Panduan Developerr

AWS Flow Framework untuk Java

Versi API 2021-04-28

AWS Flow Framework untuk Java: Panduan Developerr

Table of Contents

Apa yang dimaksud dengan AWS Flow Framework Java?

Dengan itu AWS Flow Framework, Anda dapat fokus pada penerapan logika alur kerja Anda. Di belakang layar, kerangka kerja menggunakan penjadwalan, perutean, dan kemampuan manajemen status Amazon SWF untuk mengelola eksekusi alur kerja Anda dan membuatnya dapat diskalakan, andal, dan dapat diaudit. AWS Flow Framework alur kerja berbasis sangat bersamaan. Alur kerja dapat didistribusikan ke beberapa komponen, yang dapat berjalan sebagai proses terpisah pada komputer terpisah dan diskalakan secara independen. Aplikasi dapat terus berkembang jika salah satu komponennya berjalan, sehingga sangat toleran terhadap kesalahan.

Apa yang ada di Panduan ini?

Panduan ini berisi informasi tentang cara menginstal, menyiapkan, dan menggunakan aplikasi Amazon SWF AWS Flow Framework untuk membangun aplikasi Amazon SWF.

Memulai dengan AWS Flow Framework for Java

Jika Anda baru memulai dengan AWS Flow Framework untuk Java, baca Memulai dengan AWS Flow Framework for Java bagiannya. Ini akan memandu Anda melalui mengunduh dan menginstal AWS Flow Framework untuk Java, cara mengatur lingkungan pengembangan Anda, dan mengarahkan Anda melalui contoh sederhana membuat alur kerja.

Pengertian AWS Flow Framework untuk Java

Memperkenalkan Amazon SWF AWS Flow Framework dan konsep dasar, menjelaskan struktur dasar aplikasi dan bagaimana data dipertukarkan antar bagian alur kerja terdistribusi. AWS Flow Framework

AWS Flow Framework untuk Panduan Pemrograman Java

Bab ini memberikan panduan pemrograman dasar untuk mengembangkan aplikasi alur kerja dengan AWS Flow Framework for Java, termasuk cara mendaftarkan jenis aktivitas dan alur kerja, mengimplementasikan klien alur kerja, membuat alur kerja turunan, menangani kesalahan, dan banyak lagi.

Memahami Tugas di AWS Flow Framework Java

Bab ini memberikan pandangan yang lebih mendalam tentang cara kerja Java, memberi Anda informasi tambahan tentang urutan eksekusi alur kerja asinkron dan langkah logis dari eksekusi alur kerja standar. AWS Flow Framework

Tips pemecahan masalah dan debugging untuk Java AWS Flow Framework

Bab ini memberikan informasi tentang kesalahan umum yang dapat Anda gunakan untuk memecahkan masalah alur kerja Anda, atau yang dapat Anda gunakan untuk belajar menghindari kesalahan umum.

AWS Flow Framework untuk Referensi Java

Bab ini adalah referensi ke Anotasi, Pengecualian, dan Paket yang ditambahkan AWS Flow Framework untuk Java ke SDK for Java.

Memulai dengan AWS Flow Framework for Java

Bagian ini memperkenalkan AWS Flow Framework dengan memandu Anda melalui serangkaian contoh aplikasi sederhana yang memperkenalkan model pemrograman dasar dan API. Contoh aplikasi didasarkan pada aplikasi Hello World standar yang digunakan untuk memperkenalkan C dan bahasa pemrograman terkait. Berikut ini adalah implementasi khas Java Hello World:

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

Berikut ini adalah deskripsi singkat dari contoh aplikasi. Mereka termasuk kode sumber lengkap sehingga Anda dapat menerapkan dan menjalankan aplikasi sendiri. Sebelum memulai, Anda harus terlebih dahulu mengkonfigurasi lingkungan pengembangan Anda dan membuat proyek AWS Flow Framework untuk Java, seperti diMenyiapkan AWS Flow Framework untuk Java.

- HelloWorld Aplikasi memperkenalkan aplikasi alur kerja dengan menerapkan Hello World sebagai aplikasi Java standar, tetapi menyusunnya seperti aplikasi alur kerja.
- HelloWorldWorkflow Aplikasimenggunakan AWS Flow Framework for Java untuk dikonversi HelloWorld menjadi alur kerja Amazon SWF.
- HelloWorldWorkflowAsyncAplikasi memodifikasi `HelloWorldWorkflow` untuk menggunakan metode alur kerja asinkron.
- HelloWorldWorkflowDistributed Aplikasi memodifikasi `HelloWorldWorkflowAsync` sehingga pekerja alur kerja dan aktivitas dapat berjalan pada sistem yang terpisah.
- HelloWorldWorkflowParallelAplikasi memodifikasi `HelloWorldWorkflow` untuk menjalankan dua aktivitas secara paralel.

Menyiapkan AWS Flow Framework untuk Java

The AWS Flow Framework for Java disertakan dengan file AWS SDK untuk Java. Jika Anda belum menyiapkan AWS SDK untuk Java, kunjungi Memulai di Panduan AWS SDK untuk Java Pengembang untuk informasi tentang menginstal dan mengonfigurasi SDK itu sendiri.

Tambahkan kerangka alur dengan Maven

Alat pembuatan Amazon SWF adalah sumber terbuka—untuk melihat atau mengunduh kode atau membuat alat sendiri, kunjungi repositori di. https://github.com/aws/aws-swf-build-tools

Amazon menyediakan alat build Amazon SWF di Repositori Pusat Maven.

Untuk menyiapkan kerangka alur untuk Maven, tambahkan ketergantungan berikut ke file `pom.xml` proyek Anda:

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

HelloWorld Aplikasi

Untuk memperkenalkan cara aplikasi Amazon SWF terstruktur, kita akan membuat aplikasi Java yang berperilaku seperti alur kerja, tapi itu berjalan secara lokal dalam satu proses. Tidak ada koneksi ke Amazon Web Services yang akan diperlukan.

> **Note**
>
> HelloWorldWorkflowContoh dibangun di atas yang satu ini, menghubungkan ke Amazon SWF untuk menangani pengelolaan alur kerja.

Sebuah aplikasi alur kerja terdiri dari tiga komponen dasar:

- Aktivitas pekerja mendukung satu set aktivitas, yang masing-masing merupakan metode yang mengeksekusi secara independen untuk melakukan tugas tertentu.
- pekerja alur kerja mengatur pelaksanaan aktivitas dan mengelola aliran data. Ini adalah realisasi program dari topologi alur kerja, yang pada dasarnya adalah bagan alur yang mendefinisikan ketika berbagai aktivitas mengeksekusi, apakah mereka mengeksekusi secara berurutan atau bersamaan, dan sebagainya.
- starter alur kerja memulai instans alur kerja, yang disebut eksekusi, dan dapat berinteraksi dengannya selama eksekusi.

HelloWorld diimplementasikan sebagai tiga kelas dan dua antarmuka terkait, yang dijelaskan di bagian berikut. Sebelum memulai, Anda harus mengatur lingkungan pengembangan Anda dan membuat proyek AWS Java baru seperti yang dijelaskan dalamMenyiapkan AWS Flow Framework untuk Java. Paket yang digunakan untuk panduan berikut semua bernama `helloWorld.`*`XYZ`*. Untuk menggunakan nama-nama tersebut, atur atribut `within` di aop.xml sebagai berikut:

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

Untuk mengimplementasikannya HelloWorld, buat paket Java baru di proyek AWS SDK Anda bernama `helloWorld.HelloWorld` dan tambahkan file berikut:

- Sebuah file antarmuka bernama `GreeterActivities.java`
- Sebuah file kelas bernama `GreeterActivitiesImpl.java`, yang mengimplementasikan pekerja aktivitas.
- Sebuah file antarmuka bernama `GreeterWorkflow.java`.
- Sebuah file kelas bernama `GreeterWorkflowImpl.java`, yang mengimplementasikan pekerja alur kerja.
- Sebuah file kelas bernama `GreeterMain.java`, yang menerapkan starter alur kerja.

Detail dibahas dalam bagian berikut dan termasuk kode lengkap untuk setiap komponen, yang dapat Anda tambahkan ke file yang sesuai.

HelloWorld Implementasi Kegiatan

HelloWorld memecah keseluruhan tugas mencetak `"Hello World!"` salam ke konsol menjadi tiga tugas, yang masing-masing dilakukan dengan metode aktivitas. Metode aktivitas didefinisikan dalam antarmuka `GreeterActivities`, sebagai berikut.

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

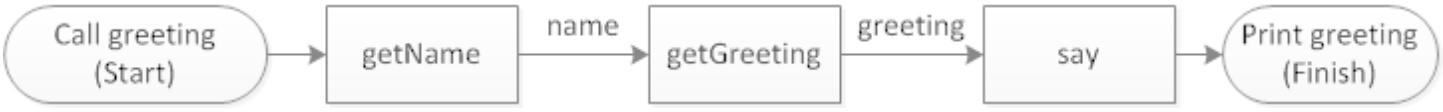
HelloWorld memiliki satu implementasi aktivitas`GreeterActivitiesImpl`, yang menyediakan `GreeterActivities` metode seperti yang ditunjukkan:

```
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);
   }
}
```

Kegiatan independen satu sama lain dan sering dapat digunakan oleh alur kerja yang berbeda. Sebagai contoh, setiap alur kerja dapat menggunakan `say` aktivitas untuk mencetak string ke konsol tersebut. Alur kerja juga dapat memiliki beberapa implementasi aktivitas, masing-masing melakukan serangkaian tugas yang berbeda.

HelloWorld Pekerja Alur Kerja

Untuk mencetak “Hello World!” ke konsol tersebut, tugas aktivitas harus dijalankan secara berurutan di urutan yang benar dengan data yang benar. Pekerja HelloWorld alur kerja mengatur eksekusi aktivitas berdasarkan topologi alur kerja linier sederhana, yang ditunjukkan pada gambar berikut.



Tiga aktivitas mengeksekusi secara berurutan, dan data mengalir dari satu aktivitas ke aktivitas berikutnya.

Pekerja HelloWorld alur kerja memiliki metode tunggal, titik masuk alur kerja, yang didefinisikan dalam `GreeterWorkflow` antarmuka, sebagai berikut:

```
public interface GreeterWorkflow {
```

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

Kelas `GreeterWorkflowImpl` mengimplementasikan antarmuka ini, sebagai berikut:

```
public class GreeterWorkflowImpl implements GreeterWorkflow{
   private GreeterActivities operations = new GreeterActivitiesImpl();

   public void greet() {
      String name = operations.getName();
      String greeting = operations.getGreeting(name);
      operations.say(greeting);
   }
}
```

`greet`Metode ini mengimplementasikan HelloWorld topologi dengan membuat instance dari`GreeterActivitiesImpl`, memanggil setiap metode aktivitas dalam urutan yang benar, dan meneruskan data yang sesuai ke setiap metode.

HelloWorld Alur Kerja Starter

starter alur kerja adalah aplikasi yang memulai eksekusi alur kerja, dan mungkin juga berkomunikasi dengan alur kerja saat mengeksekusi. `GreeterMain`Kelas mengimplementasikan HelloWorld alur kerja starter, sebagai berikut:

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

`GreeterMain` menciptakan sebuah instans dari `GreeterWorkflowImpl` dan panggilan `greet` untuk menjalankan pekerja alur kerja. Jalankan `GreeterMain` sebagai aplikasi Java dan Anda akan melihat “Hello World!” dalam output konsol tersebut.

HelloWorldWorkflow Aplikasi

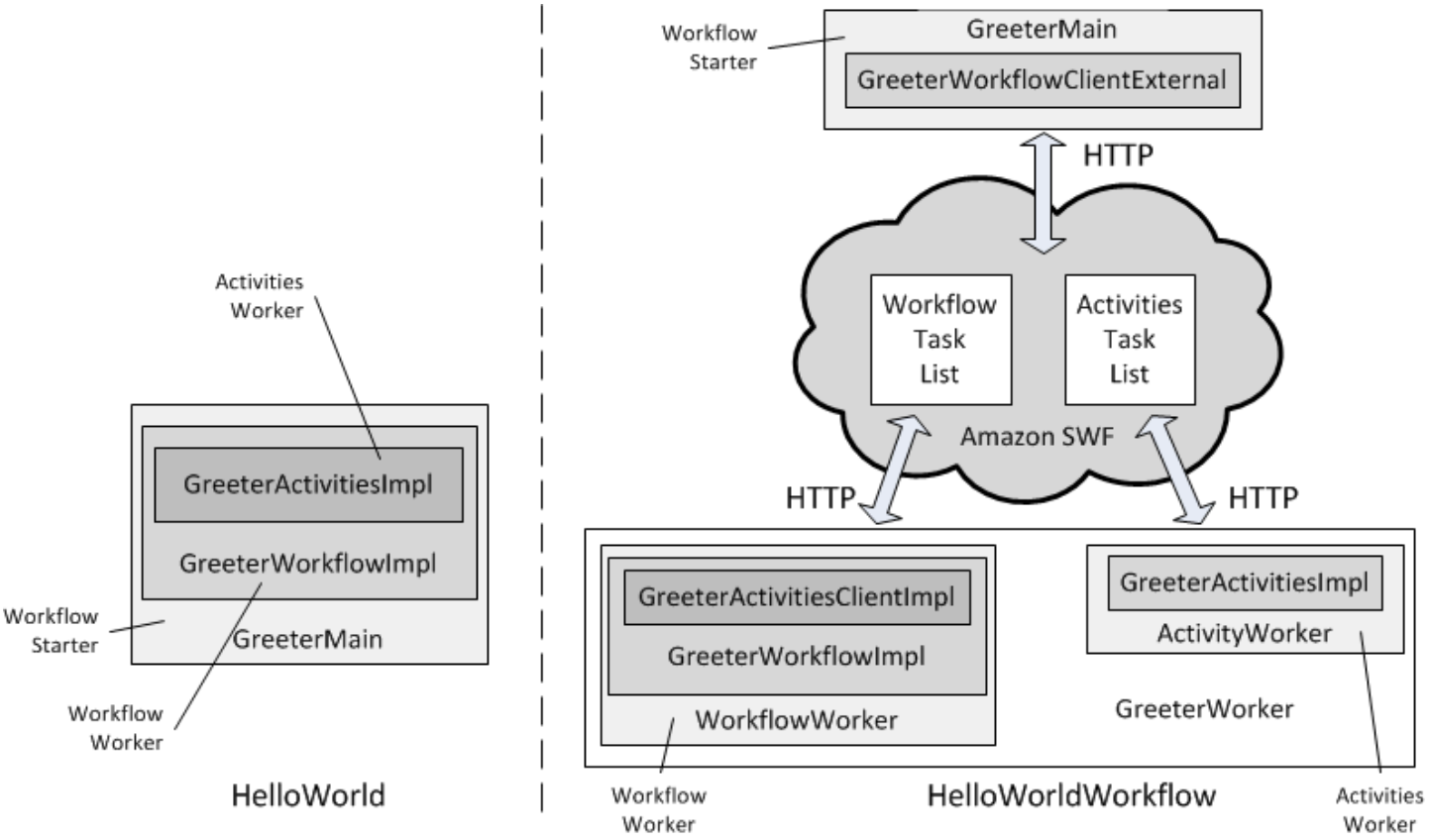
Meskipun HelloWorldcontoh dasarnya terstruktur seperti alur kerja, ini berbeda dari alur kerja Amazon SWF dalam beberapa hal utama:

Aplikasi Alur Kerja konvensional dan Amazon SWF

HelloWorld	Alur Kerja Amazon SWF
Jalankan secara lokal sebagai proses tunggal.	Berjalan sebagai beberapa proses yang dapat didistrib usikan di beberapa sistem, termasuk EC2 instans Amazon, pusat data pribadi, komputer klien, dan sebagainya. Mereka bahkan tidak perlu menjalankan sistem operasi yang sama.
Aktivitas adalah metode sinkron, yang memblok sampai mereka selesai.	Aktivitas diwakili oleh metode asinkron, yang kembali segera dan mengizinkan alur kerja untuk melakukan tugas-tugas lain sambil menunggu untuk aktivitas selesai.
Pekerja alur kerja berinteraksi dengan pekerja aktivitas dengan memanggil metode yang sesuai.	pekerja alur kerja berinteraksi dengan pekerja aktivitas dengan menggunakan permintaan HTTP, dengan Amazon SWF bertindak sebagai perantara.
Alur kerja starter berinteraksi dengan pekerja alur kerja dengan memanggil metode yang sesuai.	Alur kerja starter berinteraksi dengan pekerja alur kerja dengan menggunakan permintaan HTTP, dengan Amazon SWF bertindak sebagai perantara.

Anda dapat menerapkan aplikasi alur kerja asinkron terdistribusi dari scratch, misalnya, dengan meminta pekerja alur kerja Anda berinteraksi dengan pekerja aktivitas secara langsung melalui panggilan layanan web. Namun, Anda kemudian harus menerapkan semua kode rumit yang diperlukan untuk mengelola eksekusi asinkron beberapa aktivitas, menangani aliran data, dan sebagainya. AWS Flow Framework Untuk Java dan Amazon SWF mengurus semua detail itu, yang memungkinkan Anda untuk fokus pada penerapan logika bisnis.

HelloWorldWorkflow adalah versi modifikasi HelloWorld yang berjalan sebagai alur kerja Amazon SWF. Gambar berikut merangkum bagaimana dua aplikasi bekerja.



HelloWorld berjalan sebagai proses tunggal dan starter, pekerja alur kerja, dan pekerja aktivitas berinteraksi dengan menggunakan panggilan metode konvensional. Dengan `HelloWorldWorkflow`, starter, pekerja alur kerja, dan pekerja aktivitas mendistribusikan komponen yang berinteraksi melalui Amazon SWF dengan menggunakan permintaan HTTP. Amazon SWF mengelola interaksi dengan mempertahankan daftar alur kerja dan tugas aktivitas, yang dikirim ke komponen masing-masing. Bagian ini menjelaskan cara kerja kerangka kerja HelloWorldWorkflow.

HelloWorldWorkflow diimplementasikan dengan menggunakan AWS Flow Framework for Java API, yang menangani detail terkadang rumit berinteraksi dengan Amazon SWF di latar belakang dan menyederhanakan proses pengembangan secara signifikan. Anda dapat menggunakan proyek yang sama dengan yang Anda lakukan HelloWorld, yang sudah dikonfigurasi AWS Flow Framework untuk aplikasi Java. Namun, untuk menjalankan aplikasi, Anda harus membuat akun Amazon SWF, sebagai berikut:

- Mendaftar untuk AWS akun, jika Anda belum memilikinya, di Amazon Web Services.
- Tetapkan ID Akses dan ID rahasia akun Anda ke variabel AWS_ACCESS_KEY_ID dan AWS_SECRET_KEY lingkungan, masing-masing. Ini adalah praktik yang baik untuk tidak

mengekspos nilai-nilai kunci literal dalam kode Anda. Menyimpannya dalam variabel lingkungan adalah cara yang mudah untuk menangani masalah ini.

- Mendaftar untuk akun Amazon SWF di Amazon Simple Workflow Service.
- Masuk ke AWS Management Console dan pilih layanan Amazon SWF.
- Pilih Manage Domains (Mengelola Domain) di pojok kanan atas dan daftarkan domain Amazon SWF baru. domain adalah kontainer logis untuk sumber daya aplikasi Anda, seperti alur kerja dan aktivitas jenis, dan eksekusi alur kerja. Anda dapat menggunakan nama domain yang nyaman, tetapi penelusurannya menggunakan "". helloWorldWalkthrough

Untuk mengimplementasikan HelloWorldWorkflow, buat salinan HelloWorld. HelloWorld paket di direktori proyek Anda dan beri nama HelloWorld. HelloWorldWorkflow. Bagian berikut menjelaskan cara memodifikasi HelloWorld kode asli untuk menggunakan AWS Flow Framework untuk Java dan dijalankan sebagai aplikasi alur kerja Amazon SWF.

HelloWorldWorkflow Aktivitas Pekerja

HelloWorld melaksanakan aktivitasnya pekerja sebagai satu kelas. Pekerja AWS Flow Framework untuk aktivitas Java memiliki tiga komponen dasar:

- Metode aktivitas—yang melakukan tugas yang sebenarnya — didefinisikan dalam sebuah antarmuka dan diimplementasikan dalam kelas terkait.
- ActivityWorkerKelas mengelola interaksi antara metode aktivitas dan Amazon SWF.
- Aplikasi host aktivitas mendaftar dan memulai pekerja aktivitas, dan menangani pembersihan.

Bagian ini membahas metode aktivitas; dua kelas lainnya dibahas kemudian.

HelloWorldWorkflow mendefinisikan antarmuka aktivitas di`GreeterActivities`, sebagai berikut:

```
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);
}
```

Antarmuka ini tidak sepenuhnya diperlukan untuk HelloWorld, tetapi AWS Flow Framework untuk aplikasi Java. Perhatikan bahwa definisi antarmuka itu sendiri tidak berubah. Namun, Anda harus menerapkan dua AWS Flow Framework untuk anotasi Java, @ActivityRegistrationOptions dan@Activities, ke definisi antarmuka. Anotasi menyediakan informasi konfigurasi dan mengarahkan prosesor anotasi Java AWS Flow Framework untuk menggunakan definisi antarmuka untuk menghasilkan kelas klien aktivitas, yang akan dibahas nanti.

`@ActivityRegistrationOptions`memiliki beberapa nilai bernama yang digunakan untuk mengkonfigurasi perilaku aktivitas. HelloWorldWorkflow menentukan dua batas waktu:

- `defaultTaskScheduleToStartTimeoutSeconds` menentukan berapa lama tugas dapat antrean dalam daftar tugas aktivitas, dan diatur ke 300 detik (5 menit).
- `defaultTaskStartToCloseTimeoutSeconds` menentukan waktu maksimum aktivitas dapat mengambil untuk melakukan tugas dan diatur ke 10 detik.

Batas waktu ini memastikan bahwa aktivitas menyelesaikan tugasnya dalam jumlah waktu yang wajar. Jika batas waktu terlampaui, kerangka kerja menghasilkan kesalahan dan pekerja alur kerja harus memutuskan bagaimana menangani masalah. Untuk diskusi tentang cara menangani kesalahan tersebut, lihat Penanganan Kesalahan.

`@Activities` memiliki beberapa nilai, tetapi biasanya hanya menentukan nomor versi aktivitas, yang mengizinkan Anda untuk melacak generasi yang berbeda dari implementasi aktivitas. Jika Anda mengubah antarmuka aktivitas setelah Anda mendaftarkannya dengan Amazon SWF, termasuk mengubah nilai-nilai `@ActivityRegistrationOptions`, Anda harus menggunakan nomor versi baru.

HelloWorldWorkflow mengimplementasikan metode aktivitas di`GreeterActivitiesImpl`, sebagai berikut:

```
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);
    }
}
```

Perhatikan bahwa kode identik dengan HelloWorld implementasi. Pada intinya, AWS Flow Framework aktivitas hanyalah metode yang mengeksekusi beberapa kode dan mungkin mengembalikan hasilnya. Perbedaan antara aplikasi standar dan aplikasi alur kerja Amazon SWF terletak pada bagaimana alur kerja mengeksekusi aktivitas, di mana aktivitas mengeksekusi, dan bagaimana hasilnya dikembalikan ke pekerja alur kerja.

HelloWorldWorkflow Pekerja Alur Kerja

Seorang pekerja alur kerja Amazon SWF memiliki tiga komponen dasar.

- implementasi alur kerja, yang merupakan kelas yang melakukan tugas-tugas yang berhubungan dengan alur kerja.
- Kelas Aktivitas klien, yang pada dasarnya sebuah proksi untuk kelas aktivitas dan digunakan oleh implementasi alur kerja untuk mengeksekusi metode aktivitas secara asinkron.
- WorkflowWorkerKelas, yang mengelola interaksi antara alur kerja dan Amazon SWF.

Bagian ini membahas implementasi alur kerja dan aktivitas klien; kelas `WorkflowWorker` dibahas nanti.

HelloWorldWorkflow mendefinisikan antarmuka alur kerja di`GreeterWorkflow`, sebagai berikut:

```
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();
```

```
}
```

Antarmuka ini juga tidak sepenuhnya diperlukan untuk HelloWorld tetapi sangat penting untuk aplikasi AWS Flow Framework untuk Java. Anda harus menerapkan dua AWS Flow Framework untuk anotasi Java, @Workflow dan@WorkflowRegistrationOptions, ke definisi antarmuka alur kerja. Anotasi menyediakan informasi konfigurasi dan juga mengarahkan prosesor anotasi AWS Flow Framework untuk Java untuk menghasilkan kelas klien alur kerja berdasarkan antarmuka, seperti yang dibahas nanti.

`@Workflow`memiliki satu parameter opsional, DataConverter, yang sering digunakan dengan nilai default, NullDataConverter, yang menunjukkan bahwa harus digunakan. JsonDataConverter

`@WorkflowRegistrationOptions` juga memiliki sejumlah parameter opsional yang dapat digunakan untuk mengkonfigurasi pekerja alur kerja. Di sini, kita mengatur `defaultExecutionStartToCloseTimeoutSeconds`—yang menentukan berapa lama alur kerja dapat menjalankan—hingga 3600 detik (1 jam).

Definisi `GreeterWorkflow` antarmuka berbeda dari HelloWorld dalam satu cara penting, @Execute anotasi. Antarmuka alur kerja menentukan metode yang dapat disebut oleh aplikasi seperti alur kerja starter dan terbatas pada beberapa metode, masing-masing dengan peran tertentu. Kerangka kerja tidak menentukan nama atau daftar parameter untuk metode antarmuka alur kerja; Anda menggunakan nama dan daftar parameter yang cocok untuk alur kerja Anda dan menerapkan AWS Flow Framework untuk penjelasan Java untuk mengidentifikasi peran metode ini.

`@Execute` memiliki dua tujuan:

- Ini mengidentifikasi `greet` sebagai titik masuk alur kerja—metode yang digunakan starter alur kerja untuk memulai alur kerja. Secara umum, titik masuk dapat mengambil satu atau lebih parameter, yang memungkinkan starter menginisialisasi alur kerja, tetapi contoh ini tidak memerlukan inisialisasi.
- Ini menentukan nomor versi alur kerja, yang mengizinkan Anda untuk melacak generasi yang berbeda dari implementasi alur kerja. Untuk mengubah antarmuka alur kerja setelah Anda mendaftarkannya dengan Amazon SWF, termasuk mengubah nilai batas waktu, Anda harus menggunakan nomor versi baru.

Untuk informasi tentang metode lainnya yang dapat disertakan dalam antarmuka alur kerja, lihat Alur Kerja dan Kontrak Aktivitas.

HelloWorldWorkflow mengimplementasikan alur kerja di`GreeterWorkflowImpl`, sebagai berikut:

```
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);
   }
}
```

Kode ini mirip dengan HelloWorld, tetapi dengan dua perbedaan penting.

- `GreeterWorkflowImpl` menciptakan sebuah instans dari `GreeterActivitiesClientImpl`, klien aktivitas, bukan `GreeterActivitiesImpl`, dan mengeksekusi aktivitas dengan memanggil metode pada objek klien.
- Nama dan ucapan aktivitas mengembalikan objek `Promise<String>` daripada objek `String`.

HelloWorld adalah aplikasi Java standar yang berjalan secara lokal sebagai satu proses, sehingga `GreeterWorkflowImpl` dapat mengimplementasikan topologi alur kerja hanya dengan membuat instance`GreeterActivitiesImpl`, memanggil metode secara berurutan, dan meneruskan nilai pengembalian dari satu aktivitas ke aktivitas berikutnya. Dengan alur kerja Amazon SWF, tugas aktivitas masih dilakukan dengan metode aktivitas dari `GreeterActivitiesImpl`. Namun, metode ini tidak selalu berjalan dalam proses yang sama seperti alur kerja—bahkan mungkin tidak berjalan pada sistem yang sama—dan alur kerja perlu menjalankan aktivitas asinkron. Persyaratan ini meningkatkan masalah berikut:

- Bagaimana menjalankan metode aktivitas yang mungkin berjalan dalam proses yang berbeda, mungkin pada sistem yang berbeda.
- Bagaimana menjalankan metode aktivitas secara asinkron.
- Bagaimana mengelola nilai-nilai masukan dan mengembalikan aktivitas. Misalnya, jika nilai mengembalikan Aktivitas A adalah masukan untuk Aktivitas B, Anda harus memastikan bahwa Aktivitas B tidak mengeksekusi sampai Aktivitas A selesai.

Anda dapat menerapkan berbagai topologi alur kerja melalui aliran kontrol aplikasi dengan menggunakan familiar kontrol aliran Java dikombinasikan dengan aktivitas klien dan `Promise<T>`.

Aktivitas Klien

`GreeterActivitiesClientImpl` pada dasarnya adalah proksi untuk `GreeterActivitiesImpl` yang mengizinkan penerapan alur kerja untuk menjalankan metode `GreeterActivitiesImpl` asinkron.

Kelas `GreeterActivitiesClient` dan `GreeterActivitiesClientImpl` dibuat secara otomatis untuk Anda menggunakan informasi yang diberikan dalam anotasi yang diterapkan ke `GreeterActivities` kelas Anda. Anda tidak perlu menerapkan ini sendiri.

> **Note**
>
> Eclipse menghasilkan kelas-kelas ini ketika Anda menyimpan proyek Anda. Anda dapat melihat kode yang dihasilkan di subdirektori `.apt_generated` dari direktori proyek Anda. Untuk menghindari kesalahan kompilasi di kelas `GreeterWorkflowImpl` Anda, itu adalah praktik yang baik untuk memindahkan direktori `.apt_generated` ke atas tab Order and Export (Pesan dan Ekspor) dari kotak dialog Java Build Path (Jalur Membangun Java).

Seorang pekerja alur kerja mengeksekusi aktivitas dengan memanggil metode klien yang sesuai. Metode asinkron dan segera mengembalikan objek `Promise<T>`, tempat `T` adalah jenis kembali aktivitas. Objek `Promise<T>` kembali pada dasarnya adalah sebuah tempat untuk nilai bahwa metode aktivitas akhirnya akan kembali.

- Ketika metode klien kegiatan kembali, objek `Promise<T>` awalnya dalam sebuah kondisi tidak siap, yang menunjukkan bahwa objek belum mewakili nilai kembali valid.
- Ketika metode aktivitas yang sesuai menyelesaikan tugas dan kembali, kerangka kerja memberikan nilai kembali ke objek `Promise<T>` dan menempatkan dalam status siap.

<T>Jenis Janji

Tujuan utama dari objek `Promise<T>` adalah untuk mengelola aliran data antara komponen asinkron dan mengontrol ketika mereka mengeksekusi. Ini mengurangi aplikasi Anda dari kebutuhan untuk secara eksplisit mengelola sinkronisasi atau bergantung pada mekanisme seperti timer untuk memastikan bahwa komponen asinkron tidak menjalankan prematur. Ketika Anda memanggil metode aktivitas klien, segera kembali tetapi menunda kerangka kerja mengeksekusi metode aktivitas yang sesuai sampai masukan objek `Promise<T>` siap dan mewakili data yang valid.

Dari perspektif `GreeterWorkflowImpl`, semua tiga aktivitas metode klien kembali segera. Dari perspektif `GreeterActivitiesImpl`, kerangka kerja tidak memanggil `getGreeting` sampai `name` selesai, dan tidak memanggil `say` sampai `getGreeting` selesai.

Dengan menggunakan `Promise<T>` untuk melewatkan data dari satu aktivitas ke kegiatan berikutnya, `HelloWorldWorkflow` tidak hanya memastikan bahwa metode aktivitas tidak mencoba untuk menggunakan data yang tidak valid, itu juga mengontrol ketika kegiatan mengeksekusi dan secara implisit mendefinisikan topologi alur kerja. Melewati setiap `Promise<T>` aktivitas nilai kembali ke aktivitas berikutnya membutuhkan aktivitas untuk mengeksekusi secara berurutan, mendefinisikan topologi linear dibahas sebelumnya. Dengan AWS Flow Framework untuk Java, Anda tidak perlu menggunakan kode pemodelan khusus untuk mendefinisikan topologi yang kompleks, hanya kontrol aliran Java standar dan. `Promise<T>` Untuk contoh bagaimana menerapkan topologi paralel sederhana, lihat HelloWorldWorkflowParallelAktivitas Pekerja.

> **Note**
>
> Ketika metode aktivitas seperti `say` tidak mengembalikan nilai, metode klien yang sesuai mengembalikan objek `Promise<Void>`. Objek tidak mewakili data, tetapi awalnya tidak siap dan menjadi siap ketika aktivitas selesai. Oleh karena itu Anda dapat melewati objek `Promise<Void>` ke aktivitas lain metode klien untuk memastikan bahwa mereka menunda eksekusi sampai aktivitas asli selesai.

`Promise<T>` mengizinkan implementasi alur kerja untuk menggunakan metode klien aktivitas dan nilai-nilai kembali mereka seperti metode sinkron. Namun, Anda harus berhati-hati dalam mengakses `Promise<T>` nilai objek. Berbeda dengan jenis Java Future <T>, kerangka kerja menangani sinkronisasi untuk `Promise<T>`, bukan aplikasi. Jika Anda memanggil `Promise<T>.get` dan objeknya belum siap, `get` melempar pengecualian. Perhatikan bahwa `HelloWorldWorkflow` tidak pernah mengakses objek `Promise<T>` secara langsung; itu hanya melewati objek dari satu aktivitas ke aktivitas berikutnya. Ketika sebuah objek menjadi siap, kerangka kerja ekstrak nilai dan lolos ke metode aktivitas sebagai tipe standar.

objek `Promise<T>` harus diakses hanya dengan kode asinkron, di mana kerangka kerja menjamin bahwa objek siap dan mewakili nilai yang valid. `HelloWorldWorkflow` berkaitan dengan masalah ini dengan melewati objek `Promise<T>` untuk metode aktivitas klien. Anda dapat mengakses `Promise<T>` nilai objek dalam implementasi alur kerja Anda dengan melewatkan objek ke asynchronous workflow method (metode alur kerja asinkron), yang berperilaku seperti aktivitas. Sebagai contoh, lihat HelloWorldWorkflowAsyncAplikasi.

HelloWorldWorkflow Alur Kerja dan Implementasi Aktivitas

Alur kerja dan implementasi aktivitas memiliki kelas pekerja terkait, ActivityWorkerdan. WorkflowWorker Mereka menangani komunikasi antara Amazon SWF dan aktivitas dan implementasi alur kerja dengan polling daftar tugas Amazon SWF yang sesuai untuk tugas, mengeksekusi metode yang tepat untuk setiap tugas, dan mengelola aliran data. Untuk detail selengkapnya, lihat AWS Flow Framework Konsep Dasar: Struktur Aplikasi

Untuk mengaitkan implementasi aktivitas dan alur kerja dengan objek pekerja yang sesuai, Anda menerapkan satu atau lebih aplikasi pekerja yang:

- Mendaftar alur kerja atau aktivitas dengan Amazon SWF.
- Membuat objek pekerja dan mengasosiasikan mereka dengan alur kerja atau aktivitas implementasi pekerja.
- Mengarahkan objek pekerja untuk mulai berkomunikasi dengan Amazon SWF.

Jika Anda ingin menjalankan alur kerja dan aktivitas sebagai proses terpisah, Anda harus menerapkan alur kerja dan aktivitas yang terpisah host pekerja. Sebagai contoh, lihat HelloWorldWorkflowDistributed Aplikasi. Untuk mempermudah, HelloWorldWorkflow mengimplementasikan host pekerja tunggal yang menjalankan aktivitas dan alur kerja pekerja dalam proses yang sama, sebagai berikut:

```
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`tidak memiliki HelloWorld mitra, jadi Anda harus menambahkan kelas Java bernama `GreeterWorker` ke proyek dan menyalin kode contoh ke file itu.

Langkah pertama adalah membuat dan mengonfigurasi AmazonSimpleWorkflowClientobjek, yang memanggil metode layanan Amazon SWF yang mendasarinya. Untuk melakukannya, `GreeterWorker`:

1. Membuat ClientConfigurationobjek dan menentukan batas waktu soket 70 detik. Nilai ini menentukan lama untuk menunggu data yang akan ditransfer melalui koneksi terbuka didirikan sebelum menutup soket.
2. Membuat AWSCredentials objek Basic untuk mengidentifikasi AWS akun dan meneruskan kunci akun ke konstruktor. Untuk kenyamanan, dan untuk menghindari mengekspos mereka sebagai teks biasa dalam kode, kunci disimpan sebagai variabel lingkungan.
3. Menciptakan AmazonSimpleWorkflowClientobjek untuk mewakili alur kerja, dan meneruskan `ClientConfiguration` objek `BasicAWSCredentials` dan ke konstruktor.
4. Menetapkan URL titik akhir layanan objek klien. Amazon SWF saat ini tersedia di semua AWS wilayah.

Untuk kenyamanan, `GreeterWorker` mendefinisikan dua konstanta string.

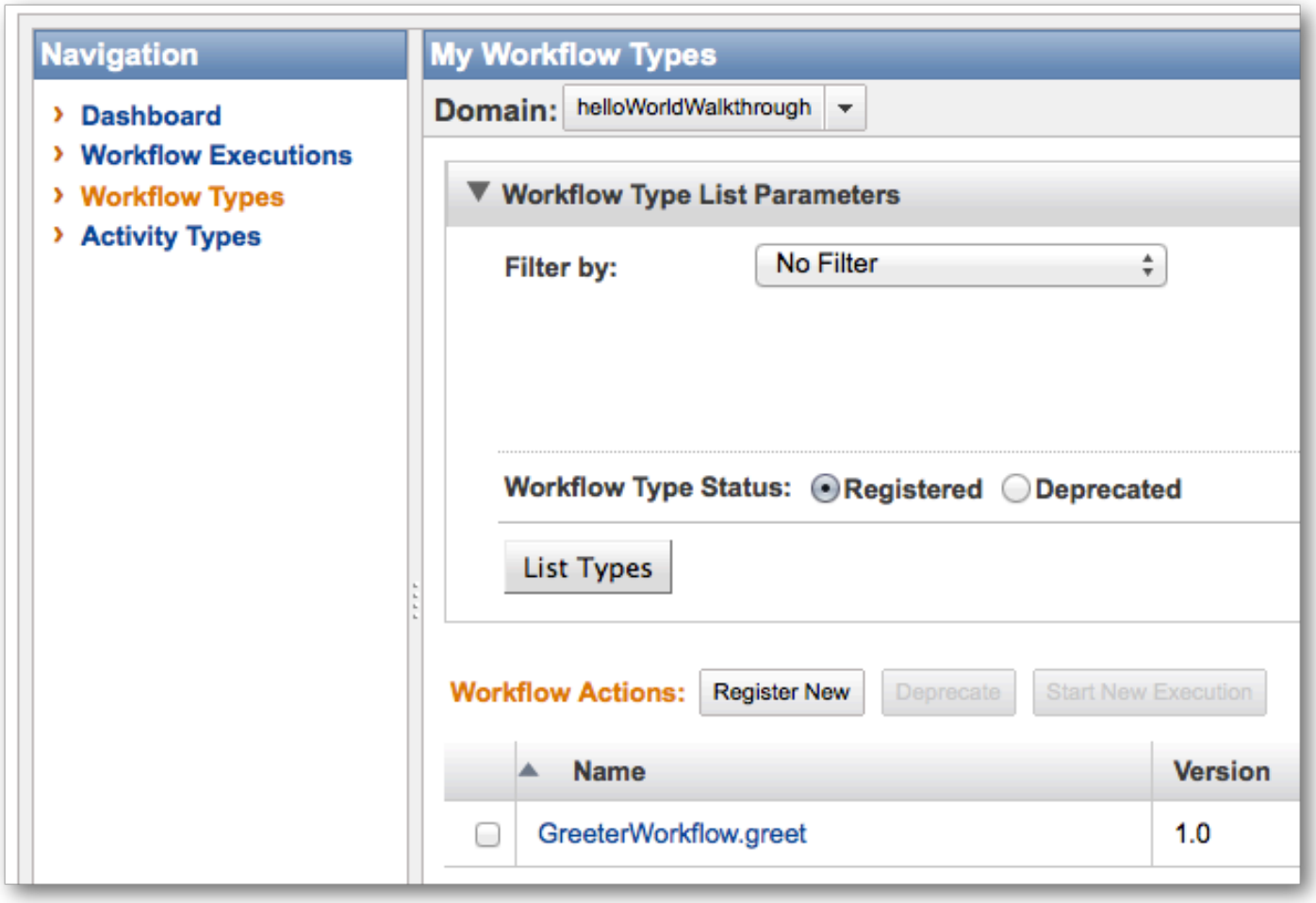
- `domain`adalah nama domain Amazon SWF alur kerja, yang Anda buat saat menyiapkan akun Amazon SWF Anda. `HelloWorldWorkflow`mengasumsikan bahwa Anda menjalankan alur kerja di domain helloWorldWalkthrough "".
- `taskListToPoll` adalah nama daftar tugas yang digunakan Amazon SWF untuk mengelola komunikasi antara alur kerja dan aktivitas pekerja. Anda dapat mengatur nama ke string yang nyaman. HelloWorldWorkflow menggunakan "HelloWorldList" untuk alur kerja dan daftar tugas aktivitas. Di belakang layar, nama-nama berakhir di namespace yang berbeda, sehingga dua daftar tugas yang berbeda.

`GreeterWorker`menggunakan konstanta string dan AmazonSimpleWorkflowClientobjek untuk membuat objek pekerja, yang mengelola interaksi antara aktivitas dan implementasi pekerja dan Amazon SWF. Khususnya, objek pekerja menangani tugas mengundi daftar tugas yang sesuai untuk tugas.

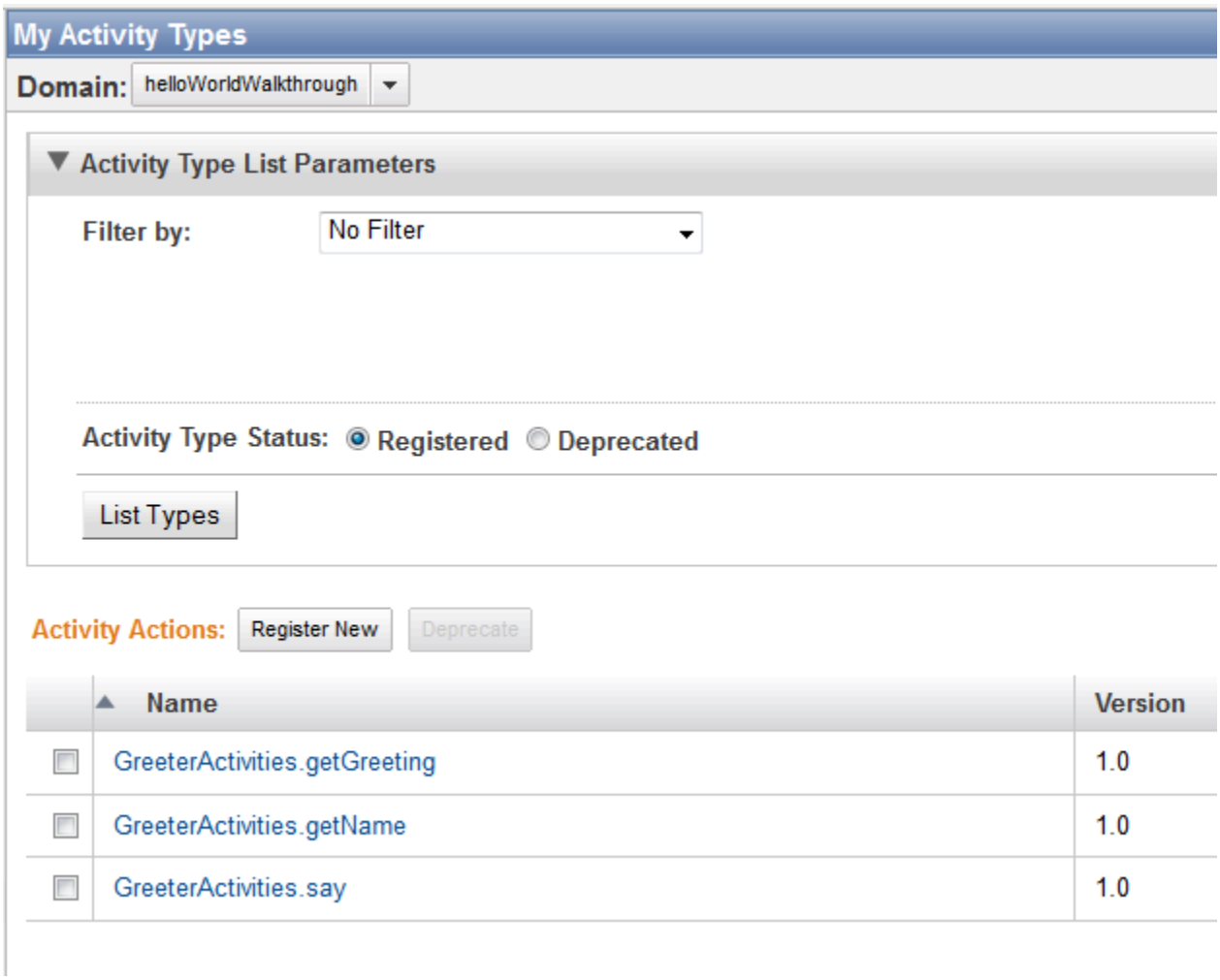
objek `GreeterWorker` membuat `ActivityWorker` dan mengkonfigurasinya untuk menangani `GreeterActivitiesImpl` dengan menambahkan instans kelas baru. `GreeterWorker` kemudian memanggil objek `ActivityWorker` metode `start`, yang mengarahkan objek untuk memulai mengundi daftar tugas aktivitas tertentu.

Objek `GreeterWorker` membuat `WorkflowWorker` dan mengkonfigurasinya untuk menangani `GreeterWorkflowImpl` dengan menambahkan nama file kelas, `GreeterWorkflowImpl.class`. Kemudian memanggil `WorkflowWorker` metode `start` objek, yang mengarahkan objek untuk memulai polling daftar tugas alur kerja yang ditentukan.

Anda dapat menjalankan `GreeterWorker` dengan berhasil pada saat ini. Ini register alur kerja dan aktivitas dengan Amazon SWF dan mulai objek pekerja polling daftar tugas masing-masing. Untuk memverifikasi ini, jalankan `GreeterWorker` dan pergi ke konsol Amazon SWF dan pilih `helloWorldWalkthrough` dari daftar domain. Jika Anda memilih Workflow Types (Jenis Alur Kerja) di panel Navigation (Navigasi), Anda akan melihat `GreeterWorkflow.greet`:



Jika Anda memilih Activity Types (Tipe Aktivitas), metode `GreeterActivities` ditampilkan:



Namun, jika Anda memilih Workflow Executions (Eksekusi Alur kerja), Anda tidak akan melihat eksekusi aktif. Meskipun alur kerja dan aktivitas pekerja membuat polling untuk tugas, kami belum memulai eksekusi alur kerja.

HelloWorldWorkflow Pemula

Bagian akhir dari teka-teki ini adalah untuk menerapkan starter alur kerja, merupakan aplikasi yang memulai eksekusi alur kerja. Status eksekusi disimpan oleh Amazon SWF, sehingga Anda dapat melihat riwayat dan status eksekusi. HelloWorldWorkflow mengimplementasikan alur kerja starter dengan memodifikasi `GreeterMain` kelas, sebagai berikut:

```
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();
   }
}
```

Objek `GreeterMain` membuat `AmazonSimpleWorkflowClient` dengan menggunakan kode yang sama seperti `GreeterWorker`. Objek kemudian menciptakan objek `GreeterWorkflowClientExternal`, yang bertindak sebagai proksi untuk alur kerja dalam banyak cara yang sama bahwa klien kegiatan dibuat di `GreeterWorkflowClientImpl` bertindak sebagai proksi untuk metode aktivitas. Daripada membuat objek klien alur kerja dengan menggunakan `new`, Anda harus:

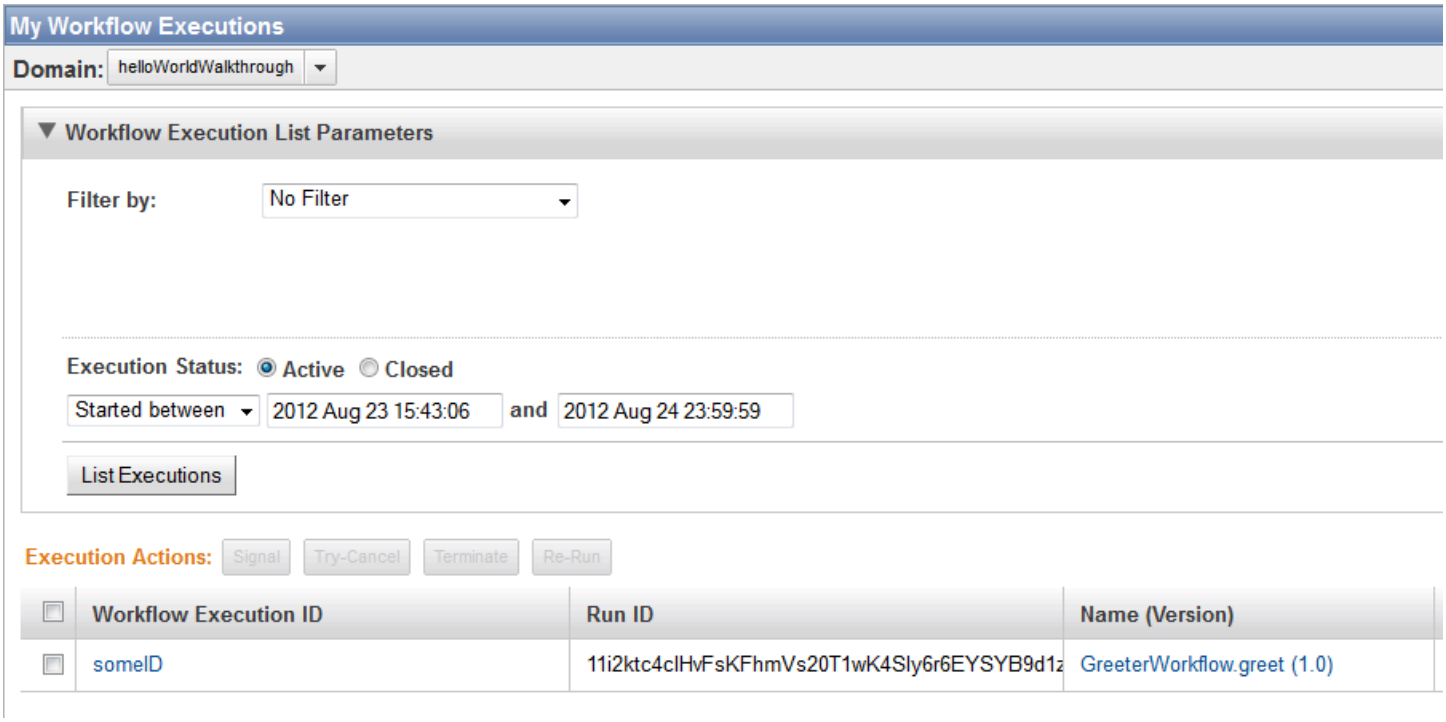
1. Membuat objek pabrik klien eksternal dan meneruskan objek `AmazonSimpleWorkflowClient` dan Amazon SWF nama domain untuk konstruktor. Objek pabrik klien dibuat oleh prosesor anotasi kerangka kerja, yang menciptakan nama objek hanya dengan menambahkan "ClientExternalFactoryImpl" ke nama antarmuka alur kerja.
2. Buat objek klien eksternal dengan memanggil `getClient` metode objek pabrik, yang menciptakan nama objek dengan menambahkan "ClientExternal" ke nama antarmuka alur kerja. Anda dapat secara opsional meneruskan `getClient` string yang Amazon SWF akan digunakan untuk mengidentifikasi instans ini dari alur kerja. Jika tidak, Amazon SWF merupakan instans alur kerja dengan menggunakan GUID yang dihasilkan.

Klien kembali dari pabrik hanya akan membuat alur kerja yang diberi nama dengan string yang diteruskan ke metode getClient, (klien yang dikembali dari pabrik sudah memiliki status di Amazon SWF). Untuk menjalankan alur kerja dengan id yang berbeda, Anda perlu kembali ke pabrik dan membuat klien baru dengan id yang berbeda telah ditentukan.

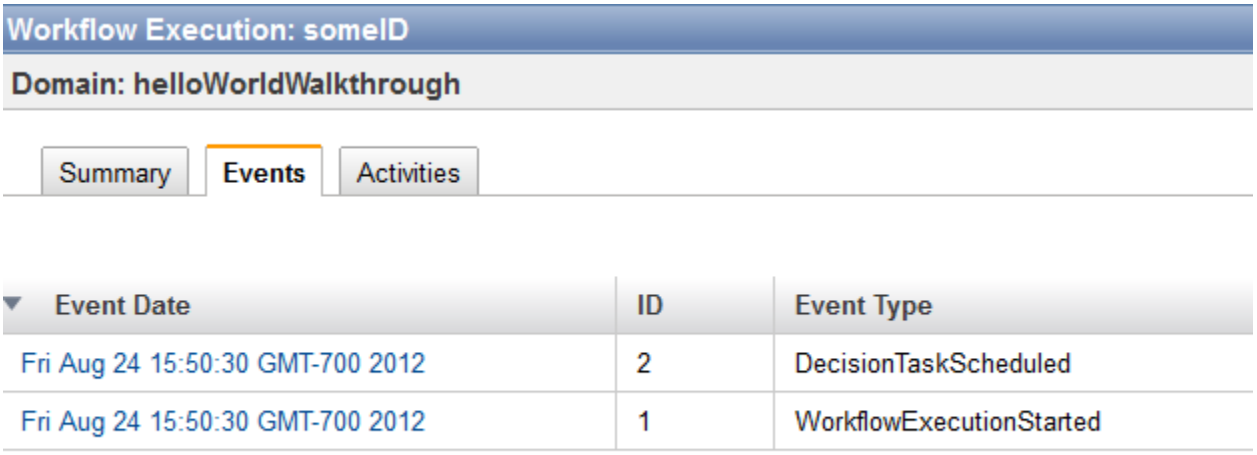
Alur kerja klien mengekspos metode `greet` yang `GreeterMain` memanggil untuk memulai alur kerja, sebagai `greet()` adalah metode yang ditentukan dengan anotasi `@Execute`.

> **Note**
>
> Prosesor anotasi juga menciptakan objek pabrik klien internal yang digunakan untuk membuat alur kerja anak. Untuk detail selengkapnya, lihat Eksekusi Alur Kerja Anak.

Matikan `GreeterWorker` untuk saat ini jika masih berjalan, dan jalankan `GreeterMain`. Anda sekarang harus melihat someID pada daftar Amazon SWF konsol eksekusi alur kerja aktif:.



Jika Anda memilih `someID` dan pilih tab Events (Peristiwa), peristiwa ditampilkan:

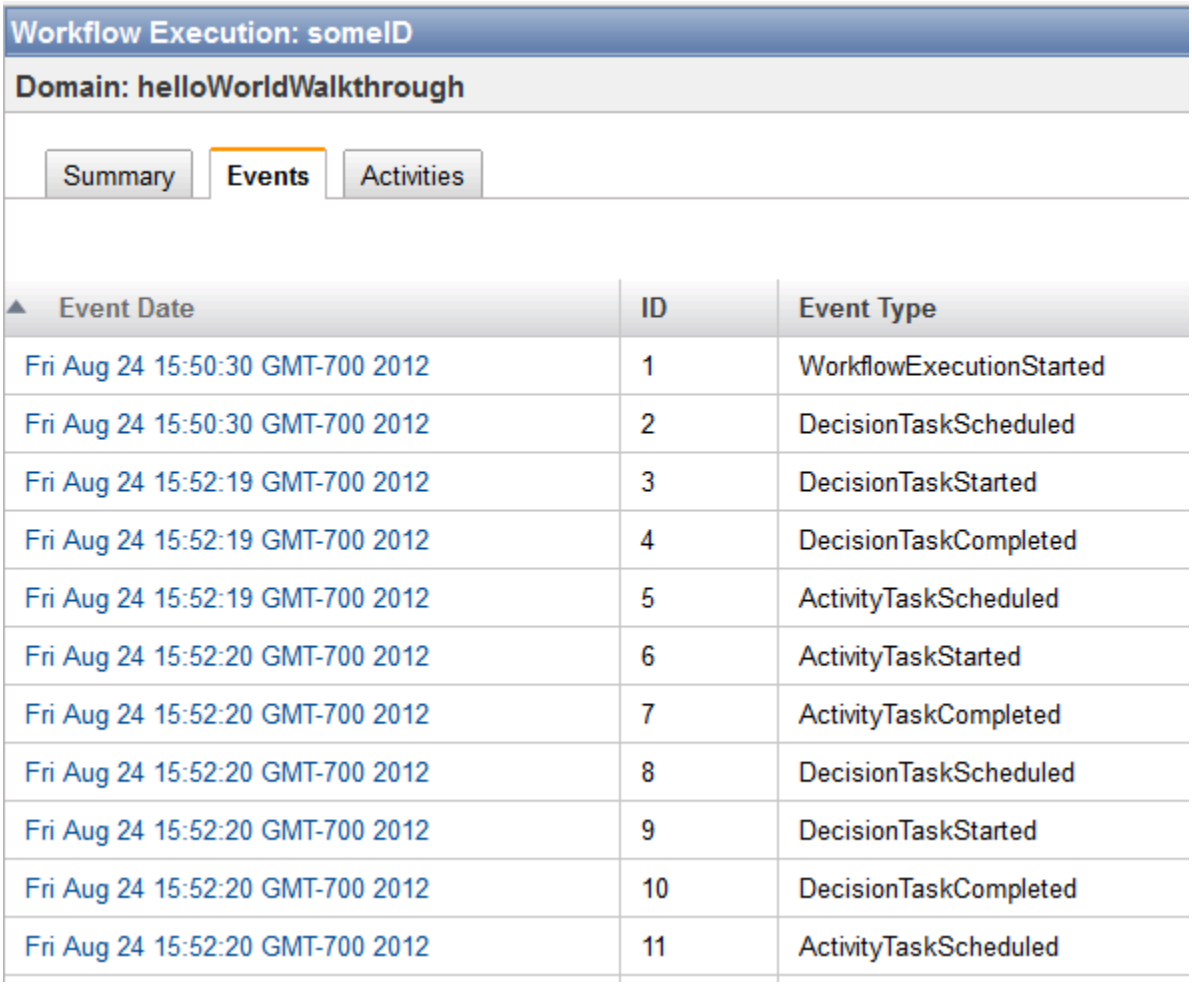


> **Note**
>
> Jika Anda memulai `GreeterWorker` sebelumnya, dan masih berjalan, Anda akan melihat daftar acara yang lebih panjang untuk alasan yang akan dibahas segera. Berhenti `GreeterWorker` dan coba jalankan `GreaterMain` lagi.

Tab Events (Peristiwa) hanya menampilkan dua acara:

- `WorkflowExecutionStarted` menunjukkan bahwa alur kerja telah mulai mengeksekusi.
- `DecisionTaskScheduled` menunjukkan bahwa Amazon SWF telah antri tugas keputusan pertama.

Alasan bahwa alur kerja diblokir pada tugas keputusan pertama adalah bahwa alur kerja didistribusikan di dua aplikasi, `GreeterMain` dan `GreeterWorker`. `GreeterMain` memulai eksekusi alur kerja, namun `GreeterWorker` tidak berjalan, sehingga pekerja tidak melakukan polling daftar dan melaksanakan tugas. Anda dapat menjalankan aplikasi baik secara independen, tetapi Anda perlu keduanya untuk eksekusi alur kerja untuk melanjutkan di luar tugas keputusan pertama. Jika Anda sekarang menjalankan `GreeterWorker`, alur kerja dan aktivitas pekerja akan mulai melakukan polling dan berbagai tugas akan selesai dengan cepat. Jika Anda sekarang memeriksa tab `Events`, batch pertama peristiwa ditampilkan.



Workflow Execution: someID

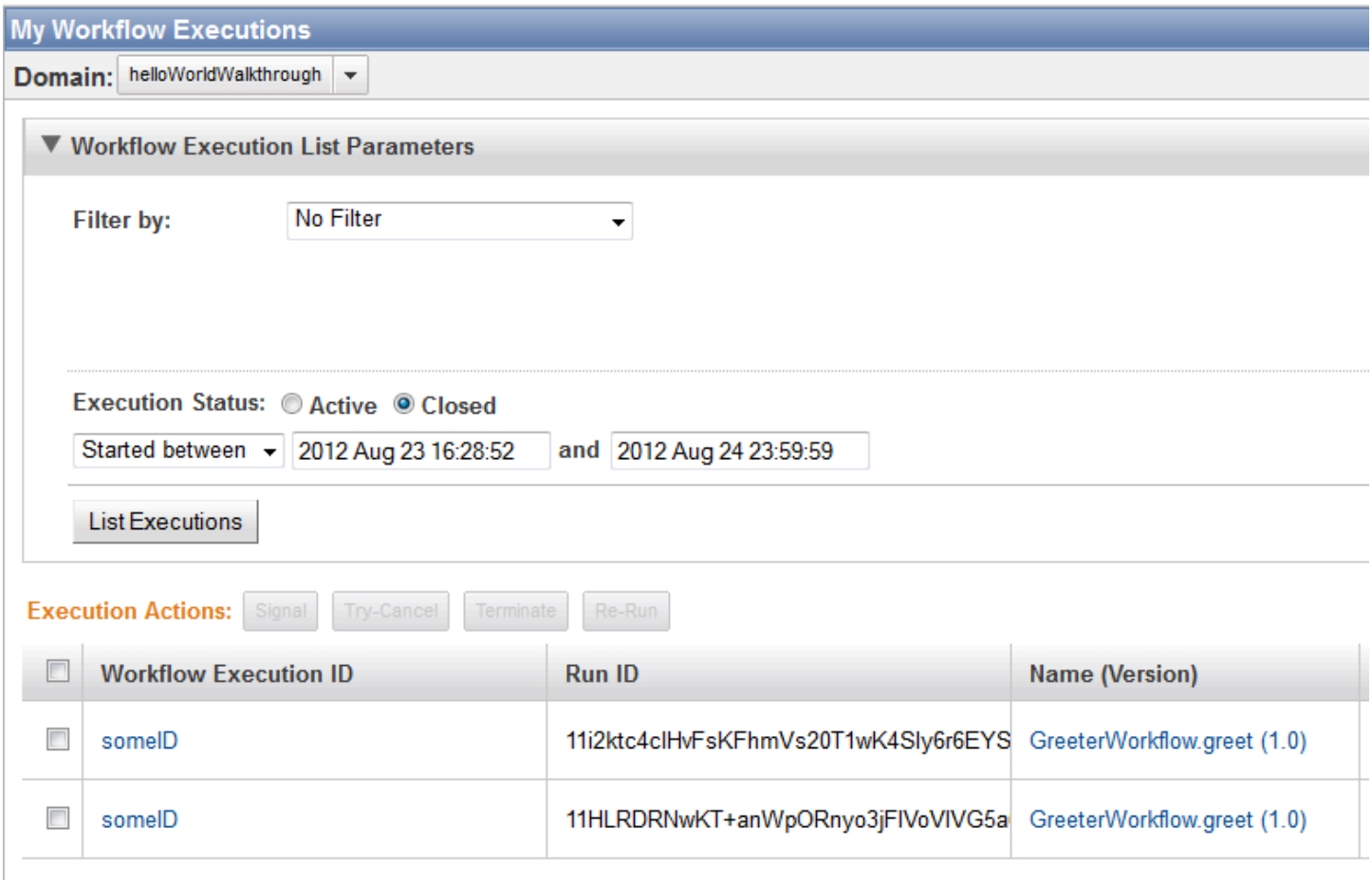
Domain: helloWorldWalkthrough

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

Anda dapat memilih acara individual untuk mendapatkan informasi lebih lanjut. Pada saat Anda selesai mencari, alur kerja seharusnya mencetak “Hello World!” ke konsol Anda.

Setelah alur kerja selesai, alur kerja tidak lagi muncul di daftar eksekusi aktif. Namun, jika Anda ingin memeriksanya, pilih tombol status eksekusi Closed (Ditutup) dan kemudian pilih List Executions (Daftar Eksekusi). Ini akan menampilkan semua instans alur kerja yang telah selesai di domain yang ditentukan (`helloWorldWalkthrough`) yang belum melebihi waktu penyimpanan mereka, yang Anda tentukan saat membuat domain.



Perhatikan bahwa setiap instans alur kerja memiliki nilai Run ID (Jalankan ID). Anda dapat menggunakan ID Alur Kerja yang sama untuk instance alur kerja yang berbeda, tetapi hanya untuk satu eksekusi aktif pada satu waktu.

HelloWorldWorkflowAsyncAplikasi

Terkadang, sebaiknya alur kerja melakukan tugas tertentu secara lokal alih-alih menggunakan aktivitas. Namun, tugas alur kerja sering melibatkan pengolahan nilai-nilai yang diwakili oleh objek `Promise<T>`. Jika Anda meneruskan objek `Promise<T>` ke metode alur kerja sinkron, metode mengeksekusi segera tetapi tidak dapat mengakses nilai objek `Promise<T>` objek siap. Anda bisa melakukan polling `Promise<T>.isReady` sampai kembali `true`, tapi itu tidak efisien dan metode mungkin memblokir untuk waktu yang lama. Pendekatan yang lebih baik adalah dengan menggunakan asynchronous method (metode asinkron).

Metode asinkron diimplementasikan seperti metode standar—sering sebagai anggota kelas implementasi alur kerja—dan berjalan dalam konteks implementasi alur kerja. Anda menetapkan

sebagai metode asinkron dengan menerapkan anotasi `@Asynchronous`, yang mengarahkan kerangka kerja untuk memperlakukannya seperti suatu aktivitas.

- Ketika implementasi alur kerja memanggil metode asinkron, alur kerja kembali dengan segera. Metode asinkron biasanya mengembalikan objek `Promise<T>`, yang telah siap ketika metode selesai.
- Jika Anda melewati metode asinkron satu atau lebih objek `Promise<T>`, metode menunda eksekusi sampai semua objek masukan siap. Sebuah metode asinkron karena itu dapat mengakses input nilai `Promise<T>` tanpa mempertaruhkan pengecualian.

> **Note**
>
> Karena cara AWS Flow Framework untuk Java mengeksekusi alur kerja, metode asinkron biasanya dijalankan beberapa kali, jadi Anda harus menggunakannya hanya untuk tugas-tugas overhead rendah cepat. Anda harus menggunakan aktivitas untuk melakukan tugas-tugas yang panjang seperti komputasi large. Untuk detail selengkapnya, lihat AWS Flow Framework Konsep Dasar: Eksekusi Terdistribusi.

Topik ini adalah panduan dari HelloWorldWorkflowAsync, versi modifikasi HelloWorldWorkflow yang menggantikan salah satu kegiatan dengan metode asinkron. Untuk mengimplementasikan aplikasi, buat salinan HelloWorld. HelloWorldWorkflow paket di direktori proyek Anda dan beri nama HelloWorld. HelloWorldWorkflowAsync.

> **Note**
>
> Topik ini didasarkan pada konsep dan file yang disajikan di topik HelloWorld Aplikasi dan HelloWorldWorkflow Aplikasi. Biasakan diri Anda dengan file dan konsep yang disajikan dalam topik tersebut sebelum melanjutkan.

Bagian berikut menjelaskan cara memodifikasi HelloWorldWorkflow kode asli untuk menggunakan metode asinkron.

HelloWorldWorkflowAsync Implementasi Kegiatan

HelloWorldWorkflowAsync mengimplementasikan antarmuka pekerja aktivitasnya di`GreeterActivities`, sebagai berikut:

```
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);
}
```

Antarmuka ini mirip dengan yang digunakan oleh HelloWorldWorkflow, dengan pengecualian berikut:

- Ini menghilangkan aktivitas `getGreeting`; tugas yang sekarang ditangani dengan metode asinkron.
- Nomor versi diatur ke 2.0. Setelah Anda mendaftarkan antarmuka aktivitas dengan Amazon SWF, Anda tidak dapat memodifikasinya kecuali Anda mengubah nomor versi.

Implementasi metode aktivitas yang tersisa identik dengan HelloWorldWorkflow. Hapus saja `getGreeting` dari `GreeterActivitiesImpl`.

HelloWorldWorkflowAsync Implementasi Alur Kerja

HelloWorldWorkflowAsync mendefinisikan antarmuka alur kerja sebagai berikut:

```
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();
}
```

Antarmuka identik dengan HelloWorldWorkflow terpisah dari nomor versi baru. Seperti aktivitas, jika Anda ingin mengubah alur kerja terdaftar, Anda harus mengubah versinya.

HelloWorldWorkflowAsync mengimplementasikan alur kerja sebagai berikut:

```
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 menggantikan `getGreeting` aktivitas dengan metode `getGreeting` asinkron tetapi metode ini bekerja dengan `greet` cara yang hampir sama:

1. Eksekusi aktivitas `getName`, yang segera mengembalikan objek `Promise<String>`, `name`, yang mewakili nama.
2. Panggil metode asinkron `getGreeting` dan meneruskan objek `name`. `getGreeting` dengan segera mengembalikan objek `Promise<String>`, `greeting`, yang mewakili ucapan.
3. Eksekusi aktivitas `say` dan meneruskan objek `greeting`.
4. Saat `getName` lengkap, `name` telah siap dan `getGreeting` menggunakan nilainya untuk membangun ucapan.
5. Saat `getGreeting` lengkap, `greeting` telah siap dan `say` mencetak string ke konsol tersebut.

Perbedaannya adalah bahwa, daripada memanggil klien aktivitas untuk mengeksekusi aktivitas `getGreeting`, menyapa panggilan metode `getGreeting` asinkron. Hasil bersih adalah sama, tetapi metode `getGreeting` bekerja agak berbeda dari aktivitas `getGreeting`.

- Pekerja alur kerja menggunakan semantik fungsi panggilan standar untuk mengeksekusi `getGreeting`. Namun, eksekusi asinkron aktivitas dimediasi oleh Amazon SWF.
- `getGreeting` menjalankan di proses implementasi alur kerja.
- `getGreeting` mengembalikan objek `Promise<String>` daripada objek `String`. Untuk mendapatkan nilai String yang dipegang oleh `Promise`, Anda menyebutnya metode `get()`. Namun, karena aktivitas dijalankan secara asinkron, nilai pengembaliannya mungkin tidak segera siap; `get()` akan memunculkan pengecualian hingga nilai pengembalian metode asinkron tersedia.

  Untuk informasi selengkapnya tentang cara kerja `Promise`, lihat AWS Flow Framework Konsep Dasar: Data Exchange Antara Aktivitas dan Alur Kerja.

`getGreeting` membuat nilai kembali dengan melewati string ucapan ke metode `Promise.asPromise`statis. Metode ini membuat objek `Promise<T>` dari jenis yang sesuai, menetapkan nilai, dan menempatkan dalam status siap.

HelloWorldWorkflowAsyncAlur Kerja dan Aktivitas Host dan Starter

HelloWorldWorkflowAsync mengimplementasikan `GreeterWorker` sebagai kelas host untuk alur kerja dan implementasi aktivitas. Ini identik dengan HelloWorldWorkflow implementasi kecuali untuk `taskListToPoll` nama, yang diatur ke "`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 mengimplementasikan alur kerja starter di`GreeterMain`; itu identik dengan implementasi. HelloWorldWorkflow

Untuk menjalankan alur kerja, jalankan `GreeterWorker` dan`GreeterMain`, seperti halnya HelloWorldWorkflow.

HelloWorldWorkflowDistributed Aplikasi

Dengan HelloWorldWorkflow dan HelloWorldWorkflowAsync, Amazon SWF memediasi interaksi antara alur kerja dan implementasi aktivitas, tetapi mereka berjalan secara lokal sebagai satu proses. `GreeterMain`dalam proses terpisah, tetapi masih berjalan pada sistem yang sama.

Sebuah fitur kunci dari Amazon SWF adalah hal itu mendukung aplikasi didistribusikan. Misalnya, Anda dapat menjalankan workflow worker di EC2 instans Amazon, starter alur kerja di komputer pusat data, dan aktivitas di komputer desktop klien. Anda bahkan dapat menjalankan aktivitas yang berbeda pada sistem yang berbeda.

HelloWorldWorkflowDistributed Aplikasi meluas HelloWorldWorkflowAsync untuk mendistribusikan aplikasi di dua sistem dan tiga proses.

- Alur kerja dan starter alur kerja berjalan sebagai proses terpisah pada satu sistem.
- Aktivitas berjalan pada sistem yang terpisah.

Untuk mengimplementasikan aplikasi, buat salinan HelloWorld. HelloWorldWorkflowAsync paket di direktori proyek Anda dan beri nama HelloWorld. HelloWorldWorkflowDistributed. Bagian berikut menjelaskan cara memodifikasi HelloWorldWorkflowAsync kode asli untuk mendistribusikan aplikasi di dua sistem dan tiga proses.

Anda tidak perlu mengubah implementasi alur kerja atau aktivitas untuk menjalankannya pada sistem yang terpisah, bahkan nomor versi tersebut. Anda juga tidak perlu memodifikasi `GreeterMain`. Yang perlu Anda ubah adalah host aktivitas dan alur kerja.

Dengan HelloWorldWorkflowAsync, satu aplikasi berfungsi sebagai alur kerja dan host aktivitas. Untuk menjalankan implementasi alur kerja dan aktivitas pada sistem terpisah, Anda harus menerapkan aplikasi terpisah. Hapus GreeterWorker dari proyek dan tambahkan dua file kelas baru, GreeterWorkflowWorker dan GreeterActivitiesWorker.

HelloWorldWorkflowDistributed mengimplementasikan kegiatan tuan rumah di GreeterActivitiesWorker, sebagai berikut:

```
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 mengimplementasikan host alur kerjanya di`GreeterWorkflowWorker`, sebagai berikut:

```
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();
   }
}
```

Catatan bahwa `GreeterActivitiesWorker` hanya `GreeterWorker` tanpa kode `WorkflowWorker` dan `GreeterWorkflowWorker` hanya `GreeterWorker` tanpa kode `ActivityWorker`.

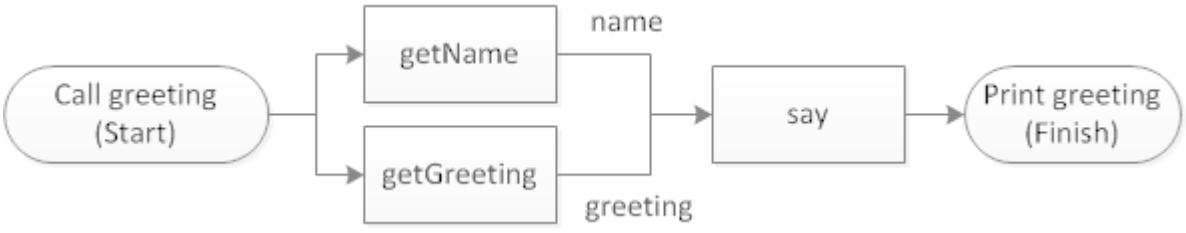
Untuk menjalankan alur kerja:

1. Buat file JAR yang dapat dijalankan dengan `GreeterActivitiesWorker` sebagai titik masuk.
2. Salin file JAR dari Langkah 1 ke sistem lain, yang dapat menjalankan sistem operasi yang mendukung Java.
3. Pastikan AWS kredensi dengan akses ke domain Amazon SWF yang sama tersedia di sistem lain.
4. Jalankan file JAR.
5. Pada sistem pengembangan Anda, gunakan Eclipse untuk menjalankan `GreeterWorkflowWorker` dan `GreeterMain`.

Selain fakta bahwa aktivitas berjalan pada sistem yang berbeda dari pekerja alur kerja dan starter alur kerja, alur kerja bekerja dengan cara yang persis sama seperti. HelloWorldAsync Namun, karena `println` memanggil yang mencetak “Hello World!” ke konsol tersebut adalah di aktivitas `say`, output akan muncul pada sistem yang menjalankan pekerja aktivitas.

HelloWorldWorkflowParallelAplikasi

Versi sebelumnya dari Hello World! semua menggunakan topologi alur kerja linear. Namun, Amazon SWF tidak terbatas pada topologi linier. HelloWorldWorkflowParallel Aplikasi ini adalah versi modifikasi HelloWorldWorkflow yang menggunakan topologi paralel, seperti yang ditunjukkan pada gambar berikut.



Dengan HelloWorldWorkflowParallel, `getName` dan `getGreeting` jalankan secara paralel dan masing-masing mengembalikan bagian dari salam. `say`kemudian menggabungkan dua string menjadi salam, dan mencetaknya ke konsol.

Untuk mengimplementasikan aplikasi, buat salinan HelloWorld. HelloWorldWorkflow paket di direktori proyek Anda dan beri nama HelloWorld. HelloWorldWorkflowParallel. Bagian berikut menjelaskan cara memodifikasi HelloWorldWorkflow kode asli untuk dijalankan `getName` dan secara `getGreeting` paralel.

HelloWorldWorkflowParallelAktivitas Pekerja

Antarmuka HelloWorldWorkflowParallel aktivitas diimplementasikan di`GreeterActivities`, seperti yang ditunjukkan pada contoh berikut.

```
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);
}
```

Antarmuka mirip dengan HelloWorldWorkflow, dengan pengecualian berikut:

- `getGreeting` tidak mengambil masukan apapun; itu hanya mengembalikan string ucapan.
- `say` mengambil dua string masukan, ucapan dan nama.
- Antarmuka memiliki nomor versi baru, yang diperlukan setiap saat Anda mengubah antarmuka yang terdaftar.

HelloWorldWorkflowParallel melaksanakan kegiatan di`GreeterActivitiesImpl`, sebagai berikut:

```
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` dan `getGreeting` sekarang hanya mengembalikan setengah dari string ucapan. `say` menggabungkan dua bagian untuk menghasilkan frase lengkap, dan mencetak ke konsol tersebut.

HelloWorldWorkflowParallelPekerja Alur Kerja

Antarmuka HelloWorldWorkflowParallel alur kerja diimplementasikan dalam`GreeterWorkflow`, sebagai berikut:

```
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();
}
```

Kelas identik dengan HelloWorldWorkflow versi, kecuali bahwa nomor versi telah diubah agar sesuai dengan pekerja aktivitas.

Alur kerja diimplementasikan dalam `GreeterWorkflowImpl`, sebagai berikut:

```
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);
    }
}
```

Sekilas, implementasi ini terlihat sangat mirip dengan HelloWorldWorkflow; tiga metode aktivitas klien mengeksekusi secara berurutan. Namun, aktivitas tidak.

- HelloWorldWorkflow diteruskan `name` ke`getGreeting`. Karena `name` adalah objek `Promise<T>`, `getGreeting` ditangguhkan melaksanakan aktivitas sampai `getName` selesai, sehingga dua aktivitas dilaksanakan secara berurutan.
- HelloWorldWorkflowParallel tidak melewati input apa pun `getName` atau`getGreeting`. Tidak ada metode yang menunda eksekusi dan metode aktivitas terkait dieksekusi segera, secara paralel.

Aktivitas `say` mengambil `greeting` dan `name` sebagai parameter input. Karena mereka adalah objek `Promise<T>`, `say` yang menunda eksekusi sampai kedua aktivitas selesai, dan kemudian membangun dan mencetak ucapan.

Perhatikan bahwa HelloWorldWorkflowParallel tidak menggunakan kode pemodelan khusus untuk menentukan topologi alur kerja. Ia melakukannya secara implisit dengan menggunakan kontrol aliran Java standar dan mengambil keuntungan dari properti objek. `Promise<T>` AWS Flow Framework untuk aplikasi Java dapat mengimplementasikan bahkan topologi kompleks hanya dengan menggunakan `Promise<T>` objek dalam hubungannya dengan konstruksi aliran kontrol Java konvensional.

HelloWorldWorkflowParallel Alur Kerja dan Aktivitas Host dan Starter

HelloWorldWorkflowParallel mengimplementasikan `GreeterWorker` sebagai kelas host untuk alur kerja dan implementasi aktivitas. Ini identik dengan HelloWorldWorkflow implementasi kecuali untuk `taskListToPoll` nama, yang diatur ke "HelloWorldParallelList”.

`HelloWorldWorkflowParallel`mengimplementasikan alur kerja starter di`GreeterMain`, dan itu identik dengan implementasi. HelloWorldWorkflow

Untuk menjalankan alur kerja, jalankan `GreeterWorker` dan `GreeterMain`, seperti dengan `HelloWorldWorkflow`.

Pengertian AWS Flow Framework untuk Java

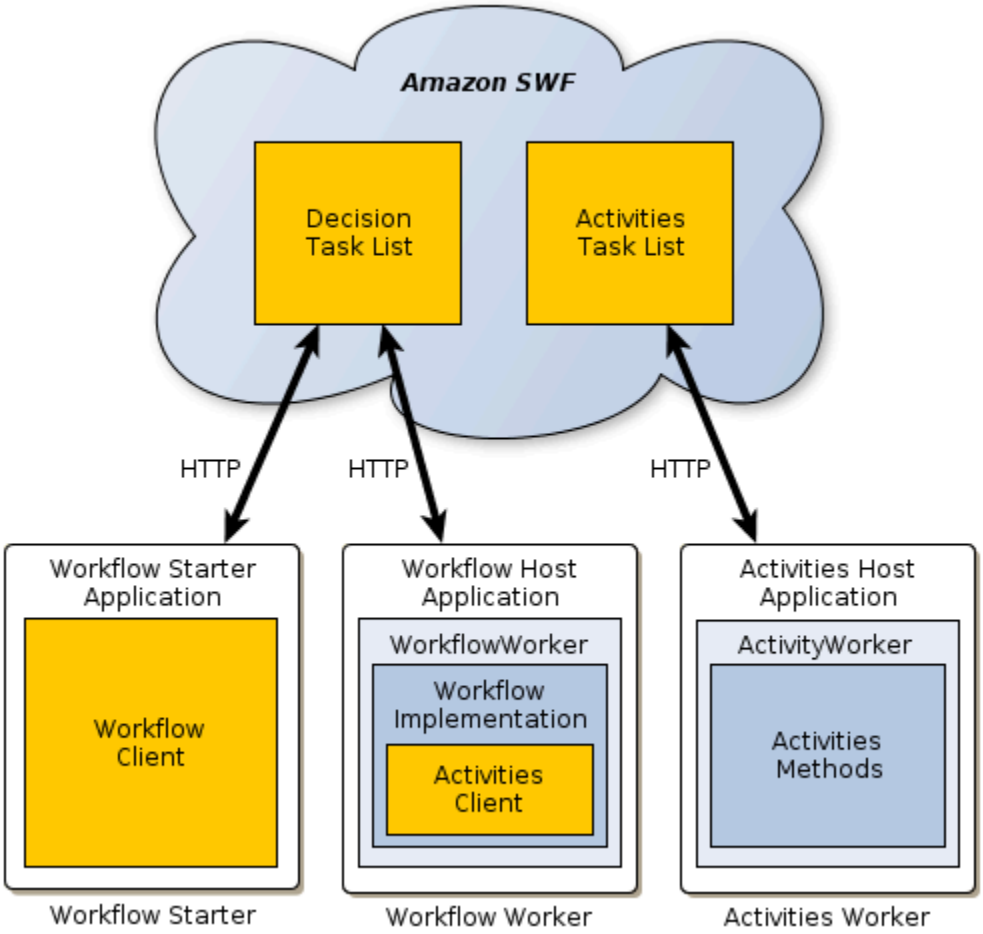
The AWS Flow Framework for Java bekerja dengan Amazon SWF untuk memudahkan pembuatan aplikasi yang dapat diskalakan dan toleran kesalahan untuk melakukan tugas asinkron yang mungkin berjalan lama, jarak jauh, atau keduanya. "Hello World!" contoh dalam Apa yang dimaksud dengan AWS Flow Framework Java? memperkenalkan dasar-dasar bagaimana menggunakan untuk AWS Flow Framework mengimplementasikan aplikasi alur kerja dasar. Bagian ini memberikan informasi konseptual tentang cara kerja AWS Flow Framework aplikasi. Bagian pertama merangkum struktur dasar AWS Flow Framework aplikasi, dan bagian yang tersisa memberikan detail lebih lanjut tentang cara kerja AWS Flow Framework aplikasi.

Topik

AWS Flow Framework Konsep Dasar: Struktur Aplikasi

Secara konseptual, AWS Flow Framework aplikasi terdiri dari tiga komponen dasar: permulaan alur kerja, pekerja alur kerja, dan pekerja aktivitas. Satu atau beberapa aplikasi host bertanggung jawab untuk mendaftarkan pekerja (alur kerja dan aktivitas) dengan Amazon SWF, memulai pekerja, dan menangani pembersihan. Para pekerja menangani mekanisme pelaksanaan alur kerja dan dapat diimplementasikan pada beberapa host.

Diagram ini merupakan AWS Flow Framework aplikasi dasar:



> **Note**
>
> Menerapkan komponen-komponen ini dalam tiga aplikasi terpisah secara konseptual nyaman, tetapi Anda dapat membuat aplikasi untuk menerapkan fungsi ini dalam berbagai cara. Misalnya, Anda dapat menggunakan aplikasi host tunggal untuk pekerja aktivitas dan alur kerja, atau menggunakan host aktivitas dan alur kerja terpisah. Anda juga dapat memiliki beberapa pekerja aktivitas, masing-masing menangani serangkaian aktivitas yang berbeda pada host yang terpisah, dan seterusnya.

Ketiga AWS Flow Framework komponen berinteraksi secara tidak langsung dengan mengirimkan permintaan HTTP ke Amazon SWF, yang mengelola permintaan. Amazon SWF melakukan hal berikut:

- Mempertahankan satu atau beberapa daftar tugas keputusan, yang menentukan langkah selanjutnya yang harus dilakukan oleh pekerja alur kerja.
- Mempertahankan satu atau beberapa daftar tugas aktivitas, yang menentukan tugas mana yang akan dilakukan oleh pekerja aktivitas.
- Mempertahankan step-by-step riwayat rinci eksekusi alur kerja.

Dengan itu AWS Flow Framework, kode aplikasi Anda tidak perlu berurusan langsung dengan banyak detail yang ditunjukkan pada gambar, seperti mengirim permintaan HTTP ke Amazon SWF. Anda cukup memanggil AWS Flow Framework metode dan kerangka kerja menangani detail di balik layar.

Peran Pekerja Aktivitas

Pekerja aktivitas melakukan berbagai tugas yang harus diselesaikan oleh alur kerja. Terdiri dari:

- Implementasi aktivitas, yang mencakup serangkaian metode aktivitas yang melakukan tugas tertentu untuk alur kerja.
- ActivityWorkerObjek, yang menggunakan permintaan polling panjang HTTP untuk melakukan polling Amazon SWF untuk tugas aktivitas yang akan dilakukan. Saat tugas diperlukan, Amazon SWF merespons permintaan tersebut dengan mengirimkan informasi yang diperlukan untuk melakukan tugas tersebut. ActivityWorkerObjek kemudian memanggil metode aktivitas yang sesuai, dan mengembalikan hasilnya ke Amazon SWF.

Peran Pekerja Alur Kerja

Pekerja alur kerja mengatur eksekusi berbagai aktivitas, mengelola aliran data, dan menangani aktivitas yang gagal. Terdiri dari:

- Implementasi alur kerja, yang mencakup logika orkestrasi aktivitas, menangani aktivitas yang gagal, dan sebagainya.
- Klien aktivitas, yang berfungsi sebagai proksi untuk pekerja aktivitas dan memungkinkan pekerja alur kerja menjadwalkan aktivitas untuk dieksekusi secara asinkron.
- Sebuah WorkflowWorkerobjek, yang menggunakan permintaan polling panjang HTTP untuk polling Amazon SWF untuk tugas keputusan. Jika ada tugas di daftar tugas alur kerja, Amazon SWF merespons permintaan tersebut dengan mengembalikan informasi yang diperlukan untuk

melakukan tugas tersebut. Kerangka kerja kemudian mengeksekusi alur kerja untuk melakukan tugas dan mengembalikan hasilnya ke Amazon SWF.

Peran Pemula Alur Kerja

Pemula alur kerja memulai instans alur kerja, juga disebut sebagai eksekusi alur kerja, dan dapat berinteraksi dengan instans selama eksekusi untuk meneruskan data tambahan ke pekerja alur kerja atau mendapatkan status alur kerja saat ini.

Starter alur kerja menggunakan klien alur kerja untuk memulai eksekusi alur kerja, berinteraksi dengan alur kerja sesuai kebutuhan selama eksekusi, dan menangani pembersihan. Starter alur kerja bisa berupa aplikasi yang dijalankan secara lokal, aplikasi web, atau bahkan aplikasi. AWS CLI AWS Management Console

Bagaimana Amazon SWF Berinteraksi dengan Aplikasi Anda

Amazon SWF memediasi interaksi antara komponen alur kerja dan mempertahankan riwayat alur kerja yang mendetail. Amazon SWF tidak memulai komunikasi dengan komponen; itu menunggu permintaan HTTP dari komponen dan mengelola permintaan sesuai kebutuhan. Sebagai contoh:

- Jika permintaan berasal dari pekerja, polling untuk tugas yang tersedia, Amazon SWF merespons langsung ke pekerja jika tugas tersedia. Untuk informasi selengkapnya tentang cara kerja polling, lihat Polling untuk Tugas di Panduan Developer Amazon Simple Workflow Service.
- Jika permintaan adalah notifikasi dari pekerja aktivitas bahwa tugas telah selesai, Amazon SWF mencatat informasi dalam riwayat eksekusi dan menambahkan tugas ke daftar tugas keputusan untuk memberi tahu pekerja alur kerja bahwa tugas telah selesai, memungkinkannya untuk melanjutkan ke langkah selanjutnya.
- Jika permintaan berasal dari pekerja alur kerja untuk menjalankan aktivitas, Amazon SWF mencatat informasi dalam riwayat eksekusi dan menambahkan tugas ke daftar tugas aktivitas untuk mengarahkan pekerja aktivitas agar menjalankan metode aktivitas yang sesuai.

Pendekatan ini memungkinkan pekerja untuk berjalan pada sistem apa pun dengan koneksi Internet, termasuk EC2 instans Amazon, pusat data perusahaan, komputer klien, dan sebagainya. Mereka bahkan tidak harus menjalankan sistem operasi yang sama. Karena permintaan HTTP berasal dari pekerja, tidak diperlukan port yang terlihat secara eksternal; pekerja dapat berjalan di belakang firewall.

Untuk Informasi Selengkapnya

Untuk diskusi yang lebih menyeluruh tentang cara kerja Amazon SWF, lihat Panduan Developer Amazon Simple Workflow Service.

AWS Flow Framework Konsep Dasar: Eksekusi yang Andal

Aplikasi terdistribusi asinkron harus menangani masalah keandalan yang tidak dihadapi oleh aplikasi konvensional, termasuk:

- Cara menyediakan komunikasi yang andal antara komponen terdistribusi asinkron, seperti komponen yang berjalan lama pada sistem jarak jauh.
- Cara memastikan bahwa hasil tidak hilang jika komponen gagal atau terputus, terutama untuk aplikasi yang berjalan lama.
- Cara menangani komponen terdistribusi yang gagal.

Aplikasi dapat mengandalkan AWS Flow Framework dan Amazon SWF untuk mengelola masalah ini. Kami akan mengeksplorasi bagaimana Amazon SWF menyediakan mekanisme untuk memastikan bahwa alur kerja Anda beroperasi dengan andal dan dengan cara yang dapat diprediksi, bahkan saat berjalan lama dan bergantung pada tugas asinkron yang dilakukan secara komputasi dan dengan interaksi manusia.

Menyediakan Komunikasi yang Andal

AWS Flow Framework menyediakan komunikasi yang andal antara pekerja alur kerja dan pekerja aktivitasnya dengan menggunakan Amazon SWF untuk mengirimkan tugas ke pekerja aktivitas terdistribusi dan mengembalikan hasilnya ke pekerja alur kerja. Amazon SWF menggunakan metode berikut untuk memastikan komunikasi yang andal antara pekerja dan aktivitasnya:

- Amazon SWF menyimpan secara tahan lama aktivitas terjadwal dan tugas alur kerja serta menjamin bahwa tugas tersebut akan dilakukan paling banyak satu kali.
- Amazon SWF menjamin bahwa tugas aktivitas akan berhasil diselesaikan dan mengembalikan hasil yang valid atau akan memberi tahu pekerja alur kerja bahwa tugas tersebut gagal.
- Amazon SWF menyimpan secara tahan lama setiap hasil aktivitas yang diselesaikan atau, untuk aktivitas yang gagal, Amazon SWF menyimpan informasi kesalahan yang relevan.

AWS Flow Framework Kemudian menggunakan hasil aktivitas dari Amazon SWF untuk menentukan cara melanjutkan eksekusi alur kerja.

Memastikan bahwa Hasil Tidak Hilang

Mempertahankan Riwayat Alur Kerja

Aktivitas yang melakukan operasi penambangan data pada petabyte data mungkin membutuhkan waktu berjam-jam untuk diselesaikan, dan aktivitas yang mengarahkan pekerja manusia untuk melakukan tugas yang kompleks mungkin membutuhkan waktu berhari-hari, atau bahkan berminggu-minggu untuk diselesaikan!

Untuk mengakomodasi skenario seperti ini, AWS Flow Framework alur kerja dan aktivitas dapat memakan waktu lama untuk diselesaikan: hingga batas satu tahun untuk eksekusi alur kerja. Eksekusi proses yang berjalan lama dengan andal memerlukan mekanisme untuk menyimpan riwayat eksekusi alur kerja tahan lama secara berkelanjutan.

AWS Flow Framework Menangani ini dengan bergantung pada Amazon SWF, yang mempertahankan riwayat berjalan setiap instance alur kerja. Riwayat alur kerja menyediakan catatan lengkap dan otoritatif tentang kemajuan alur kerja, termasuk semua alur kerja dan tugas aktivitas yang telah dijadwalkan dan diselesaikan, dan informasi yang dikembalikan oleh aktivitas yang diselesaikan atau gagal.

AWS Flow Framework aplikasi biasanya tidak perlu berinteraksi dengan riwayat alur kerja secara langsung, meskipun mereka dapat mengaksesnya jika perlu. Untuk sebagian besar tujuan, aplikasi dapat dengan mudah membiarkan kerangka kerja berinteraksi dengan riwayat alur kerja di belakang layar. Untuk diskusi lengkap tentang riwayat alur kerja, lihat Riwayat Alur Kerja di Panduan Developer Amazon Simple Workflow Service.

Eksekusi Stateless

Riwayat eksekusi memungkinkan pekerja alur kerja menjadi stateless. Jika Anda memiliki beberapa instans alur kerja atau pekerja aktivitas, pekerja mana pun dapat melakukan tugas apa pun. Pekerja menerima semua informasi status yang diperlukan untuk melakukan tugas dari Amazon SWF.

Pendekatan ini membuat alur kerja lebih dapat diandalkan. Misalnya, jika pekerja aktivitas gagal, Anda tidak perlu memulai ulang alur kerja. Cukup mulai ulang pekerja dan itu akan mulai mengumpulkan daftar tugas dan memproses tugas apa pun yang ada di daftar, terlepas dari kapan kegagalan terjadi. Anda dapat membuat alur kerja Anda secara keseluruhan toleran terhadap

kesalahan dengan menggunakan dua atau beberapa alur kerja dan pekerja aktivitas, mungkin pada sistem yang terpisah. Kemudian, jika salah satu pekerja gagal, pekerja lainnya akan terus menangani tugas terjadwal tanpa gangguan dalam kemajuan alur kerja.

Menangani Komponen Terdistribusi yang Gagal

Aktivitas sering gagal karena alasan singkat, seperti pemutusan hubungan singkat, jadi strategi umum untuk menangani aktivitas yang gagal adalah mencoba kembali aktivitas tersebut. Alih-alih menangani proses coba lagi dengan menerapkan strategi penyampaian pesan yang kompleks, aplikasi dapat bergantung pada AWS Flow Framework. Ini menyediakan beberapa mekanisme untuk mencoba kembali aktivitas yang gagal, dan menyediakan mekanisme penanganan pengecualian bawaan yang bekerja dengan eksekusi tugas yang tidak sinkron dan terdistribusi dalam alur kerja.

AWS Flow Framework Konsep Dasar: Eksekusi Terdistribusi

Instans alur kerja pada dasarnya adalah utas eksekusi virtual yang dapat menjangkau aktivitas dan logika orkestrasi yang berjalan di beberapa komputer jarak jauh. Amazon SWF dan AWS Flow Framework fungsinya sebagai sistem operasi yang mengelola instance alur kerja pada CPU virtual dengan:

- Mempertahankan status eksekusi setiap instans.
- Beralih antar instans.
- Melanjutkan eksekusi sebuah instans pada saat itu dimatikan.

Memutar Ulang Alur Kerja

Karena aktivitas dapat berjalan lama, tidak diinginkan jika alur kerja diblokir begitu saja hingga selesai. Sebagai gantinya, AWS Flow Framework mengelola eksekusi alur kerja dengan menggunakan mekanisme pemutaran ulang, yang bergantung pada riwayat alur kerja yang dikelola oleh Amazon SWF untuk menjalankan alur kerja dalam episode.

Setiap episode memutar ulang logika alur kerja dengan cara yang mengeksekusi setiap aktivitas hanya sekali, dan memastikan bahwa aktivitas dan metode asinkron tidak dijalankan hingga objek Promise mereka siap.

Pemula alur kerja memulai episode putar ulang pertama saat memulai eksekusi alur kerja. Kerangka kerja memanggil metode titik masuk alur kerja dan:

1. Mengeksekusi semua tugas alur kerja yang tidak bergantung pada penyelesaian aktivitas, termasuk memanggil semua metode klien aktivitas.
2. Memberi Amazon SWF daftar tugas aktivitas yang dijadwalkan untuk dieksekusi. Untuk episode pertama, daftar ini hanya terdiri dari aktivitas-aktivitas yang tidak bergantung pada Promise dan dapat segera dieksekusi.
3. Memberi tahu Amazon SWF bahwa episode telah selesai.

Amazon SWF menyimpan tugas aktivitas dalam riwayat alur kerja dan menjadwalkannya untuk dieksekusi dengan menempatkannya di daftar tugas aktivitas. Pekerja aktivitas mengumpulkan daftar tugas dan mengeksekusi tugas.

Saat pekerja aktivitas menyelesaikan tugas, ia mengembalikan hasilnya ke Amazon SWF, yang mencatatnya dalam riwayat eksekusi alur kerja dan menjadwalkan tugas alur kerja baru untuk pekerja alur kerja dengan menempatkannya di daftar tugas alur kerja. Pekerja alur kerja polling daftar tugas dan ketika menerima tugas, menjalankan episode putar ulang berikutnya, sebagai berikut:

1. Kerangka kerja menjalankan metode titik masuk alur kerja lagi dan:
   - Mengeksekusi semua tugas alur kerja yang tidak bergantung pada penyelesaian aktivitas, termasuk memanggil semua metode klien aktivitas. Namun, kerangka kerja memeriksa riwayat eksekusi dan tidak menjadwalkan tugas aktivitas duplikat.
   - Memeriksa riwayat untuk melihat tugas aktivitas mana yang telah diselesaikan dan menjalankan metode alur kerja asinkron yang bergantung pada aktivitas tersebut.
2. Ketika semua tugas alur kerja yang dapat dijalankan telah selesai, kerangka kerja melaporkan kembali ke Amazon SWF:
   - Ini memberi Amazon SWF daftar aktivitas apa pun yang objek `Promise<T>` inputnya telah siap sejak episode terakhir dan dapat dijadwalkan untuk dieksekusi.
   - Jika episode tidak menghasilkan tugas aktivitas tambahan tetapi masih ada aktivitas yang belum selesai, kerangka kerja akan memberi tahu Amazon SWF bahwa episode telah selesai. Kemudian menunggu aktivitas lain selesai, memulai episode putar ulang berikutnya.
   - Jika episode tidak menghasilkan tugas aktivitas tambahan dan semua aktivitas telah selesai, kerangka kerja akan memberi tahu Amazon SWF bahwa eksekusi alur kerja telah selesai.

Untuk contoh perilaku putar ulang, lihat AWS Flow Framework untuk Java Replay Behavior.

Metode Alur Kerja Putar Ulang dan Asinkron

Metode alur kerja asinkron sering digunakan seperti aktivitas, karena metode menunda eksekusi hingga semua objek `Promise<T>` input siap. Namun, mekanisme putar ulang menangani metode asinkron secara berbeda dari aktivitas.

- Putar ulang tidak menjamin bahwa metode asinkron akan dijalankan hanya sekali. Ini menunda eksekusi pada metode asinkron hingga objek Promise inputnya siap, tetapi kemudian mengeksekusi metode itu untuk semua episode berikutnya.
- Ketika metode asinkron selesai, itu tidak memulai episode baru.

Contoh memutar ulang alur kerja asinkron disediakan di AWS Flow Framework untuk Java Replay Behavior.

Implementasi Putar Ulang dan Alur Kerja

Untuk sebagian besar, Anda tidak perlu khawatir dengan detail mekanisme putar ulang. Ini pada dasarnya adalah sesuatu yang terjadi di belakang layar. Namun, putar ulang memiliki dua implikasi penting untuk implementasi alur kerja Anda.

- Jangan gunakan metode alur kerja untuk melakukan tugas yang sudah berjalan lama, karena putar ulang akan mengulangi tugas tersebut beberapa kali. Bahkan metode alur kerja asinkron biasanya berjalan lebih dari sekali. Sebaliknya, gunakan aktivitas untuk tugas yang berjalan lama; putar ulang mengeksekusi aktivitas hanya sekali.
- Logika alur kerja Anda harus sepenuhnya deterministik; setiap episode harus mengambil jalur aliran kontrol yang sama. Misalnya, jalur aliran kontrol tidak boleh bergantung pada waktu saat ini. Untuk penjelasan mendetail tentang putar ulang dan persyaratan determinisme, lihat Nondeterminisme.

AWS Flow Framework Konsep Dasar: Daftar Tugas dan Eksekusi Tugas

Amazon SWF mengelola alur kerja dan tugas aktivitas dengan mempostingnya ke daftar bernama. Amazon SWF mempertahankan setidaknya dua daftar tugas, satu untuk pekerja alur kerja dan satu untuk pekerja aktivitas.

> **Note**
>
> Anda dapat menentukan daftar tugas sebanyak yang Anda butuhkan, dengan pekerja berbeda yang ditugaskan ke setiap daftar. Tidak ada batasan jumlah daftar tugas. Anda biasanya menentukan daftar tugas pekerja di aplikasi host pekerja saat Anda membuat objek pekerja.

Kutipan berikut dari aplikasi host `HelloWorldWorkflow` membuat pekerja aktivitas baru dan menetapkannya ke daftar tugas aktivitas `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();
    ...
  }
}
```

Secara default, Amazon SWF menjadwalkan tugas pekerja dalam daftar `HelloWorldList`. Kemudian pekerja tersebut melakukan polling terhadap daftar tugas tersebut. Anda dapat menetapkan nama apa pun ke daftar tugas. Anda bahkan dapat menggunakan nama yang sama untuk alur kerja dan daftar aktivitas. Secara internal, Amazon SWF menempatkan nama daftar tugas alur kerja dan aktivitas di namespace yang berbeda, sehingga kedua daftar tersebut akan berbeda.

Jika Anda tidak menentukan daftar tugas, daftar default akan AWS Flow Framework menentukan daftar default saat pekerja mendaftarkan jenis dengan Amazon SWF. Untuk informasi selengkapnya, lihat Alur Kerja dan Jenis Aktivitas Pendaftaran.

Terkadang ada gunanya meminta pekerja atau grup pekerja tertentu melakukan tugas tertentu. Misalnya, alur kerja pemrosesan citra mungkin menggunakan satu aktivitas untuk mengunduh citra dan aktivitas lain untuk memproses citra. Ini lebih efisien untuk melakukan kedua tugas pada sistem yang sama, dan menghindari overhead mentransfer file besar melalui jaringan.

Untuk mendukung skenario tersebut, Anda dapat secara eksplisit menentukan daftar tugas saat Anda memanggil metode klien aktivitas dengan menggunakan kelebihan yang menyertakan parameter `schedulingOptions`. Anda menentukan daftar tugas dengan meneruskan metode `ActivitySchedulingOptions` objek yang dikonfigurasi dengan tepat.

Misalnya, anggaplah aktivitas `say` aplikasi `HelloWorldWorkflow` yang di-host oleh pekerja aktivitas yang berbeda dari `getName` dan `getGreeting`. Contoh berikut menunjukkan cara memastikan bahwa `say`menggunakan daftar tugas yang sama seperti `getName` dan `getGreeting`, bahkan jika mereka awalnya ditetapkan ke daftar yang berbeda.

```
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);
  }
}
```

Metode `runSay` asinkron mendapatkan daftar tugas `getGreeting` dari objek kliennya. Kemudian ia membuat dan mengonfigurasi objek `ActivitySchedulingOptions` yang memastikan bahwa `say` melakukan polling daftar tugas yang sama dengan `getGreeting` .

> **Note**
>
> Saat Anda meneruskan parameter `schedulingOptions` ke metode klien aktivitas, itu menimpa daftar tugas asli hanya untuk eksekusi aktivitas tersebut. Jika Anda memanggil metode klien aktivitas lagi tanpa menentukan daftar tugas, Amazon SWF menetapkan tugas ke daftar asli, dan pekerja aktivitas akan melakukan polling daftar itu.

AWS Flow Framework Konsep Dasar: Aplikasi yang Dapat Diskalakan

Amazon SWF memiliki dua fitur kunci yang memudahkan penskalaan aplikasi alur kerja untuk menangani beban saat ini:

- Riwayat eksekusi alur kerja yang lengkap, yang memungkinkan Anda untuk menerapkan aplikasi stateless.
- Penjadwalan tugas yang digabungkan secara longgar dengan eksekusi tugas, yang memudahkan penskalaan aplikasi Anda untuk memenuhi tuntutan saat ini.

Amazon SWF menjadwalkan tugas dengan mempostingnya ke daftar tugas yang dialokasikan secara dinamis, bukan dengan berkomunikasi langsung dengan alur kerja dan pekerja aktivitas. Sebagai gantinya, para pekerja menggunakan permintaan HTTP untuk mengumpulkan daftar tugas mereka masing-masing. Pendekatan ini secara longgar menggabungkan penjadwalan tugas dengan pelaksanaan tugas dan memungkinkan pekerja untuk berjalan pada sistem yang sesuai, termasuk EC2 instans Amazon, pusat data perusahaan, komputer klien, dan sebagainya. Karena permintaan HTTP berasal dari pekerja, tidak perlu port yang terlihat secara eksternal, yang memungkinkan pekerja untuk bahkan berjalan di belakang firewall.

Mekanisme polling panjang yang digunakan pekerja untuk melakukan polling untuk tugas memastikan bahwa pekerja tidak kelebihan beban. Bahkan jika ada lonjakan tugas terjadwal, pekerja menarik tugas dengan kecepatan mereka sendiri. Namun, karena pekerja stateless, Anda dapat menskalakan aplikasi secara dinamis untuk memenuhi peningkatan beban dengan memulai instans pekerja tambahan. Bahkan jika mereka berjalan pada sistem yang berbeda, setiap instans mengumpulkan daftar tugas yang sama dan instans pekerja pertama yang tersedia menjalankan setiap tugas, terlepas dari di mana pekerja berada atau kapan dimulai. Ketika beban menurun, Anda dapat mengurangi jumlah pekerja yang sesuai.

AWS Flow Framework Konsep Dasar: Data Exchange Antara Aktivitas dan Alur Kerja

Saat Anda memanggil metode klien aktivitas asinkron, itu segera mengembalikan objek Promise (juga dikenal sebagai Future), yang mewakili nilai kembali metode aktivitas. Awalnya, Promise dalam keadaan tidak siap dan nilai kembali tidak terdefinisi. Setelah metode aktivitas menyelesaikan

tugasnya dan mengembalikan, kerangka kerja menyusun nilai kembalian di seluruh jaringan ke pekerja alur kerja, yang memberikan nilai ke Promise dan menempatkan objek dalam keadaan siap.

Bahkan jika metode aktivitas tidak memiliki nilai kembali, Anda masih dapat menggunakan Promise untuk mengelola eksekusi alur kerja. Jika Anda meneruskan Promise yang dikembalikan ke metode klien aktivitas atau metode alur kerja asinkron, itu akan menunda eksekusi hingga objek siap.

Jika Anda meneruskan satu atau beberapa Promise ke metode klien aktivitas, kerangka kerja akan mengantrekan tugas tetapi menunda penjadwalannya hingga semua objek siap. Kerangka kerja kemudian mengekstrak data dari setiap Promise dan menyusunnya di internet ke pekerja aktivitas, yang meneruskannya ke metode aktivitas sebagai tipe standar.

> **Note**
>
> Jika Anda perlu mentransfer data dalam jumlah besar antara alur kerja dan pekerja aktivitas, pendekatan yang lebih disukai adalah menyimpan data di lokasi yang nyaman dan hanya meneruskan informasi pengambilan. Misalnya, Anda dapat menyimpan data dalam bucket Amazon S3 dan meneruskan URL terkait.

<T>Jenis Janji

Tipe `Promise<T>` mirip dalam beberapa hal dengan tipe `Future<T>` Java. Kedua jenis mewakili nilai-nilai yang dikembalikan oleh metode asinkronus dan awalnya tidak terdefinisi. Anda mengakses nilai obyek dengan memanggil nya Metode `get`. Di luar itu, kedua jenis berperilaku cukup berbeda.

- `Future<T>` adalah konstruksi sinkronisasi yang memungkinkan aplikasi menunggu penyelesaian metode asinkron. Jika Anda menelepon `get` dan objek tidak siap, itu akan memblokir hingga objek siap.
- Dengan `Promise<T>`, sinkronisasi ditangani oleh kerangka kerja. Jika Anda menelepon `get` dan objeknya belum siap, `get` melempar pengecualian.

Tujuan utama dari `Promise<T>` adalah untuk mengelola aliran data dari satu aktivitas ke aktivitas lainnya. Ini memastikan bahwa suatu kegiatan tidak mengeksekusi sampai input data valid. Dalam banyak kasus, pekerja alur kerja tidak perlu mengakses objek `Promise<T>` secara langsung; mereka hanya meneruskan objek dari satu aktivitas ke aktivitas lainnya dan membiarkan kerangka kerja dan pekerja aktivitas menangani detailnya. Untuk mengakses nilai objek `Promise<T>` dalam pekerja alur kerja, Anda harus yakin bahwa objek siap sebelum memanggil nya Metode `get`.

- Pendekatan yang disukai adalah meneruskan objek `Promise<T>` ke metode alur kerja asinkron dan memproses nilai di sana. Metode asinkron menunda eksekusi hingga semua objek `Promise<T>` inputnya siap, yang menjamin bahwa Anda dapat mengakses nilainya dengan aman.
- `Promise<T>` mengekspos sebuah metode `isReady` yang mengembalikan `true` jika objek sudah siap. Menggunakan `isReady` untuk polling objek `Promise<T>` tidak disarankan, tetapi `isReady` berguna dalam keadaan tertentu.

AWS Flow Framework Untuk Java juga mencakup `Settable<T>` tipe, yang berasal dari `Promise<T>` dan memiliki perilaku serupa. Perbedaannya adalah bahwa kerangka kerja biasanya menetapkan nilai `Promise<T>` objek dan pekerja alur kerja bertanggung jawab untuk menetapkan nilai a`Settable<T>`.

Ada beberapa keadaan di mana pekerja alur kerja perlu membuat objek `Promise<T>` dan menetapkan nilainya. Misalnya, metode asinkron yang mengembalikan objek `Promise<T>` perlu membuat nilai kembalian.

- Untuk membuat objek yang mewakili nilai yang diketik, panggil metode `Promise.asPromise` statis, yang membuat objek `Promise<T>` dengan tipe yang sesuai, menetapkan nilainya, dan menempatkannya dalam status siap.
- Untuk membuat objek `Promise<Void>`, panggil metode `Promise.Void` statis.

> **Note**
>
> `Promise<T>` dapat mewakili jenis yang valid. Namun, jika data harus disusun di internet, jenisnya harus kompatibel dengan konverter data. Lihat bagian selanjutnya untuk rinciannya.

Konverter Data dan Marshaling

Data AWS Flow Framework marshals di internet dengan menggunakan konverter data. Secara default, kerangka menggunakan konverter data yang didasarkan pada Prosesor Jackson JSON. Namun, konverter ini memiliki beberapa keterbatasan. Sebagai contoh, tidak dapat menyusun peta yang tidak menggunakan string sebagai kunci. Jika konverter default tidak cukup untuk aplikasi Anda, Anda dapat menerapkan konverter data kustom. Untuk detail selengkapnya, lihat DataConverters.

AWS Flow Framework Konsep Dasar: Data Exchange Antara Aplikasi dan Eksekusi Alur Kerja

Metode titik masuk alur kerja dapat memiliki satu atau beberapa parameter, yang memungkinkan starter alur kerja meneruskan data awal ke alur kerja. Hal ini juga dapat berguna untuk memberikan data tambahan ke alur kerja selama eksekusi. Misalnya, jika pelanggan mengubah alamat pengiriman mereka, Anda dapat memberitahu alur kerja pemrosesan pesanan sehingga dapat membuat perubahan yang sesuai.

Amazon SWF memungkinkan alur kerja untuk menerapkan sinyal, yang memungkinkan aplikasi seperti starter alur kerja untuk menyampaikan data ke alur kerja kapan saja. Metode sinyal dapat memiliki nama dan parameter yang mudah digunakan. Anda menunjuknya sebagai metode sinyal dengan memasukkannya dalam definisi antarmuka alur kerja Anda, dan menerapkan anotasi `@Signal` ke deklarasi metode.

Contoh berikut menunjukkan antarmuka alur kerja pemrosesan urutan yang menyatakan metode sinyal, `changeOrder`, yang memungkinkan starter alur kerja mengubah urutan awal setelah alur kerja dimulai.

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

Pemroses anotasi kerangka kerja membuat metode klien alur kerja dengan nama yang sama dengan metode sinyal dan starter alur kerja memanggil metode klien untuk meneruskan data ke alur kerja. Sebagai contoh, lihat Resep AWS Flow Framework

Tipe Batas Waktu Amazon SWF

Untuk memastikan eksekusi alur kerja berjalan dengan benar, Anda dapat mengatur berbagai jenis batas waktu dengan Amazon SWF. Beberapa batas waktu menentukan berapa lama alur kerja dapat berjalan secara keseluruhan. Batas waktu lainnya menentukan berapa lama tugas aktivitas dapat berjalan sebelum ditugaskan ke pekerja dan berapa lama waktu yang bisa mereka gunakan untuk

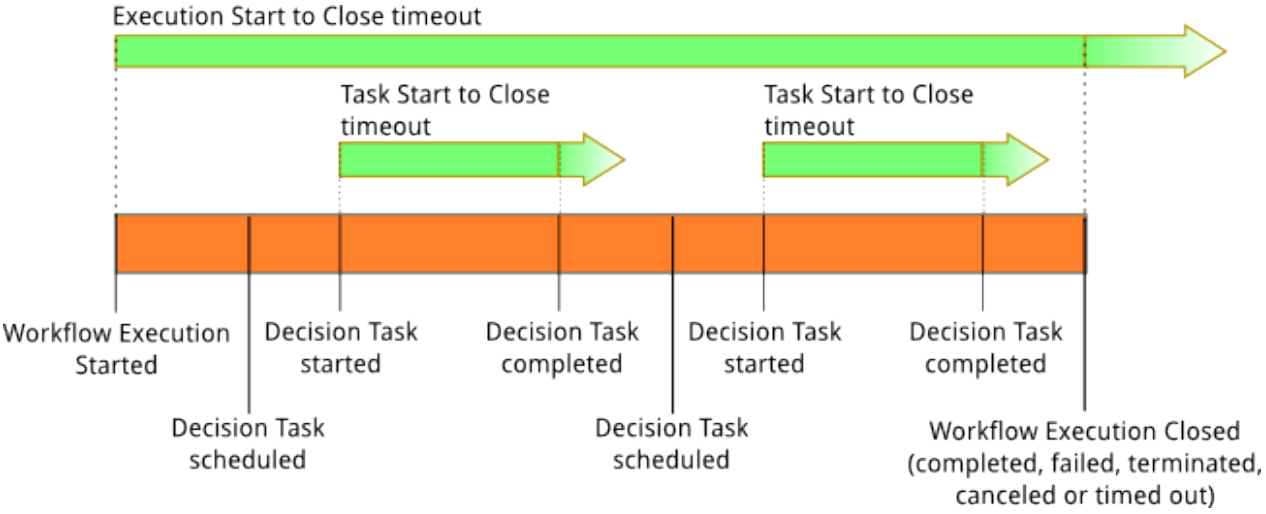
menyelesaikan dari waktu yang dijadwalkan. Semua batas waktu di API Amazon SWF ditentukan dalam hitungan detik. Amazon SWF juga mendukung string `NONE` sebagai nilai batas waktu, yang menunjukkan tidak ada batas waktu.

Untuk batas waktu yang berkaitan dengan tugas keputusan dan tugas aktivitas, Amazon SWF menambahkan sebuah kejadian ke riwayat eksekusi alur kerja. Atribut acara memberikan informasi tentang jenis batas waktu apa yang terjadi dan tugas keputusan atau aktivitas tugas yang terpengaruh. Amazon SWF juga menjadwalkan tugas keputusan. Ketika penentu menerima tugas keputusan baru, ia akan melihat peristiwa batas waktu dalam sejarah dan mengambil tindakan yang tepat dengan memanggil tindakan. RespondDecisionTaskCompleted

Tugas dianggap terbuka dari saat yang dijadwalkan sampai ditutup. Oleh karena itu tugas dilaporkan sebagai terbuka sementara pekerja sedang memproses tugas. Tugas ditutup ketika seorang pekerja melaporkannya sebagai selesai, dibatalkan, atau gagal. Sebuah tugas juga dapat ditutup oleh Amazon SWF sebagai hasil dari batas waktu.

Batas Waktu dalam Alur Kerja dan Tugas Keputusan

Diagram berikut menunjukkan bagaimana alur kerja dan batas waktu keputusan terkait dengan masa pakai alur kerja:



Ada dua tipe batas waktu timeout yang relevan dengan alur kerja dan tugas keputusan:

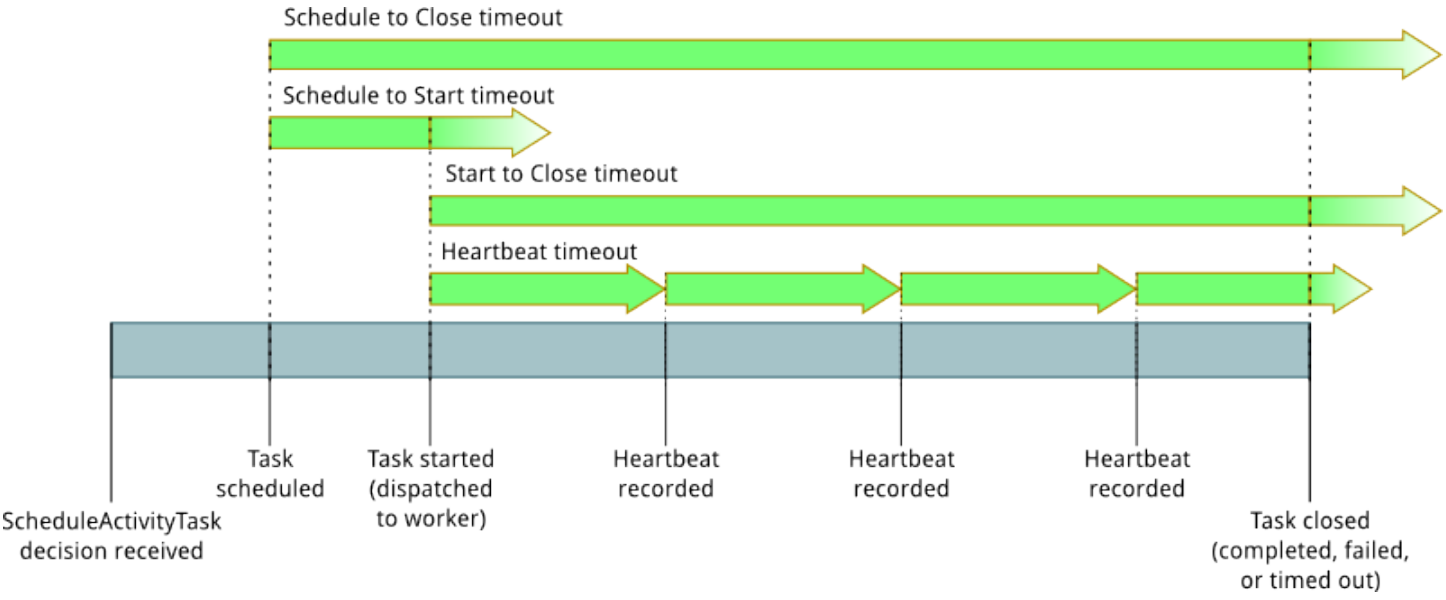
- Alur Kerja Start to Close (**`timeoutType: START_TO_CLOSE`**) – Batas Waktu ini menentukan waktu maksimum yang dapat diambil untuk menyelesaikan eksekusi alur kerja. Batas ini diatur sebagai default selama pendaftaran alur kerja, tetapi dapat diambil alih dengan nilai yang berbeda ketika alur kerja dimulai. Jika batas waktu ini terlampaui, Amazon SWF menutup eksekusi alur

kerja dan menambahkan WorkflowExecutionTimedOutjenis peristiwa ke riwayat eksekusi alur kerja. Selain `timeoutType`, atribut kejadian menentukan `childPolicy` yang berlaku untuk eksekusi alur kerja ini. Kebijakan anak menentukan bagaimana eksekusi alur kerja anak ditangani jika eksekusi alur kerja induk mencapai batas waktu atau berakhir. Contohnya, jika `childPolicy` diatur menjadi TERMINATE (AKHIRI), maka eksekusi alur kerja anak akan diakhiri. Setelah eksekusi alur kerja mencapai batas waktu, Anda tidak dapat mengambil tindakan apa pun selain panggilan visibilitas.

- Tugas Keputusan Start to Close (**`timeoutType: START_TO_CLOSE`**) – Batas waktu ini menentukan waktu maksimum yang dapat digunakan decider yang sesuai untuk menyelesaikan tugas keputusan. Hal ini diatur selama pendaftaran tipe alur kerja. Jika batas waktu ini terlampaui, tugas ditandai sebagai kehabisan waktu dalam riwayat eksekusi alur kerja, dan Amazon SWF menambahkan jenis peristiwa ke riwayat alur kerja. DecisionTaskTimedOut Atribut event akan menyertakan IDs untuk peristiwa yang sesuai dengan saat tugas keputusan ini dijadwalkan (`scheduledEventId`) dan saat dimulai (`startedEventId`). Selain menambahkan kejadian tersebut, Amazon SWF juga menjadwalkan tugas keputusan baru untuk memberitahu decider bahwa tugas keputusan ini mencapai batas waktu. Setelah batas waktu ini terjadi, upaya untuk menyelesaikan tugas keputusan yang mencapai batas waktu menggunakan `RespondDecisionTaskCompleted` akan gagal.

Batas Waktu dalam Tugas Aktivitas

Diagram berikut menunjukkan bagaimana batas waktu terkait dengan masa pakai tugas kegiatan:



Ada empat tipe batas waktu yang relevan dengan tugas aktivitas:

- Tugas Aktivitas Start to Close (**`timeoutType: START_TO_CLOSE`**) – Batas waktu ini menentukan waktu maksimum yang dapat diambil seorang pekerja aktivitas untuk memproses tugas setelah pekerja menerima tugas. Upaya untuk menutup tugas aktivitas yang habis waktu menggunakan RespondActivityTaskCanceled, RespondActivityTaskCompleted, dan RespondActivityTaskFailedakan gagal.
- Tugas Aktivitas Heartbeat (**`timeoutType: HEARTBEAT`**) – Batas waktu ini menentukan waktu maksimum untuk menjalankan tugas sebelum menyediakan kemajuannya melalui tindakan `RecordActivityTaskHeartbeat`.
- Tugas Aktivitas Schedule to Start (**`timeoutType: SCHEDULE_TO_START`**) - Batas waktu ini menentukan berapa lama Amazon SWF menunggu sebelum mencapai batas waktu tugas aktivitas jika tidak ada pekerja yang tersedia untuk melakukan tugas. Setelah batas waktu tercapai, tugas yang kedaluwarsa tidak akan ditugaskan ke pekerja lain.
- Tugas Aktivitas Schedule to Close (**`timeoutType: SCHEDULE_TO_CLOSE`**) - Batas waktu ini menentukan berapa lama waktu yang bisa digunakan sejak waktu yang dijadwalkan hingga tugas selesai. Sebagai praktik terbaik, nilai ini tidak boleh lebih besar dari jumlah schedule-to-start batas waktu tugas dan batas start-to-close waktu tugas.

> **Note**
>
> Setiap tipe batas waktu memiliki nilai default, yang umumnya diatur ke `NONE` (tak terbatas). Namun, waktu maksimum untuk setiap eksekusi aktivitas terbatas pada satu tahun.

Anda menetapkan nilai default untuk nilai ini selama pendaftaran tipe aktivitas, tetapi Anda dapat mengambil alih dengan nilai baru ketika Anda menjadwalkan tugas aktivitas. Ketika salah satu batas waktu ini terjadi, Amazon SWF akan menambahkan ActivityTaskTimedOutjenis peristiwa ke riwayat alur kerja. Atribut `timeoutType` nilai dari kejadian ini akan menentukan batas waktu mana yang akan terjadi. Untuk setiap batas waktu, nilai `timeoutType` ditampilkan dalam tanda kurung. Atribut event juga akan menyertakan IDs untuk peristiwa yang sesuai dengan saat tugas aktivitas dijadwalkan (`scheduledEventId`) dan saat dimulai (`startedEventId`). Selain menambahkan kejadian tersebut, Amazon SWF juga menjadwalkan tugas keputusan baru untuk memberitahu decider bahwa batas waktu terjadi.

Memahami Tugas di AWS Flow Framework Java

Topik

Tugas

Primitif mendasar yang digunakan AWS Flow Framework untuk Java untuk mengelola eksekusi kode asinkron adalah kelas. `Task` Sebuah objek tipe `Task` mewakili pekerjaan yang harus dilakukan secara asinkron. Ketika Anda memanggil metode asinkronus, kerangka kerja menciptakan `Task` untuk mengeksekusi kode dalam metode itu dan menempatkannya dalam daftar untuk eksekusi di lain waktu. Demikian pula, ketika Anda memanggil `Activity`, sebuah `Task` diciptakan untuk itu. Pemanggilan metode kembali setelah ini, biasanya mengembalikan `Promise<T>` sebagai hasil pemanggilan di masa mendatang.

Kelas `Task` adalah publik dan dapat digunakan secara langsung. Sebagai contoh, kita dapat menulis ulang contoh Hello World untuk menggunakan `Task` bukan metode asinkronus.

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

Kerangka kerja ini memanggil metode `doExecute()` ketika semua `Promise` diteruskan ke konstruktor dari `Task` menjadi siap. Untuk detail selengkapnya tentang `Task` kelas, lihat AWS SDK untuk Java dokumentasi.

Kerangka kerja ini juga mencakup kelas yang disebut `Functor` yang mewakili `Task` yang juga merupakan `Promise<T>`. `Functor` objek menjadi siap ketika `Task` selesai. Pada contoh berikut, `Functor` dibuat untuk mendapatkan pesan ucapan:

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

Urutan eksekusi

Tugas menjadi memenuhi syarat untuk dieksekusi hanya ketika semua parameter yang bertipe `Promise<T>`, diteruskan ke metode atau aktivitas asinkron yang sesuai, telah siap. `Task` yang siap untuk eksekusi secara logis dipindahkan ke antrean siap. Dengan kata lain, dijadwalkan untuk eksekusi. Kelas pekerja mengeksekusi tugas dengan menerapkan kode yang Anda tulis dalam isi metode asyinkronus, atau dengan penjadwalan tugas aktivitas di Amazon Simple Workflow Service (AWS) jika ada metode aktivitas.

Saat tugas dijalankan dan menghasilkan hasil, maka itu menyebabkan tugas lain menjadi siap dan eksekusi program terus bergerak maju. Cara kerangka kerja menjalankan tugas penting untuk memahami urutannya di mana kode asinkron Anda dijalankan. Kode yang muncul secara berurutan dalam program Anda mungkin tidak benar-benar mengeksekusi dalam urutan itu.

```
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!");
}
```

Kode dalam daftar di atas akan mencetak berikut:

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

Ini mungkin bukan apa yang Anda harapkan tetapi dapat dengan mudah dijelaskan dengan memikirkan bagaimana tugas-tugas untuk metode asinkronus dieksekusi:

1. Panggilan ke `getUserName` membuat `Task`. Mari kita sebut saja `Task1`. Karena `getUserName` tidak mengambil parameter apapun, segera `Task1` dimasukkan ke dalam antrian siap.
2. Selanjutnya, panggilan ke `printHelloName` membuat `Task` yang perlu menunggu hasil dari `getUserName`. Mari kita sebut saja `Task2`. Karena nilai yang diperlukan belum siap, `Task2` dimasukkan ke dalam daftar tunggu.
3. Kemudian tugas untuk `printHelloWorld` dibuat dan ditambahkan ke antrean siap. Mari kita sebut saja `Task3`.
4. Pernyataan `println` kemudian mencetak "Hello, Amazon!" ke konsol.
5. Pada titik ini, `Task1` dan `Task3` berada dalam antrean siap dan `Task2` ada di daftar tunggu.
6. Pekerja mengeksekusi `Task1`, dan hasilnya membuat `Task2` siap. `Task2` akan ditambahkan ke antrean siap di belakang `Task3`.
7. `Task3` dan `Task2` kemudian dieksekusi dalam urutan itu.

Pelaksanaan kegiatan mengikuti pola yang sama. Ketika Anda memanggil metode pada klien aktivitas, itu menciptakan `Task` yang, setelah eksekusi, menjadwalkan kegiatan di Amazon SWF.

Kerangka kerja ini bergantung pada fitur seperti generasi kode dan proxy dinamis untuk menyuntikkan logika untuk mengonversi panggilan metode ke pemanggilan aktivitas dan tugas asinkron dalam program Anda.

Eksekusi alur kerja

Eksekusi implementasi alur kerja juga dikelola oleh kelas pekerja. Saat Anda memanggil metode pada klien alur kerja, itu akan memanggil Amazon SWF untuk membuat instans alur kerja. Tugas di Amazon SWF tidak boleh disamakan dengan tugas dalam kerangka kerja. Tugas di Amazon SWF adalah tugas aktivitas atau tugas keputusan. Pelaksanaan tugas aktivitas sederhana. Kelas pekerja aktivitas menerima tugas aktivitas dari Amazon SWF, memanggil metode aktivitas yang sesuai dalam pelaksanaan Anda, dan mengembalikan hasilnya ke Amazon SWF.

Eksekusi tugas keputusan lebih terlibat. Pekerja alur kerja menerima tugas keputusan dari Amazon SWF. Tugas keputusan secara efektif merupakan permintaan yang menanyakan logika alur kerja apa yang harus dilakukan selanjutnya. Tugas keputusan pertama dibuat untuk contoh alur kerja saat dimulai melalui klien alur kerja. Setelah menerima tugas keputusan ini, kerangka kerja mulai mengeksekusi kode dalam metode alur kerja dijelaskan dengan `@Execute`. Metode ini mengeksekusi logika koordinasi yang menjadwalkan kegiatan. Saat status instans alur kerja berubah —misalnya, saat aktivitas selesai—tugas keputusan lebih lanjut dijadwalkan. Pada titik ini, logika alur kerja dapat memutuskan untuk mengambil tindakan berdasarkan hasil aktivitas; misalnya, mungkin memutuskan untuk menjadwalkan aktivitas lain.

Kerangka kerja menyembunyikan semua detail ini dari developer dengan menerjemahkan tugas keputusan secara mulus ke logika alur kerja. Dari sudut pandang developer, kode terlihat seperti program biasa. Secara rahasia, kerangka kerja memetakannya ke panggilan ke Amazon SWF dan tugas keputusan menggunakan riwayat yang dikelola oleh Amazon SWF. Ketika tugas keputusan tiba, kerangka kerja memutar ulang eksekusi program memasukkan hasil kegiatan yang diselesaikan sejauh ini. Metode dan aktivitas asinkronus yang menunggu hasil ini tidak diblokir, dan eksekusi program bergerak maju.

Eksekusi contoh alur kerja pemrosesan gambar dan riwayat terkait ditampilkan dalam tabel berikut.

Eksekusi alur kerja thumbnail

Eksekusi program alur kerja	Riwayat dikelola oleh Amazon SWF
Eksekusi awal	
1. Loop pengiriman 2. getImageUrls 3. downloadImage	1. Instans alur kerja dimulai, id="1" 2. downloadImage terjadwal

Eksekusi program alur kerja	Riwayat dikelola oleh Amazon SWF
4. createThumbnail (tugas dalam antrean tunggu) 5. uploadImage (tugas dalam antrean tunggu) 6. <iterasi berikutnya dari loop>	
Memutar ulang	
1. Loop pengiriman 2. getImageUrls 3. gambar downloadImage path="foo" 4. createThumbnail 5. uploadImage (tugas dalam antrean tunggu) 6. <iterasi berikutnya dari loop>	1. Instans alur kerja dimulai, id="1" 2. downloadImage terjadwal 3. downloadImage selesai, return="foo" 4. createThumbnail terjadwal
Memutar ulang	
1. Loop pengiriman 2. getImageUrls 3. gambar downloadImage path="foo" 4. createThumbnail thumbnail path="bar" 5. uploadImage 6. <iterasi berikutnya dari loop>	1. Instans alur kerja dimulai, id="1" 2. downloadImage terjadwal 3. downloadImage selesai, return="foo" 4. createThumbnail terjadwal 5. createThumbnail selesai, return="bar" 6. uploadImage terjadwal
Memutar ulang	

Eksekusi program alur kerja	Riwayat dikelola oleh Amazon SWF
1. Loop pengiriman 2. getImageUrls 3. gambar downloadImage path="foo" 4. createThumbnail thumbnail path="bar" 5. UploaDimage 6. <iterasi berikutnya dari loop>	1. Instans alur kerja dimulai, id="1" 2. downloadImage terjadwal 3. downloadImage selesai, return="foo" 4. createThumbnail terjadwal 5. createThumbnail selesai, return="bar" 6. uploadImage terjadwal 7. uploadImage selesai ...

Ketika panggilan ke `processImage` dibuat, kerangka kerja menciptakan instans alur kerja baru di Amazon SWF. Ini adalah catatan tahan lama dari instans alur kerja yang sedang dimulai. Program ini mengeksekusi sampai panggilan ke kegiatan `downloadImage`, yang meminta Amazon SWF untuk menjadwalkan suatu kegiatan. Alur kerja mengeksekusi lebih jauh dan membuat tugas untuk kegiatan berikutnya, tetapi tidak dapat dijalankan sampai kegiatan `downloadImage` selesai; maka, episode putar ulang ini berakhir. Amazon SWF mengirimkan tugas untuk kegiatan `downloadImage` untuk dieksekusi, dan setelah selesai, catatan dibuat dalam riwayat bersama dengan hasilnya. Alur kerja sekarang siap untuk bergerak maju dan tugas keputusan dihasilkan oleh Amazon SWF. Kerangka kerja menerima tugas keputusan dan replay alur kerja memasukkan hasil gambar download seperti yang tercatat dalam riwayat. Ini membuka blokir tugas untuk `createThumbnail`, dan pelaksanaan program berlanjut lebih jauh dengan penjadwalan tugas aktivitas `createThumbnail` di Amazon SWF. Proses yang sama berulang untuk `uploadImage`. Pelaksanaan program berlanjut dengan cara ini sampai alur kerja telah memproses semua gambar dan tidak ada tugas yang tertunda. Karena tidak ada status eksekusi yang disimpan secara lokal, setiap tugas keputusan berpotensi dieksekusi pada mesin yang berbeda. Hal ini mengizinkan Anda untuk dengan mudah menulis program yang toleran kesalahan dan dapat diskalakan.

Nondeterminisme

Karena kerangka kerja bergantung pada pemutaran ulang, penting bahwa kode orkestrasi (semua kode alur kerja dengan pengecualian implementasi aktivitas) bersifat deterministik. Misalnya, aliran kontrol dalam program Anda tidak harus bergantung pada nomor acak atau waktu saat ini. Karena hal-hal ini akan berubah di antara pemanggilan, pemutaran ulang mungkin tidak mengikuti jalur yang

sama melalui logika orkestrasi. Hal ini akan menyebabkan hasil yang tidak terduga atau kesalahan. Kerangka kerja ini menyediakan `WorkflowClock` yang dapat Anda gunakan untuk mendapatkan waktu saat ini dengan cara deterministik. Lihat bagian di Konteks Eksekusi untuk lebih detailnya.

> **Note**
>
> Pengkabelan pegas yang salah dari objek implementasi alur kerja juga dapat menyebabkan nondeterminisme. Alur kerja implementasi benas serta beans yang mereka andalkan harus berada dalam lingkup alur kerja (`WorkflowScope`). Misalnya, kabel alur kerja implementasi beans ke beans yang membuat status dan dalam konteks global akan menghasilkan perilaku tak terduga. Lihat Integrasi Spring bagian untuk detail lebih lanjut.

AWS Flow Framework untuk Panduan Pemrograman Java

Bagian ini memberikan rincian tentang cara menggunakan fitur untuk Java AWS Flow Framework untuk mengimplementasikan aplikasi alur kerja.

Topik

Menerapkan Aplikasi Alur Kerja dengan AWS Flow Framework

Langkah-langkah khas yang terlibat dalam mengembangkan alur kerja dengan AWS Flow Framework adalah:

1. Tentukan kontrak aktivitas dan alur kerja. Analisis persyaratan aplikasi Anda, lalu tentukan aktivitas yang diperlukan dan topologi alur kerja. Aktivitas menangani tugas pemrosesan yang diperlukan, sedangkan topologi alur kerja menentukan struktur dasar dan logika bisnis alur kerja .

   Misalnya, aplikasi pemrosesan media mungkin perlu mengunduh file, memprosesnya, lalu mengunggah file yang diproses ke bucket Amazon Simple Storage Service (S3). Ini dapat dipecah menjadi empat tugas aktivitas:

   1. unduh file dari server
   2. memproses file (misalnya, dengan transcoding ke format media yang berbeda)
   3. unggah file ke bucket S3
   4. lakukan pembersihan dengan menghapus file lokal

   Alur kerja ini akan memiliki metode titik masuk dan akan menerapkan topologi linier sederhana yang menjalankan aktivitas secara berurutan, seperti HelloWorldWorkflow Aplikasi.
2. Terapkan antarmuka aktivitas dan alur kerja. Kontrak alur kerja dan aktivitas ditentukan oleh antarmuka Java, membuat konvensi pemanggilannya dapat diprediksi oleh SWF, dan memberi Anda fleksibilitas saat menerapkan logika alur kerja dan tugas aktivitas Anda. Berbagai bagian program Anda dapat bertindak sebagai konsumen data satu sama lain, namun tidak perlu mengetahui banyak detail implementasi dari bagian lain mana pun.

   Misalnya, Anda dapat menentukan antarmuka `FileProcessingWorkflow` dan menyediakan implementasi alur kerja yang berbeda untuk pengkodean video, kompresi, gambar mini, dan sebagainya. Masing-masing alur kerja tersebut dapat memiliki alur kontrol yang berbeda dan dapat memanggil metode aktivitas yang berbeda; starter alur kerja Anda tidak perlu tahu. Dengan menggunakan antarmuka, juga mudah untuk menguji alur kerja Anda dengan menggunakan implementasi tiruan yang nantinya dapat diganti dengan kode kerja.
3. Hasilkan klien aktivitas dan alur kerja. AWS Flow Framework Ini menghilangkan kebutuhan bagi Anda untuk menerapkan rincian mengelola eksekusi asinkron, mengirim permintaan HTTP, menyusun data, dan sebagainya. Sebagai gantinya, starter alur kerja mengeksekusi instans alur kerja dengan memanggil metode pada klien alur kerja, dan implementasi alur kerja mengeksekusi aktivitas dengan memanggil metode pada klien aktivitas. Kerangka kerja menangani detail interaksi ini di latar belakang.

   Jika Anda menggunakan Eclipse dan Anda telah mengonfigurasi proyek Anda, seperti diMenyiapkan AWS Flow Framework untuk Java, prosesor AWS Flow Framework anotasi

menggunakan definisi antarmuka untuk secara otomatis menghasilkan alur kerja dan aktivitas klien yang mengekspos kumpulan metode yang sama dengan antarmuka yang sesuai.

4. Terapkan aplikasi host aktivitas dan alur kerja. Alur kerja dan implementasi aktivitas Anda harus disematkan dalam aplikasi host yang melakukan polling Amazon SWF untuk tugas, menyusun data apa pun, dan memanggil metode implementasi yang sesuai. AWS Flow Framework untuk Java termasuk WorkflowWorkerdan ActivityWorkerkelas yang membuat penerapan aplikasi host mudah dan mudah dilakukan.
5. Uji alur kerja Anda. AWS Flow Framework untuk Java menyediakan JUnit integrasi yang dapat Anda gunakan untuk menguji alur kerja Anda secara inline dan lokal.
6. Deploy pekerja. Anda dapat menerapkan pekerja sesuai kesesuaian—misalnya, Anda dapat menerapkannya ke instans EC2 Amazon atau ke komputer di pusat data Anda. Setelah di-deploy dan dimulai, pekerja mulai melakukan polling Amazon SWF untuk tugas dan menanganinya sesuai kebutuhan.
7. Mulai eksekusi. Aplikasi memulai instans alur kerja dengan menggunakan klien alur kerja untuk memanggil titik masuk alur kerja. Anda juga dapat memulai alur kerja dengan menggunakan konsol Amazon SWF. Terlepas dari cara Anda memulai instans alur kerja, Anda dapat menggunakan konsol Amazon SWF untuk memantau instans alur kerja yang berjalan dan memeriksa riwayat alur kerja untuk instans yang berjalan, selesai, dan gagal.

AWS SDK untuk JavaTermasuk satu set AWS Flow Framework untuk sampel Java yang dapat Anda jelajahi dan jalankan dengan mengikuti instruksi dalam file readme.html di direktori root. Ada juga satu set resep —aplikasi sederhana — yang menunjukkan cara menangani berbagai masalah pemrograman tertentu, yang tersedia dari Resep AWS Flow Framework.

Alur Kerja dan Kontrak Aktivitas

Antarmuka Java digunakan untuk menyatakan tanda tangan dari alur kerja dan kegiatan. Antarmuka membentuk kontrak antara pelaksanaan alur kerja (atau aktivitas) dan klien alur kerja tersebut (atau aktivitas). Sebagai contoh, jenis alur kerja `MyWorkflow` didefinisikan menggunakan antarmuka yang dijelaskan dengan anotasi `@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();
}
```

Kontrak tidak memiliki pengaturan khusus implementasi. Penggunaan kontrak implementasi-netral memungkinkan klien untuk dipisahkan dari pelaksanaan dan karenanya memberikan fleksibilitas untuk mengubah detail pelaksanaan tanpa melanggar klien. Sebaliknya, Anda juga dapat mengubah klien tanpa memerlukan perubahan alur kerja atau aktivitas yang dikonsumsi. Sebagai contoh, klien dapat dimodifikasi untuk memanggil aktivitas asynchronously menggunakan janji-janji (`Promise<T>`) tanpa memerlukan perubahan pada pelaksanaan kegiatan. Demikian pula, pelaksanaan kegiatan dapat diubah sehingga selesai secara asinkron, misalnya, oleh orang yang mengirim email—tanpa memerlukan klien dari kegiatan yang akan diubah.

Dalam contoh di atas, antarmuka alur kerja `MyWorkflow` berisi sebuah metode, `startMyWF`, untuk memulai eksekusi baru. Metode ini dijelaskan dengan anotasi `@Execute` dan harus memiliki tipe pengembalian dari `void` atau `Promise<>`. Dalam antarmuka alur kerja yang diberikan, paling banyak satu metode dapat dijelaskan dengan anotasi ini. Metode ini adalah titik masuk logika alur kerja, dan kerangka kerja memanggil metode ini untuk mengeksekusi logika alur kerja ketika tugas keputusan diterima.

Antarmuka alur kerja juga mendefinisikan sinyal yang dapat dikirim ke alur kerja. Metode sinyal akan dipanggil ketika sinyal dengan nama yang cocok diterima oleh eksekusi alur kerja. Misalnya, antarmuka `MyWorkflow` menyatakan metode sinyal, `signal1`, dianotasi dengan anotasi `@Signal`.

Anotasi `@Signal` diperlukan pada metode sinyal. Jenis kembali dari metode sinyal harus `void`. Sebuah antarmuka alur kerja mungkin memiliki nol atau lebih metode sinyal didefinisikan di dalamnya. Anda dapat mendeklarasikan antarmuka alur kerja tanpa metode `@Execute` dan beberapa metode `@Signal` untuk menghasilkan klien yang tidak dapat memulai eksekusi mereka tetapi dapat mengirim sinyal untuk menjalankan eksekusi.

Metode dijelaskan dengan `@Execute` dan anotasi `@Signal` mungkin memiliki sejumlah parameter dari jenis apa pun selain `Promise<T>` atau turunannya. Hal ini memungkinkan Anda meneruskan input yang diketik dengan kuat ke eksekusi alur kerja di awal dan saat sedang berjalan. Tipe pengembalian metode `@Execute` harus `void` atau `Promise<>`.

Selain itu, Anda juga dapat menyatakan metode dalam antarmuka alur kerja untuk melaporkan status terbaru eksekusi alur kerja, misalnya, metode `getState` dalam contoh sebelumnya. Status ini tidak seluruh aplikasi status alur kerja. Tujuan penggunaan fitur ini adalah untuk mengizinkan Anda menyimpan hingga 32 KB data untuk menunjukkan status terbaru eksekusi. Sebagai contoh, dalam alur kerja pemrosesan urutan, Anda dapat menyimpan string yang menunjukkan bahwa pesanan telah diterima, diproses, atau dibatalkan. Metode ini disebut oleh kerangka kerja setiap kali tugas keputusan selesai untuk mendapatkan status terbaru. Status disimpan di Amazon Simple Workflow Service (Amazon SWF) dan dapat diambil menggunakan klien eksternal yang dihasilkan. Hal ini mengizinkan Anda untuk memeriksa status terbaru dari eksekusi alur kerja. Metode dijelaskan dengan `@GetState` tidak boleh mengambil argumen apapun dan tidak boleh memiliki sebuah jenis pengembalian `void`. Anda dapat mengembalikan jenis apa pun, yang sesuai dengan kebutuhan Anda, dari metode ini. Dalam contoh di atas, objek `MyWorkflowState` (lihat definisi di bawah) dikembalikan oleh metode yang digunakan untuk menyimpan status string dan persen numerik lengkap. Metode ini diharapkan untuk melakukan akses baca saja dari objek implementasi alur kerja dan dipanggil serentak, yang melarang penggunaan operasi asinkron seperti memanggil metode dijelaskan dengan `@Asynchronous`. Paling banyak satu metode dalam antarmuka alur kerja dapat dijelaskan dengan anotasi `@GetState`.

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

Demikian pula, satu set aktivitas didefinisikan menggunakan antarmuka yang dianotasi dengan anotasi `@Activities`. Setiap metode dalam antarmuka sesuai dengan aktivitas—misalnya:

```
@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);
}
```

Antarmuka mengizinkan Anda untuk mengelompokkan serangkain aktivitas terkait. Anda dapat menentukan sejumlah aktivitas dalam antarmuka aktivitas, dan Anda dapat menentukan sebanyak kegiatan antarmuka yang Anda inginkan. Serupa dengan `@Execute` dan metode `@Signal`, metode aktivitas dapat mengambil sejumlah argumen dari jenis apa pun selain `Promise<T>` atau turunannya. Jenis kembalinya suatu aktivitas tidak boleh `Promise<T>` atau turunannya.

Alur Kerja dan Jenis Aktivitas Pendaftaran

Amazon SWF membutuhkan aktivitas dan alur kerja jenis untuk didaftarkan sebelum mereka dapat digunakan. Kerangka kerja secara otomatis mendaftarkan alur kerja dan aktivitas dalam implementasi yang Anda tambahkan ke pekerja. Kerangka kerja ini mencari jenis yang mengimplementasikan alur kerja dan aktivitas dan daftarkan mereka dengan Amazon SWF. Secara default, kerangka kerja menggunakan definisi antarmuka untuk menyimpulkan pilihan pendaftaran untuk alur kerja dan jenis aktivitas. Semua antarmuka alur kerja yang diperlukan untuk memiliki anotasi `@WorkflowRegistrationOptions` atau anotasi `@SkipRegistration`. Pekerja alur kerja mendaftarkan semua jenis alur kerja yang dikonfigurasi dengan yang memiliki anotasi `@WorkflowRegistrationOptions`. Demikian pula, setiap metode aktivitas diperlukan untuk dijelaskan dengan baik anotasi `@ActivityRegistrationOptions` atau anotasi `@SkipRegistration` atau salah satu anotasi ini harus ada di antarmuka `@Activities`. Pekerja aktivitas mendaftarkan semua jenis aktivitas yang dikonfigurasi dengan anotasi `@ActivityRegistrationOptions` yang berlaku. Pendaftaran dilakukan secara otomatis ketika Anda memulai salah satu pekerja. Alur kerja dan jenis aktivitas yang memiliki anotasi `@SkipRegistration` tidak terdaftar. `@ActivityRegistrationOptions` , dan anotasi `@SkipRegistration` telah menimpa semantik dan yang paling spesifik diterapkan pada jenis aktivitas.

Perhatikan bahwa Amazon SWF tidak mengizinkan Anda untuk mendaftar ulang atau memodifikasi jenis setelah telah terdaftar. Kerangka kerja akan mencoba untuk mendaftarkan semua jenis, tetapi jika jenis sudah terdaftar maka tidak akan terdaftar ulang dan tidak ada kesalahan yang akan dilaporkan.

Jika Anda perlu memodifikasi pengaturan terdaftar, Anda harus mendaftarkan versi baru dari jenis ini. Anda juga dapat mengganti pengaturan terdaftar saat memulai eksekusi baru atau saat memanggil aktivitas yang menggunakan klien yang dihasilkan.

Pendaftaran memerlukan nama jenis dan beberapa pilihan pendaftaran lain. Implementasi default menentukan ini sebagai berikut:

Nama dan Versi Jenis Alur Kerja

Kerangka kerja menentukan nama jenis alur kerja dari antarmuka alur kerja. Bentuk nama tipe alur kerja default adalah {*prefix*} {*name*}. {*prefix*} diatur ke nama `@Workflow` antarmuka diikuti oleh '.' dan {*name*} diatur ke nama `@Execute` metode. Nama default dari jenis alur kerja dalam contoh sebelumnya adalah `MyWorkflow.startMyWF`. Anda dapat mengganti nama default menggunakan parameter nama metode `@Execute`. Nama default dari jenis alur kerja dalam contoh adalah `startMyWF`. Nama tidak harus string kosong. Perhatikan bahwa ketika Anda mengganti nama menggunakan `@Execute`, kerangka kerja tidak secara otomatis menambahkan prefiks untuk itu. Anda bebas untuk menggunakan skema penamaan Anda sendiri.

Versi alur kerja ditentukan menggunakan parameter `version` anotasi `@Execute`. Tidak ada default untuk `version` dan harus ditentukan secara eksplisit; `version`adalah bentuk string gratis, dan Anda bebas untuk menggunakan skema versioning Anda sendiri.

Nama Sinyal

Nama sinyal dapat ditentukan dengan menggunakan parameter nama anotasi `@Signal`. Jika tidak ditentukan, itu adalah default untuk nama metode sinyal.

Nama dan Versi Jenis Aktivitas

Kerangka kerja menentukan nama jenis aktivitas dari antarmuka aktivitas. Bentuk nama tipe aktivitas default adalah {*prefix*} {*name*}. {*prefix*} diatur ke nama `@Activities` antarmuka diikuti oleh '.' dan {*name*} diatur ke nama metode. {*prefix*} default dapat diganti dalam `@Activities` anotasi pada antarmuka aktivitas. Anda juga dapat menentukan nama jenis aktivitas menggunakan anotasi `@Activity` pada metode aktivitas. Perhatikan bahwa ketika Anda mengganti nama menggunakan `@Activity`, kerangka kerja tidak akan secara otomatis menambahkan prefiks untuk itu. Anda bebas untuk pengguna skema penamaan Anda sendiri.

Versi aktivitas ditentukan menggunakan parameter versi anotasi `@Activities`. Versi ini digunakan sebagai default untuk semua kegiatan didefinisikan dalam antarmuka dan dapat diganti secara per-aktivitas menggunakan anotasi `@Activity`.

Daftar Tugas Default

Daftar tugas default dapat dikonfigurasi menggunakan `@WorkflowRegistrationOptions` dan anotasi `@ActivityRegistrationOptions` dan mengatur parameter `defaultTaskList`. Secara default, nilainya diatur ke `USE_WORKER_TASK_LIST`. Ini adalah nilai khusus yang

menginstruksikan kerangka kerja untuk menggunakan daftar tugas yang dikonfigurasi pada objek pekerja yang digunakan untuk mendaftar jenis aktivitas atau alur kerja. Anda juga dapat memilih untuk tidak mendaftarkan daftar tugas default dengan mengatur daftar tugas default ke `NO_DEFAULT_TASK_LIST` menggunakan anotasi ini. Ini dapat digunakan dalam kasus-kasus di mana Anda ingin meminta bahwa daftar tugas ditentukan pada waktu aktif. Jika tidak ada daftar tugas default telah terdaftar, maka Anda harus menentukan daftar tugas saat memulai alur kerja atau memanggil metode aktivitas menggunakan `StartWorkflowOptions` dan parameter `ActivitySchedulingOptions` pada kelebihan metode masing-masing klien yang dihasilkan.

Pilihan Pendaftaran lainnya

Semua opsi pendaftaran alur kerja dan jenis aktivitas yang diizinkan oleh API Amazon SWF dapat ditentukan melalui kerangka kerja.

Untuk daftar lengkap opsi pendaftaran alur kerja, lihat berikut ini:

- @Workflow
- @Execute
- @WorkflowRegistrationOptions
- @Signal

Untuk daftar lengkap opsi pendaftaran aktivitas, lihat berikut ini:

- @Activity
- @Activities
- @ActivityRegistrationOptions

Jika Anda ingin memiliki kontrol penuh atas pendaftaran jenis, lihat Ekstensibilitas Pekerja.

Aktivitas dan alur kerja klien

Alur kerja dan aktivitas klien yang dihasilkan oleh kerangka kerja berdasarkan antarmuka `@Workflow` dan `@Activities`. Antarmuka klien terpisah yang dihasilkan yang berisi metode dan pengaturan yang masuk akal hanya pada klien. Jika Anda mengembangkan menggunakan Eclipse, hal ini dilakukan oleh plugin Amazon SWF Eclipse setiap kali Anda menyimpan file yang berisi antarmuka yang sesuai. Kode yang dihasilkan ditempatkan di direktori sumber yang dihasilkan dalam proyek Anda dalam paket yang sama dengan antarmuka.

> **Note**
>
> Perhatikan bahwa nama direktori default yang digunakan oleh Eclipse adalah .apt_generated. Eclipse tidak menampilkan direktori yang namanya dimulai dengan '.' di Package Explorer. Menggunakan nama direktori yang berbeda jika Anda ingin melihat file yang dihasilkan dalam Project Explorer. Dalam Eclipse, klik kanan paket dalam Paket Explorer, dan kemudian pilih Property (Properti), Java Compiler (Kompiler Java), Annotation processing (Memproses anotasi), dan modifikasi pengaturan Generate source directory (Hasilkan direktori sumber).

Alur Kerja klien

Artifact yang dihasilkan untuk alur kerja berisi tiga antarmuka sisi klien dan kelas yang menerapkannya. Klien yang dihasilkan meliputi:

- Asynchronous client (Klien asinkron) dimaksudkan untuk dikonsumsi dari dalam implementasi alur kerja yang menyediakan metode asinkronus untuk memulai eksekusi alur kerja dan mengirim sinyal
- External client (Klien eksternal) yang dapat digunakan untuk memulai eksekusi dan mengirim sinyal dan mengambil keadaan alur kerja dari luar cakupan implementasi alur kerja
- Self client (Klien mandiri) yang dapat digunakan untuk membuat alur kerja berkesinambungan

Sebagai contoh, antarmuka klien yang dihasilkan untuk contoh Antarmuka `MyWorkflow` adalah:

```
//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);
```

Antarmuka memiliki kelebihan beban metode yang sesuai dengan masing-masing metode di antarmuka `@Workflow` yang Anda nyatakan.

Klien eksternal mencerminkan metode pada antarmuka `@Workflow` dengan satu kelebihan beban tambahan metode `@Execute` yang mengambil `StartWorkflowOptions`. Anda dapat menggunakan kelebihan beban ini untuk melewati opsi tambahan saat memulai eksekusi alur kerja baru. Pilihan ini mengizinkan Anda untuk menimpa daftar tugas default, pengaturan batas waktu, dan mengaitkan tag dengan eksekusi alur kerja.

Di sisi lain, klien asinkronus memiliki metode yang mengizinkan permintaan asinkronus metode `@Execute`. Kelebihan metode berikut dihasilkan di antarmuka klien untuk metode `@Execute` di antarmuka alur kerja:

1. Kelebihan yang mengambil argumen asli apa adanya. Jenis kembalinya kelebihan beban ini akan `Promise<Void>` jika metode asli kembali `void`; jika tidak, itu akan menjadi `Promise<>` seperti yang dinyatakan pada metode asli. Sebagai contoh:

   Metode asli:

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

Metode yang dihasilkan:

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

Kelebihan beban ini harus digunakan ketika semua argumen alur kerja tersedia dan tidak perlu ditunggu.

2. Overload yang mengambil argumen asli sebagaimana adanya dan argumen variabel tambahan tipe `Promise<?>`. Jenis kembalinya kelebihan beban ini akan `Promise<Void>` jika metode asli kembali `void`; jika tidak, itu akan menjadi `Promise<>` seperti yang dinyatakan pada metode asli. Sebagai contoh:

   Metode asli:

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

   Metode yang dihasilkan:

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

   Kelebihan beban ini harus digunakan ketika semua argumen alur kerja tersedia dan tidak perlu ditunggu, tetapi Anda ingin menunggu beberapa janji lain untuk menjadi siap. Argumen variabel dapat digunakan untuk meneruskan objek `Promise<?>` yang tidak dideklarasikan sebagai argumen, tetapi Anda ingin menunggu sebelum menjalankan panggilan.

3. Kelebihan yang mengambil argumen asli apa adanya, argumen tambahan dari tipe `StartWorkflowOptions` dan argumen variabel tambahan dari tipe `Promise<?>`. Jenis kembalinya kelebihan beban ini akan `Promise<Void>` jika metode asli kembali `void`; jika tidak, itu akan menjadi `Promise<>` seperti yang dinyatakan pada metode asli. Sebagai contoh:

   Metode asli:

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

   Metode yang dihasilkan:

   ```
   Promise<void> startMyWF(
      int a,
      String b,
   ```

```
   StartWorkflowOptions optionOverrides,
   Promise<?>...waitFor);
```

Kelebihan beban ini harus digunakan ketika semua argumen alur kerja tersedia dan tidak perlu ditunggu, ketika Anda ingin mengganti pengaturan default yang digunakan untuk memulai eksekusi alur kerja, atau ketika Anda ingin menunggu beberapa janji lain untuk menjadi siap. Argumen variabel dapat digunakan untuk meneruskan objek `Promise<?>` yang tidak dideklarasikan sebagai argumen, tetapi Anda ingin menunggu sebelum menjalankan panggilan.

4. Kelebihan beban dengan setiap argumen dalam metode asli diganti dengan pembungkus `Promise<>`. Jenis kembalinya kelebihan beban ini akan `Promise<Void>` jika metode asli kembali `void`; jika tidak, itu akan menjadi `Promise<>` seperti yang dinyatakan pada metode asli. Sebagai contoh:

   Metode asli:

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

   Metode yang dihasilkan:

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

   Kelebihan beban ini harus digunakan ketika argumen yang akan diteruskan ke eksekusi alur kerja harus dievaluasi secara asinkronus. Panggilan ke kelebihan metode ini tidak akan dijalankan sampai semua argumen yang diteruskan ke metode ini menjadi siap.

   Jika beberapa argumen sudah siap, kemudian mengkonversinya ke `Promise` yang sudah dalam keadaan siap melalui metode `Promise.asPromise(`*`value`*`)`. Sebagai contoh:

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

5. Kelebihan beban dengan setiap argumen dalam metode asli diganti dengan pembungkus `Promise<>`. Kelebihan beban juga memiliki argumen variabel tambahan tipe `Promise<?>`. Jenis kembalinya kelebihan beban ini akan `Promise<Void>` jika metode asli kembali `void`; jika tidak, itu akan menjadi `Promise<>` seperti yang dinyatakan pada metode asli. Sebagai contoh:

Metode asli:

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

Metode yang dihasilkan:

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

Kelebihan ini harus digunakan ketika argumen yang akan diteruskan ke eksekusi alur kerja harus dievaluasi secara asinkronus dan Anda ingin menunggu beberapa janji lain untuk menjadi siap juga. Panggilan ke kelebihan metode ini tidak akan dijalankan sampai semua argumen yang diteruskan ke metode ini menjadi siap.

6. Kelebihan beban dengan setiap argumen dalam metode asli diganti dengan pembungkus `Promise<?>`. Kelebihan beban juga mempunyai argumen tambahan jenis `StartWorkflowOptions` dan argumen variabel dari tipe `Promise<?>`. Jenis kembalinya kelebihan beban ini akan `Promise<Void>` jika metode asli kembali `void`; jika tidak, itu akan menjadi `Promise<>` seperti yang dinyatakan pada metode asli. Sebagai contoh:

   Metode asli:

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

   Metode yang dihasilkan:

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

   Gunakan kelebihan ini ketika argumen yang akan diteruskan ke eksekusi alur kerja akan dievaluasi secara asinkron dan Anda ingin mengganti pengaturan default yang digunakan untuk

memulai eksekusi alur kerja. Panggilan ke kelebihan metode ini tidak akan dijalankan sampai semua argumen yang diteruskan ke metode ini menjadi siap.

Sebuah metode juga dihasilkan sesuai dengan setiap sinyal dalam antarmuka alur kerja—misalnya:

Metode asli:

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

Metode yang dihasilkan:

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

Klien asinkron tidak berisi metode yang sesuai dengan metode yang dianotasi dengan `@GetState` di antarmuka asli. Karena pengambilan status memerlukan panggilan layanan web, itu tidak cocok untuk digunakan dalam alur kerja. Oleh karena itu, hanya disediakan melalui klien eksternal.

Klien mandiri dimaksudkan untuk digunakan dari dalam alur kerja untuk memulai eksekusi baru pada penyelesaian eksekusi saat ini. Metode pada klien ini mirip dengan yang ada di klien asinkronus, tapi mengembalikan `void`. Klien ini tidak memiliki metode yang sesuai dengan metode dianotasikan dengan `@Signal` dan `@GetState`. Untuk detail selengkapnya, lihat Alur Kerja berkelanjutan.

Klien yang dihasilkan berasal dari antarmuka dasar: `WorkflowClient` dan `WorkflowClientExternal`, masing-masing menyediakan metode yang dapat Anda gunakan untuk membatalkan atau mengakhiri eksekusi alur kerja. Untuk detail lebih lanjut tentang antarmuka ini, lihat dokumentasi AWS SDK untuk Java .

Klien yang dihasilkan mengizinkan Anda berinteraksi dengan eksekusi alur kerja dengan cara yang diketik dengan kuat. Setelah dibuat, sebuah instans dari klien yang dihasilkan terikat ke eksekusi alur kerja tertentu dan hanya dapat digunakan untuk eksekusi tersebut. Selain itu, kerangka kerja juga menyediakan klien dinamis yang tidak spesifik untuk jenis alur kerja atau eksekusi. Klien yang dihasilkan bergantung pada klien ini secara rahasia. Anda juga dapat langsung menggunakan klien ini. Lihat bagian di Klien dinamis.

Kerangka kerja ini juga menghasilkan pabrik untuk menciptakan klien yang diketik dengan kuat. Pabrik klien yang dihasilkan untuk contoh antarmuka `MyWorkflow` adalah:

```
//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);
}
```

Antarmuka `WorkflowClientFactory` dasar adalah:

```
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);
}
```

Anda harus menggunakan pabrik-pabrik ini untuk membuat contoh klien. Pabrik memungkinkan Anda untuk mengkonfigurasi klien generik (klien generik harus digunakan untuk menyediakan

implementasi klien kustom) dan `DataConverter` digunakan oleh klien untuk menyusun data, serta opsi yang digunakan untuk memulai Eksekusi alur kerja. Untuk detail selengkapnya, lihat bagian DataConverters dan Eksekusi Alur Kerja Anak. `StartWorkflowOptions` berisi pengaturan yang dapat Anda gunakan untuk menimpa default—misalnya, batas waktu—ditentukan pada waktu pendaftaran. Untuk detail lebih lanjut tentang `StartWorkflowOptions` kelas, lihat AWS SDK untuk Java dokumentasi.

Klien eksternal dapat digunakan untuk memulai eksekusi alur kerja dari luar lingkup alur kerja sementara klien asinkron dapat digunakan untuk memulai eksekusi alur kerja dari kode dalam alur kerja. Untuk memulai eksekusi, Anda cukup menggunakan klien yang dihasilkan untuk memanggil metode yang sesuai dengan metode dijelaskan dengan `@Execute` dalam antarmuka alur kerja.

Kerangka kerja ini juga menghasilkan kelas implementasi untuk antarmuka klien. Klien ini membuat dan mengirim permintaan ke Amazon SWF untuk melakukan tindakan yang tepat. Versi klien dari `@Execute` metode ini memulai eksekusi alur kerja baru atau membuat eksekusi alur kerja turunan menggunakan Amazon SWF. APIs Demikian pula, versi klien dari `@Signal` metode ini menggunakan Amazon SWF APIs untuk mengirim sinyal.

> **Note**
>
> Klien alur kerja eksternal harus dikonfigurasi dengan klien Amazon SWF dan domain. Anda dapat menggunakan konstruktor pabrik klien yang menggunakan ini sebagai parameter atau meneruskan implementasi klien generik yang sudah dikonfigurasi dengan klien dan domain Amazon SWF.
>
> Kerangka kerja menjalankan hirarki jenis antarmuka alur kerja dan juga menghasilkan antarmuka klien untuk antarmuka alur kerja induk dan berasal dari mereka.

Klien aktivitas

Mirip dengan klien alur kerja, klien dibuat untuk setiap antarmuka dijelaskan dengan `@Activities`. Artifact yang dihasilkan termasuk antarmuka sisi klien dan kelas klien. Antarmuka yang dihasilkan untuk contoh antarmuka `@Activities` di atas (`MyActivities`) adalah sebagai berikut:

```
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);
}
```

Antarmuka berisi satu set metode kelebihan beban yang sesuai dengan setiap metode aktivitas di antarmuka `@Activities`. Kelebihan beban ini disediakan untuk kenyamanan dan mengizinkan kegiatan panggilan secara asinkronus. Untuk setiap metode aktivitas di antarmuka `@Activities`, metode berikut kelebihan beban yang dihasilkan dalam antarmuka klien:

1. Kelebihan yang mengambil argumen asli apa adanya. Jenis kembalinya kelebihan beban ini adalah `Promise<`*T*`>`, dimana *T* adalah jenis kembali dari metode asli. Sebagai contoh:

   Metode asli:

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

   Metode yang dihasilkan:

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

   Kelebihan beban ini harus digunakan ketika semua argumen alur kerja tersedia dan tidak perlu ditunggu.

2. Kelebihan beban yang mengambil argumen asli apa adanya, argumen tipe `ActivitySchedulingOptions` dan argumen variabel tambahan tipe `Promise<?>`. Jenis kembalinya kelebihan beban ini adalah `Promise<`*T*`>`, dimana *T* adalah jenis kembali dari metode asli. Sebagai contoh:

   Metode asli:

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

Metode yang dihasilkan:

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

Kelebihan beban ini harus digunakan ketika semua argumen dari alur kerja tersedia dan tidak perlu ditunggu, ketika Anda ingin mengganti pengaturan default, atau ketika Anda ingin menunggu tambahan `Promise` untuk menjadi siap. Argumen variabel dapat digunakan untuk meneruskan objek `Promise<?>` tambahan yang tidak dideklarasikan sebagai argumen, tetapi Anda ingin menunggu sebelum menjalankan panggilan.

3. Kelebihan beban dengan setiap argumen dalam metode asli diganti dengan pembungkus `Promise<>`. Jenis kembalinya kelebihan beban ini adalah `Promise<`*T*`>`, dimana *T* adalah jenis kembali dari metode asli. Sebagai contoh:

   Metode asli:

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

   Metode yang dihasilkan:

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

   Kelebihan beban ini harus digunakan ketika argumen yang akan diteruskan ke aktivitas akan dievaluasi secara asinkronus. Panggilan ke kelebihan metode ini tidak akan dijalankan sampai semua argumen yang diteruskan ke metode ini menjadi siap.

4. Kelebihan beban dengan setiap argumen dalam metode asli diganti dengan pembungkus `Promise<>`. Kelebihan beban juga mempunyai argumen tambahan jenis `ActivitySchedulingOptions` dan argumen variabel dari tipe `Promise<?>`. Jenis kembalinya kelebihan beban ini adalah `Promise<`*T*`>`, dimana *T* adalah jenis kembali dari metode asli. Sebagai contoh:

   Metode asli:

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

Metode yang dihasilkan:

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

Kelebihan beban ini harus digunakan ketika argumen yang akan dilewatkan ke aktivitas akan dievaluasi secara asinkronus, ketika Anda ingin menimpa pengaturan default terdaftar dengan tipe, atau ketika Anda ingin menunggu tambahan `Promise` untuk menjadi siap. Panggilan ke kelebihan metode ini tidak akan dijalankan sampai semua argumen yang diteruskan ke metode ini menjadi siap. Kelas klien yang dihasilkan mengimplementasikan antarmuka ini. Implementasi setiap metode antarmuka membuat dan mengirimkan permintaan ke Amazon SWF untuk menjadwalkan tugas aktivitas dari jenis yang sesuai menggunakan Amazon SWF. APIs

5. Overload yang mengambil argumen asli sebagaimana adanya dan argumen variabel tambahan tipe `Promise<?>`. Jenis kembalinya kelebihan beban ini adalah `Promise<`*T*`>`, dimana *T* adalah jenis kembali dari metode asli. Sebagai contoh:

   Metode asli:

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

   Metode yang dihasilkan:

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

   Kelebihan beban ini harus digunakan ketika semua argumen aktivitas tersedia dan tidak perlu ditunggu, tetapi anda ingin menunggu objek `Promise` untuk menjadi siap.

6. Kelebihan beban dengan setiap argumen dalam metode asli diganti dengan pembungkus `Promise` dan argumen variabel tambahan dari tipe `Promise<?>`. Jenis kembalinya kelebihan beban ini adalah `Promise<`*T*`>`, dimana *T* adalah jenis kembali dari metode asli. Sebagai contoh:

   Metode asli:

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

Metode yang dihasilkan:

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

Kelebihan beban ini harus digunakan saat semua argumen aktivitas akan menunggu secara asinkronus dan Anda juga ingin menunggu beberapa `Promise` lainnya untuk menjadi siap. Panggilan ke kelebihan metode ini akan dieksekusi secara tidak asinkron ketika semua objek `Promise` yang diteruskan menjadi siap.

Aktivitas klien yang dihasilkan juga memiliki metode yang dilindungi sesuai dengan setiap metode aktivitas, bernama {*activity method name*}`Impl()`, yang dipanggil oleh semua aktivitas yang kelebihan beban. Anda dapat mengganti metode ini untuk membuat implementasi klien mock. Metode ini sebagai argumen: semua argumen untuk metode asli di pembungkus `Promise<>`, `ActivitySchedulingOptions`, dan argumen variabel tipe `Promise<?>`. Sebagai contoh:

Metode asli:

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

Metode yang dihasilkan:

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

Opsi Penjadwalan

Klien aktivitas yang dihasilkan memungkinkan Anda untuk meneruskan `ActivitySchedulingOptions` sebagai argumen. struktur `ActivitySchedulingOptions` berisi pengaturan yang menentukan konfigurasi tugas aktivitas yang dijadwalkan kerangka kerja di Amazon SWF. Pengaturan ini menimpa default yang ditetapkan sebagai pilihan pendaftaran. Untuk menentukan pilihan penjadwalan secara dinamis, buat objek `ActivitySchedulingOptions`,

mengkonfigurasi seperti yang diinginkan, dan menyebarkannya ke metode aktivitas. Pada contoh berikut, kami telah menentukan daftar tugas yang harus digunakan untuk tugas aktivitas. Ini akan menimpa daftar tugas terdaftar default untuk pemanggilan aktivitas ini.

```
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);
    }
}
```

Klien dinamis

Selain klien yang dihasilkan, kerangka kerja juga menyediakan tujuan umum klien —`DynamicWorkflowClient` dan `DynamicActivityClient`—yang dapat Anda gunakan untuk secara dinamis memulai eksekusi alur kerja, mengirim sinyal, menjadwalkan aktivitas, dll. Misalnya, Anda mungkin ingin menjadwalkan aktivitas yang jenisnya tidak diketahui pada waktu desain. Anda dapat menggunakan `DynamicActivityClient` untuk penjadwalan seperti tugas kegiatan. Demikian pula, Anda dapat secara dinamis menjadwalkan eksekusi alur kerja anak dengan menggunakan `DynamicWorkflowClient`. Pada contoh berikut, alur kerja mencari aktivitas dari basis data dan menggunakan klien aktivitas dinamis untuk menjadwalkannya:

```
//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);
}
```

Untuk lebih jelasnya, lihat AWS SDK untuk Java dokumentasi.

Memberi Sinyal dan Membatalkan Eksekusi Alur Kerja

Klien alur kerja yang dihasilkan memiliki metode yang sesuai dengan setiap sinyal yang dapat dikirim ke alur kerja. Anda dapat menggunakannya dari dalam alur kerja untuk mengirim sinyal ke eksekusi alur kerja lainnya. Ini menyediakan mekanisme diketik untuk mengirim sinyal. Namun, kadang Anda mungkin perlu secara dinamis menentukan nama sinyal—misalnya, ketika nama sinyal diterima dalam pesan. Anda dapat menggunakan klien alur kerja dinamis untuk secara dinamis mengirim sinyal ke setiap eksekusi alur kerja. Demikian pula, Anda dapat menggunakan klien untuk meminta pembatalan eksekusi alur kerja lain.

Dalam contoh berikut, alur kerja mencari eksekusi untuk dikirimi sinyal dari database dan mengirim sinyal secara dinamis menggunakan klien alur kerja dinamis.

```
//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);
}
```

Implementasi Alur Kerja

Untuk menerapkan alur kerja, Anda menulis kelas yang menerrapkan antarmuka `@Workflow` yang diinginkan. Misalnya, contoh antarmuka alur kerja (`MyWorkflow`) dapat diterapkan seperti:

```
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);
  }
}
```

Metode `@Execute` di kelas ini adalah titik masuk logika alur kerja. Karena kerangka kerja menggunakan replay untuk merekonstruksi status objek ketika tugas keputusan akan diproses, objek baru dibuat untuk setiap tugas keputusan.

Penggunaan `Promise<`*T*`>` sebagai parameter tidak diperbolehkan dalam metode `@Execute` dalam antarmuka `@Workflow`. Ini dilakukan karena membuat panggilan asinkron adalah murni keputusan

pemanggil. Implementasi alur kerja itu sendiri tidak bergantung pada apakah pemanggilan itu sinkron atau asinkron. Oleh karena itu, antarmuka klien yang dihasilkan memiliki kelebihan yang mengambil parameter `Promise<T>` sehingga metode ini dapat dipanggil secara asinkron.

Jenis pengembalian suatu metode `@Execute` hanya dapat berupa `void` atau `Promise<T>`. Perhatikan bahwa jenis pengembalian klien eksternal yang sesuai adalah `void` dan bukan `Promise<>`. Karena klien eksternal tidak dimaksudkan untuk digunakan dari kode asinkron, klien eksternal tidak mengembalikan objek. `Promise` Untuk mendapatkan hasil eksekusi alur kerja yang dinyatakan secara eksternal, Anda bisa mendesain alur kerja untuk memperbarui status di penyimpanan data eksternal melalui aktivitas. Visibilitas Amazon SWF juga APIs dapat digunakan untuk mengambil hasil alur kerja untuk tujuan diagnostik. Anda tidak disarankan untuk menggunakan visibilitas APIs untuk mengambil hasil eksekusi alur kerja sebagai praktik umum karena panggilan API ini mungkin dibatasi oleh Amazon SWF. Visibilitas APIs mengharuskan Anda untuk mengidentifikasi eksekusi alur kerja menggunakan struktur. `WorkflowExecution` Anda bisa mendapatkan struktur ini dari klien alur kerja yang dihasilkan dengan memanggil metode `getWorkflowExecution`. Metode ini akan mengembalikan struktur `WorkflowExecution` yang sesuai dengan eksekusi alur kerja yang terikat pada klien. Lihat Referensi API Layanan Alur Kerja Sederhana Amazon untuk detail selengkapnya tentang APIs visibilitas.

Saat memanggil aktivitas dari implementasi alur kerja Anda, Anda harus menggunakan klien aktivitas yang dihasilkan. Demikian pula, untuk mengirim sinyal, gunakan klien alur kerja yang dihasilkan.

Konteks Keputusan

Kerangka kerja menyediakan konteks sekitar kapan saja kode alur kerja dijalankan oleh kerangka kerja. Konteks ini menyediakan fungsionalitas khusus konteks yang dapat Anda akses dalam implementasi alur kerja Anda, seperti membuat pengatur waktu. Lihat bagian di Konteks Eksekusi untuk informasi selengkapnya.

Mengekspos Status Eksekusi

Amazon SWF memungkinkan Anda menambahkan status kustom dalam riwayat alur kerja. Status terbaru yang dilaporkan oleh eksekusi alur kerja dikembalikan kepada Anda melalui panggilan visibilitas ke layanan Amazon SWF dan di konsol Amazon SWF. Misalnya, dalam alur kerja pemrosesan pesanan, Anda dapat melaporkan status pesanan pada tahapan yang berbeda seperti 'pesanan diterima', 'pesanan dikirim', dan seterusnya. Di AWS Flow Framework for Java, ini dilakukan melalui metode pada antarmuka alur kerja Anda yang dianotasi dengan anotasi `@GetState`. Ketika penentu selesai memproses tugas keputusan, itu memanggil metode ini untuk mendapatkan status terbaru dari implementasi alur kerja. Selain panggilan visibilitas, status juga dapat diambil

menggunakan klien eksternal yang dihasilkan (yang menggunakan panggilan API visibilitas secara internal).

Contoh berikut menunjukkan cara mengatur konteks eksekusi.

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

Klien eksternal yang dihasilkan dapat digunakan untuk mengambil status terbaru dari eksekusi alur kerja kapan saja.

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

Dalam contoh di atas, status eksekusi dilaporkan pada berbagai tahap. Saat instans alur kerja dimulai, `periodicWorkflow` melaporkan status awal sebagai 'Baru Dimulai'. Setiap panggilan ke

`callPeriodicActivity` kemudian memperbarui status alur kerja. Setelah `activity1` dipanggil 100 kali, metode kembali dan instans alur kerja selesai.

Alur Kerja Lokal

Terkadang, Anda mungkin perlu menggunakan variabel statis dalam implementasi alur kerja Anda. Misalnya, Anda mungkin ingin menyimpan penghitung yang akan diakses dari berbagai tempat (mungkin kelas yang berbeda) dalam implementasi alur kerja. Namun, Anda tidak dapat mengandalkan variabel statis dalam alur kerja Anda karena variabel statis dibagikan di seluruh utas, yang bermasalah karena seorang pekerja dapat memproses tugas keputusan yang berbeda pada utas yang berbeda secara bersamaan. Atau, Anda dapat menyimpan status tersebut di bidang pada implementasi alur kerja, tetapi kemudian Anda harus meneruskan objek implementasi. Untuk mengatasi kebutuhan ini, kerangka kerja menyediakan kelas `WorkflowExecutionLocal<?>`. Status apa pun yang perlu memiliki variabel statis seperti semantik harus disimpan sebagai instans lokal menggunakan `WorkflowExecutionLocal<?>`. Anda dapat mendeklarasikan dan menggunakan variabel statis jenis ini. Misalnya, dalam cuplikan berikut, `WorkflowExecutionLocal<String>` digunakan untuk menyimpan nama pengguna.

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

Implementasi Aktivitas

Kegiatan dilaksanakan dengan menyediakan implementasi dari Antarmuka `@Activities`. The AWS Flow Framework for Java menggunakan instance implementasi aktivitas yang dikonfigurasi pada pekerja untuk memproses tugas aktivitas pada waktu berjalan. Pekerja secara otomatis mencari implementasi aktivitas dari jenis yang sesuai.

Anda bisa menggunakan properti dan bidang untuk meneruskan sumber daya ke instans aktivitas, seperti koneksi database. Karena objek implementasi aktivitas dapat diakses dari beberapa utas, sumber daya bersama harus aman untuk utas.

Perhatikan bahwa implementasi kegiatan tidak mengambil parameter jenis `Promise<>` atau mengembalikan objek dari jenis itu. Hal ini karena implementasi kegiatan tidak harus bergantung pada bagaimana hal itu dipanggil (serentak atau asinkron).

Antarmuka kegiatan yang ditampilkan sebelumnya dapat diimplementasikan seperti ini:

```
public class MyActivitiesImpl implements MyActivities {

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

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

Sebuah thread konteks lokal tersedia untuk implementasi kegiatan yang dapat digunakan untuk mengambil objek tugas, objek konverter data yang digunakan, dll. Konteks saat ini dapat diakses melalui `ActivityExecutionContextProvider.getActivityExecutionContext()`. Untuk lebih jelasnya, lihat AWS SDK untuk Java dokumentasi untuk `ActivityExecutionContext` dan bagiannyaKonteks Eksekusi.

Menyelesaikan Aktivitas Secara Manual

Anotasi `@ManualActivityCompletion` pada contoh di atas adalah anotasi opsional. Hal ini diperbolehkan hanya pada metode yang menerapkan kegiatan dan digunakan untuk mengkonfigurasi aktivitas untuk tidak secara otomatis selesai ketika metode aktivitas kembali. Hal ini dapat berguna ketika Anda ingin menyelesaikan aktivitas asinkron—misalnya, secara manual setelah tindakan manusia telah selesai.

Secara default, kerangka menganggap aktivitas selesai ketika metode aktivitas Anda kembali. Ini berarti pekerja aktivitas melaporkan penyelesaian tugas aktivitas ke Amazon SWF dan memberikan hasilnya (jika ada). Namun, ada kasus penggunaan di mana Anda tidak ingin tugas aktivitas ditandai selesai ketika metode aktivitas kembali. Ini sangat berguna jika Anda memodelkan tugas manusia. Misalnya, metode aktivitas dapat mengirim email ke orang yang harus menyelesaikan beberapa pekerjaan sebelum tugas aktivitas selesai. Dalam kasus seperti itu, Anda dapat memberi anotasi pada metode aktivitas dengan anotasi `@ManualActivityCompletion` untuk memberi tahu pekerja aktivitas bahwa tidak harus menyelesaikan aktivitas secara otomatis. Untuk menyelesaikan aktivitas secara manual, Anda dapat menggunakan `ManualActivityCompletionClient` yang disediakan dalam kerangka kerja atau menggunakan metode `RespondActivityTaskCompleted` pada klien Java Amazon SWF yang disediakan di Amazon SWF SDK. Untuk detail selengkapnya, lihat AWS SDK untuk Java dokumentasi.

Dalam rangka untuk menyelesaikan tugas kegiatan, Anda perlu memberikan token tugas. Token tugas digunakan oleh Amazon SWF untuk mengidentifikasi tugas secara unik. Anda dapat mengakses token ini dari `ActivityExecutionContext` dalam implementasi aktivitas Anda. Anda harus mengirim token ini ke pihak yang bertanggung jawab untuk menyelesaikan tugas. Token ini dapat diambil dari `ActivityExecutionContext` dengan memanggil `ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken()`.

Aktivitas `getName` dari contoh Hello World dapat diimplementasikan untuk mengirim email yang meminta seseorang untuk memberikan pesan ucapan:

```
@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";
}
```

Potongan kode berikut dapat digunakan untuk memberikan ucapan dan menutup tugas dengan menggunakan `ManualActivityCompletionClient`. Atau, Anda juga dapat gagal dalam tugas:

```
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);
    }
}
```

Menerapkan AWS Lambda Tugas

Topik

- Tentang AWS Lambda
- Manfaat dan pembatasan dalam menggunakan tugas Lambda
- Menggunakan tugas Lambda di alur kerja AWS Flow Framework untuk Java Anda
- Lihat HelloLambda sampelnya

Tentang AWS Lambda

AWS Lambda adalah layanan komputasi yang dikelola sepenuhnya yang menjalankan kode Anda sebagai respons terhadap peristiwa yang dihasilkan oleh kode khusus atau dari berbagai AWS layanan seperti Amazon S3, DynamoDB, Amazon Kinesis, Amazon SNS, dan Amazon Cognito. Untuk informasi selengkapnya tentang Lambda, lihat Panduan Developer AWS Lambda.

Amazon Simple Workflow Service menyediakan tugas Lambda sehingga Anda dapat menjalankan fungsi Lambda untuk menggantikan, atau digunakan bersama kegiatan Amazon SWF tradisional.

> **Important**
>
> AWS Akun Anda akan dikenakan biaya untuk eksekusi (permintaan) Lambda yang dijalankan oleh Amazon SWF atas nama Anda. Untuk detail tentang harga Lambda, lihat https://aws.amazon.com/lambda/ harga/.

Manfaat dan pembatasan dalam menggunakan tugas Lambda

Ada sejumlah manfaat menggunakan tugas Lambda dibandingkan menjalankan kegiatan Amazon SWF tradisional:

- Tugas Lambda tidak perlu didaftarkan atau diversikan seperti tipe aktivitas Amazon SWF.
- Anda dapat menggunakan fungsi Lambda yang ada yang telah Anda tetapkan dalam alur kerja Anda.
- Fungsi Lambda dipanggil langsung oleh Amazon SWF; Anda tidak perlu menerapkan program pekerja untuk mengeksekusi fungsi seperti yang harus Anda lakukan pada kegiatan tradisional.
- Lambda menyediakan metrik dan pencatatan untuk melacak dan menganalisis eksekusi fungsi Anda.

Ada juga sejumlah batasan mengenai tugas Lambda yang harus Anda sadari:

- Tugas Lambda hanya dapat dijalankan di AWS wilayah yang memberikan dukungan untuk Lambda. Lihat Wilayah dan Titik Akhir Lambda di Referensi Umum Amazon Web Services untuk detail tentang wilayah yang saat ini didukung untuk Lambda.
- Tugas Lambda saat ini hanya didukung oleh API HTTP SWF dasar dan di untuk Java. AWS Flow Framework Saat ini tidak ada dukungan untuk tugas Lambda di Ruby AWS Flow Framework .

Menggunakan tugas Lambda di alur kerja AWS Flow Framework untuk Java Anda

Ada tiga persyaratan untuk menggunakan tugas Lambda dalam alur kerja AWS Flow Framework untuk Java Anda:

- Sebuah fungsi Lambda untuk mengeksekusi. Anda dapat menggunakan fungsi Lambda yang telah Anda tetapkan. Untuk informasi lebih lanjut tentang cara membuat fungsi Lambda, lihat AWS Lambda Developer Guide (Panduan Developer).
- IAM role yang menyediakan akses untuk menjalankan fungsi Lambda dari alur kerja Amazon SWF Anda.
- Kode untuk menjadwalkan tugas Lambda dari dalam alur kerja Anda.

Mengatur IAM role

Sebelum Anda dapat memanggil fungsi Lambda dari Amazon SWF Anda harus menyediakan IAM role yang menyediakan akses ke Lambda dari Amazon SWF. Anda dapat:

- pilih peran yang telah ditentukan sebelumnya, AWSLambdaPeran, untuk memberikan izin alur kerja Anda untuk menjalankan fungsi Lambda apa pun yang terkait dengan akun Anda.
- tentukan kebijakan Anda sendiri dan peran terkait untuk memberikan izin alur kerja untuk menjalankan fungsi Lambda tertentu, yang ditentukan oleh Nama Sumber Daya Amazon (). ARNs

Batasi izin pada peran IAM

Anda dapat membatasi izin pada peran IAM yang Anda berikan ke Amazon SWF dengan menggunakan kunci konteks `SourceAccount` dan dalam `SourceArn` kebijakan kepercayaan sumber daya Anda. Kunci ini membatasi penggunaan kebijakan IAM sehingga hanya digunakan dari eksekusi Amazon Simple Workflow Service yang termasuk dalam ARN domain yang ditentukan. Jika Anda menggunakan kedua kunci konteks kondisi global, `aws:SourceAccount` nilai dan akun yang

direferensikan dalam `aws:SourceArn` nilai harus menggunakan ID akun yang sama saat digunakan dalam pernyataan kebijakan yang sama.

Dalam contoh kebijakan kepercayaan berikut, kami menggunakan kunci `SourceArn` konteks untuk membatasi peran layanan IAM agar hanya digunakan dalam eksekusi Amazon Simple Workflow Service yang termasuk `someDomain` dalam akun,. `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"
       }
      }
    }
  ]
}
```

Dalam contoh kebijakan kepercayaan berikut, kami menggunakan kunci `SourceAccount` konteks untuk membatasi peran layanan IAM agar hanya digunakan dalam eksekusi Amazon Simple Workflow Service di akun,. `123456789012`

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

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

Menyediakan akses bagi Amazon SWF untuk memanggil setiap peran Lambda

Anda dapat menggunakan peran yang telah ditentukan sebelumnya, AWSLambdaPeran, untuk memberikan alur kerja Amazon SWF kemampuan untuk menjalankan fungsi Lambda apa pun yang terkait dengan akun Anda.

Untuk menggunakan AWSLambda Peran untuk memberi Amazon SWF akses untuk menjalankan fungsi Lambda

1. Buka Amazon IAM console (Konsol Amazon IAM).
2. Pilih Roles (Peran), kemudian Create New Role (Buat Peran Baru).
3. Beri nama bagi peran Anda, misalnya `swf-lambda` dan pilih Next Step (Langkah Selanjutnya).
4. Di bawah AWS Service Roles (Peran Layanan), pilih Amazon SWF, dan pilih Next Step (Langkah Berikutnya).
5. Pada layar Lampirkan Kebijakan, pilih AWSLambdaPeran dari daftar.
6. Pilih Next Step (Langkah Selanjutnya) dan kemudian Create Role (Buat Baru) setelah Anda meninjau peran tersebut.

Menentukan IAM role untuk menyediakan akses guna memanggil fungsi Lambda tertentu

Jika Anda ingin menyediakan akses untuk memanggil fungsi Lambda tertentu dari alur kerja Anda, Anda perlu menentukan kebijakan IAM Anda sendiri.

Untuk membuat kebijakan IAM untuk menyediakan akses ke fungsi Lambda tertentu

1. Buka Amazon IAM console (Konsol Amazon IAM).
2. Pilih Policies (Kebijakan), kemudian Create Policy (Buat Kebijakan).
3. Pilih Salin Kebijakan AWS Terkelola dan pilih AWSLambdaPeran dari daftar. Sebuah kebijakan akan dibuat untuk Anda. Anda memiliki opsi untuk mengedit nama dan deskripsi kebijakan untuk menyesuaikan kebutuhan Anda.

4. Di bidang Sumber Daya di Dokumen Kebijakan, tambahkan ARN dari satu atau beberapa fungsi Lambda Anda. Sebagai contoh:

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

> **Note**
>
> Untuk keterangan lengkap tentang cara menentukan sumber daya di IAM role, lihat Gambaran Umum Kebijakan IAM di Menggunakan IAM.

5. Pilih Create Policy (Buat Kebijakan) untuk menyelesaikan pembuatan kebijakan Anda.

Anda kemudian dapat memilih kebijakan ini saat membuat IAM role baru, dan menggunakan peran tersebut untuk memberikan akses panggilan ke alur kerja Amazon SWF Anda. Prosedur ini sangat mirip dengan membuat peran dengan kebijakan Peranan. sebagai gantinya, pilih kebijakan Anda sendiri saat membuat AWSLambdaperan.

Untuk membuat peran Amazon SWF menggunakan kebijakan Lambda Anda

1. Buka Amazon IAM console (Konsol Amazon IAM).
2. Pilih Roles (Peran), kemudian Create New Role (Buat Peran Baru).
3. Beri nama bagi peran Anda, misalnya `swf-lambda-function` dan pilih Next Step (Langkah Selanjutnya).
4. Di bawah AWS Service Roles (Peran Layanan), pilih Amazon SWF, dan pilih Next Step (Langkah Berikutnya).

5. Pada layar Attach Policy (Lampirkan Kebijakan), pilih kebijakan khusus fungsi Lambda Anda dari daftar.
6. Pilih Next Step (Langkah Selanjutnya) dan kemudian Create Role (Buat Baru) setelah Anda meninjau peran tersebut.

Jadwalkan tugas Lambda untuk eksekusi

Setelah Anda menentukan IAM role yang mengizinkan Anda untuk memanggil fungsi Lambda, Anda dapat menjadwalkan mereka untuk eksekusi sebagai bagian dari alur kerja Anda.

> **Note**
>
> Proses ini sepenuhnya ditunjukkan oleh HelloLambda sampel di AWS SDK untuk Java.

Untuk menjadwalkan tugas Lambda untuk eksekusi

1. Dalam penerapan alur kerja Anda, dapatkan instans `LambdaFunctionClient` dengan memanggil `getLambdaFunctionClient()` pada instans `DecisionContext`.

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

2. Jadwal tugas menggunakan metode `scheduleLambdaFunction()` di `LambdaFunctionClient`, dengan meneruskan metode nama fungsi Lambda yang Anda buat dan data input apa pun untuk tugas 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. Di starter eksekusi alur kerja Anda, tambahkan peran lambda IAM ke opsi alur kerja default dengan menggunakan `StartWorkflowOptions.withLambdaRole()`, dan kemudian teruskan pilihan ketika memulai alur kerja.

```
// 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);
```

Lihat HelloLambda sampelnya

Contoh yang menyediakan implementasi alur kerja yang menggunakan tugas Lambda disediakan di AWS SDK untuk Java. Untuk melihat dan/atau menjalankannya, download the source (unduh sumber).

Deskripsi lengkap tentang cara membangun dan menjalankan HelloLambdasampel disediakan dalam file README yang disediakan dengan sampel AWS Flow Framework untuk Java.

Menjalankan Program yang Ditulis dengan AWS Flow Framework untuk Java

Topik

- WorkflowWorker
- ActivityWorker
- Model Threading Pekerja
- Ekstensibilitas Pekerja

Kerangka kerja ini menyediakan kelas pekerja untuk menginisialisasi waktu aktif AWS Flow Framework for Java dan berkomunikasi dengan Amazon SWF. Untuk menerapkan alur kerja atau pekerja aktivitas, Anda harus membuat dan memulai instans kelas pekerja. Kelas pekerja ini bertanggung jawab untuk mengelola operasi asinkron yang sedang berlangsung, menjalankan metode asinkron yang tidak diblokir, dan berkomunikasi dengan Amazon SWF. Mereka dapat dikonfigurasi dengan alur kerja dan implementasi aktivitas, jumlah utas, daftar tugas untuk polling, dan sebagainya.

Kerangka kerja ini dilengkapi dengan dua kelas pekerja, satu untuk aktivitas dan satu untuk alur kerja. Untuk menjalankan logika alur kerja, Anda menggunakan kelas `WorkflowWorker`. Demikian pula untuk aktivitas kelas `ActivityWorker` digunakan. Kelas-kelas ini secara otomatis melakukan polling Amazon SWF untuk tugas aktivitas dan memanggil metode yang sesuai dalam implementasi Anda.

Contoh berikut menunjukkan cara membuat instans `WorkflowWorker` dan memulai polling untuk tugas:

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

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

Langkah-langkah dasar untuk membuat instans `ActivityWorker` dari dan memulai polling untuk tugas adalah sebagai berikut:

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

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

Saat Anda ingin mematikan aktivitas atau penentu, aplikasi Anda harus mematikan instans kelas pekerja yang digunakan serta instans klien Java Amazon SWF. Ini akan memastikan bahwa semua sumber daya yang digunakan oleh kelas pekerja dilepaskan dengan benar.

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

Untuk memulai eksekusi, cukup buat instans dari klien eksternal yang dihasilkan dan panggil metode `@Execute`.

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

WorkflowWorker

Seperti namanya, kelas pekerja ini dimaksudkan untuk digunakan oleh implementasi alur kerja. Ini dikonfigurasi dengan daftar tugas dan jenis implementasi alur kerja. Kelas pekerja menjalankan loop untuk polling untuk tugas keputusan dalam daftar tugas yang ditentukan. Ketika tugas keputusan diterima, itu membuat instans dari implementasi alur kerja dan memanggil metode `@Execute` untuk memproses tugas.

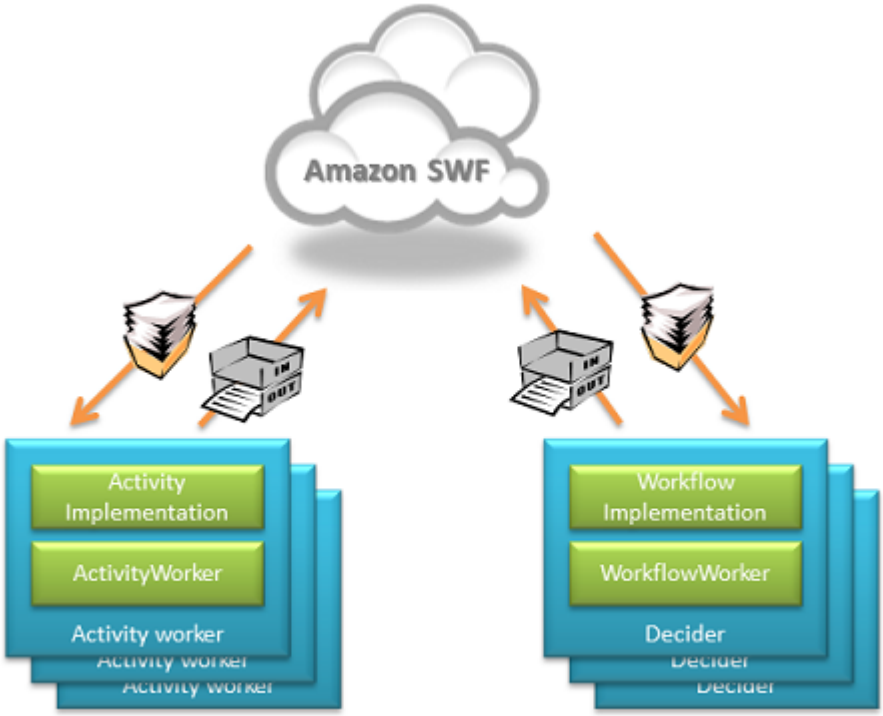
ActivityWorker

Untuk menerapkan pekerja aktivitas, Anda dapat menggunakan kelas `ActivityWorker` untuk dengan mudah melakukan polling daftar tugas untuk tugas aktivitas. Anda mengonfigurasi pekerja aktivitas dengan objek implementasi aktivitas. Kelas pekerja ini menjalankan loop untuk polling untuk tugas aktivitas dalam daftar tugas yang ditentukan. Saat tugas aktivitas diterima, ia mencari implementasi yang sesuai yang Anda berikan dan memanggil metode aktivitas untuk memproses tugas. Berbeda dengan `WorkflowWorker`, yang memanggil pabrik untuk membuat instans baru untuk setiap tugas keputusan, `ActivityWorker` hanya menggunakan objek yang Anda berikan.

`ActivityWorker`Kelas menggunakan anotasi AWS Flow Framework for Java untuk menentukan opsi pendaftaran dan eksekusi.

Model Threading Pekerja

Dalam AWS Flow Framework for Java, perwujudan dari suatu aktivitas atau decider adalah contoh dari kelas pekerja. Aplikasi Anda bertanggung jawab untuk mengonfigurasi dan membuat instans objek pekerja pada setiap mesin dan proses yang harus bertindak sebagai pekerja. Objek pekerja kemudian secara otomatis menerima tugas dari Amazon SWF, mengirimkannya ke aktivitas atau implementasi alur kerja Anda dan melaporkan hasilnya ke Amazon SWF. Dimungkinkan untuk satu instans alur kerja untuk menjangkau banyak pekerja. Saat Amazon SWF memiliki satu atau beberapa tugas aktivitas yang tertunda, ia menetapkan tugas ke pekerja pertama yang tersedia, lalu pekerja berikutnya, dan seterusnya. Hal ini memungkinkan tugas-tugas milik instans alur kerja yang sama untuk diproses pada pekerja yang berbeda secara bersamaan.

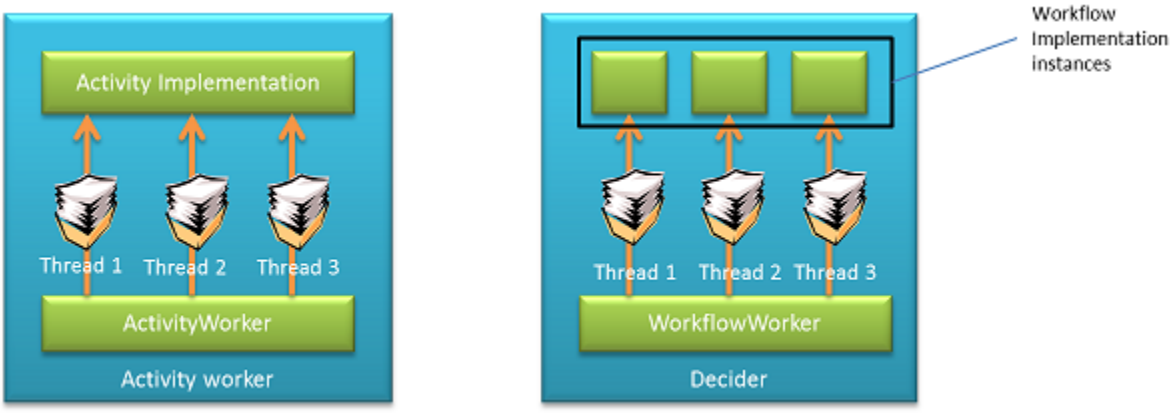


Selain itu, setiap pekerja dapat dikonfigurasi untuk memproses tugas di beberapa utas. Ini berarti bahwa tugas aktivitas dari instans alur kerja dapat berjalan secara bersamaan meskipun hanya ada satu pekerja.

Tugas keputusan berperilaku serupa dengan pengecualian bahwa Amazon SWF menjamin bahwa untuk eksekusi alur kerja tertentu, hanya satu keputusan yang dapat dieksekusi pada satu waktu. Eksekusi alur kerja tunggal biasanya memerlukan beberapa tugas keputusan; oleh karena itu, ini mungkin berakhir pada beberapa proses dan utas juga. Penentu dikonfigurasi dengan jenis implementasi alur kerja. Ketika tugas keputusan diterima oleh penentu, itu menciptakan sebuah

instans (objek) dari implementasi alur kerja. Kerangka kerja ini menyediakan pola pabrik yang dapat diperluas untuk membuat instans ini. Pabrik alur kerja default membuat objek baru setiap saat. Anda dapat menyediakan pabrik khusus untuk mengganti perilaku ini.

Bertentangan dengan penentu, yang dikonfigurasi dengan jenis implementasi alur kerja, pekerja aktivitas dikonfigurasikan dengan instans (objek) dari implementasi aktivitas. Ketika tugas aktivitas diterima oleh pekerja aktivitas, itu dikirim ke objek implementasi aktivitas yang sesuai.



Pekerja alur kerja mempertahankan satu kolam utas dan menjalankan alur kerja pada utas yang sama yang digunakan untuk polling Amazon SWF untuk tugas tersebut. Karena aktivitas berjalan lama (setidaknya jika dibandingkan dengan logika alur kerja), kelas pekerja aktivitas memelihara dua kumpulan utas yang terpisah; satu untuk polling Amazon SWF untuk tugas aktivitas dan yang lainnya untuk memproses tugas dengan menjalankan implementasi aktivitas. Ini memungkinkan Anda untuk mengonfigurasi jumlah utas untuk polling untuk tugas yang terpisah dari jumlah utas untuk menjalankannya. Misalnya, Anda dapat memiliki sejumlah kecil utas untuk polling dan sejumlah besar utas untuk menjalankan tugas. Kelas pekerja aktivitas melakukan polling Amazon SWF untuk tugas hanya jika memiliki utas polling gratis serta utas gratis untuk memproses tugas.

Perilaku threading dan instans ini menyiratkan bahwa:

1. Implementasi aktivitas harus stateless. Anda tidak boleh menggunakan variabel instans untuk menyimpan status aplikasi di objek aktivitas. Namun, Anda dapat menggunakan bidang untuk menyimpan sumber daya seperti koneksi basis data.
2. Implementasi aktivitas harus utas aman. Karena instance yang sama dapat digunakan untuk memproses tugas dari thread yang berbeda pada saat yang sama, akses ke sumber daya bersama dari kode aktivitas harus disinkronkan.
3. Implementasi alur kerja dapat berupa stateful, dan variabel instans dapat digunakan untuk menyimpan status. Meskipun instans baru dari implementasi alur kerja dibuat untuk memproses

setiap tugas keputusan, kerangka kerja akan memastikan bahwa status dibuat ulang dengan benar. Namun, implementasi alur kerja harus deterministik. Lihat bagian Memahami Tugas di AWS Flow Framework Java untuk lebih jelasnya.

4. Implementasi alur kerja tidak perlu utas aman saat menggunakan pabrik default. Implementasi default memastikan bahwa hanya satu utas yang menggunakan instans implementasi alur kerja pada satu waktu.

Ekstensibilitas Pekerja

The AWS Flow Framework for Java juga berisi beberapa kelas pekerja tingkat rendah yang memberi Anda kontrol halus serta ekstensibilitas. Dengan menggunakannya, Anda dapat sepenuhnya menyesuaikan alur kerja dan pendaftaran jenis aktivitas dan mengatur pabrik untuk membuat objek implementasi. Para pekerja ini adalah `GenericWorkflowWorker` dan `GenericActivityWorker`.

`GenericWorkflowWorker` dapat dikonfigurasi dengan pabrik untuk membuat pabrik definisi alur kerja. Pabrik definisi alur kerja bertanggung jawab untuk membuat contoh implementasi alur kerja dan untuk menyediakan pengaturan konfigurasi seperti opsi pendaftaran. Dalam keadaan normal, Anda harus menggunakan kelas `WorkflowWorker` secara langsung. Ini akan secara otomatis membuat dan mengonfigurasi implementasi pabrik yang disediakan dalam kerangka kerja, `POJOWorkflowDefinitionFactoryFactory` dan `POJOWorkflowDefinitionFactory`. Pabrik mengharuskan kelas implementasi alur kerja harus memiliki konstruktor tanpa argumen. Konstruktor ini digunakan untuk membuat instans objek alur kerja pada saat waktu aktif. Pabrik melihat anotasi yang Anda gunakan pada antarmuka alur kerja dan implementasi untuk membuat opsi pendaftaran dan eksekusi yang sesuai.

Anda dapat menyediakan implementasi pabrik Anda sendiri dengan menerapkan `WorkflowDefinitionFactory`, `WorkflowDefinitionFactoryFactory`, dan `WorkflowDefinition`. Kelas `WorkflowDefinition` digunakan oleh kelas pekerja untuk mengirimkan tugas keputusan dan sinyal. Dengan menerapkan kelas dasar ini, Anda dapat sepenuhnya menyesuaikan pabrik dan pengiriman permintaan ke implementasi alur kerja. Misalnya, Anda dapat menggunakan titik ekstensibilitas ini untuk menyediakan model pemrograman kustom untuk menulis alur kerja, misalnya, berdasarkan anotasi Anda sendiri atau membuatnya dari WSDL alih-alih pendekatan kode pertama yang digunakan oleh kerangka kerja. Untuk menggunakan pabrik kustom Anda, Anda harus menggunakan kelas `GenericWorkflowWorker`. Untuk detail selengkapnya tentang kelas-kelas ini, lihat AWS SDK untuk Java dokumentasi.

Demikian pula, `GenericActivityWorker` memungkinkan Anda untuk menyediakan pabrik implementasi aktivitas kustom. Dengan menerapkan kelas `ActivityImplementationFactory` and `ActivityImplementation` Anda dapat sepenuhnya mengontrol pembuatan instans aktivitas serta menyesuaikan opsi pendaftaran dan eksekusi. Untuk detail lebih lanjut dari kelas-kelas ini, lihat AWS SDK untuk Java dokumentasi.

Konteks Eksekusi

Topik

Kerangka kerja ini memberikan konteks sekitar untuk alur kerja dan implementasi aktivitas. Konteks ini khusus untuk tugas yang sedang diproses dan menyediakan beberapa utilitas yang dapat Anda gunakan dalam pelaksanaan Anda. Sebuah objek konteks dibuat setiap kali tugas baru diproses oleh pekerja.

Konteks Keputusan

Ketika tugas keputusan dijalankan, kerangka kerja menyediakan konteks untuk implementasi alur kerja melalui kelas `DecisionContext`. `DecisionContext` menyediakan informasi konteks-sensitif seperti eksekusi alur kerja menjalankan Id dan jam dan fungsi timer.

Mengakses dalam Implementasi DecisionContext Alur Kerja

Anda dapat mengakses `DecisionContext` dalam penerapan alur kerja Anda menggunakan kelas `DecisionContextProviderImpl`. Atau, Anda dapat menyuntikkan konteks di lapangan atau properti pelaksanaan alur kerja Anda menggunakan Spring seperti yang ditunjukkan di bagian injeksi Testabilitas dan Ketergantungan.

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

Membuat Jam dan Timer

`DecisionContext` berisi properti tipe `WorkflowClock` yang menyediakan fungsi timer dan jam. Karena logika alur kerja harus deterministik, Anda tidak boleh langsung menggunakan jam

sistem dalam implementasi alur kerja Anda. Metode `currentTimeMills` pada `WorkflowClock` mengembalikan waktu acara awal keputusan yang sedang diproses. Hal ini memastikan bahwa Anda mendapatkan nilai waktu yang sama selama replay, maka, membuat logika alur kerja Anda deterministik.

`WorkflowClock` juga memiliki metode `createTimer` yang mengembalikan sebuah objek `Promise` yang menjadi siap setelah interval tertentu. Anda dapat menggunakan nilai ini sebagai parameter untuk metode asinkron lainnya untuk menunda eksekusi mereka dengan jangka waktu tertentu. Dengan cara ini Anda dapat secara efektif menjadwalkan metode asinkron atau aktivitas untuk eksekusi di lain waktu.

Contoh dalam daftar berikut menunjukkan cara memanggil suatu aktivitas secara berkala.

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

Dalam daftar di atas, metode asinkron `callPeriodicActivity` memanggil `activity1` dan kemudian menciptakan timer menggunakan `AsyncDecisionContext` saat ini. Ini meneruskan kembali `Promise` sebagai argumen untuk panggilan rekursif untuk dirinya sendiri. Panggilan rekursif ini menunggu sampai timer menyala (1 jam dalam contoh ini) sebelum mengeksekusi.

Konteks Eksekusi Aktivitas

Sama seperti `DecisionContext` memberikan informasi konteks ketika tugas keputusan sedang diproses, `ActivityExecutionContext` menyediakan informasi konteks yang sama ketika tugas aktivitas sedang diproses. Konteks ini tersedia untuk kode aktivitas Anda melalui kelas `ActivityExecutionContextProviderImpl`.

```
ActivityExecutionContextProvider provider
    = new ActivityExecutionContextProviderImpl();
```

```
ActivityExecutionContext aec = provider.getActivityExecutionContext();
```

Menggunakan `ActivityExecutionContext`, Anda dapat melakukan hal berikut:

Heartbeat Aktivitas Berjalan Panjang

Jika aktivitas ini berjalan lama, itu harus secara berkala melaporkan kemajuan ke Amazon SWF untuk membiarkan ia tahu bahwa tugas masih membuat kemajuan. Dengan tidak adanya heartbeat seperti itu, tugas mungkin timeout jika tugas waktu habis heartbeat ditetapkan pada jenis aktivitas pendaftaran atau saat penjadwalan kegiatan. Untuk mengirim heartbeat, Anda dapat menggunakan metode `recordActivityHeartbeat` pada `ActivityExecutionContext`. Heartbeat juga menyediakan mekanisme untuk membatalkan aktivitas yang sedang berlangsung. Lihat bagian Penanganan Kesalahan untuk detai lebih lanjut dan contoh.

Dapatkan Detail Tugas Aktivitas

Jika Anda mau, Anda bisa mendapatkan semua detail tugas aktivitas yang disahkan oleh Amazon SWF saat pelaksana mendapat tugas. Ini termasuk informasi mengenai masukan untuk tugas, jenis tugas, tugas token, dll. Jika Anda ingin menerapkan aktivitas yang diselesaikan secara manual—misalnya, dengan tindakan manusia—maka Anda harus menggunakan `ActivityExecutionContext` untuk mengambil token tugas dan menyebarkannya ke proses yang akhirnya akan menyelesaikan tugas aktivitas. Lihat bagian di Menyelesaikan Aktivitas Secara Manual untuk lebih detailnya.

Dapatkan Amazon SWF Client Object yang Sedang Digunakan oleh Pelaksana

Objek klien Amazon SWF yang digunakan oleh pelaksana dapat diambil dengan memanggil metode `getService` pada `ActivityExecutionContext`. Hal ini berguna jika Anda ingin membuat panggilan langsung ke layanan Amazon SWF.

Eksekusi Alur Kerja Anak

Dalam contoh sejauh ini, kita sudah mulai eksekusi alur kerja langsung dari aplikasi. Namun, eksekusi alur kerja dapat dimulai dari dalam alur kerja dengan memanggil metode titik entri alur kerja pada klien yang dihasilkan. Ketika eksekusi alur kerja dimulai dari konteks eksekusi alur kerja lain, hal itu disebut eksekusi alur kerja anak. Hal ini memungkinkan Anda untuk memfaktor ulang (refactor) alur kerja yang kompleks menjadi unit yang lebih kecil dan berpotensi membagikannya di alur kerja yang berbeda. Misalnya, Anda dapat membuat alur kerja pemrosesan pembayaran dan memanggilnya dari alur kerja pemrosesan pesanan.

Secara semantik, eksekusi alur kerja anak berperilaku sama dengan alur kerja mandiri kecuali untuk perbedaan berikut:

1. Bila alur kerja induk berakhir karena tindakan eksplisit oleh pengguna—misalnya, dengan memanggil Amazon SWF API `TerminateWorkflowExecution`, atau diakhiri karena timeout—maka nasib eksekusi alur kerja anak akan ditentukan oleh kebijakan anak. Anda dapat menyetel kebijakan anak ini untuk menghentikan, membatalkan, atau mengabaikan (terus berjalan) eksekusi alur kerja anak.
2. Output dari alur kerja anak (nilai kembali dari metode entry point) dapat digunakan oleh eksekusi alur kerja induk seperti `Promise<T>` yang dikembalikan oleh metode asinkronus. Ini berbeda dari eksekusi mandiri di mana aplikasi harus mendapatkan output dengan menggunakan Amazon SWF. APIs

Pada contoh berikut, alur `OrderProcessor` membuat alur kerja anak `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);

}
```

Alur Kerja berkelanjutan

Dalam beberapa kasus penggunaan, Anda mungkin memerlukan alur kerja yang dijalankan selamanya atau berjalan dalam waktu lama, misalnya alur kerja yang memantau kondisi armada server.

> **Note**
>
> Karena Amazon SWF menyimpan seluruh riwayat eksekusi alur kerja, riwayat akan terus bertambah seiring waktu. Kerangka kerja mengambil riwayat ini dari Amazon SWF ketika melakukan replay, dan akan menjadi mahal jika ukuran sejarah terlalu besar. Dalam alur kerja yang berjalan lama atau berkelanjutan seperti itu, Anda harus secara berkala menutup eksekusi saat ini dan mulai yang baru untuk melanjutkan pemrosesan.

Ini merupakan kelanjutan logis dari eksekusi alur kerja. Klien mandiri yang dihasilkan dapat digunakan untuk tujuan ini. Dalam penerapan alur kerja Anda, cukup hubungi metode `@Execute` pada klien mandiri. Setelah eksekusi saat ini selesai, kerangka kerja akan memulai eksekusi baru menggunakan Id alur kerja yang sama.

Anda juga dapat melanjutkan eksekusi dengan memanggil metode `continueAsNewOnCompletion` pada `GenericWorkflowClient` yang dapat Anda ambil dari `DecisionContext`. Misalnya, implementasi alur kerja berikut menyetel timer untuk diaktifkan setelah satu hari dan memanggil titik masuknya sendiri untuk memulai eksekusi baru.

```
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();
    }
}
```

Ketika alur kerja secara rekursif memanggil alur kerjanya sendiri, kerangka kerja akan menutup alur kerja saat semua tugas tertunda telah selesai dan memulai eksekusi alur kerja baru. Perhatikan bahwa selama ada tugas tertunda, eksekusi alur kerja saat ini tidak akan tertutup. Eksekusi baru tidak akan secara otomatis mewarisi riwayat atau data dari eksekusi asli; jika Anda ingin membawa beberapa status ke eksekusi baru, maka Anda harus meneruskannya secara eksplisit sebagai input.

Menetapkan prioritas tugas di Amazon SWF

Secara default, tugas pada daftar tugas dikirimkan berdasarkan waktu kedatangan: tugas yang dijadwalkan pertama biasanya dijalankan pertama, sejauh mungkin. Dengan mengatur prioritas tugas opsional, Anda dapat memberikan prioritas bagi tugas tertentu: Amazon SWF akan mencoba untuk mengirimkan tugas dengan prioritas yang lebih tinggi pada daftar tugas terlebih dahulu sebelum tugas yang dengan prioritas lebih rendah.

Anda dapat mengatur prioritas tugas untuk alur kerja dan aktivitas. Prioritas tugas alur kerja tidak memengaruhi prioritas tugas aktivitas apa pun yang dijadwalkan, juga tidak memengaruhi alur kerja anak yang dimulai oleh alur kerja tersebut. Prioritas default untuk aktivitas atau alur kerja diatur (baik oleh Anda atau oleh Amazon SWF) selama pendaftaran, dan prioritas tugas terdaftar selalu digunakan kecuali diabil alih saat menjadwalkan aktivitas atau memulai eksekusi alur kerja.

Nilai prioritas tugas dapat berkisar antara “-2147483648” hingga “2147483647”, dengan angka yang lebih tinggi menunjukkan prioritas yang lebih tinggi. Jika Anda tidak mengatur prioritas tugas untuk aktivitas atau alur kerja, prioritas akan diberikan prioritas nol ("0").

Topik

- Mengatur Prioritas Tugas untuk Alur Kerja
- Mengatur Prioritas Tugas untuk Aktivitas

Mengatur Prioritas Tugas untuk Alur Kerja

Anda dapat mengatur prioritas tugas untuk alur kerja saat Anda mendaftarkannya atau memulainya. Prioritas tugas yang diatur saat tipe alur kerja terdaftar digunakan sebagai default untuk setiap eksekusi tipe alur kerja tersebut, kecuali jika diambil alih saat memulai eksekusi alur kerja.

Untuk mendaftarkan jenis alur kerja dengan prioritas tugas default, setel defaultTaskPriorityopsi WorkflowRegistrationOptionssaat mendeklarasikannya:

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

Anda juga dapat mengatur taskPriority untuk alur kerja saat Anda memulainya, menimpa prioritas tugas (default) terdaftar.

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

PriorityWorkflowClientExternalFactory cf
    = new PriorityWorkflowClientExternalFactoryImpl(swfService, domain);

priority_workflow_client = cf.getClient();

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

Selain itu, Anda dapat menetapkan prioritas tugas saat memulai alur kerja anak atau melanjutkan alur kerja sebagai baru. Misalnya, Anda dapat mengatur opsi ContinueAsNewWorkflowExecutionParametersTaskPriority di atau di. StartChildWorkflowExecutionParameters

Mengatur Prioritas Tugas untuk Aktivitas

Anda dapat mengatur prioritas tugas untuk suatu aktivitas baik ketika mendaftarkan maupun ketika menjadwalkan aktivitas tersebut. Prioritas tugas yang diatur ketika mendaftarkan tipe aktivitas digunakan sebagai prioritas default ketika aktivitas dijalankan, kecuali diambil alih ketika menjadwalkan aktivitas.

Untuk mendaftarkan jenis aktivitas dengan prioritas tugas default, setel defaultTaskPriorityopsi ActivityRegistrationOptionssaat mendeklarasikannya:

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

Anda juga dapat mengatur taskPriority untuk suatu aktivitas ketika Anda menjadwalkan prioritas tugas, menimpa prioritas tugas (default) terdaftar.

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

ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();

activityClient.doSomethingImportant(activityOptions);
```

DataConverters

Saat implementasi alur kerja Anda memanggil aktivitas jarak jauh, input yang diteruskan ke sana dan hasil eksekusi aktivitas harus diserialisasi sehingga dapat dikirim melalui kabel. Framework menggunakan DataConverter kelas untuk tujuan ini. Ini adalah kelas abstrak yang dapat Anda menerapkan untuk memberikan serializer Anda sendiri. Implementasi default berbasis serializer Jackson, `JsonDataConverter`, disediakan dalam kerangka kerja. Untuk detail selengkapnya, lihat Dokumentasi AWS SDK untuk Java. Lihat dokumentasi Jackson JSON Processor untuk detail tentang bagaimana Jackson melakukan serialisasi serta penjelasan Jackson yang dapat digunakan untuk mempengaruhinya. Format wire yang digunakan dianggap sebagai bagian dari kontrak. Oleh

karena itu, Anda dapat menentukan `DataConverter` pada antarmuka aktivitas dan alur kerja Anda dengan menetapkan properti `DataConverter` dari anotasi `@Activities` dan `@Workflow`.

Kerangka kerja ini akan membuat objek dari tipe `DataConverter` yang Anda tentukan pada anotasi `@Activities` untuk membuat serial input ke aktivitas dan untuk membatalkan serialisasi hasilnya. Demikian pula, objek dari tipe `DataConverter` yang Anda tentukan di anotasi `@Workflow` akan digunakan untuk cerita bersambung parameter yang Anda berikan ke alur kerja, dan dalam kasus alur kerja anak, untuk deserialize hasilnya. Selain input, kerangka kerja juga melewati data tambahan ke Amazon SWF—misalnya, detail pengecualian—serializer alur kerja akan digunakan untuk serialisasi data ini juga.

Anda juga dapat memberikan sebuah instans dari `DataConverter` jika Anda tidak ingin kerangka kerja untuk secara otomatis membuatnya. Klien yang dihasilkan memiliki overloads konstruktor yang mengambil `DataConverter`.

Jika Anda tidak menentukan tipe `DataConverter` dan tidak meneruskan objek `DataConverter`, `JsonDataConverter` digunakan secara default.

Meneruskan Data ke Metode Asinkron

Topik

Penggunaan `Promise<T>` telah dijelaskan di bagian sebelumnya. Beberapa kasus penggunaan `Promise<T>` lanjutan dibahas di sini.

Meneruskan Koleksi dan Maps ke Metode Asinkron

Kerangka ini mendukung meneruskan array, koleksi, dan peta sebagai jenis `Promise` metode asinkron. Sebagai contoh, metode asinkron dapat mengambil `Promise<ArrayList<String>>` sebagai argumen seperti yang ditunjukkan dalam daftar berikut.

```
@Asynchronous
```

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

Secara semantik, ini berperilaku seperti parameter `Promise` lain yang diketik dan metode asinkron akan menunggu hingga koleksi tersedia sebelum dieksekusi. Jika anggota koleksi adalah objek `Promise`, maka anda bisa membuat framework menunggu semua member siap seperti terlihat pada cuplikan berikut. Ini akan membuat metode asinkron menunggu pada setiap anggota koleksi untuk menjadi tersedia.

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

Perhatikan bahwa anotasi `@Wait` harus digunakan pada parameter untuk menunjukkan bahwa itu berisi Objek `Promise`.

Perhatikan juga bahwa aktivitas `printActivity` mengambil argumen `String` tetapi metode yang dihasilkan menggunakan Promise<String>. Kami memanggil metode pada klien dan tidak memanggil metode aktivitas secara langsung.

Dapat diatur <T>

`Settable<T>` adalah jenis turunan dari `Promise<T>` yang menyediakan satu set metode yang memungkinkan Anda untuk secara manual mengatur nilai `Promise`. Sebagai contoh, alur kerja berikut menunggu sinyal yang akan diterima dengan menunggu `Settable<?>`, yang diatur dalam metode sinyal:

```
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;
    }
}
```

Sebuah `Settable<?>` juga dapat dirantai ke janji lain pada suatu waktu. Anda dapat menggunakan `AndPromise` dan `OrPromise` untuk janji kelompok. Anda dapat melepaskan rantai `Settable` yang dirantai dengan memanggil metode `unchain()` di atasnya. Ketika dirantai, `Settable<?>` secara otomatis menjadi siap ketika janji yang dirantai menjadi siap. Perangkaian sangat berguna ketika Anda ingin menggunakan janji yang dikembalikan dari dalam lingkup `doTry()` di bagian lain dari program Anda. Karena `TryCatchFinally` digunakan sebagai kelas bersarang, Anda tidak dapat mendeklarasikan a `Promise<>` dalam lingkup induk dan mengaturnya. `doTry()` Hal ini karena Java membutuhkan variabel yang akan dinyatakan dalam lingkup induk dan digunakan dalam kelas bersarang untuk ditandai sebagai akhir. Sebagai contoh:

```
@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;
}
```

Sebuah `Settable` dapat dirantai ke satu janji pada satu waktu. Anda dapat melepaskan rantai `Settable` yang dirantai dengan memanggil metode `unchain()` di atasnya.

@NoWait

Ketika Anda meneruskan `Promise` ke metode asinkron, secara default, kerangka kerja akan menunggu `Promise` untuk menjadi siap sebelum mengeksekusi metode (kecuali untuk jenis koleksi). Anda dapat mengganti perilaku ini dengan menggunakan anotasi `@NoWait` pada parameter dalam deklarasi metode asinkron. Ini berguna jika Anda meneruskan di `Settable<T>`, yang akan ditetapkan oleh metode asinkron itu sendiri.

Janji <Void>

Dependensi dalam metode asinkron diimplementasikan dengan meneruskan `Promise` yang dikembalikan oleh satu metode sebagai argumen yang lain. Namun, mungkin ada kasus di mana Anda ingin mengembalikan `void` dari sebuah metode, tetapi masih ingin metode asinkron lainnya dijalankan setelah selesai. Dalam kasus seperti itu, Anda bisa menggunakan `Promise<Void>` sebagai tipe pengembalian metode. Kelas `Promise` menyediakan metode `Void` statis yang dapat Anda gunakan untuk membuat objek `Promise<Void>`. `Promise` Ini akan menjadi siap ketika metode asinkron selesai dijalankan. Anda bisa meneruskan `Promise` ini ke metode asinkron lain seperti objek `Promise` yang lain. Jika Anda menggunakan `Settable<Void>`, panggil metode set di atasnya dengan null untuk membuatnya siap.

AndPromise dan OrPromise

`AndPromise` dan `OrPromise` memungkinkan Anda untuk mengelompokkan beberapa objek `Promise<>` menjadi janji logis tunggal. `AndPromise` menjadi siap ketika semua janji yang digunakan untuk membangunnya siap. `OrPromise` menjadi siap ketika janji dalam koleksi janji yang digunakan untuk membangunnya siap. Anda dapat menelepon `getValues()` pada `AndPromise` dan `OrPromise` untuk mengambil daftar nilai dari janji konstituen.

Testabilitas dan Injeksi Ketergantungan

Topik

Kerangka kerja ini dirancang agar ramah Inversion of Control (IoC). Implementasi aktivitas dan alur kerja serta kerangka kerja yang disediakan pekerja dan objek konteks dapat dikonfigurasi dan dipakai menggunakan kontainer seperti Spring. Di luar kotak, kerangka kerja menyediakan integrasi dengan Kerangka Kerja Spring. Selain itu, integrasi dengan JUnit telah disediakan untuk alur kerja pengujian unit dan implementasi aktivitas.

Integrasi Spring

Paket com.amazonaws.services.simpleworkflow.flow.spring berisi kelas-kelas yang memudahkan penggunaan kerangka kerja Spring dalam aplikasi Anda. Ini mencakup aktivitas dan pekerja alur kerja yang sadar Scope dan Spring kustom: `WorkflowScope`, `SpringWorkflowWorker` dan `SpringActivityWorker`. Kelas-kelas ini memungkinkan Anda untuk mengonfigurasi implementasi alur kerja dan aktivitas Anda serta pekerja sepenuhnya melalui Spring.

WorkflowScope

`WorkflowScope` adalah implementasi Lingkup Musim Semi khusus yang disediakan oleh kerangka kerja. Lingkup ini memungkinkan Anda membuat objek di kontainer Spring yang masa pakainya dicakup dengan tugas keputusan. Bean dalam lingkup ini dipakai setiap kali tugas keputusan baru diterima oleh pekerja. Anda harus menggunakan lingkup ini untuk bean implementasi alur kerja dan bean lain yang bergantung padanya. Lingkup tunggal dan prototipe yang disediakan Spring tidak boleh digunakan untuk implementasi alur kerja karena kerangka kerja mengharuskan bean baru dibuat untuk setiap tugas keputusan. Kegagalan untuk melakukannya akan menghasilkan perilaku yang tidak terduga.

Contoh berikut menunjukkan cuplikan konfigurasi Spring yang mendaftarkan `WorkflowScope` lalu menggunakannya untuk mengonfigurasi bean implementasi alur kerja dan bean klien aktivitas.

```
<!-- 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>
```

Baris konfigurasi: `<aop:scoped-proxy proxy-target-class="false" />`, digunakan dalam konfigurasi bean `workflowImpl`, diperlukan karena `WorkflowScope` tidak mendukung proksi menggunakan CGLIB. Anda harus menggunakan konfigurasi ini untuk bean apa pun di `WorkflowScope` yang disambungkan ke bean lain dalam lingkup yang berbeda. Dalam hal ini, bean `workflowImpl` perlu disambungkan ke bean pekerja alur kerja dalam lingkup tunggal (lihat contoh lengkap di bawah).

Anda dapat mempelajari selengkapnya tentang menggunakan lingkup kustom di dokumentasi Kerangka Kerja Spring.

Pekerja Sadar Spring

Saat menggunakan Spring, Anda harus menggunakan kelas pekerja sadar Spring yang disediakan oleh kerangka kerja: `SpringWorkflowWorker` dan `SpringActivityWorker`. Pekerja ini dapat disuntikkan dalam aplikasi Anda menggunakan Spring seperti yang ditunjukkan pada contoh berikutnya. Pekerja sadar Spring menerapkan antarmuka `SmartLifecycle` Spring dan, secara default, secara otomatis memulai polling untuk tugas ketika konteks Spring diinisialisasi. Anda dapat menonaktifkan fungsi ini dengan menyetel properti `disableAutoStartup` pekerja ke `true`.

Contoh berikut menunjukkan cara mengonfigurasi penentu. Contoh ini menggunakan antarmuka `MyActivities` dan `MyWorkflow` (tidak ditampilkan di sini) dan implementasi yang sesuai, `MyActivitiesImpl` dan `MyWorkflowImpl`. Antarmuka dan implementasi

klien yang dihasilkan adalah `MyWorkflowClient`/`MyWorkflowClientImpl` and `MyActivitiesClient`/`MyActivitiesClientImpl` (juga tidak ditampilkan di sini).

Klien aktivitas disuntikkan dalam implementasi alur kerja menggunakan fitur kabel otomatis Spring:

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

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

Konfigurasi Spring untuk penentu adalah sebagai berikut:

```
<?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>
```

Karena sepenuhnya `SpringWorkflowWorker` dikonfigurasi di Musim Semi dan secara otomatis memulai polling ketika konteks Spring diinisialisasi, proses host untuk decider sederhana:

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

Demikian pula, pekerja aktivitas dapat dikonfigurasi sebagai berikut:

```
<?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>
```

Proses host pekerja aktivitas mirip dengan penentu:

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

Menyuntikkan Konteks Keputusan

Jika implementasi alur kerja Anda bergantung pada objek konteks, maka Anda dapat dengan mudah menyuntikkannya melalui Spring juga. Kerangka kerja secara otomatis mendaftarkan bean terkait konteks di kontainer Spring. Misalnya, dalam cuplikan berikut, berbagai objek konteks telah disambungkan secara otomatis. Tidak ada konfigurasi Spring lain dari objek konteks yang diperlukan.

```
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();
   }
}
```

Jika Anda ingin mengonfigurasi objek konteks dalam implementasi alur kerja melalui konfigurasi Spring XML, gunakan nama bean yang dideklarasikan di kelas `WorkflowScopeBeanNames` dalam paket com.amazonaws.services.simpleworkflow.flow.spring. Sebagai contoh:

```
<!-- 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>
```

Atau, Anda dapat menyuntikkan `DecisionContextProvider` dalam bean implementasi alur kerja dan menggunakannya untuk membuat konteks. Ini dapat berguna jika Anda ingin memberikan implementasi kustom dari penyedia dan konteks.

Menyuntikkan Sumber Daya dalam Aktivitas

Anda dapat membuat instans dan mengonfigurasi implementasi aktivitas menggunakan kontainer Inversion of Control (IoC) dan dengan mudah menyuntikkan sumber daya seperti koneksi basis data dengan mendeklarasikannya sebagai properti dari kelas implementasi aktivitas. Sumber daya tersebut biasanya akan dicakup sebagai tunggal. Perhatikan bahwa implementasi aktivitas dipanggil oleh pekerja aktivitas di beberapa utas. Oleh karena itu, akses ke sumber daya bersama harus disinkronkan.

JUnit Integrasi

Kerangka kerja menyediakan JUnit ekstensi serta implementasi pengujian objek konteks, seperti jam pengujian, yang dapat Anda gunakan untuk menulis dan menjalankan pengujian unit. JUnit Dengan ekstensi ini, Anda dapat menguji implementasi alur kerja Anda secara lokal di barisan.

Menulis Pengujian Unit Sederhana

Untuk menulis pengujian untuk alur kerja Anda, gunakan kelas `WorkflowTest` dalam paket com.amazonaws.services.simpleworkflow.flow.junit. Kelas ini adalah JUnit `MethodRule` implementasi khusus kerangka kerja dan menjalankan kode alur kerja Anda secara lokal, memanggil aktivitas sebaris sebagai lawan melalui Amazon SWF. Ini memberi Anda fleksibilitas untuk menjalankan pengujian sesering yang Anda inginkan tanpa dikenakan biaya apa pun.

Untuk menggunakan kelas ini, cukup deklarasikan bidang jenis `WorkflowTest` dan beri anotasi dengan anotasi `@Rule`. Sebelum menjalankan pengujian Anda, buat objek `WorkflowTest` baru dan tambahkan implementasi aktivitas serta alur kerja Anda ke dalamnya. Anda kemudian dapat menggunakan pabrik klien alur kerja yang dihasilkan untuk membuat klien dan memulai eksekusi alur kerja. Kerangka kerja ini juga menyediakan JUnit pelari khusus`FlowBlockJUnit4ClassRunner`, yang harus Anda gunakan untuk pengujian alur kerja Anda. Misalnya:

```
@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);
    }
}
```

Anda juga dapat menentukan daftar tugas terpisah untuk setiap implementasi aktivitas yang Anda tambahkan ke `WorkflowTest`. Misalnya, jika Anda memiliki implementasi alur kerja yang menjadwalkan aktivitas dalam daftar tugas khusus host, Anda dapat mendaftarkan aktivitas dalam daftar tugas setiap host:

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

Perhatikan bahwa kode dalam `@Test` asinkron. Oleh karena itu, Anda harus menggunakan klien alur kerja asinkron untuk memulai eksekusi. Untuk memverifikasi hasil tes Anda, kelas bantuan `AsyncAssert` juga disediakan. Kelas ini memungkinkan Anda menunggu hingga janji siap sebelum memverifikasi hasil. Dalam contoh ini, kami menunggu hasil eksekusi alur kerja siap sebelum memverifikasi hasil pengujian.

Jika Anda menggunakan Spring, maka kelas `SpringWorkflowTest` dapat digunakan sebagai pengganti kelas `WorkflowTest`. `SpringWorkflowTest` menyediakan properti yang dapat Anda gunakan untuk mengonfigurasi aktivitas dan implementasi alur kerja dengan mudah melalui konfigurasi Spring. Sama seperti pekerja sadar Spring, Anda harus menggunakan `WorkflowScope` untuk mengonfigurasi bean implementasi alur kerja. Ini memastikan bahwa bean implementasi alur kerja baru dibuat untuk setiap tugas keputusan. Pastikan untuk mengonfigurasi kacang ini dengan proxy-target-class pengaturan proxy cakupan yang disetel ke. `false` Lihat bagian Integrasi Spring untuk lebih detail. Contoh konfigurasi Spring yang ditampilkan di bagian Integrasi Spring dapat diubah untuk menguji alur kerja menggunakan `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>
```

Meniru Implementasi Aktivitas

Anda dapat menggunakan implementasi aktivitas nyata selama pengujian, tetapi jika Anda ingin pengujian unit hanya logika alur kerja, Anda harus meniru aktivitas tersebut. Ini dapat dilakukan dengan menyediakan implementasi tiruan dari antarmuka aktivitas ke kelas `WorkflowTest`. Sebagai contoh:

```
@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);
    }
}
```

Atau, Anda dapat memberikan implementasi tiruan dari klien aktivitas dan menyuntikkannya ke dalam implementasi alur kerja Anda.

Objek konteks pengujian

Jika implementasi alur kerja Anda bergantung pada objek konteks kerangka kerja—misalnya, `DecisionContext`—Anda tidak perlu melakukan sesuatu yang khusus untuk menguji alur kerja tersebut. Saat pengujian dijalankan melalui `WorkflowTest`, ia secara otomatis menyuntikkan objek konteks pengujian. Saat implementasi alur kerja Anda mengakses objek konteks—misalnya, menggunakan `DecisionContextProviderImpl`—ia akan mendapatkan implementasi pengujian. Anda dapat memanipulasi objek konteks pengujian ini dalam kode pengujian (metode `@Test`) Anda untuk membuat kasus pengujian yang menarik. Misalnya, jika alur kerja Anda membuat penghitung waktu, Anda bisa mengaktifkan penghitung dengan memanggil metode `clockAdvanceSeconds` di kelas `WorkflowTest` untuk menggerakkan jam maju tepat waktu. Anda juga dapat mempercepat jam untuk mengaktifkan penghitung lebih awal dari biasanya menggunakan properti `ClockAccelerationCoefficient` di `WorkflowTest`. Misalnya, jika alur kerja Anda membuat penghitung waktu selama satu jam, Anda dapat menyetel `ClockAccelerationCoefficient` ke 60 untuk mengaktifkan penghitung waktu dalam satu menit. Secara default, `ClockAccelerationCoefficient` disetel ke 1.

Untuk detail selengkapnya tentang paket com.amazonaws.services.simpleworkflow.flow.test dan com.amazonaws.services.simpleworkflow.flow.junit, lihat dokumentasi AWS SDK untuk Java .

Penanganan Kesalahan

Topik

`try/catch/finally` membangun di Java membuatnya sederhana untuk menangani kesalahan dan digunakan di mana-mana. Hal ini memungkinkan Anda untuk mengasosiasikan handler kesalahan untuk blok kode. Secara internal, ini bekerja dengan isian metadata tambahan tentang handler kesalahan pada tumpukan panggilan. Ketika pengecualian dilemparkan, waktu aktif terlihat di tumpukan panggilan untuk handler kesalahan terkait dan memanggil waktu aktif; dan jika tidak ada handler kesalahan yang tepat ditemukan, waktu aktif menyebarkan pengecualian ke atas rantai panggilan.

Ini bekerja dengan baik untuk kode sinkron, tapi penanganan kesalahan dalam program asinkron dan distribusi menimbulkan tantangan tambahan. Karena panggilan asinkron segera kembali, pemanggil tidak berada di tumpukan panggilan saat kode asinkron dijalankan. Ini berarti bahwa pengecualian yang tidak tertangani dalam kode asinkron tidak dapat ditangani oleh pemanggil dengan cara biasa. Biasanya, pengecualian yang berasal dalam kode asinkron ditangani dengan melewati status kesalahan ke panggilan balik yang diteruskan ke metode asinkron. Atau, jika `Future<?>` sedang digunakan, itu melaporkan kesalahan saat Anda mencoba mengaksesnya. Ini kurang dari ideal karena kode yang menerima pengecualian (callback atau kode yang menggunakan `Future<?>`) tidak memiliki konteks panggilan asli dan mungkin tidak dapat menangani pengecualian secara memadai. Selain itu, dalam sistem asinkron terdistribusi, dengan komponen yang berjalan bersamaan, lebih dari satu kesalahan dapat terjadi secara bersamaan. Kesalahan ini bisa dari berbagai jenis dan tingkat kepelikan dan perlu ditangani dengan tepat.

Membersihkan sumber daya setelah panggilan asinkron juga sulit. Tidak seperti kode sinkron, Anda tidak dapat menggunakan try/catch/finally dalam kode panggilan untuk membersihkan sumber daya karena pekerjaan yang dimulai di blok coba mungkin masih berlangsung ketika blok akhirnya dijalankan.

Framework ini menyediakan mekanisme yang membuat penanganan kesalahan dalam kode asinkron terdistribusi mirip dengan, dan hampir sesederhana, 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();
    }
  };
}
```

Kelas `TryCatchFinally` dan varian kelasnya, `TryFinally` dan `TryCatch`, pekerjaan yang mirip dengan Java `try`/`catch`/`finally`. Menggunakannya, Anda dapat mengasosiasikan handler pengecualian untuk blok kode alur kerja yang dapat mengeksekusi tugas sebagai asinkron dan jarak jauh. Metode `doTry()` secara logis setara dengan blok `try`. Kerangka kerja secara

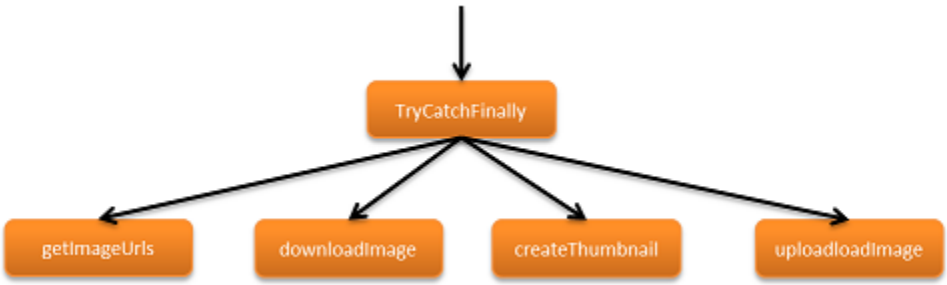
otomatis mengeksekusi kode di `doTry()`. Daftar objek `Promise` dapat diteruskan ke konstruktor `TryCatchFinally`. Metode `doTry` akan dieksekusi ketika semua objek `Promise` diteruskan ke konstruktor telah siap. Jika pengecualian dimunculkan oleh kode yang secara asinkron dipanggil dari dalam `doTry()`, setiap pekerjaan yang tertunda di `doTry()` dibatalkan dan `doCatch()` dipanggil untuk menangani pengecualian. Misalnya, dalam daftar di atas, jika `downloadImage` melempar pengecualian, maka `createThumbnail` dan `uploadImage` akan dibatalkan. Akhirnya, `doFinally()` dipanggil ketika semua pekerjaan asinkron dilakukan (selesai, gagal, atau dibatalkan). Hal ini dapat digunakan untuk pembersihan sumber daya. Anda juga dapat menyusun kelas ini untuk menyesuaikan dengan kebutuhan Anda.

Ketika pengecualian dilaporkan dalam `doCatch()`, kerangka kerja menyediakan tumpukan panggilan logis lengkap yang mencakup panggilan asinkron dan jarak jauh. Hal ini dapat membantu ketika debugging, terutama jika Anda memiliki metode asinkron memanggil metode asinkron lainnya. Sebagai contoh, pengecualian dari downloadImage akan menghasilkan pengecualian seperti ini:

```
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 Semantik

Eksekusi program AWS Flow Framework untuk Java dapat divisualisasikan sebagai pohon cabang yang mengeksekusi secara bersamaan. Panggilan ke metode asinkron, aktivitas, dan `TryCatchFinally` menciptakan sebuah cabang baru di pohon eksekusi ini. Misalnya, alur kerja pemrosesan citra dapat dilihat sebagai pohon yang ditunjukkan pada gambar berikut.



Kesalahan dalam satu cabang eksekusi akan menyebabkan pelepasan cabang itu, hanya sebagai pengecualian yang menyebabkan pelepasan tumpukan panggilan dalam program Java. Pelepasan

terus bergerak ke atas cabang eksekusi sampai baik kesalahan ditangani atau akar pohon tercapai, dalam hal eksekusi alur kerja yang diakhiri.

Kerangka kerja melaporkan kesalahan yang terjadi saat memproses tugas sebagai pengecualian. Alur kerja Ini mengaitkan handler pengecualian (metode `doCatch()`) didefinisikan dalam `TryCatchFinally` dengan semua tugas yang dibuat oleh kode yang sesuai `doTry()`. Jika tugas gagal-misalnya, karena batas waktu atau pengecualian tidak tertangani - maka pengecualian yang sesuai akan dimunculkan dan `doCatch()` akan dipanggil untuk menanganinya. Untuk mencapai hal ini, kerangka kerja bekerja bersama-sama dengan Amazon SWF untuk menyebarkan kesalahan jarak jauh dan membangkitkan mereka sebagai pengecualian dalam konteks pemanggil.

Pembatalan

Ketika pengecualian terjadi dalam kode sinkron, kontrol melompat langsung ke blok `catch`, melewatkan lebih dari kode yang tersisa di blok `try`. Sebagai contoh:

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

Dalam kode ini, jika `b()` melempar pengecualian, maka `c()` tidak pernah dipanggil. Membandingkan itu dengan alur kerja:

```
new TryCatch() {

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

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

```
};
```

Dalam hal ini, panggilan ke `activityA`, `activityB`, dan `activityC` semua kembali berhasil dan menghasilkan penciptaan tiga tugas yang akan dieksekusi secara asinkron. Katakanlah di lain waktu bahwa tugas untuk `activityB` menghasilkan kesalahan. Kesalahan ini dicatatkan di riwayat oleh Amazon SWF. Untuk menangani hal ini, kerangka kerja pertama akan mencoba untuk membatalkan semua tugas-tugas lain yang berasal dalam lingkup `doTry()` yang sama; dalam hal ini, `activityA` dan `activityC`. Ketika semua tugas tersebut selesai (membatalkan, gagal, atau berhasil diselesaikan), metode `doCatch()` yang cocok akan dipanggil untuk menangani kesalahan.

Berbeda dengan contoh sinkron, dimana `c()` tidak pernah dieksekusi, `activityC` dipanggil dan tugas dijadwalkan untuk eksekusi; oleh karena itu, kerangka kerja akan membuat upaya untuk membatalkannya, tetapi tidak ada jaminan bahwa itu akan dibatalkan. Pembatalan tidak dapat dijamin karena aktivitas mungkin telah selesai, dapat mengabaikan permintaan pembatalan, atau mungkin gagal karena kesalahan. Namun, kerangka kerja tidak memberikan jaminan bahwa `doCatch()` disebut hanya setelah semua tugas dimulai dari yang sesuai `doTry()` telah selesai. Hal ini juga menjamin bahwa `doFinally()` disebut hanya setelah semua tugas dimulai dari `doTry()` dan `doCatch()` telah selesai. Jika, misalnya, kegiatan dalam contoh di atas tergantung satu sama lain, katakanlah `activityB` tergantung pada `activityA` dan `activityC` pada `activityB`, maka pembatalan `activityC` akan segera karena tidak dijadwalkan di Amazon SWF sampai `activityB` selesai:

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

Aktivitas Heartbeat

Mekanisme pembatalan kooperatif AWS Flow Framework for Java memungkinkan tugas aktivitas dalam penerbangan dibatalkan dengan anggun. Ketika pembatalan dipicu, tugas yang diblokir atau menunggu untuk ditugaskan ke pekerja secara otomatis dibatalkan. Jika, bagaimanapun, tugas sudah ditugaskan ke pekerja, kerangka kerja akan meminta aktivitas untuk membatalkan. Implementasi aktivitas Anda harus secara eksplisit menangani permintaan pembatalan tersebut. Hal ini dilakukan dengan melaporkan heartbeat aktivitas Anda.

Pelaporan heartbeat mengizinkan pelaksanaan kegiatan untuk melaporkan kemajuan tugas kegiatan yang sedang berlangsung, yang berguna untuk pemantauan, dan memungkinkan pemeriksaan aktivitas untuk permintaan pembatalan. Metode `recordActivityHeartbeat` akan melemparkan `CancellationException` jika pembatalan telah diminta. Pelaksanaan kegiatan dapat menangkap pengecualian ini dan bertindak atas permintaan pembatalan, atau dapat mengabaikan permintaan dengan menelan pengecualian. Untuk menghormati permintaan pembatalan, aktivitas harus melakukan pembersihan yang diinginkan, jika ada, dan kemudian melempar kembali `CancellationException`. Ketika pengecualian ini dilemparkan dari pelaksanaan aktivitas, kerangka kerja mencatat bahwa tugas aktivitas telah selesai dalam status dibatalkan.

Contoh berikut menunjukkan aktivitas yang mengunduh dan memproses citra. Ini heartbeat setelah memproses setiap citra, dan jika pembatalan diminta, itu membersihkan dan melempar kembali pengecualian untuk mengakui pembatalan.

```
@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;
        }
    }
```

```
}
```

Melaporkan aktivitas heartbeat tidak diperlukan, tetapi disarankan jika aktivitas Anda berjalan lama atau mungkin melakukan operasi mahal yang ingin dibatalkan dalam kondisi kesalahan. Anda harus menelepon `heartbeatActivityTask` secara berkala dari pelaksanaan aktivitas.

Jika aktivitas habis, `ActivityTaskTimedOutException` akan dilempar dan `getDetails` pada objek pengecualian akan mengembalikan data yang diteruskan ke panggilan sukses terakhir ke `heartbeatActivityTask` untuk tugas aktivitas yang sesuai. Implementasi alur kerja dapat menggunakan informasi ini untuk menentukan berapa banyak kemajuan yang dibuat sebelum tugas aktivitas habis.

> **Note**
>
> Ini bukan praktik yang baik untuk heartbeat terlalu sering karena Amazon SWF mungkin throttle permintaan heartbeat. Lihat Panduan Developer Amazon Simple Workflow Service untuk batas yang ditempatkan oleh Amazon SWF.

Secara eksplisit Membatalkan tugas

Selain kondisi kesalahan, ada kasus lain di mana Anda dapat secara eksplisit membatalkan tugas. Misalnya, aktivitas untuk memproses pembayaran menggunakan kartu kredit mungkin perlu dibatalkan jika pengguna membatalkan pesanan. Kerangka kerja ini mengizinkan Anda untuk secara eksplisit membatalkan tugas yang dibuat dalam lingkup `TryCatchFinally`. Pada contoh berikut, tugas pembayaran dibatalkan jika sinyal diterima saat pembayaran sedang diproses.

```
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);
        }
    }
}
```

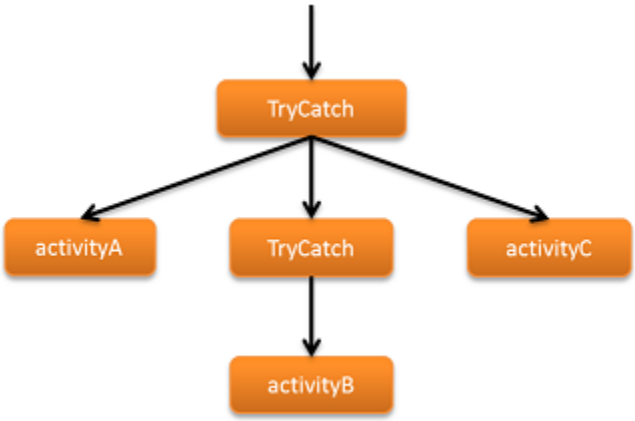
Menerima Notifikasi Tugas Dibatalkan

Ketika tugas selesai dalam keadaan dibatalkan, kerangka kerja menginformasikan logika alur kerja dengan melemparkan `CancellationException`. Ketika suatu aktivitas selesai dalam keadaan dibatalkan, catatan dibuat dalam riwayat dan kerangka kerja panggilan yang sesuai `doCatch()` dengan `CancellationException`. Seperti yang ditunjukkan dalam contoh sebelumnya, ketika pembayaran proses tugas dibatalkan, alur kerja menerima `CancellationException`.

Sebuah `CancellationException` tidak tertangani disebarkan cabang eksekusi seperti exception lainnya. Namun, metode `doCatch()` akan menerima `CancellationException` jika tidak ada pengecualian lain dalam ruang lingkup; pengecualian lain diprioritaskan lebih tinggi dari pembatalan.

Bersarang TryCatchFinally

Anda dapat mengumpulkan `TryCatchFinally` untuk menyesuaikan dengan kebutuhan Anda. Karena masing-masing `TryCatchFinally` membuat cabang baru di pohon eksekusi, Anda dapat membuat cakupan bersarang. Exception dalam lingkup induk akan menyebabkan upaya pembatalan dari semua tugas yang diprakarsai oleh `TryCatchFinally` bersarang ada di dalamnya. Namun, pengecualian dalam `TryCatchFinally` bersarang tidak secara otomatis menyebarkan ke induk. Jika Anda ingin menyebarkan pengecualian dari `TryCatchFinally` bersarang ke yang mengandung `TryCatchFinally`, Anda harus membuang kembali pengecualian di `doCatch()`. Dengan kata lain, hanya pengecualian tidak tertangani yang menggelegak, seperti `try/catch` Java. Jika Anda membatalkan `TryCatchFinally` bersarang dengan memanggil metode pembatalan, `TryCatchFinally` bersarang akan dibatalkan tetapi yang mengandung `TryCatchFinally` tidak akan secara otomatis dibatalkan.




```
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);
    }
};
```

Coba lagi Aktivitas Gagal

Kegiatan kadang-kadang gagal karena alasan sementara, seperti kehilangan konektivitas sementara. Di lain waktu, aktivitas mungkin berhasil, sehingga cara yang tepat untuk menangani kegagalan aktivitas sering untuk mencoba lagi aktivitas, mungkin beberapa kali.

Ada berbagai strategi untuk mencoba kembali aktivitas; yang terbaik tergantung pada detail alur kerja Anda. Strategi jatuh ke dalam tiga kategori dasar:

- retry-until-successStrategi hanya terus mencoba kembali aktivitas sampai selesai.
- Strategi coba lagi eksponensial meningkatkan interval waktu antara coba lagi secara eksponensial sampai aktivitas selesai atau proses mencapai titik pemfilteran stopword tertentu, seperti jumlah maksimum upaya.
- Strategi coba lagi kustom memutuskan apakah atau cara mencoba lagi aktivitas setelah setiap upaya gagal.

Bagian berikut menjelaskan cara menerapkan strategi ini. Contoh pekerja alur kerja semua menggunakan aktivitas tunggal, `unreliableActivity`, yang secara acak melakukan salah satu dari berikut ini:

- Selesaikan segera
- Gagal sengaja dengan melebihi nilai waktu habis
- Gagal sengaja dengan melempar `IllegalStateException`

Retry-Until-Success Strategi

Strategi coba lagi yang paling sederhana adalah terus mencoba kembali aktivitas setiap kali gagal sampai akhirnya berhasil. Pola dasar adalah:

1. Menerapkan sebuah `TryCatch` bersarang atau kelas `TryCatchFinally` dalam metode titik masuk alur kerja Anda.
2. Jalankan aktivitas di `doTry`
3. Jika aktivitas gagal, kerangka kerja memanggil `doCatch`, yang menjalankan metode titik masuk lagi.
4. Ulangi Langkah 2 - 3 hingga aktivitas selesai dengan sukses.

Alur kerja berikut mengimplementasikan strategi. retry-until-success Antarmuka alur kerja diimplementasikan dalam `RetryActivityRecipeWorkflow` dan memiliki satu metode, `runUnreliableActivityTillSuccess`, yang merupakan titik masuk alur kerja. Pekerja alur kerja diimplementasikan dalam `RetryActivityRecipeWorkflowImpl`, sebagai berikut:

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

Alur kerja bekerja sebagai berikut:

1. `runUnreliableActivityTillSuccess` membuat objek `Settable<Boolean>` bernama `retryActivity` yang digunakan untuk menunjukkan apakah aktivitas gagal dan harus dicoba lagi. `Settable<T>` berasal dari `Promise<T>` dan bekerja banyak dengan cara yang sama, tetapi Anda menetapkan nilai objek `Settable<T>` secara manual.
2. `runUnreliableActivityTillSuccess` mengimplementasikan sebuah kelas `TryCatch` bersarang anonim kelas untuk menangani setiap pengecualian yang dilemparkan oleh aktivitas `unreliableActivity`. Untuk diskusi lebih lanjut tentang cara menangani pengecualian dilemparkan oleh kode asinkron, lihat Penanganan Kesalahan.
3. `doTry` mengeksekusi aktivitas `unreliableActivity`, yang mengembalikan objek `Promise<Void>` bernama `activityRanSuccessfully`.
4. `doTry` memanggil metode `setRetryActivityToFalse` asinkron, yang memiliki dua parameter:
   - `activityRanSuccessfully` mengambil objek `Promise<Void>` yang dikembalikan oleh aktivitas `unreliableActivity`.
   - `retryActivity` mengambil objek `retryActivity`.

   Jika `unreliableActivity` selesai, `activityRanSuccessfully` menjadi siap dan `setRetryActivityToFalse` mengatur `retryActivity` ke SALAH. Jika tidak, `activityRanSuccessfully` tidak pernah menjadi siap dan `setRetryActivityToFalse` tidak mengeksekusi.

5. Jika `unreliableActivity` melempar pengecualian, kerangka kerja memanggil `doCatch` dan meneruskan objek pengecualian. `doCatch` set `retryActivity` ke betul.
6. `runUnreliableActivityTillSuccess` memanggil metode `restartRunUnreliableActivityTillSuccess` asinkron dan meneruskannya objek `retryActivity`. Karena `retryActivity` adalah jenis `Promise<T>`, `restartRunUnreliableActivityTillSuccess` menunda eksekusi sampai `retryActivity` sudah siap, yang terjadi setelahnya `TryCatch` selesai.
7. Saat `retryActivity` sudah siap, `restartRunUnreliableActivityTillSuccess` mengekstrak nilai.
   - Jika nilai adalah `false`, coba lagi berhasil. `restartRunUnreliableActivityTillSuccess` tidak dan urutan coba lagi berakhir.
   - Jika nilai benar, coba lagi gagal. `restartRunUnreliableActivityTillSuccess` memanggil `runUnreliableActivityTillSuccess` untuk menjalankan aktivitas lagi.
8. Langkah 1 - 7 ulangi sehingga `unreliableActivity` selesai.

> **Note**
>
> `doCatch` tidak menangani pengecualian; itu hanya menetapkan objek `retryActivity` ke BETUL untuk menunjukkan bahwa aktivitas gagal. Coba lagi ditangani oleh metode `restartRunUnreliableActivityTillSuccess` asinkron, yang menunda eksekusi sampai `TryCatch` selesai. Alasan untuk pendekatan ini adalah bahwa, jika Anda mencoba kembali aktivitas di `doCatch`, Anda tidak dapat membatalkannya. Mencoba kembali aktivitas di `restartRunUnreliableActivityTillSuccess` mengizinkan Anda untuk melaksanakan kegiatan yang dapat dibatalkan.

Strategi Coba Kembali Eksponensial

Dengan strategi coba lagi eksponensial, kerangka kerja mengeksekusi aktivitas gagal lagi setelah jangka waktu tertentu, N detik. Jika upaya gagal kerangka kerja mengeksekusi aktivitas lagi setelah 2N detik, dan kemudian 4N detik dan seterusnya. Karena waktu tunggu bisa mendapatkan cukup besar, Anda biasanya menghentikan upaya coba lagi di beberapa titik daripada berlanjut tanpa batas..

Kerangka kerja ini menyediakan tiga cara untuk menerapkan strategi coba lagi eksponensial:

- Anotasi `@ExponentialRetry` adalah pendekatan yang paling sederhana, tetapi Anda harus mengatur opsi konfigurasi coba lagi pada waktu kompilasi.
- Kelas `RetryDecorator` mengizinkan Anda untuk mengatur konfigurasi coba lagi pada waktu aktif dan mengubahnya sesuai kebutuhan.
- Kelas `AsyncRetryingExecutor` mengizinkan Anda untuk mengatur konfigurasi coba lagi pada waktu aktif dan mengubahnya sesuai kebutuhan. Selain itu, kerangka kerja memanggil metode `AsyncRunnable.run` yang diimplementasikan pengguna untuk menjalankan setiap upaya coba lagi.

Semua pendekatan mendukung opsi konfigurasi berikut, di mana nilai waktu dalam detik:

- Waktu tunggu percobaan ulang awal.
- Koefisien back-off, yang digunakan untuk komputasi interval coba lagi, sebagai berikut:

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

  Nilai default adalah 2.0.
- Jumlah maksimum upaya coba lagi. Nilai default tidak terbatas.
- Interval coba lagi maksimum. Nilai default tidak terbatas.
- Waktu kedaluwarsa. Coba lagi upaya berhenti ketika total durasi proses melebihi nilai ini. Nilai default tidak terbatas.
- Pengecualian yang akan memicu proses coba lagi. Secara default, setiap pengecualian memicu proses coba lagi.
- Pengecualian yang tidak akan memicu upaya coba lagi. Secara default, tidak ada pengecualian yang dikecualikan.

Bagian berikut menjelaskan berbagai cara yang Anda dapat menerapkan strategi coba lagi eksponensial.

Coba lagi eksponensial dengan @ ExponentialRetry

Cara termudah untuk menerapkan strategi coba lagi eksponensial untuk suatu kegiatan adalah dengan menerapkan anotasi `@ExponentialRetry` untuk aktivitas dalam definisi antarmuka. Jika

aktivitas gagal, kerangka kerja menangani proses coba lagi secara otomatis, berdasarkan nilai opsi yang ditentukan. Pola dasar adalah:

1. Terapkan `@ExponentialRetry` untuk aktivitas yang sesuai dan menentukan konfigurasi coba lagi.
2. Jika aktivitas dijelaskan gagal, kerangka kerja secara otomatis mencoba ulang aktivitas sesuai dengan konfigurasi yang ditentukan oleh argumen penjelasan ini.

Pekerja alur kerja `ExponentialRetryAnnotationWorkflow` mengimplementasikan strategi coba lagi eksponensial dengan menggunakan anotasi `@ExponentialRetry`. anotasi tersebut menggunakan aktivitas `unreliableActivity` yang definisi antarmukanya diimplementasikan dalam `ExponentialRetryAnnotationActivities` sebagai berikut:

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

Opsi `@ExponentialRetry` menentukan strategi berikut:

- Coba lagi hanya jika aktivitas melempar `IllegalStateException`.
- Gunakan waktu tunggu awal 5 detik.
- Tidak lebih dari 5 upaya coba lagi.

Antarmuka alur kerja diimplementasikan dalam `RetryWorkflow` dan memiliki satu metode, `process`, yang merupakan titik masuk alur kerja. Pekerja alur kerja diimplementasikan dalam `ExponentialRetryAnnotationWorkflowImpl`, sebagai berikut:

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

```
    }

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

Alur kerja bekerja sebagai berikut:

1. `process` menjalankan metode `handleUnreliableActivity` sinkron.
2. `handleUnreliableActivity` mengeksekusi aktivitas `unreliableActivity`.

Jika aktivitas gagal dengan melempar `IllegalStateException`, kerangka kerja secara otomatis menjalankan strategi coba lagi yang ditentukan dalam `ExponentialRetryAnnotationActivities`.

Coba Ulang Eksponensial dengan Kelas RetryDecorator

`@ExponentialRetry` mudah digunakan. Namun, konfigurasi statis dan ditetapkan pada waktu kompilasi, sehingga kerangka kerja menggunakan strategi coba lagi yang sama setiap kali aktivitas gagal. Anda dapat menerapkan strategi coba lagi eksponensial yang lebih fleksibel dengan menggunakan kelas `RetryDecorator`, yang mengizinkan Anda untuk menentukan konfigurasi pada waktu aktif dan mengubahnya sesuai kebutuhan. Pola dasar adalah:

1. Buat dan konfigurasi objek `ExponentialRetryPolicy` yang menentukan konfigurasi coba lagi.
2. Buat objek `RetryDecorator` dan melalui objek `ExponentialRetryPolicy` dari Langkah 1 untuk konstruktor.
3. Terapkan objek dekorator untuk aktivitas dengan melewati nama kelas klien aktivitas ke objek `RetryDecorator` menghiasi metode.
4. Mengeksekusi aktivitas.

Jika aktivitas gagal, kerangka kerja mencoba ulang aktivitas sesuai dengan konfigurasi objek `ExponentialRetryPolicy`. Anda dapat mengubah konfigurasi coba lagi yang diperlukan dengan memodifikasi objek ini.

> **Note**
>
> Anotasi `@ExponentialRetry` dan kelas `RetryDecorator` sama-sama eksklusif. Anda tidak dapat menggunakan `RetryDecorator` untuk secara dinamis mengesampingkan kebijakan coba lagi yang ditentukan oleh anotasi `@ExponentialRetry`.

Implementasi alur kerja berikut menunjukkan cara menggunakan kelas `RetryDecorator` untuk menerapkan strategi coba lagi eksponensial. Implementasi menggunakan aktivitas `unreliableActivity` yang tidak memiliki anotasi `@ExponentialRetry`. Antarmuka alur kerja diimplementasikan dalam `RetryWorkflow` dan memiliki satu metode, `process`, yang merupakan titik masuk alur kerja. Pekerja alur kerja diimplementasikan dalam `DecoratorRetryWorkflowImpl`, sebagai berikut:

```
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();
  }
}
```

Alur kerja bekerja sebagai berikut:

1. `process` membuat dan mengkonfigurasi objek `ExponentialRetryPolicy` oleh:
   - Melewati interval coba lagi awal untuk konstruktor.
   - Memanggil metode objek `withMaximumAttempts` untuk mengatur jumlah maksimum upaya untuk 5. `ExponentialRetryPolicy` mengekspos lainnya objek `with` yang dapat Anda gunakan untuk menentukan opsi konfigurasi lainnya.

2. `process` membuat objek `RetryDecorator` bernama `retryDecorator` dan meneruskan objek `ExponentialRetryPolicy` dari Langkah 1 ke konstruktor.
3. `process` menerapkan dekorator untuk aktivitas dengan memanggil metode `retryDecorator.decorate` dan meneruskan itu nama kelas aktivitas klien.
4. `handleUnreliableActivity` mengeksekusi aktivitas.

Jika aktivitas gagal, kerangka kerja coba lagi sesuai dengan konfigurasi yang ditentukan pada Langkah 1.

> **Note**
>
> Beberapa `ExponentialRetryPolicy` metode kelas `with` memiliki metode `set` yang sesuai yang dapat Anda hubungi untuk mengubah opsi konfigurasi yang sesuai setiap saat: `setBackoffCoefficient`, `setMaximumAttempts`, `setMaximumRetryIntervalSeconds`, dan `setMaximumRetryExpirationIntervalSeconds`.

Coba Ulang Eksponensial dengan Kelas AsyncRetryingExecutor

Kelas `RetryDecorator` menyediakan lebih banyak fleksibilitas dalam mengkonfigurasi proses coba lagi daripada `@ExponentialRetry`, namun kerangka kerja masih menjalankan upaya coba lagi secara otomatis, berdasarkan objek konfigurasi `ExponentialRetryPolicy` saat ini. Pendekatan yang lebih fleksibel adalah dengan menggunakan kelas `AsyncRetryingExecutor`. Selain mengizinkan Anda untuk mengkonfigurasi proses coba lagi pada waktu aktif, kerangka kerja memanggil metode `AsyncRunnable.run` yang diimplementasikan pengguna untuk menjalankan setiap upaya coba lagi bukan hanya mengeksekusi aktivitas.

Pola dasar adalah:

1. Buat dan konfigurasikan objek `ExponentialRetryPolicy` untuk menentukan konfigurasi coba lagi.
2. Buat objek `AsyncRetryingExecutor`, dan meneruskannya objek `ExponentialRetryPolicy` dan sebuah instans dari jam alur kerja.
3. Menerapkan sebuah `TryCatch` bersarang anonim atau kelas `TryCatchFinally`.
4. Menerapkan kelas `AsyncRunnable` anonim dan menimpa metode `run` untuk menerapkan kode kustom untuk menjalankan aktivitas.

5. Ganti doTry untuk memanggil AsyncRetryingExecutor metode execute objek dan meneruskannya kelas AsyncRunnable dari Langkah 4. Objek AsyncRetryingExecutor memanggil AsyncRunnable.run untuk menjalankan aktivitas.
6. Jika aktivitas gagal, objek AsyncRetryingExecutor memanggil metode AsyncRunnable.run lagi, sesuai dengan kebijakan coba lagi yang ditentukan pada Langkah 1.

Alur kerja berikut menunjukkan cara menggunakan kelas AsyncRetryingExecutor untuk menerapkan strategi coba lagi eksponensial. Alur kerja menggunakan aktivitas unreliableActivity yang sama sebagai alur kerja DecoratorRetryWorkflowyang dibahas sebelumnya. Antarmuka alur kerja diimplementasikan dalam RetryWorkflow dan memiliki satu metode, process, yang merupakan titik masuk alur kerja. Pekerja alur kerja diimplementasikan dalam AsyncExecutorRetryWorkflowImpl, sebagai berikut:

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

Alur kerja bekerja sebagai berikut:

1. `process` memanggil metode `handleUnreliableActivity` dan meneruskannya pengaturan konfigurasi.
2. `handleUnreliableActivity` menggunakan pengaturan konfigurasi dari Langkah 1 untuk membuat objek `ExponentialRetryPolicy`, `retryPolicy`.
3. `handleUnreliableActivity` membuat objek `AsyncRetryExecutor`, `executor`, dan meneruskan objek `ExponentialRetryPolicy` dari Langkah 2 dan sebuah instans dari jam alur kerja ke konstruktor
4. `handleUnreliableActivity` mengimplementasikan sebuah kelas `TryCatch` bersarang anonim dan menimpa `doTry` dan metode `doCatch` untuk menjalankan upaya coba lagi dan menangani pengecualian apapun.
5. `doTry` membuat kelas `AsyncRunnable` anonim dan menimpa metode `run` untuk menerapkan kode kustom untuk mengeksekusi `unreliableActivity`. Untuk kesederhanaan, `run` hanya mengeksekusi aktivitas, tetapi Anda dapat menerapkan pendekatan yang lebih canggih yang sesuai.
6. `doTry` memanggil `executor.execute` dan meneruskannya objek `AsyncRunnable`. `execute` memanggil metode `AsyncRunnable` objek `run` untuk menjalankan aktivitas.
7. Jika aktivitas gagal, pelaksana memanggil `run` lagi, menurut konfigurasi objek `retryPolicy`.

Untuk diskusi lebih lanjut tentang cara menggunakan kelas `TryCatch` untuk menangani kesalahan, lihat AWS Flow Framework untuk Pengecualian Java.

Strategi Coba Lagi Kustom

Pendekatan yang paling fleksibel untuk mencoba kembali aktivitas yang gagal adalah strategi khusus, yang secara rekursif memanggil metode asinkron yang menjalankan upaya coba lagi, seperti strategi. retry-until-success Namun, bukan hanya menjalankan aktivitas lagi, Anda menerapkan logika

kustom yang memutuskan apakah dan cara menjalankan setiap upaya coba lagi berturut-turut. Pola dasar adalah:

1. Buat objek status `Settable<T>`, yang digunakan untuk menunjukkan apakah aktivitas gagal.
2. Menerapkan sebuah `TryCatch` bersarang atau kelas `TryCatchFinally`.
3. `doTry` mengeksekusi aktivitas.
4. Jika aktivitasnya gagal, `doCatch` menetapkan objek status untuk menunjukkan bahwa aktivitas gagal.
5. Memanggil metode penanganan kegagalan asinkron dan menyebarkannya objek status. Metode menunda eksekusi sampai `TryCatch` atau `TryCatchFinally` selesai.
6. Metode penanganan kegagalan memutuskan apakah akan mencoba kembali aktivitas, dan jika demikian, kapan.

Alur kerja berikut menunjukkan cara menerapkan strategi coba lagi kustom. Alur kerja menggunakan aktivitas `unreliableActivity` yang sama sebagai `DecoratorRetryWorkflow` dan alur kerja `AsyncExecutorRetryWorkflow`. Antarmuka alur kerja diimplementasikan dalam `RetryWorkflow` dan memiliki satu metode, `process`, yang merupakan titik masuk alur kerja. Pekerja alur kerja diimplementasikan dalam `CustomLogicRetryWorkflowImpl`, sebagai berikut:

```
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;
    }
}
```

Alur kerja bekerja sebagai berikut:

1. `process` memanggil metode `callActivityWithRetry` asinkron.
2. `callActivityWithRetry` membuat objek `Settable<Throwable>` bernama gagal yang digunakan untuk menunjukkan apakah aktivitas telah gagal. `Settable<T>` berasal dari `Promise<T>` dan banyak bekerja dengan cara yang sama, tetapi Anda menetapkan nilai objek `Settable<T>` secara manual.
3. `callActivityWithRetry` mengimplementasikan sebuah kelas `TryCatchFinally` bersarang anonim untuk menangani pengecualian yang dilemparkan oleh `unreliableActivity`. Untuk diskusi lebih lanjut tentang cara menangani pengecualian dilemparkan oleh kode asinkron, lihat AWS Flow Framework untuk Pengecualian Java.
4. `doTry` mengeksekusi `unreliableActivity`.
5. Jika `unreliableActivity` melempar pengecualian, kerangka kerja memanggil `doCatch` dan meneruskannya objek pengecualian. `doCatch` mengatur `failure` ke objek pengecualian, yang menunjukkan bahwa aktivitas gagal dan menempatkan objek dalam status siap.
6. `doFinally` memeriksa apakah `failure` sudah siap, yang akan benar hanya jika `failure` telah ditetapkan oleh `doCatch`.
   - Jika `failure` sudah siap, `doFinally` tidak melakukan apa-apa.
   - Jika `failure` belum siap, aktivitas selesai dan `doFinally` menetapkan gagal ke `null`.
7. `callActivityWithRetry` memanggil metode `retryOnFailure` asinkron dan meneruskannya dengan gagal. Karena gagal adalah jenis `Settable<T>`, `callActivityWithRetry` menunda eksekusi sampai gagal siap, yang terjadi setelah `TryCatchFinally` selesai.

8. `retryOnFailure` mendapat nilai dari gagal.
   - Jika gagal diatur ke nol, upaya coba lagi berhasil. `retryOnFailure` tidak melakukan apa-apa, yang mengakhiri proses coba lagi.
   - Jika gagal diatur ke objek pengecualian dan `shouldRetry` mengembalikan BETUL, `retryOnFailure` memanggil `callActivityWithRetry` untuk mencoba kembali aktivitas.

     `shouldRetry` mengimplementasikan logika kustom untuk memutuskan apakah akan mencoba lagi aktivitas gagal. Untuk kesederhanaan, `shouldRetry` selalu mengembalikan `true` dan `retryOnFailure` mengeksekusi aktivitas segera, tetapi Anda dapat menerapkan logika yang lebih canggih sesuai kebutuhan.
9. Langkah 2-8 ulangi hingga `unreliableActivity` melengkapi atau `shouldRetry` memutuskan untuk menghentikan proses.

> **Note**
>
> `doCatch` tidak menangani proses coba lagi; itu hanya menetapkan gagal untuk menunjukkan bahwa aktivitas gagal. Proses coba lagi ditangani oleh metode `retryOnFailure` asinkron, yang menunda eksekusi sampai `TryCatch` selesai. Alasan untuk pendekatan ini adalah bahwa, jika Anda mencoba kembali aktivitas di `doCatch`, Anda tidak dapat membatalkannya. Mencoba kembali aktivitas di `retryOnFailure` mengizinkan Anda untuk melaksanakan kegiatan yang dapat dibatalkan.

Tugas Daemon

The AWS Flow Framework for Java memungkinkan penandaan tugas-tugas tertentu sebagai`daemon`. Ini memungkinkan Anda membuat tugas yang melakukan beberapa pekerjaan latar belakang yang harus dibatalkan saat semua pekerjaan lain selesai. Sebagai contoh, tugas pemantauan kesehatan harus dibatalkan ketika sisa alur kerja selesai. Anda dapat melakukan ini dengan menetapkan bendera `daemon` pada metode asinkronus atau instans dari `TryCatchFinally`. Pada contoh berikut, metode asinkronus `monitorHealth()` ditandai sebagai `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();
  }
}
```

Dalam contoh di atas, saat doUsefulWorkActivity diselesaikan, monitoringHealth akan dibatalkan secara otomatis. Hal ini pada gilirannya akan membatalkan seluruh cabang eksekusi berakar pada metode asinkronus ini. Semantik pembatalan sama seperti di TryCatchFinally. Demikian pula, Anda dapat menandai daemon TryCatchFinally dengan menenruskan bendera Boolean untuk konstruktor.

```
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
            }
        };
    }
}
```

Tugas daemon yang dimulai dalam TryCatchFinally dicakupkan ke konteks pembuatannya—yaitu, akan dicakup ke metode doTry(), doCatch(), atau doFinally(). Misalnya, dalam contoh berikut, metode asinkron startMonitoring ditandai daemon dan dipanggil dari doTry(). Tugas yang

dibuat untuk itu akan dibatalkan segera setelah tugas-tugas lain (`doUsefulWorkActivity` dalam kasus ini) yang dimulai dalam `doTry()` diselesaikan.

```
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 untuk Java Replay Behavior

Topik ini membahas contoh perilaku memutar ulang, menggunakan contoh di bagian Apa yang dimaksud dengan AWS Flow Framework Java?. Baik skenario synchronous (sinkron) dan asynchronous (asinkron) dibahas.

Contoh 1: Putar Ulang Sinkron

Untuk contoh cara kerja replay dalam alur kerja sinkron, ubah alur kerja dan implementasi aktivitas dengan menambahkan `println` panggilan dalam implementasi masing-masing, sebagai berikut: HelloWorldWorkflow

```
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);
   }
}
```

Untuk detail tentang kode, lihat HelloWorldWorkflow Aplikasi. Berikut ini adalah versi output yang telah diedit, dengan komentar yang menunjukkan awal setiap episode replay.

```
//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
```

Proses replay untuk contoh ini bekerja sebagai berikut:

- Episode pertama menjadwalkan tugas aktivitas `getName`, yang tidak memiliki dependensi.
- Episode kedua menjadwalkan tugas aktivitas `getGreeting`, yang tergantung pada `getName`.
- Episode ketiga menjadwalkan tugas aktivitas `say`, yang tergantung pada`getGreeting`.
- Episode terakhir tidak menjadwalkan tugas tambahan dan tidak menemukan aktivitas yang belum selesai, mengakhiri eksekusi alur kerja.

> **Note**
>
> Tiga metode klien aktivitas dipanggil sekali untuk setiap episode. Namun, hanya satu dari panggilan itu yang menghasilkan tugas aktivitas, sehingga setiap tugas yang dilakukan hanya sekali.

Contoh 2: Pemutaran Ulang Asinkron

Demikian pula dengan synchronous replay example (contoh pemutaran ulang sinkron), Anda dapat memodifikasi HelloWorldWorkflowAsyncAplikasi untuk melihat bagaimana pemutaran ulang asinkron bekerja. Pemutaran ulang sinkron menghasilkan output berikut:

```
//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 menggunakan tiga episode replay karena hanya ada dua kegiatan. Aktivitas `getGreeting` digantikan oleh metode alur kerja asinkron getGreeting, yang tidak memulai episode pemutaran ulang ketika selesai.

Episode pertama tidak memanggil `getGreeting`, karena itu tergantung pada selesainya aktivitas nama. Namun, setelah getName selesai, memutar ulang panggilan getGreeting sekali untuk setiap episode yang berhasil.

Lihat juga

- AWS Flow Framework Konsep Dasar: Eksekusi Terdistribusi

Praktik terbaik

Gunakan praktik terbaik ini untuk memaksimalkan AWS Flow Framework Java.

Topik

- Membuat perubahan pada Decider Code: Versioning dan Feature Flag

Membuat perubahan pada Decider Code: Versioning dan Feature Flag

Bagian ini menunjukkan cara menghindari perubahan yang tidak kompatibel ke belakang pada decider menggunakan dua metode:

- Versioning menyediakan solusi dasar.
- Versioning with Feature Flag dibangun di atas solusi Versioning: Tidak ada versi baru alur kerja yang diperkenalkan, dan tidak perlu mendorong kode baru untuk memperbarui versi.

Sebelum Anda mencoba solusi ini, biasakan diri Anda dengan bagian Contoh skenario yang menjelaskan penyebab dan efek dari perubahan decider yang tidak kompatibel ke belakang.

Proses Replay dan Perubahan Kode

Ketika seorang pekerja penentu AWS Flow Framework untuk Java mengeksekusi tugas keputusan, pertama-tama harus membangun kembali status eksekusi saat ini sebelum dapat menambahkan langkah-langkah ke dalamnya. Decider melakukan hal ini menggunakan proses yang disebut replay.

Proses replay mengeksekusi ulang kode decider dari awal, sekaligus menelusuri history kejadian yang sudah terjadi. Menelusuri riwayat peristiwa memungkinkan kerangka kerja bereaksi terhadap sinyal atau penyelesaian tugas dan membuka blokir objek `Promise` dalam kode.

Saat kerangka kerja mengeksekusi kode decider, ia menetapkan ID untuk setiap tugas terjadwal (aktivitas, fungsi Lambda, timer, alur kerja anak, atau sinyal keluar) dengan menambahkan penghitung. Kerangka ini mengkomunikasikan ID ini ke Amazon SWF, dan menambahkan ID untuk peristiwa sejarah, seperti `ActivityTaskCompleted`.

Agar proses replay berhasil, penting bagi kode decider untuk menjadi deterministik, dan untuk menjadwalkan tugas yang sama dalam urutan yang sama untuk setiap keputusan dalam setiap

eksekusi alur kerja. Jika Anda tidak mematuhi persyaratan ini, kerangka kerja mungkin, misalnya, gagal mencocokkan ID dalam peristiwa `ActivityTaskCompleted` dengan objek `Promise` yang ada.

Contoh skenario

Ada kelas perubahan kode yang dianggap tidak kompatibel ke belakang. Perubahan ini mencakup pembaruan yang mengubah nomor, jenis, atau urutan tugas terjadwal. Pertimbangkan contoh berikut:

Anda menulis kode decider untuk menjadwalkan dua tugas timer. Anda memulai eksekusi dan menjalankan keputusan. Akibatnya, dua tugas timer dijadwalkan, dengan IDs 1 dan2.

Jika Anda memperbarui kode penentu untuk menjadwalkan hanya satu timer sebelum keputusan berikutnya dieksekusi, selama tugas keputusan berikutnya, kerangka kerja akan gagal memutar ulang peristiwa `TimerFired` kedua, karena ID 2 tidak cocok dengan tugas timer apa pun yang dihasilkan kode.

Skenario

Garis besar berikut menunjukkan langkah-langkah skenario ini. Tujuan akhir dari skenario ini adalah untuk bermigrasi ke sistem yang hanya menjadwalkan satu timer tetapi tidak menyebabkan kegagalan dalam eksekusi yang dimulai sebelum migrasi.

1. Versi Decider Awal
    a. Tulis decider.
    b. Mulai decider.
    c. Decider menjadwalkan dua timer.
    d. Decider mulai lima eksekusi.
    e. Hentikan decider.
2. Perubahan Decider Tidak Kompatibel untuk Mundur
    a. Modifikasi decider.
    b. Mulai decider.
    c. Decider menjadwalkan satu timer.
    d. Decider mulai lima eksekusi.

Bagian berikut termasuk contoh kode Java yang menunjukkan bagaimana menerapkan skenario ini. Contoh kode di Solusi menunjukkan berbagai cara untuk memperbaiki perubahan yang tidak kompatibel ke belakang.

> **Note**
>
> Anda dapat menggunakan versi terbaru AWS SDK untuk Java untuk menjalankan kode ini.

Kode Umum

Kode Java berikut tidak berubah di antara contoh-contoh dalam skenario ini.

`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;
    }

}
```

Menulis Kode Decider Awal

Berikut ini adalah kode Java awal dari decider. Ini terdaftar sebagai versi 1 dan menjadwalkan dua tugas timer lima detik.

`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);
        }

    }
}
```

Mensimulasikan Perubahan yang Tidak Kompatibel untuk Mundur

Kode Java decider berikut yang dimodifikasi adalah contoh yang baik dari perubahan yang tidak kompatibel ke belakang. Kode ini masih terdaftar sebagai versi 1, tetapi hanya menjadwalkan satu timer.

`ModifiedDecider.java`

```
package sample.v1.modified;

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

import sample.Input;

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

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

    public static class Impl implements Foo {

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

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

    }
}
```

Berikut kode Java memungkinkan Anda untuk mensimulasikan masalah membuat perubahan kebelakang yang kompatibel dengan menjalankan decider dimodifikasi.

`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();
    }

}
```

Saat Anda menjalankan program, tiga eksekusi yang gagal adalah yang dimulai di bawah versi awal dari penentu dan dilanjutkan setelah migrasi.

Solusi

Anda dapat menggunakan solusi berikut untuk menghindari perubahan tidak sesuai ke belakang. Untuk informasi selengkapnya, lihat Membuat Perubahan pada Kode Decider dan Contoh skenario.

Menggunakan versioning

Dalam solusi ini, Anda menyalin decider ke kelas baru, memodifikasi decider, dan kemudian mendaftarkan decider di bawah versi alur kerja baru.

`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);
        }

    }

}
```

Dalam kode Java yang diperbarui, pekerja decider kedua menjalankan kedua versi alur kerja, memungkinkan eksekusi dalam penerbangan untuk terus mengeksekusi secara independen dari perubahan dalam versi 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();
    }

}
```

Ketika Anda menjalankan program, semua eksekusi berhasil diselesaikan.

Menggunakan Feature Flag

Solusi lain untuk masalah kompatibilitas mundur adalah kode cabang untuk mendukung dua implementasi di kelas yang sama adalah bercabang berdasarkan data input alih-alih versi alur kerja.

Ketika Anda mengambil pendekatan ini, Anda menambahkan bidang untuk (atau memodifikasi bidang yang ada) objek masukan Anda setiap kali Anda memperkenalkan perubahan yang sensitif. Untuk eksekusi yang dimulai sebelum migrasi, objek input tidak akan memiliki bidang (atau akan memiliki nilai yang berbeda). Dengan demikian, Anda tidak perlu menambah nomor versi.

> **Note**
>
> Jika Anda menambahkan bidang baru, pastikan bahwa proses deserialization JSON kompatibel ke belakang. Objek yang diserialisasi sebelum pengenalan bidang harus tetap berhasil dideserialisasi setelah migrasi. Karena JSON menetapkan nilai `null` setiap kali bidang tidak ada, selalu gunakan tipe kotak (`Boolean` alih-alih `boolean`) dan tangani kasus di mana nilainya adalah `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);
            }
        }

    }
}
```

Dalam kode Java yang diperbarui, kode untuk kedua versi alur kerja masih terdaftar untuk versi 1. Namun, setelah migrasi, eksekusi baru dimulai dengan bidang `skipSecondTimer` input data diatur ke `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();
    }

}
```

Ketika Anda menjalankan program, semua eksekusi berhasil diselesaikan.

Tips pemecahan masalah dan debugging untuk Java AWS Flow Framework

Topik

Bagian ini menjelaskan beberapa jebakan umum yang mungkin Anda hadapi saat mengembangkan alur kerja menggunakan AWS Flow Framework untuk Java. Ini juga memberikan beberapa tips untuk membantu Anda mendiagnosis dan men-debug masalah.

Kesalahan kompilasi

Jika Anda menggunakan opsi penenunan waktu kompilasi AspectJ, Anda mungkin mengalami kesalahan waktu kompilasi di mana penyusun tidak dapat menemukan kelas klien yang dihasilkan untuk alur kerja dan aktivitas Anda. Kemungkinan penyebab kesalahan kompilasi tersebut adalah bahwa pembuat AspectJ mengabaikan klien yang dihasilkan selama kompilasi. Anda dapat memperbaiki masalah ini dengan menghapus kemampuan AspectJ dari proyek dan mengaktifkannya kembali. Perhatikan bahwa Anda perlu melakukan ini setiap kali antarmuka alur kerja atau aktivitas Anda berubah. Karena masalah ini, kami menyarankan Anda untuk menggunakan opsi penenunan waktu unggah. Lihat bagian Menyiapkan AWS Flow Framework untuk Java untuk lebih jelasnya.

Kesalahan sumber daya tidak diketahui

Amazon SWF mengembalikan kesalahan sumber daya yang tidak diketahui saat Anda mencoba melakukan operasi pada sumber daya yang tidak tersedia. Penyebab umum untuk kesalahan ini adalah:

- Anda mengonfigurasi pekerja dengan domain yang tidak ada. Untuk memperbaikinya, pertama-tama daftarkan domain menggunakan konsol Amazon SWF atau API layanan Amazon SWF.
- Anda mencoba membuat eksekusi alur kerja atau tugas aktivitas jenis yang belum terdaftar. Ini dapat terjadi jika Anda mencoba membuat eksekusi alur kerja sebelum pekerja dijalankan. Karena pekerja mendaftarkan tipe mereka saat dijalankan untuk pertama kalinya, Anda harus menjalankannya setidaknya sekali sebelum mencoba memulai eksekusi (atau mendaftarkan tipe secara manual menggunakan Console atau API layanan). Perhatikan bahwa setelah jenis didaftarkan, Anda dapat membuat eksekusi meskipun tidak ada pekerja yang berjalan.
- Seorang pekerja mencoba untuk menyelesaikan tugas yang telah habis waktunya. Misalnya, jika seorang pekerja membutuhkan waktu terlalu lama untuk memproses tugas dan melebihi batas waktu, itu akan mendapatkan UnknownResource kesalahan ketika mencoba menyelesaikan atau gagal tugas. Para AWS Flow Framework pekerja akan terus melakukan polling Amazon SWF dan memproses tugas-tugas tambahan. Namun, Anda harus mempertimbangkan untuk menyesuaikan batas waktu. Menyesuaikan batas waktu mengharuskan Anda mendaftarkan versi baru dari jenis aktivitas.

Pengecualian saat memanggil get () pada Promise

Tidak seperti Java Future, `Promise` adalah konstruksi non-pemblokiran dan memanggil `get()` pada `Promise` yang belum siap akan mengeluarkan pengecualian alih-alih memblokir. Cara yang benar untuk menggunakan `Promise` adalah dengan meneruskannya ke metode asinkron (atau tugas) dan mengakses nilainya dalam metode asinkron. AWS Flow Framework for Java memastikan bahwa metode asinkron dipanggil hanya ketika semua argumen `Promise` yang diteruskan ke metode tersebut telah siap. Jika Anda yakin kode Anda benar atau jika Anda mengalami ini saat menjalankan salah satu AWS Flow Framework sampel, maka kemungkinan besar karena aspectJ tidak dikonfigurasi dengan benar. Untuk detailnya, lihat bagian Menyiapkan AWS Flow Framework untuk Java.

Alur kerja nondeterministik

Seperti yang dijelaskan di bagian Nondeterminisme, implementasi alur kerja Anda harus bersifat deterministik. Beberapa kesalahan umum yang dapat menyebabkan nondeterminisme adalah penggunaan jam sistem, penggunaan angka acak, dan generasi. GUIDs Karena konstruksi ini dapat mengembalikan nilai yang berbeda pada waktu yang berbeda, alur kontrol alur kerja Anda mungkin mengambil jalur yang berbeda setiap kali dijalankan (lihat bagian AWS Flow Framework Konsep Dasar: Eksekusi Terdistribusi dan Memahami Tugas di AWS Flow Framework Java untuk detailnya).

Jika kerangka kerja mendeteksi nondeterminisme saat menjalankan alur kerja, pengecualian akan dilemparkan.

Masalah karena pembuatan versi

Saat Anda mengimplementasikan versi baru alur kerja atau aktivitas Anda—misalnya, saat Anda menambahkan fitur baru—Anda harus meningkatkan versi tipe dengan menggunakan anotasi yang sesuai: `@Workflow`, `@Activites`, atau `@Activity`. Saat versi baru alur kerja di-deploy, sering kali Anda akan memiliki eksekusi dari versi yang sudah ada yang sudah berjalan. Oleh karena itu, Anda perlu memastikan bahwa pekerja dengan versi alur kerja dan aktivitas yang sesuai mendapatkan tugas. Anda dapat melakukannya dengan menggunakan kumpulan daftar tugas yang berbeda untuk setiap versi. Misalnya, Anda dapat menambahkan nomor versi ke nama daftar tugas. Ini memastikan bahwa tugas-tugas milik versi berbeda dari alur kerja dan aktivitas ditugaskan ke pekerja yang sesuai.

Memecahkan masalah dan men-debug eksekusi alur kerja

Langkah pertama dalam memecahkan masalah eksekusi alur kerja adalah menggunakan konsol Amazon SWF untuk melihat riwayat alur kerja. Riwayat alur kerja adalah catatan lengkap dan otoritatif dari semua peristiwa yang mengubah status eksekusi dari eksekusi alur kerja. Riwayat ini dikelola oleh Amazon SWF dan sangat berharga untuk mendiagnosis masalah. Konsol Amazon SWF memungkinkan Anda mencari eksekusi alur kerja dan menelusuri peristiwa riwayat individu.

AWS Flow Framework menyediakan `WorkflowReplayer` kelas yang dapat Anda gunakan untuk memutar ulang eksekusi alur kerja secara lokal dan men-debugnya. Dengan menggunakan kelas ini, Anda dapat men-debug tertutup dan menjalankan eksekusi alur kerja. `WorkflowReplayer`bergantung pada riwayat yang disimpan di Amazon SWF untuk melakukan pemutaran ulang. Anda dapat mengarahkannya ke eksekusi alur kerja di akun Amazon SWF Anda atau menyediakannya dengan peristiwa riwayat (misalnya, Anda dapat mengambil riwayat dari Amazon SWF dan membuat cerita bersambung secara lokal untuk digunakan nanti). Saat Anda memutar ulang eksekusi alur kerja menggunakan `WorkflowReplayer`, itu tidak memengaruhi eksekusi alur kerja yang berjalan di akun Anda. Pemutaran ulang dilakukan sepenuhnya pada klien. Anda dapat men-debug alur kerja, membuat titik putus, dan memasukkan kode menggunakan alat debug seperti biasa. Jika Anda menggunakan Eclipse, pertimbangkan untuk menambahkan filter langkah untuk AWS Flow Framework memfilter paket.

Misalnya, cuplikan kode berikut dapat digunakan untuk memutar ulang eksekusi alur kerja:

```
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 juga memungkinkan Anda untuk mendapatkan dump thread asinkron dari eksekusi alur kerja Anda. Pembuangan utas ini memberi Anda tumpukan panggilan dari semua tugas asinkron terbuka. Informasi ini dapat berguna untuk menentukan tugas mana dalam eksekusi yang tertunda dan mungkin macet. Sebagai contoh:

```
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);
}
```

Tugas yang hilang

Terkadang Anda mungkin menutup pekerja dan memulai yang baru secara berurutan hanya untuk mengetahui bahwa tugas terkirim ke pekerja lama. Hal ini dapat terjadi karena kondisi balapan dalam sistem, yang didistribusikan di beberapa proses. Masalahnya juga dapat muncul saat Anda menjalankan pengujian unit dalam lingkaran yang ketat. Menghentikan pengujian di Eclipse terkadang juga dapat menyebabkan hal ini karena penangan shutdown mungkin tidak dipanggil.

Untuk memastikan bahwa masalahnya sebenarnya karena pekerja lama mendapatkan tugas, Anda harus melihat riwayat alur kerja untuk menentukan proses mana yang menerima tugas yang Anda harapkan akan diterima pekerja baru. Misalnya, peristiwa `DecisionTaskStarted` dalam riwayat berisi identitas pekerja alur kerja yang menerima tugas. Id yang digunakan oleh Flow Framework adalah dari bentuk: {*`processId`*} `@` {*`host name`*}. Misalnya, berikut adalah detail peristiwa `DecisionTaskStarted` di konsol Amazon SWF untuk contoh eksekusi:

Stempel Waktu Acara	Sen 20 Feb 11:52:40 GMT-800 2012
Identitas	2276 @ip -0A6C1 DF5
Id Peristiwa Terjadwal	33

Untuk menghindari situasi ini, gunakan daftar tugas yang berbeda untuk setiap pengujian. Juga, pertimbangkan untuk menambahkan penundaan antara mematikan pekerja lama dan memulai yang baru.

Kegagalan validasi karena kendala panjang parameter API

Amazon SWF memberlakukan batasan panjang pada parameter API. Anda akan menerima `HTTP 400` kesalahan jika alur kerja atau implementasi aktivitas Anda melebihi kendala. Misalnya, saat memanggil `recordActivityHeartbeat` `ActivityExecutionContext` untuk mengirim detak jantung untuk aktivitas yang sedang berjalan, string tidak boleh lebih dari 2048 karakter.

Skenario umum lainnya adalah ketika suatu aktivitas gagal karena pengecualian. Kerangka kerja melaporkan kegagalan aktivitas ke Amazon SWF dengan menelepon RespondActivityTaskFaileddengan pengecualian serial sebagai detail. Panggilan API akan melaporkan kesalahan 400 jika pengecualian serial memiliki panjang lebih dari 32.768 byte. Untuk

mengurangi situasi ini, Anda dapat memotong pesan pengecualian atau penyebab agar sesuai dengan batasan panjang.

AWS Flow Framework untuk Referensi Java

Topik

AWS Flow Framework untuk Anotasi Java

Topik

@Activities

Anotasi ini dapat digunakan pada sebuah antarmuka untuk mendeklarasikan satu set jenis aktivitas. Setiap metode dalam antarmuka dijelaskan dengan anotasi ini merupakan jenis aktivitas. Antarmuka tidak dapat memiliki kedua anotasi `@Workflow` dan `@Activities`-

Parameter berikut dapat ditentukan pada anotasi ini:

`activityNamePrefix`

Menentukan prefiks nama jenis kegiatan dinyatakan dalam antarmuka. Jika diatur ke string kosong (yang merupakan default), nama antarmuka diikuti oleh '.' digunakan sebagai prefiks.

`version`

Menentukan versi default dari jenis aktivitas dinyatakan dalam antarmuka. Nilai default-nya adalah `1.0`.

`dataConverter`

Menentukan jenis `DataConverter` untuk digunakan untuk membuat serial/deserialisasi data saat membuat tugas dari jenis aktivitas ini dan hasilnya. Atur ke `NullDataConverter` secara default, yang menunjukkan bahwa `JsonDataConverter` harus digunakan.

@Activity

Anotasi ini dapat digunakan pada metode dalam sebuah antarmuka yang dianotasi dengan `@Activities`.

Parameter berikut dapat ditentukan pada anotasi ini:

`name`

Menentukan nama jenis aktivitas. Defaultnya adalah string kosong, yang menunjukkan bahwa prefiks default dan nama metode aktivitas harus digunakan untuk menentukan nama tipe aktivitas (yang merupakan bentuk {prefiks} {nama}). Perhatikan bahwa ketika Anda menentukan nama dalam anotasi `@Activity`, kerangka kerja tidak akan secara otomatis menambahkan prefiks untuk itu. Anda bebas untuk menggunakan skema penamaan Anda sendiri.

`version`

Menentukan versi dari jenis aktivitas. Ini menimpa versi default yang ditentukan dalam anotasi `@Activities` pada antarmuka yang memuatnya. Defaultnya adalah string kosong.

@ActivityRegistrationOptions

Menentukan pilihan pendaftaran dari jenis kegiatan. Anotasi ini dapat digunakan pada sebuah antarmuka yang dianotasi dengan `@Activities` atau metode di dalamnya. Jika ditentukan di kedua tempat, maka anotasi yang digunakan pada metode ini berlaku.

Parameter berikut dapat ditentukan pada anotasi ini:

`defaultTasklist`

Menentukan daftar tugas default untuk didaftarkan dengan Amazon SWF untuk jenis kegiatan ini. Default ini dapat diganti ketika memanggil metode aktivitas pada klien yang dihasilkan menggunakan Parameter `ActivitySchedulingOptions`. Diatur ke `USE_WORKER_TASK_LIST` secara default. Ini adalah nilai khusus yang menunjukkan bahwa daftar tugas yang digunakan oleh pekerja, yang melakukan pendaftaran, harus digunakan.

`defaultTaskScheduleToStartTimeoutSeconds`

Menentukan defaultTaskSchedule ToStartTimeout terdaftar dengan Amazon SWF untuk jenis aktivitas ini. Ini adalah waktu maksimum yang diizinkan untuk menunggu tugas jenis aktivitas ini sebelum ditugaskan ke pekerja. Lihat Referensi API Amazon Simple Workflow Service untuk lebih jelasnya.

`defaultTaskHeartbeatTimeoutSeconds`

Menentukan `defaultTaskHeartbeatTimeout` yang terdaftar dengan Amazon SWF untuk jenis kegiatan ini. Pekerja aktivitas harus memberikan detak jantung dalam durasi ini; jika tidak, tugas akan habis. Setel ke -1 secara default, yang merupakan nilai khusus yang menunjukkan batas waktu ini harus dinonaktifkan. Lihat Referensi API Amazon Simple Workflow Service untuk lebih jelasnya.

`defaultTaskStartToCloseTimeoutSeconds`

Menentukan defaultTaskStart ToCloseTimeout terdaftar dengan Amazon SWF untuk jenis aktivitas ini. Batas waktu ini menentukan waktu maksimum yang dapat diambil pekerja untuk memproses tugas aktivitas jenis ini. Lihat Referensi API Amazon Simple Workflow Service untuk lebih jelasnya.

`defaultTaskScheduleToCloseTimeoutSeconds`

Menentukan `defaultScheduleToCloseTimeout` yang terdaftar dengan Amazon SWF untuk jenis kegiatan ini. Batas waktu ini menentukan total durasi tugas dapat tetap dalam keadaan terbuka. Setel ke -1 secara default, yang merupakan nilai khusus yang menunjukkan batas waktu ini harus dinonaktifkan. Lihat Referensi API Amazon Simple Workflow Service untuk lebih jelasnya.

@Asynchronous

Ketika digunakan pada metode dalam logika koordinasi alur kerja, menunjukkan bahwa metode harus dijalankan secara asinkron. Panggilan ke metode akan segera kembali, tetapi eksekusi sebenarnya akan terjadi secara asinkron ketika semua parameter Promise<> yang diteruskan ke metode menjadi siap. Metode yang dianotasi dengan @Asynchronous harus memiliki tipe pengembalian Promise<> atau batal.

`daemon`

Menunjukkan apakah tugas yang dibuat untuk metode asinkron harus berupa tugas daemon. `False` secara default.

@Execute

Saat digunakan pada metode dalam antarmuka yang dianotasi dengan anotasi `@Workflow`, mengidentifikasi titik masuk alur kerja.

> **Important**
>
> Hanya satu metode di antarmuka yang bisa dihias dengan `@Execute`.

Parameter berikut dapat ditentukan pada anotasi ini:

`name`

Menentukan nama jenis alur kerja. Jika tidak diatur, nama default ke {prefiks}{nama}, di mana {prefiks} adalah nama antarmuka alur kerja diikuti oleh '.' dan {nama} adalah nama metode yang didekorasi dengan `@Execute` dalam alur kerja.

`version`

Menentukan versi jenis alur kerja.

@ExponentialRetry

Ketika digunakan pada suatu kegiatan atau metode asinkronus, tetapkan kebijakan percobaan ulang eksponensial jika metode memunculkan pengecualian yang tidak tertangani. Upaya percobaan ulang dilakukan setelah periode back-off, yang dihitung dengan kekuatan jumlah upaya.

Parameter berikut dapat ditentukan pada anotasi ini:

`intialRetryIntervalSeconds`

Menentukan durasi menunggu sebelum upaya percobaan ulang pertama. Nilai ini seharusnya tidak lebih besar dari `maximumRetryIntervalSeconds` dan `retryExpirationSeconds`.

`maximumRetryIntervalSeconds`

Menentukan durasi maksimum antara upaya percobaan ulang. Setelah tercapai, interval percobaan ulang dibatasi ke nilai ini. Atur ke -1 secara default, yang berarti durasi tak terbatas.

`retryExpirationSeconds`

Menentukan durasi setelah percobaan ulang eksponensial akan berhenti. Set ke -1 secara default, yang berarti tidak ada kedaluwarsa.

`backoffCoefficient`

Menentukan koefisien yang digunakan untuk menghitung interval percobaan ulang. Lihat Strategi Coba Kembali Eksponensial.

`maximumAttempts`

Menentukan jumlah upaya setelah percobaan ulang eksponensial akan berhenti. Set ke -1 secara default, yang berarti tidak ada batas pada jumlah upaya coba lagi.

`exceptionsToRetry`

Menentukan daftar jenis pengecualian yang harus memicu percobaan ulang. Pengecualian yang tidak tertangani dari jenis ini tidak akan menyebar lebih jauh dan metode akan dicoba lagi setelah interval percobaan ulang yang dihitung. Secara default, daftar berisi `Throwable`.

`excludeExceptions`

Menentukan daftar jenis pengecualian yang seharusnya tidak memicu coba lagi. Pengecualian tidak tertangani jenis ini akan diizinkan untuk menyebarkan. Daftar ini kosong secara default.

@GetState

Ketika digunakan pada sebuah metode dalam sebuah antarmuka dianotasikan dengan anotasi `@Workflow`, mengidentifikasi bahwa metode ini digunakan untuk mengambil status eksekusi alur kerja terbaru. Paling banyak ada satu metode dengan anotasi ini dalam sebuah antarmuka dengan

anotasi `@Workflow`. Metode dengan anotasi ini tidak boleh mengambil parameter apapun dan harus memiliki tipe pengembalian selain `void`.

@ManualActivityCompletion

Anotasi ini dapat digunakan pada metode aktivitas untuk menunjukkan bahwa tugas aktivitas tidak harus diselesaikan ketika metode kembali. Tugas aktivitas tidak akan secara otomatis selesai dan akan perlu diselesaikan secara manual langsung menggunakan API Amazon SWF. Hal ini berguna untuk kasus penggunaan di mana tugas aktivitas didelegasikan ke beberapa sistem eksternal yang tidak otomatis atau memerlukan intervensi manusia untuk diselesaikan.

@Signal

Ketika digunakan pada sebuah metode dalam sebuah antarmuka dianotasikan dengan anotasi `@Workflow`, mengidentifikasi sinyal yang dapat diterima oleh eksekusi dari jenis alur kerja dinyatakan oleh antarmuka. Penggunaan anotasi ini diperlukan untuk menentukan metode sinyal.

Parameter berikut dapat ditentukan pada anotasi ini:

`name`

Menentukan bagian nama dari nama sinyal. Jika tidak diatur, nama metode akan digunakan.

@SkipRegistration

Ketika digunakan pada sebuah antarmuka dianotasikan dengan anotasi `@Workflow`, menunjukkan bahwa jenis alur kerja tidak harus terdaftar dengan Amazon SWF. Salah satu anotasi `@WorkflowRegistrationOptions` dan `@SkipRegistrationOptions` harus digunakan pada sebuah antarmuka yang dianotasi dengan `@Workflow`, tapi tidak keduanya.

@Wait dan @ NoWait

Anotasi ini dapat digunakan pada parameter tipe `Promise<>` untuk menunjukkan apakah AWS Flow Framework untuk Java harus menunggu untuk menjadi siap sebelum menjalankan metode. Secara default, parameter `Promise<>` diteruskan ke metode `@Asynchronous` harus menjadi siap sebelum eksekusi metode terjadi. Dalam skenario tertentu, hal ini diperlukan untuk menimpa perilaku default ini. Parameter `Promise<>` dilewatkan ke `@Asynchronous` dan dianotasi dengan `@NoWait` tidak ditunggu.

Koleksi parameter (atau subclass dari) yang berisi janji, seperti `List<Promise<Int>>`, harus dianotasi dengan anotasi `@Wait`. Secara default, kerangka kerja tidak menunggu anggota koleksi.

@Workflow

Anotasi ini digunakan pada sebuah antarmuka untuk mendeklarasikan sebuah jenis alur Kerja. Sebuah antarmuka dihiasi dengan anotasi ini harus berisi tepat satu metode yang dihiasi dengan @Execute untuk mendeklarasikan titik masuk untuk alur kerja Anda.

> **Note**
>
> Antarmuka tidak dapat memiliki kedua anotasi `@Workflow` dan `@Activities` dideklarasikan sekaligus; mereka saling eksklusif.

Parameter berikut dapat ditentukan pada anotasi ini:

`dataConverter`

Menentukan `DataConverter` untuk digunakan saat mengirim permintaan, dan menerima hasil, eksekusi alur kerja jenis alur kerja ini.

Defaultnya adalah `NullDataConverter` yang, pada gilirannya, kembali `JsonDataConverter` untuk memproses semua data permintaan dan respons sebagai JavaScript Object Notation (JSON).

Contoh

```
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

Saat digunakan pada antarmuka yang dianotasi dengan `@Workflow`, menyediakan pengaturan default yang digunakan oleh Amazon SWF saat mendaftarkan jenis alur kerja.

> **Note**
>
> Entah `@WorkflowRegistrationOptions` atau `@SkipRegistrationOptions` harus digunakan pada sebuah antarmuka yang dianotasi dengan `@Workflow`, tetapi tidak dapat menentukan keduanya.

Parameter berikut dapat ditentukan pada anotasi ini:

Deskripsi

Deskripsi teks opsional dari jenis alur kerja.

`defaultExecutionStartToCloseTimeoutSeconds`

Menentukan `defaultExecutionStartToCloseTimeout` yang terdaftar dengan Amazon SWF untuk jenis alur kerja. Ini adalah total waktu yang dapat diselesaikan oleh eksekusi alur kerja jenis ini.

Untuk informasi selengkapnya tentang timeout alur Kerja, lihat Tipe Batas Waktu Amazon SWF .

`defaultTaskStartToCloseTimeoutSeconds`

Menentukan `defaultTaskStartToCloseTimeout` yang terdaftar dengan Amazon SWF untuk jenis alur kerja. Ini menentukan waktu yang diperlukan untuk menyelesaikan tugas keputusan tunggal untuk eksekusi alur kerja jenis ini.

Jika Anda tidak menentukan `defaultTaskStartToCloseTimeout`, defaultnya adalah 30 detik.

Untuk informasi selengkapnya tentang timeout alur Kerja, lihat Tipe Batas Waktu Amazon SWF .

`defaultTaskList`

Daftar tugas default digunakan untuk tugas keputusan untuk eksekusi jenis alur kerja ini. Set default di sini dapat diganti dengan menggunakan `StartWorkflowOptions` saat memulai eksekusi alur kerja.

Jika Anda tidak menentukan `defaultTaskList`, maka akan diatur ke `USE_WORKER_TASK_LIST` secara default. Ini menunjukkan bahwa daftar tugas yang digunakan oleh pekerja yang melakukan pendaftaran alur kerja harus digunakan.

`defaultChildPolicy`

Menentukan kebijakan yang akan digunakan untuk alur kerja anak jika eksekusi jenis ini dihentikan. Nilai default-nya adalah `ABANDON`. Kemungkinan nilai adalah:

- `ABANDON` – Izinkan eksekusi alur kerja anak untuk terus berjalan
- `TERMINATE` – Menghentikan eksekusi alur kerja anak
- `REQUEST_CANCEL` – Meminta pembatalan eksekusi alur kerja anak

AWS Flow Framework untuk Pengecualian Java

Pengecualian berikut digunakan oleh AWS Flow Framework for Java. Bagian ini memberikan gambaran umum pengecualian. Untuk detail selengkapnya, lihat AWS SDK untuk Java dokumentasi pengecualian individu.

Topik

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

- WorkflowException

ActivityFailureException

Pengecualian ini digunakan oleh kerangka kerja internal untuk mengkomunikasikan kegagalan aktivitas. Ketika suatu aktivitas gagal karena pengecualian tidak tertangani, itu dibungkus dalam `ActivityFailureException` dan dilaporkan ke Amazon SWF. Anda perlu berurusan dengan pengecualian ini hanya jika Anda menggunakan poin pekerja aktivitas yang diperpanjang. Kode aplikasi Anda tidak perlu berurusan dengan pengecualian ini.

ActivityTaskException

Ini adalah kelas dasar untuk pengecualian aktivitas tugas kegagalan: `ScheduleActivityTaskFailedException`, `ActivityTaskFailedException`, `ActivityTaskTimedoutException`. Kelas Ini berisi tugas Id dan jenis aktivitas tugas gagal. Anda dapat menangkap pengecualian ini dalam pelaksanaan alur kerja Anda untuk menangani kegagalan aktivitas dengan cara umum.

ActivityTaskFailedException

Pengecualian tidak tertangani dalam aktivitas yang dilaporkan kembali ke implementasi alur kerja dengan melemparkan `ActivityTaskFailedException`. Pengecualian asli dapat diambil dari properti penyebab pengecualian ini. Pengecualian juga menyediakan informasi lain yang berguna untuk tujuan debugging, seperti pengenal aktivitas unik dalam riwayat.

Kerangka kerja ini mampu memberikan pengecualian jarak jauh dengan menserialisasikan pengecualian asli dari pekerja aktivitas.

ActivityTaskTimedOutException

Pengecualian ini dibuang jika waktu aktivitas habis oleh Amazon SWF. Hal ini bisa terjadi jika tugas aktivitas tidak dapat ditugaskan kepada pekerja dalam jangka waktu yang ditentukan atau tidak dapat diselesaikan oleh pekerja dalam waktu yang diperlukan. Anda dapat mengatur waktu habis ini pada aktivitas menggunakan anotasi `@ActivityRegistrationOptions` atau menggunakan parameter `ActivitySchedulingOptions` ketika memanggil metode aktivitas.

ChildWorkflowException

kelas dasar untuk pengecualian yang digunakan untuk melaporkan kegagalan eksekusi alur kerja anak. Pengecualian berisi Id eksekusi alur kerja anak serta jenis alur kerja. Anda dapat menangkap pengecualian ini untuk menangani kegagalan eksekusi alur kerja anak dengan cara umum.

ChildWorkflowFailedException

Pengecualian tidak tertangani dalam alur kerja anak yang dilaporkan kembali ke implementasi alur kerja induk dengan melemparkan `ChildWorkflowFailedException`. Pengecualian asli dapat diambil dari properti `cause` pengecualian ini. Pengecualian juga menyediakan informasi lain yang berguna untuk tujuan debugging, seperti pengenal unik eksekusi anak.

ChildWorkflowTerminatedException

Pengecualian ini dilemparkan dalam eksekusi alur kerja induk untuk melaporkan penghentian eksekusi alur kerja anak. Anda harus menangkap pengecualian ini jika Anda ingin berurusan dengan penghentian alur kerja anak, misalnya, untuk melakukan pembersihan atau kompensasi.

ChildWorkflowTimedOutException

Pengecualian ini dilemparkan dalam eksekusi alur kerja induk untuk melaporkan bahwa eksekusi alur kerja anak habis dan ditutup oleh Amazon SWF. Anda harus menangkap pengecualian ini jika Anda ingin berurusan dengan penutupan paksa alur kerja anak, misalnya, untuk melakukan pembersihan atau kompensasi.

DataConverterException

Kerangka kerja menggunakan komponen `DataConverter` untuk menyusun dan membongkar data yang dikirim melalui kawat. Pengecualian ini dilemparkan jika `DataConverter` gagal untuk menyusun dan membongkar data. Hal ini bisa terjadi karena berbagai alasan, misalnya, karena ketidakcocokan dalam komponen `DataConverter` yang digunakan untuk menyusun dan membongkar data.

DecisionException

Ini adalah kelas dasar untuk pengecualian yang mewakili gagal untuk memberlakukan keputusan oleh Amazon SWF. Anda dapat menangkap pengecualian ini untuk secara umum menangani exception tersebut.

ScheduleActivityTaskFailedException

Pengecualian ini dilemparkan jika Amazon SWF gagal untuk menjadwalkan tugas aktivitas. Hal ini dapat terjadi karena berbagai alasan—misalnya, aktivitas tidak lagi digunakan, atau batas Amazon SWF pada akun Anda telah tercapai. Properti `failureCause` di pengecualian menentukan penyebab pasti kegagalan untuk jadwal aktivitas.

SignalExternalWorkflowException

Pengecualian ini dilemparkan jika Amazon SWF gagal untuk memproses permintaan oleh eksekusi alur kerja untuk sinyal eksekusi alur kerja lain. Hal ini terjadi jika eksekusi alur kerja target tidak dapat ditemukan—yaitu eksekusi alur kerja yang Anda tentukan tidak ada atau dalam keadaan tertutup.

StartChildWorkflowFailedException

Pengecualian ini dilemparkan jika Amazon SWF gagal untuk memulai eksekusi alur kerja anak. Hal ini dapat terjadi karena berbagai alasan-misalnya, jenis alur kerja anak yang ditentukan tidak lagi digunakan, atau batas Amazon SWF pada akun Anda telah tercapai. Properti `failureCause` di pengecualian menentukan penyebab pasti gagal untuk memulai eksekusi alur kerja anak.

StartTimerFailedException

Pengecualian ini dilemparkan jika Amazon SWF gagal untuk memulai timer diminta oleh eksekusi alur kerja. Hal ini bisa terjadi jika ID timer yang ditentukan sudah digunakan, atau batas Amazon SWF pada akun Anda telah tercapai. Properti `failureCause` di pengecualian menentukan penyebab pasti gagal.

TimerException

Ini adalah kelas dasar untuk pengecualian yang terkait dengan timer.

WorkflowException

Pengecualian ini digunakan secara internal oleh kerangka kerja untuk melaporkan kegagalan dalam eksekusi alur kerja. Anda harus berurusan dengan pengecualian ini hanya jika Anda menggunakan titik yang diperpanjang pekerja alur kerja.

AWS Flow Framework untuk Paket Java

Bagian ini memberikan gambaran umum dari paket-paket yang disertakan dengan AWS Flow Framework untuk Java. Untuk informasi lebih lanjut tentang setiap paket, lihat com.amazonaws.services.simpleworkflow.flow di Referensi API AWS SDK untuk Java.

com.amazonaws.services.simpleworkflow.flow

Berisi komponen yang mengintegrasikan dengan Amazon SWF.

com.amazonaws.services.simpleworkflow.flow.annotations

Berisi anotasi yang digunakan oleh AWS Flow Framework untuk model pemrograman Java.

com.amazonaws.services.simpleworkflow.flow.aspectj

Berisi AWS Flow Framework untuk komponen Java yang diperlukan untuk fitur seperti @Asynchronous dan@ExponentialRetry.

com.amazonaws.services.simpleworkflow.flow.common

Berisi utilitas umum seperti konstanta kerangka kerja yang didefinisikan.

com.amazonaws.services.simpleworkflow.flow.core

Berisi fitur inti seperti `Task` dan `Promise`.

com.amazonaws.services.simpleworkflow.flow.generic

Berisi komponen inti, seperti klien generik, yang dibangun oleh fitur lain.

com.amazonaws.services.simpleworkflow.flow.interceptors

Berisi implementasi dari framework yang disediakan dekorator termasuk `RetryDecorator`.

com.amazonaws.services.simpleworkflow.flow.junit

Berisi komponen yang menyediakan integrasi Junit.

com.amazonaws.services.simpleworkflow.flow.pojo

Berisi kelas yang mengimplementasikan aktivitas dan definisi alur kerja untuk model pemrograman berbasis penjelasan.

com.amazonaws.services.simpleworkflow.flow.spring

Berisi komponen yang menyediakan integrasi Spring.

com.amazonaws.services.simpleworkflow.flow.test

Berisi kelas helper, seperti `TestWorkflowClock`, untuk implementasi alur kerja pengujian unit.

com.amazonaws.services.simpleworkflow.flow.worker

Berisi implementasi aktivitas dan pekerja alur kerja.

Riwayat Dokumen

Tabel berikut menjelaskan perubahan penting pada dokumentasi sejak rilis terakhir dari Panduan Developer AWS Flow Framework untuk Java.

- Versi API: 25-01-2012
- Pembaruan dokumentasi terbaru: 25 Juni 2018

Perubahan	Deskripsi	Tanggal Diubah
Pembaruan	Memperbaiki kesalahan dalam deskripsi `backoffCoefficient` untuk `@ExponentialRetry` . Lihat @ExponentialRetry.	25 Juni 2018
Pembaruan	Bersihkan contoh kode di seluruh panduan ini.	5 Juni 2017
Pembaruan	Menyederhanakan dan meningkatkan organisasi dan isi panduan ini.	19 Mei 2017
Pembaruan	Menyederhanakan dan meningkatkan bagian Membuat perubahan pada Decider Code: Versioning dan Feature Flag.	10 April 2017
Pembaruan	Menambahkan bagian Praktik terbaik baru dengan panduan baru untuk membuat perubahan pada kode penentu.	3 Maret 2017
Fitur baru	Anda dapat menentukan tugas Lambda selain tugas Aktivitas tradisional di alur kerja Anda. Untuk informasi selengkapnya, lihat Menerapkan AWS Lambda Tugas.	21 Juli 2015
Fitur baru	Amazon SWF menyertakan dukungan untuk menetapkan prioritas tugas pada daftar tugas, mencoba mengirimkan tugas dengan prioritas lebih tinggi sebelum tugas dengan prioritas lebih rendah. Untuk informasi selengkapnya, lihat Menetapkan prioritas tugas di Amazon SWF.	17 Desember 2014

Perubahan	Deskripsi	Tanggal Diubah
Pembaruan	Membuat pembaruan dan perbaikan.	1 Agustus 2013
Pembaruan	• Membuat pembaruan dan perbaikan, termasuk pembaruan Instruksi pengaturan untuk Eclipse 4.3 dan AWS SDK untuk Java 1.4.7. • Menambahkan satu set tutorial baru untuk membuat skenario pemula	28 Juni 2013
Fitur baru	Rilis awal AWS Flow Framework untuk Java.	27 Februari 2012

Terjemahan disediakan oleh mesin penerjemah. Jika konten terjemahan yang diberikan bertentangan dengan versi bahasa Inggris aslinya, utamakan versi bahasa Inggris.