não deve ser registrado no Amazon SWF. Uma das anotações `@WorkflowRegistrationOptions` e `@SkipRegistrationOptions` deve ser usada em uma interface anotada com `@Workflow`, mas não ambas.

@Wait e @ NoWait

Essas anotações podem ser usadas em um parâmetro do tipo `Promise<>` para indicar se o AWS Flow Framework for Java deve esperar que fique pronto antes de executar o método. Por padrão, os parâmetros `Promise<>` passados para métodos `@Asynchronous` devem estar prontos para que a execução do método ocorra. Em determinadas situações, é necessário substituir esse comportamento padrão. Os parâmetros `Promise<>` passados em métodos `@Asynchronous` e anotados com `@NoWait` não são esperados.

Parâmetros de coleções (ou subclasses de) que contêm promessas, como `List<Promise<Int>>`, devem ser anotados com a anotação `@Wait`. Por padrão, a estrutura não espera os membros de uma coleção.

@Fluxo de trabalho

Essa anotação é usada em uma interface para declarar um tipo de fluxo de trabalho. Uma interface decorada com essa anotação deve conter exatamente um método decorado com a anotação @Execute para declarar um ponto de entrada para o fluxo de trabalho.

> **Note**
>
> Uma interface não pode ter as duas anotações `@Workflow` e `@Activities` declaradas de uma vez pois são mutuamente exclusivas.

Os seguintes parâmetros podem ser especificados nessa anotação:

`dataConverter`

Especifica qual `DataConverter` usar ao enviar solicitações e receber resultados das execuções de fluxo de trabalho desse tipo de fluxo de trabalho.

O padrão é `NullDataConverter` que, por sua vez, volta `JsonDataConverter` para processar todos os dados de solicitação e resposta como JavaScript Object Notation (JSON).

Exemplo

```
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

Quando usado em uma interface anotada com `@Workflow`, fornece as configurações padrão usadas pelo Amazon SWF ao registrar o tipo de fluxo de trabalho.

> **Note**
>
> `@WorkflowRegistrationOptions` ou `@SkipRegistrationOptions` deve ser usada em uma interface anotada com `@Workflow`, mas as duas não podem ser especificadas.

Os seguintes parâmetros podem ser especificados nessa anotação:

Descrição

Uma descrição de texto opcional do tipo do fluxo de trabalho.

`defaultExecutionStartToCloseTimeoutSeconds`

Especifica o `defaultExecutionStartToCloseTimeout` registrado com o Amazon SWF para o tipo de fluxo de trabalho. Esse é o tempo total que uma execução de fluxo de trabalho desse tipo pode levar para ser concluída.

Para obter mais informações sobre tempos limite de fluxo de trabalho, consulte Tipos de tempo limite do Amazon SWF .

`defaultTaskStartToCloseTimeoutSeconds`

Especifica o `defaultTaskStartToCloseTimeout` registrado com o Amazon SWF para o tipo de fluxo de trabalho. Especifica o tempo que uma única tarefa de decisão de uma execução de fluxo de trabalho desse tipo pode levar para ser concluída.

Se você não especificar o `defaultTaskStartToCloseTimeout`, ele será padronizado para 30 segundos.

Para obter mais informações sobre tempos limite de fluxo de trabalho, consulte Tipos de tempo limite do Amazon SWF .

`defaultTaskList`

A lista de tarefas padrão usada para tarefas de decisão para execuções desse tipo de fluxo de trabalho. O conjunto padrão aqui pode ser substituído usando `StartWorkflowOptions` ao iniciar uma execução de fluxo de trabalho.

Se você não especificar `defaultTaskList`, ela será definida como `USE_WORKER_TASK_LIST` por padrão. Isso indica que a lista de tarefas usada pelo operador que está executando o registro do fluxo de trabalho deve ser usada.

`defaultChildPolicy`

Especifica a política a ser usada para fluxos de trabalho filhos se uma execução desse tipo for encerrada. O valor padrão é `ABANDON`. Os valores possíveis são:

- `ABANDON`: permitir que as execuções do fluxo de trabalho filho continuem em execução
- `TERMINATE`: encerrar execuções de fluxo de trabalho filho
- `REQUEST_CANCEL`: solicitar o cancelamento das execuções do fluxo de trabalho filho

AWS Flow Framework para exceções de Java

As exceções a seguir são usadas pelo AWS Flow Framework for Java. Esta seção fornece uma visão geral da exceção. Para obter mais detalhes, consulte a AWS SDK para Java documentação das exceções individuais.

Tópicos

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

ActivityFailureException

Essa exceção é usada internamente pela estrutura para comunicar falha de atividade. Quando uma atividade falha devido a uma exceção não tratada, ela é envolvida em `ActivityFailureException` e relatada ao Amazon SWF. Você precisará tratar essa exceção apenas se usar os pontos de extensibilidade do operador de atividades. O código de seu aplicativo nunca precisará tratar essa exceção.

ActivityTaskException

Essa é a classe base para exceções de falha de tarefas de atividades: `ScheduleActivityTaskFailedException`, `ActivityTaskFailedException`, `ActivityTaskTimedoutException`. Ela contém o Id da tarefa e o tipo de atividade da tarefa com falha. Você pode capturar essa exceção em sua implementação de fluxo de trabalho para tratar falhas de atividades de forma genérica.

ActivityTaskFailedException

As exceções não tratadas em atividades são relatadas de volta para a implementação de fluxo de trabalho gerando uma `ActivityTaskFailedException`. A exceção original pode ser recuperada da propriedade cause dessa exceção. A exceção também fornece outras informações úteis para fins de depuração, como o identificador exclusivo da atividade no histórico.

A estrutura pode fornecer a exceção remota serializando a exceção original no operador de atividades.

ActivityTaskTimedOutException

Essa exceção é lançada se uma atividade tiver sido interrompida pelo Amazon SWF. Isso poderá acontecer se a tarefa de atividade não puder ser atribuída ao operador no período necessário ou não puder ser concluída pelo operador no tempo requerido. Você pode definir esses tempos limite na atividade usando a anotação `@ActivityRegistrationOptions` ou o parâmetro `ActivitySchedulingOptions` ao chamar o método da atividade.

ChildWorkflowException

Classe base de exceções usadas para relatar falha na execução de fluxo de trabalho filho. A exceção contém os Ids de execução do fluxo de trabalho filho assim como o tipo de seu fluxo de

trabalho. Você pode capturar essa exceção para tratar falhas na execução de fluxos de trabalho filhos de forma genérica.

ChildWorkflowFailedException

As exceções não tratadas em fluxos de trabalho filhos são relatadas de volta para a implementação do fluxo de trabalho pai gerando uma `ChildWorkflowFailedException`. A exceção original pode ser recuperada da propriedade `cause` dessa exceção. A exceção também fornece outras informações úteis para fins de depuração, como os identificadores exclusivos da execução filho.

ChildWorkflowTerminatedException

Essa exceção é gerada na execução do fluxo de trabalho pai para relatar o término de uma execução de fluxo de trabalho filho. Você deve capturar essa exceção se desejar tratar o término do fluxo de trabalho filho, por exemplo, executar uma limpeza ou uma compensação.

ChildWorkflowTimedOutException

Essa exceção é lançada na execução do fluxo de trabalho pai para relatar que a execução de um fluxo de trabalho filho foi expirada e encerrada pelo Amazon SWF. Você deve capturar essa exceção se desejar tratar o fechamento forçado do fluxo de trabalho filho, por exemplo, executar uma limpeza ou uma compensação.

DataConverterException

A estrutura usa o componente `DataConverter` para executar marshal e unmarshal nos dados enviados pela conexão. Essa exceção será gerada se `DataConverter` não executar marshal ou unmarshal nos dados. Isso pode acontecer por vários motivos, por exemplo, devido a uma incompatibilidade nos componentes de `DataConverter` que estão sendo usados para executar marshal e unmarshal nos dados.

DecisionException

Esta é a classe base para exceções que representam falhas na execução de uma decisão pelo Amazon SWF. Você pode capturar essa exceção para tratar genericamente essas exceções.

ScheduleActivityTaskFailedException

Essa exceção é lançada se o Amazon SWF não conseguir agendar uma tarefa de atividade. Isso pode ocorrer por vários motivos; por exemplo, a atividade foi descontinuada ou um limite do Amazon

SWF em sua conta foi atingido. A propriedade `failureCause` na exceção especifica a causa exata da falha na programação da atividade.

SignalExternalWorkflowException

Essa exceção é lançada se o Amazon SWF não conseguir processar uma solicitação da execução do fluxo de trabalho para sinalizar outra execução do fluxo de trabalho. Isso acontece se a execução do fluxo de trabalho de destino não puder ser encontrada, ou seja, a execução do fluxo de trabalho que você especificou não existe ou está em estado fechado.

StartChildWorkflowFailedException

Essa exceção é lançada se o Amazon SWF não conseguir iniciar a execução de um fluxo de trabalho filho. Isso pode ocorrer por vários motivos: por exemplo, o tipo de fluxo de trabalho principal especificado foi preterido ou o limite do Amazon SWF em sua conta foi atingido. A propriedade `failureCause` na exceção especifica a causa exata da falha na inicialização da execução do fluxo de trabalho filho.

StartTimerFailedException

Essa exceção é lançada se o Amazon SWF não conseguir iniciar um cronômetro solicitado pela execução do fluxo de trabalho. Isso pode acontecer se o ID do timer especificado já estiver em uso ou se o limite do Amazon SWF em sua conta tiver sido atingido. A propriedade `failureCause` na exceção especifica a causa exata da falha.

TimerException

Essa é a classe base de exceções relacionadas a temporizadores.

WorkflowException

Essa exceção é usada internamente pela estrutura para relatar falhas na execução do fluxo de trabalho. Você precisará tratar essa exceção apenas se estiver usando um ponto de extensibilidade do operador do fluxo de trabalho.

AWS Flow Framework para pacotes Java

Esta seção fornece uma visão geral dos pacotes incluídos no AWS Flow Framework for Java. Para obter mais informações sobre cada pacote, consulte o pacote com.amazonaws.services.simpleworkflow.flow na Referência da API do AWS SDK para Java.

com.amazonaws.services.simpleworkflow.flow

Contém componentes que se integram ao Amazon SWF.

com.amazonaws.services.simpleworkflow.flow.annotations

Contém as anotações usadas pelo modelo de programação AWS Flow Framework para Java.

com.amazonaws.services.simpleworkflow.flow.aspectj

Contém AWS Flow Framework os componentes Java necessários para recursos como @Assíncrono @ExponentialRetry e.

com.amazonaws.services.simpleworkflow.flow.common

Contém utilitários comuns como constantes definidas pela estrutura.

com.amazonaws.services.simpleworkflow.flow.core

Contém recursos principais como `Task` e `Promise`.

com.amazonaws.services.simpleworkflow.flow.generic

Contém componentes principais, como clientes genéricos, nos quais outros recursos se baseiam.

com.amazonaws.services.simpleworkflow.flow.interceptors

Contém implementações de decoradores fornecidos pela estrutura, incluindo o `RetryDecorator`.

com.amazonaws.services.simpleworkflow.flow.junit

Contém componentes que fornecem integração com o Junit.

com.amazonaws.services.simpleworkflow.flow.pojo

Contém as classes que implementam definições de atividade e de fluxo de trabalho para o modelo de programação com base em anotação.

com.amazonaws.services.simpleworkflow.flow.spring

Contém componentes que fornecem integração com o Spring.

com.amazonaws.services.simpleworkflow.flow.test

Contém classes auxiliares, como `TestWorkflowClock`, para teste de unidade de implementações de fluxo de trabalho.

com.amazonaws.services.simpleworkflow.flow.worker

Contém implementações de operadores de atividade e de fluxo de trabalho.

Histórico do documento

A tabela a seguir descreve as alterações importantes na documentação desde a última versão do Guia do desenvolvedor do AWS Flow Framework para Java.

- Versão da API: 25-01-2012
- Última atualização de documentação: 25 de junho de 2018

Alteração	Descrição	Alterado em
Atualizar	Corrigido um erro na descrição de `backoffCo efficient` para `@ExponentialRetry` . Consulte @ExponentialRetry.	25 de junho de 2018
Atualizar	Limpeza dos exemplos de código neste guia.	5 de junho de 2017
Atualizar	Simplificação e melhoria da organização e do conteúdo deste guia.	19 de maio de 2017
Atualizar	A seção Como fazer alterações no código do agente de decisão: sinalizadores de versionamento e de recursos foi simplificada e melhorada.	10 de abril de 2017
Atualizar	A nova seção Práticas recomendadas foi adicionada com novas orientações sobre como fazer alterações no código do agente de decisão.	3 de março de 2017
Novo recurso	Você pode especificar tarefas Lambda além das tarefas de atividade tradicionais em seus fluxos de trabalho. Para obter mais informações, consulte Implementando AWS Lambda tarefas.	21 de julho de 2015
Novo recurso	O Amazon SWF inclui suporte para definir a prioridade da tarefa em uma lista de tarefas, tentando entregar as tarefas com prioridade mais alta antes das tarefas com prioridad	17 de dezembro de 2014

Alteração	Descrição	Alterado em
	e mais baixa. Para obter mais informações, consulte Definindo a prioridade da tarefa no Amazon SWF.	
Atualizar	Foram feitas atualizações e correções.	1 de agosto de 2013
Atualizar	• Foram feitas atualizações e correções, inclusive atualizações nas instruções de configuração do Eclipse 4.3 e do AWS SDK para Java 1.4.7. • Foi adicionado um novo conjunto de tutoriais para criar cenários de início	28 de junho de 2013
Novo recurso	A versão inicial do AWS Flow Framework para Java.	27 de fevereiro de 2012

As traduções são geradas por tradução automática. Em caso de conflito entre o conteúdo da tradução e da versão original em inglês, a versão em inglês prevalecerá.