개발자 가이드

AWS Flow Framework 자바용

API 버전 2021-04-28

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Flow Framework 자바용: 개발자 가이드

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 상표 및 트레이드 드레스는 Amazon 외 제품 또는 서비스와 함께 사용되어서는 안되며, 고객에게 혼동을 일으키거나 Amazon 브랜드 이미지를 떨어뜨리고 폄하하는 방식으로 이용할 수 없습니다. Amazon이 소유하지 않은 기타 모든 상표는 Amazon 계열사, 관련 업체 또는 Amazon의 지원 업체 여부에 상관없이 해당 소유자의 자산입니다.

Table of Contents

AWS Flow Framework Java용이란 무엇입니까? 1
이 가이드의 내용은 무엇입니까? 1
시작하기 3
프레임워크 설정 3
Maven용으로 설치 4
Eclipse용 설치 4
HelloWorld 애플리케이션 13
HelloWorld 활동 구현 14
HelloWorld 워크플로 작업자 15
HelloWorld 워크플로 시작자 16
HelloWorldWorkflow 애플리케이션 16
HelloWorldWorkflow 액티비티 워커 18
HelloWorldWorkflow 워크플로 워커 20
HelloWorldWorkflow 워크플로우 및 활동 구현 24
HelloWorldWorkflow 스타터 28
HelloWorldWorkflowAsync 애플리케이션 33
HelloWorldWorkflowAsync 활동 구현 34
HelloWorldWorkflowAsync 워크플로 구현 35
HelloWorldWorkflowAsync 워크플로 및 활동 호스트와 시작자 37
HelloWorldWorkflowDistributed 애플리케이션 38
HelloWorldWorkflowParallel 애플리케이션 41
HelloWorldWorkflowParallel 활동 작업자 41
HelloWorldWorkflowParallel 워크플로 작업자 43
HelloWorldWorkflowParallel 워크플로 및 활동 호스트와 시작자 44
AWS Flow Framework 자바의 작동 방식 45
애플리케이션 구조 45
활동 작업자의 역할 47
워크플로 작업자의 역할 47
워크플로 시작자의 역할 47
Amazon SWF가 애플리케이션과 상호 작용하는 방식 48
자세한 정보 48
안정적 실행 48
안정적 통신 제공 49
결과 분실 방지 49

- 실패한 분산 구성 요소 처리 ... 50
- 분산 실행 ... 50
- 워크플로 다시 재생 ... 51
- 다시 재생 및 비동기식 워크플로 메서드 ... 52
- 다시 재생 및 워크플로 구현 ... 52
- 작업 목록 및 작업 실행 ... 53
- 확장 가능 애플리케이션 ... 55
- 활동과 워크플로 간 데이터 교환 ... 55
- Promise<T> 유형 ... 56
- 데이터 변환기 및 마샬링 ... 57
- 애플리케이션과 워크플로 실행 간 데이터 교환 ... 57
- 제한 시간 유형 ... 58
- 워크플로 및 결정 작업의 제한 시간 ... 59
- 활동 작업의 제한 시간 ... 60
- 모범 사례 ... 62
- 결정자 코드 변경 ... 62
- 다시 재생 프로세스 및 코드 변경 ... 62
- 예제 시나리오 ... 63
- 솔루션 ... 69
- 프로그래밍 가이드 ... 75
- 워크플로 애플리케이션 구현 ... 75
- 워크플로 및 활동 계약 ... 77
- 워크플로 및 활동 유형 등록 ... 79
- 워크플로 유형 이름 및 버전 ... 80
- 신호 이름 ... 81
- 활동 유형 이름 및 버전 ... 81
- 기본 작업 목록 ... 81
- 기타 등록 옵션 ... 81
- 활동 및 워크플로 클라이언트 ... 82
- 워크플로 클라이언트 ... 82
- 활동 클라이언트 ... 91
- 예약 옵션 ... 95
- 동적 클라이언트 ... 95
- 워크플로 구현 ... 97
- 결정 컨텍스트 ... 98
- 실행 상태 노출 ... 98

워크플로 로컬 101
활동 구현 102
활동을 수동으로 완료 103
Lambda 작업 구현 104
AWS Lambda 정보 105
Lambda 작업 사용의 이점 및 제한 사항 105
AWS Flow Framework for Java 워크플로에서 Lambda 작업 사용 106
HelloLambda 샘플 보기 110
AWS Flow Framework for Java로 작성한 프로그램 실행 111
WorkflowWorker 112
ActivityWorker 112
작업자 스레딩 모델 113
작업자 확장성 115
실행 콘텍스트 115
결정 컨텍스트 116
활동 실행 컨텍스트 118
하위 워크플로 실행 119
연속 워크플로 121
작업 우선 순위 설정 122
워크플로의 작업 우선 순위 설정 123
활동의 작업 우선 순위 설정 124
DataConverters 124
데이터를 비동기식 메서드로 전달 125
모음 및 맵을 비동기식 메서드로 전달 125
Settable<T> 126
@NoWait 128
Promise<Void> 128
AndPromise 및 OrPromise 128
테스트 가능성 및 종속성 주입 128
Spring 통합 129
JUnit 통합 135
오류 처리 141
TryCatchFinally 의미론 143
취소 144
중첩 TryCatchFinally 148
실패한 활동 재시도 150

성공할 때까지 재시도하는 전략 150
기하급수적 재시도 전략 153
사용자 지정 재시도 전략 159
대몬(daemon) 작업 162
다시 재생 동작 164
예 1: 동기식 다시 재생 164
예 2: 비동기식 다시 재생 166
참고 항목 167
Under the Hood 168
태스크 168
실행 순서 169
워크플로 실행 170
불확정성 173
문제 해결 및 디버깅 팁 174
컴파일 오류 174
알 수 없는 리소스 장애 174
약속에서 get()을 호출할 때 발생하는 예외 175
불확정적 워크플로 175
버전 관리로 인한 문제 175
워크플로 실행 관련 문제 해결 및 디버깅 176
손실된 작업 177
참조 179
주석 179
@활동 179
@활동 180
@ActivityRegistrationOptions 180
@비동기식 181
@실행 182
@ExponentialRetry 182
@GetState 183
@ManualActivityCompletion 183
@신호 184
@SkipRegistration 184
@Wait 및 @NoWait 184
@워크플로 184
@WorkflowRegistrationOptions 185

- 예외 ... 186
  - ActivityFailureException ... 187
  - ActivityTaskException ... 187
  - ActivityTaskFailedException ... 188
  - ActivityTaskTimedOutException ... 188
  - ChildWorkflowException ... 188
  - ChildWorkflowFailedException ... 188
  - ChildWorkflowTerminatedException ... 188
  - ChildWorkflowTimedOutException ... 188
  - DataConverterException ... 189
  - DecisionException ... 189
  - ScheduleActivityTaskFailedException ... 189
  - SignalExternalWorkflowException ... 189
  - StartChildWorkflowFailedException ... 189
  - StartTimerFailedException ... 189
  - TimerException ... 190
  - WorkflowException ... 190
- 패키지 ... 190

문서 기록 ... 192
AWS 용어집 ... 194
... cxcv

AWS Flow Framework Java용이란 무엇입니까?

를 AWS Flow Framework 사용하면 워크플로 로직을 구현하는 데 집중할 수 있습니다. 프레임워크는 Amazon SWF의 일정 관리, 라우팅 및 상태 관리 기능을 사용하여 워크플로의 실행을 관리하고 확장 가능하고 안정적이며 감사 가능하도록 만듭니다. 프레임워크 기반 워크플로는 매우 동시 실행됩니다. 워크플로를 여러 구성 요소에 분산하여 개별 컴퓨터에서 별도의 프로세스로 실행하고 독립적으로 확장할 수 있습니다. 구성 요소 중 하나라도 실행 중인 경우 애플리케이션을 계속 진행할 수 있으므로 내결함성이 높습니다.

이 가이드의 내용은 무엇입니까?

이 안내서에는 프레임워크를 설치, 설정 및 사용하여 Amazon SWF 애플리케이션을 구축하는 방법에 대한 정보가 있습니다.

AWS Flow Framework for Java 시작하기

AWS Flow Framework Java용 버전을 막 시작하는 경우 AWS Flow Framework for Java 시작하기 섹션을 읽어보십시오. AWS Flow Framework Java용 다운로드 및 설치, 개발 환경 설정 방법, 워크플로우 생성의 간단한 예를 안내합니다.

AWS Flow Framework 자바의 작동 방식

기본 Amazon SWF와 프레임워크 개념을 소개하고, 프레임워크 애플리케이션의 기본 구조와 분산 워크플로의 일부 간에 데이터가 교환되는 방식을 설명합니다.

AWS Flow Framework for Java 프로그래밍 가이드

이 장에서는 활동 및 워크플로 유형 등록, 워크플로 클라이언트 구현, 하위 워크플로 생성, 오류 처리 등 AWS Flow Framework for Java를 이용한 워크플로 애플리케이션 개발과 관련해 기본 프로그래밍 지침을 제공합니다.

Under the Hood

이 장에서는 AWS Flow Framework for Java의 작동 방식을 보다 심층적으로 살펴보고 비동기 워크플로의 실행 순서에 대한 추가 정보와 표준 워크플로 실행의 논리적인 단계를 제공합니다.

문제 해결 및 디버깅 팁

이 장에서는 워크플로 관련 문제를 해결하거나 흔한 오류를 방지하는 방법을 학습하는 데 사용할 수 있는 흔한 오류에 관한 정보를 제공합니다.

AWS Flow Framework for Java 참조

이 장에서는 Java용 SDK가 AWS Flow Framework Java용 SDK에 추가하는 주석, 예외 및 패키지에 대한 참조입니다.

문서 기록

이 장에서는 설명서의 주요 변경 사항에 관한 세부 정보를 제공합니다. 여기에는 새 단원 및 주제와 상당 부분 개정된 주제가 나열되어 있습니다.

AWS Flow Framework for Java 시작하기

이 단원에서는 기본적인 프로그래밍 모델 및 API를 소개하는 일련의 단순한 예시 애플리케이션을 차례로 살펴봄으로써 AWS Flow Framework의 개요를 이해할 수 있습니다. 예시 애플리케이션은 C 언어 및 관련 프로그래밍 언어를 소개하는 데 사용되는 표준 Hello World 애플리케이션에 기반을 두고 있습니다. Hello World를 Java로 구현한 일반적인 형태는 다음과 같습니다.

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

다음은 예시 애플리케이션에 관해 간략히 설명한 것입니다. 이 예시에는 온전한 소스 코드가 포함되어 있으므로 사용자는 직접 애플리케이션을 구현 및 실행할 수 있습니다. 시작하기 전에 먼저 AWS Flow Framework for Java 설정에 설명된 것처럼 개발 환경을 구성하고 AWS Flow Framework for Java 프로젝트를 생성해야 합니다.

- HelloWorld 애플리케이션에서는 Hello World를 표준 Java 애플리케이션으로 구현하되 워크플로 애플리케이션처럼 구성하는 방식으로 워크플로 애플리케이션을 소개합니다.
- HelloWorldWorkflow 애플리케이션에서는 AWS Flow Framework for Java를 사용하여 HelloWorld를 Amazon SWF 워크플로로 변환합니다.
- HelloWorldWorkflowAsync 애플리케이션에서는 `HelloWorldWorkflow`가 비동기 워크플로 메서드를 사용하도록 수정합니다.
- HelloWorldWorkflowDistributed 애플리케이션에서는 워크플로 및 활동 작업자가 별도의 시스템에서 실행될 수 있도록 `HelloWorldWorkflowAsync`를 수정합니다.
- HelloWorldWorkflowParallel 애플리케이션에서는 두 가지 활동을 병렬로 실행할 수 있도록 `HelloWorldWorkflow`를 수정합니다.

AWS Flow Framework for Java 설정

AWS Flow Framework for Java는 AWS SDK for Java에 포함되어 있습니다. AWS SDK for Java를 아직 설정하지 않은 경우 AWS SDK for Java 개발자 안내서의 시작하기를 방문하여 SDK 자체 설치 및 구성에 대한 자세한 내용을 확인하십시오.

이 단원에서는 AWS Flow Framework for Java를 사용하는 데 필요한 추가 절차에 관한 정보를 제공합니다. 이 절차는 Eclipse 및 Maven 용도로 제공됩니다.

주제

- Maven용으로 설치
- Eclipse용 설치

Maven용으로 설치

Amazon은 Maven Central Repository에서 Amazon SWF 빌드 도구를 제공하여 Maven 프로젝트에서 AWS Flow Framework for Java를 설정할 수 있도록 지원합니다.

Maven용 플로우 프레임워크를 설정하려면 프로젝트의 `pom.xml` 파일에 다음 종속성을 추가합니다.

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-swf-build-tools</artifactId>
  <version>1.0</version>
</dependency>
```

Amazon SWF 빌드 도구는 오픈 소스입니다. 코드를 보거나 다운로드하거나 도구를 직접 빌드하려면 https://github.com/aws/aws-swf-build-tools 리포지토리를 방문하십시오.

Eclipse용 설치

Eclipse IDE를 사용하는 경우 AWS Toolkit for Eclipse를 사용하여 AWS Flow Framework for Java를 설치합니다.

주제

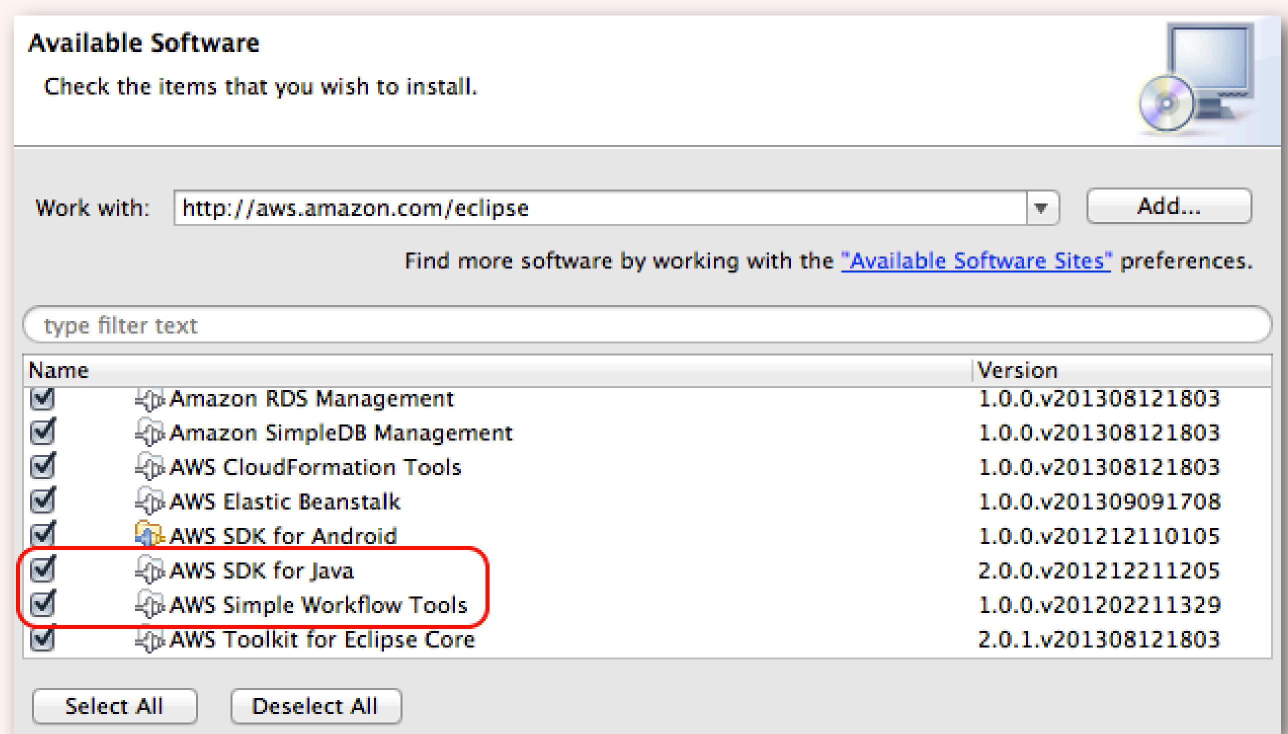
- AWS Toolkit for Eclipse 설치
- AWS Flow Framework for Java 프로젝트 생성

AWS Toolkit for Eclipse 설치

AWS Flow Framework for Java를 시작하는 가장 쉬운 방법은 Toolkit for Eclipse를 설치하는 것입니다. Toolkit for Eclipse를 설치하려면 AWS Toolkit for Eclipse 시작 안내서의 AWS Toolkit for Eclipse 설정을 참조하십시오.

Important

Eclipse의 [Available Software] 대화 상자에서 설치할 패키지를 선택할 때 AWS SDK for Java 및 AWS Simple Workflow 도구를 모두 포함해야 합니다.



AWS Toolkit for Eclipse 최상위 노드를 선택하거나 모두 선택을 선택하여 사용 가능한 패키지를 모두 설치한 경우 두 패키지가 모두 자동으로 선택되어 설치됩니다.

AWS Flow Framework for Java 프로젝트 생성

Eclipse에 적절히 구성된 AWS Flow Framework for Java 프로젝트를 생성하는 작업에는 다음과 같은 여러 단계가 수반됩니다.

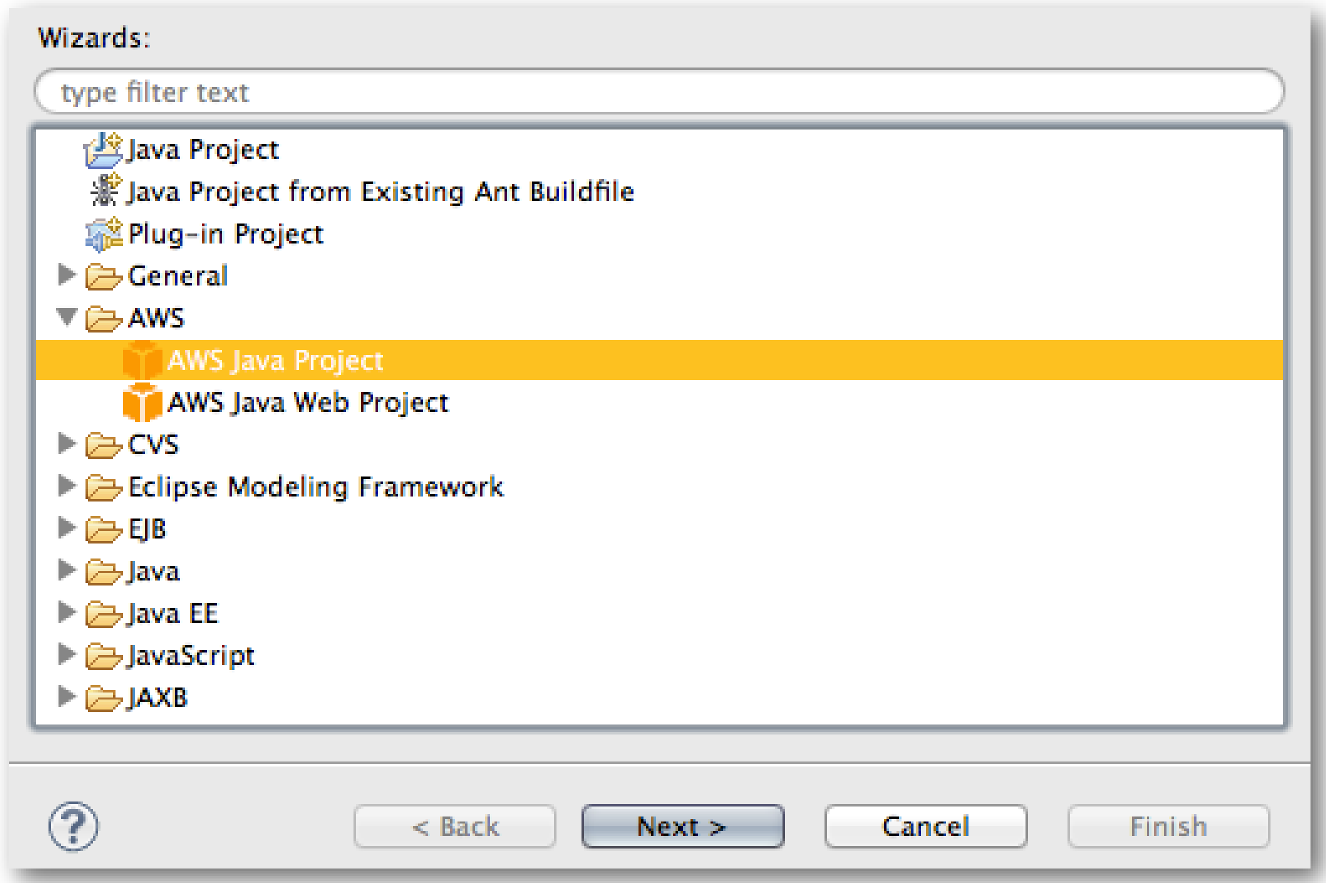
1. AWS Java 프로젝트를 생성합니다.
2. 프로젝트에 대해 주석 처리 활성화

3. AspectJ 활성화 및 구성

이제 이 절차의 각 단계에 관해 설명하겠습니다.

AWS Java 프로젝트를 생성하려면

1. Eclipse를 시작합니다.
2. Java 관점을 선택하려면 [Window], [Open Perspective], [Java]를 선택합니다.
3. [File], [New], [AWS Java Project]를 선택합니다.



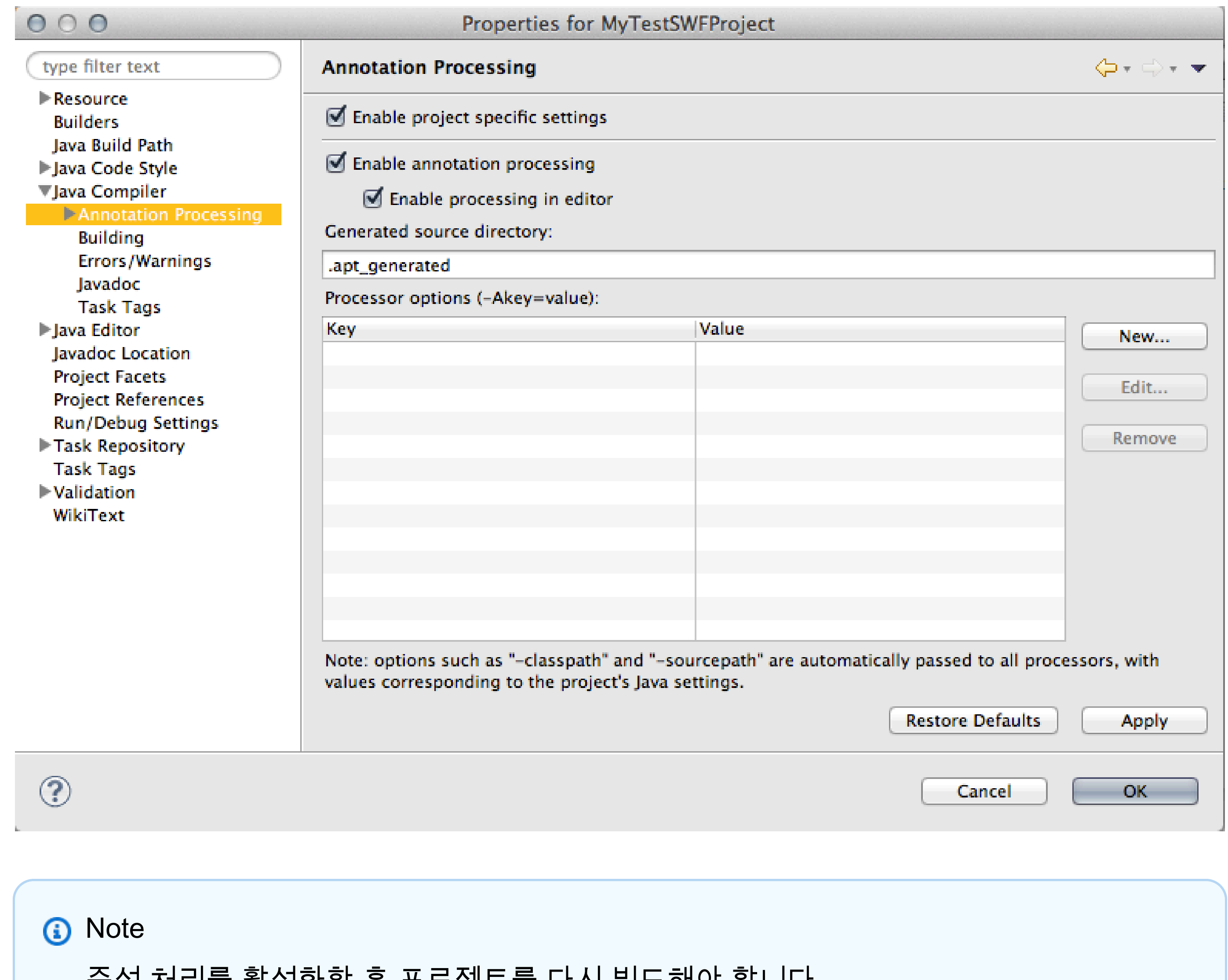
4. AWS Java 프로젝트 마법사를 사용하여 새 프로젝트를 생성합니다.

> **Note**
>
> Eclipse를 사용해 AWS Java 프로젝트를 처음 생성하는 경우 프로젝트 마법사가 시작되면 SDK for Java가 자동으로 다운로드되어 설치됩니다.

AWS Java 프로젝트를 생성한 후에 프로젝트에 대해 주석 처리를 활성화합니다. AWS Flow Framework for Java에는 주석이 붙은 소스 코드에 따라 몇 가지 키 클래스를 생성하는 주석 프로세서가 포함되어 있습니다.

주석 처리를 활성화하려면

1. [Project Explorer]에서 프로젝트를 마우스 오른쪽 버튼으로 클릭하고 [Properties]를 선택합니다.
2. [Properties] 대화 상자에서 [Java Compiler] > [Annotation Processing] 순으로 이동합니다.
3. [Enable project specific settings]([Enable annotation processing]도 활성화됨. 그러나 이 옵션의 확인란도 선택되도록 보장하지는 않는 경우)의 확인란을 선택합니다. 그 다음에 확인(OK)를 선택합니다.



> **Note**
>
> 주석 처리를 활성화한 후 프로젝트를 다시 빌드해야 합니다.

AspectJ 활성화 및 구성

그다음에는 AspectJ를 활성화 및 구성해야 합니다. `@Asynchronous`과 같은 특정 AWS Flow Framework for Java 주석에는 AspectJ가 필요합니다. AspectJ를 직접 사용할 필요는 없지만 로드 시간 위빙 또는 컴파일 시간 위빙을 통해 활성화해야 합니다.

Note

권장 접근 방식은 로드 시간 위빙을 사용하는 것입니다.

주제

- 필수 조건
- AspectJ 로드 시간 위빙 구성
- AspectJ 컴파일 시간 위빙
- AspectJ 및 Eclipse 관련 문제 해결

필수 조건

AspectJ를 구성하기 전에 다음과 같이 Java 버전과 매치되는 AspectJ 버전이 필요합니다.

- Java 8을 사용 중이라면 최신 AspectJ 1.8.X 릴리스를 다운로드하십시오.
- Java 7을 사용 중이라면 최신 AspectJ 1.7.X 릴리스를 다운로드하십시오.
- Java 6를 사용 중이라면 최신 AspectJ 1.6.X 릴리스를 다운로드하십시오.

Eclipse 다운로드 페이지에서 이 AspectJ 버전 중 원하는 것을 다운로드할 수 있습니다.

AspectJ 다운로드를 마친 후에는 다운로드한 `.jar` 파일을 실행하여 AspectJ를 설치합니다. AspectJ 설치에서는 바이너리를 설치하고 싶은 위치를 묻고, 최종 화면에서는 설치 완료를 위한 권장 절차를 제공합니다. aspectjweaver.jar 파일의 위치를 기억해 둡니다,. Eclipse에서 AspectJ를 구성하는 데 필요하기 때문입니다.

AspectJ 로드 시간 위빙 구성

AWS Flow Framework for Java 프로젝트에 대해 AspectJ 로드 시간 위빙을 구성하려면 먼저 AspectJ JAR 파일을 Java 에이전트로 지정한 다음, 프로젝트에 `aop.xml` 파일을 추가하여 이를 구성합니다.

AspectJ를 Java 에이전트로 추가하려면

1. [Preferences] 대화 상자를 열려면 [Window], [Preferences]를 선택합니다.
2. [Java] > [Installed JREs] 순으로 이동합니다.
3. 적절한 JRE를 선택하고 [Edit]를 선택합니다.
4. [Default VM arguments] 상자에 설치된 AspectJ 바이너리로 가는 경로를 입력합니다. 이 경로의 예를 들면 `/home/user/aspectj1.7/lib/aspectjweaver.jar`와 같은 것인데, 이는 사용자의 운영 체제와 다운로드한 AspectJ의 버전에 다라 달라집니다.

   Linux, macOS 또는 Unix에서는 다음을 사용합니다.

   ```
   -javaagent:/your_path/aspectj/lib/aspectjweaver.jar
   ```

   Windows에서는 그 대신에 다음과 같이 표준 Windows 스타일 경로를 사용합니다.

   ```
   -javaagent:C:\your_path\aspectj\lib\aspectjweaver.jar
   ```

AWS Flow Framework for Java에 대해 AspectJ를 구성하려면 `aop.xml` 파일을 프로젝트에 추가합니다.

aop.xml 파일을 추가하려면

1. 프로젝트의 [src] 디렉터리에 `META-INF`라는 디렉터리를 추가합니다.
2. `META-INF`에 다음 내용과 함께 `aop.xml`이라는 파일을 추가합니다.

```
<aspectj>
   <aspects>
      <aspect
 name="com.amazonaws.services.simpleworkflow.flow.aspectj.AsynchronousAspect"/>
      <aspect
 name="com.amazonaws.services.simpleworkflow.flow.aspectj.ExponentialRetryAspect"/>
```

```
    </aspects>
    <weaver options="-verbose">
      <include within="MySimpleWorkflow.*"/>
    </weaver>
</aspectj>
```

`<include within=""/>`의 값은 프로젝트의 패키지에 어떤 이름을 붙이느냐에 따라 달라집니다. 위 예에서는 프로젝트의 패키지가 MySimpleWorkflow.* 패턴을 따른다고 가정합니다. 프로젝트의 패키지에 적절한 값을 사용합니다.

AspectJ 컴파일 시간 위빙

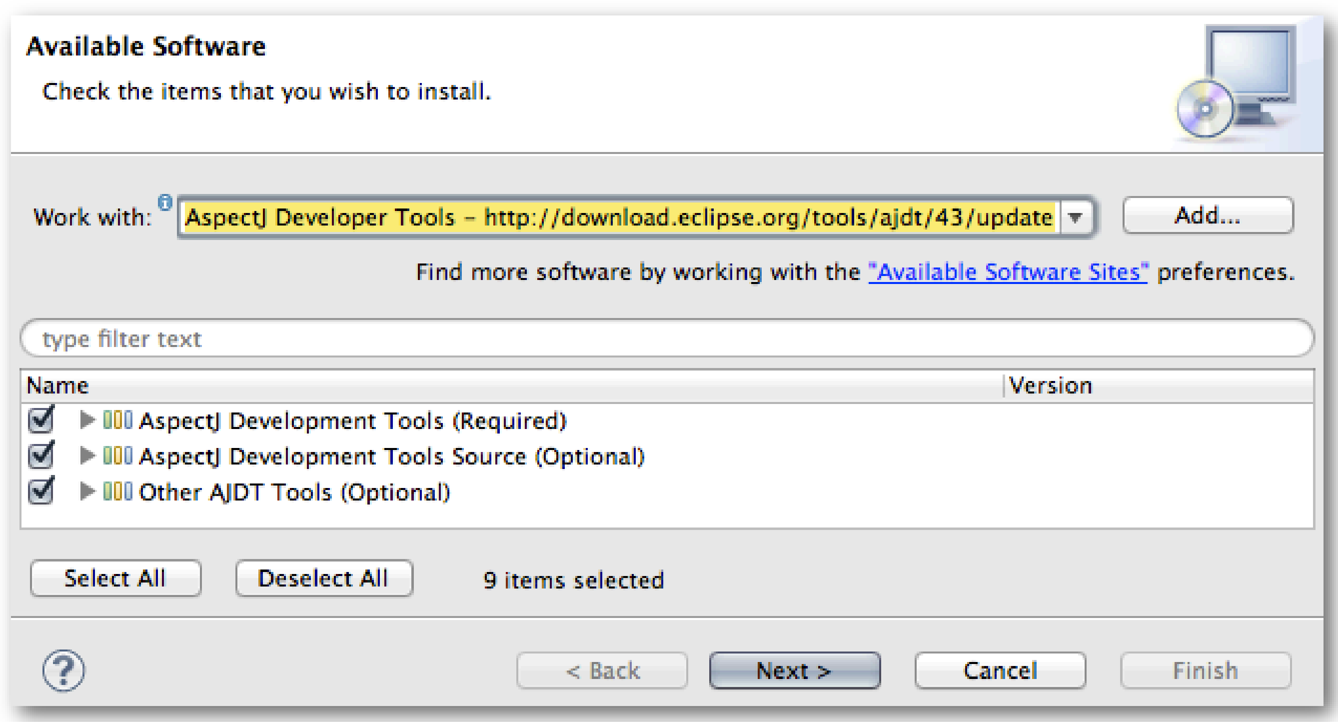
AspectJ 컴파일 시간 위빙을 활성화 및 구성하려면 먼저 Eclipse에 대해 AspectJ 개발자 도구를 설치해야 합니다. 이 도구는 http://www.eclipse.org/aspectj/downloads.php에서 얻을 수 있습니다.

Eclipse에서 AspectJ 개발자 도구를 설치하려면

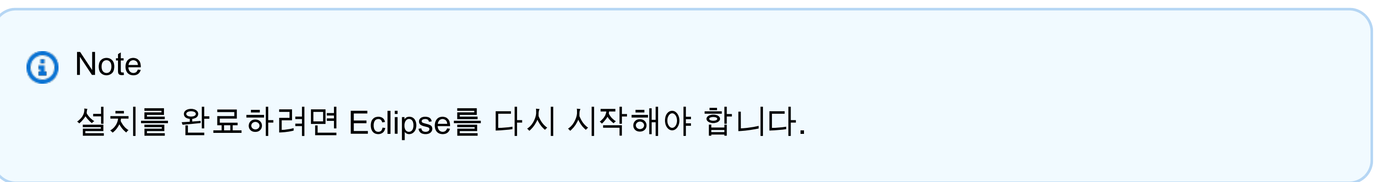
1. [Help] 메뉴에서 [Install New Software]를 선택합니다.
2. [Available Software] 대화 상자에 `http://download.eclipse.org/tools/ajdt/version/dev/update`를 입력합니다. 여기에서 version은 Eclipse 버전 번호를 나타냅니다. 예를 들어 Eclipse 4.6을 사용하는 경우 `http://download.eclipse.org/tools/ajdt/46/dev/update`이라고 입력합니다.

   > **Important**
   >
   > AspectJ 버전이 Java 버전과 매치되어야 합니다. 그렇지 않으면 AspectJ 설치가 실패합니다.

3. [Add]를 선택하여 위치를 추가합니다. 위치가 추가되면 AspectJ 개발자 도구가 나열됩니다.



4. [Select all]을 선택하여 AspectJ 개발자 도구를 모두 선택한 다음, [Next]를 선택하여 설치합니다.



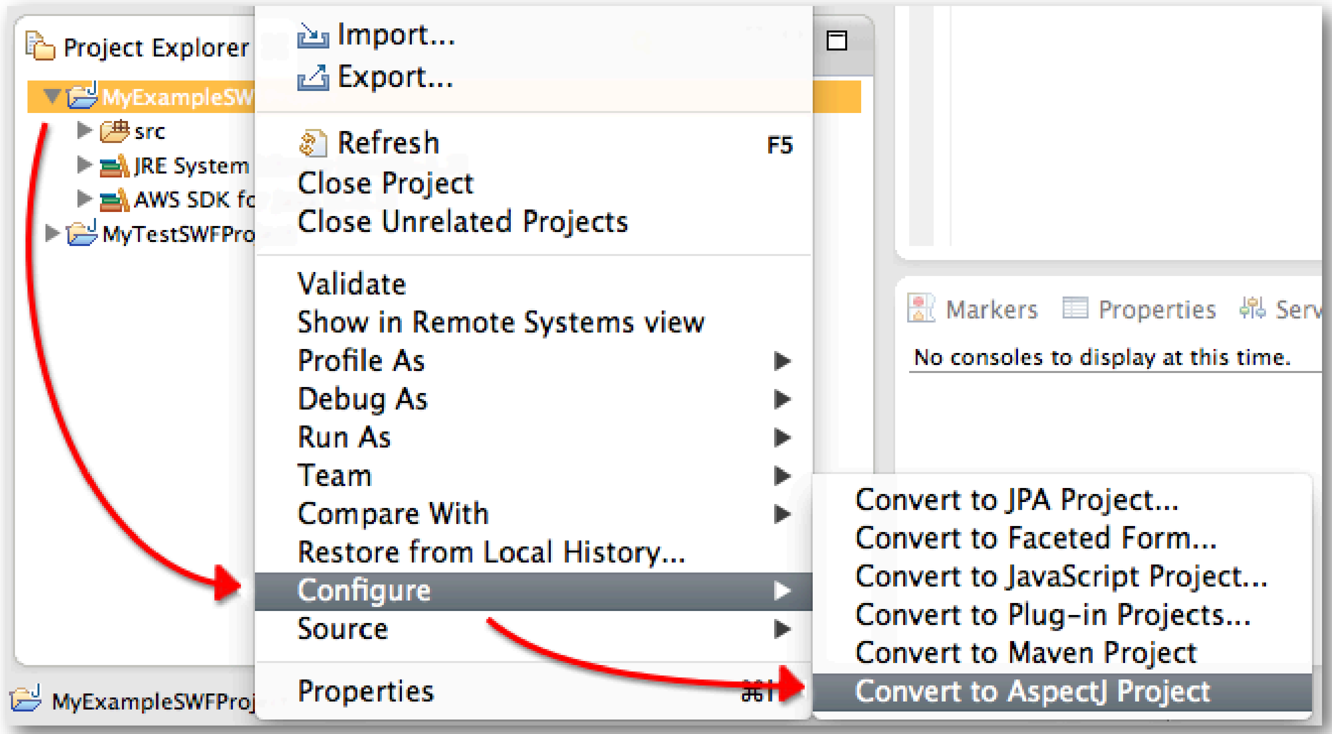
Note

설치를 완료하려면 Eclipse를 다시 시작해야 합니다.

그다음에는 프로젝트를 구성해야 합니다.

AspectJ 컴파일 시간 위빙에 대해 프로젝트를 구성하려면

1. [Project Explorer]에서 프로젝트를 마우스 오른쪽 버튼으로 클릭하고 [Configure] > [Convert to AspectJ Project]를 선택합니다.



프로젝트에 AspectJ 실행 시간 라이브러리가 추가됩니다.

2. 프로젝트를 다시 마우스 오른쪽 버튼을 클릭하고 [Properties]를 선택합니다.
3. [AspectJ Build]를 선택한 다음 [Aspect Path] 탭을 선택합니다.
4. Add External JARs(외부 JAR 추가)를 선택하고 AWS SDK for Java JAR 파일을 프로젝트의 Aspect Path(Aspect 경로)에 추가합니다.

> **Note**
>
> AWS Toolkit for Eclipse는 `.metadata/.plugins/com.amazonaws.eclipse.core/aws-java-sdk/`*`AWS Version`*`/lib` 디렉터리의 사용자 작업 영역에 AWS SDK for Java JAR 파일을 설치합니다. 이때 *`AWS Version`*을 설치된 AWS SDK 버전 번호로 바꿉니다. 또는 일반 AWS SDK 설치와 함께 내장된 JAR 파일을 사용할 수 있습니다. 이 파일은 `lib` 디렉터리에 있습니다.

AspectJ 및 Eclipse 관련 문제 해결

AspectJ Eclipse 플러그인에는 생성된 코드가 컴파일되는 것을 방지하는 것과 관련된 문제가 있습니다. 다시 컴파일한 후 생성된 코드를 강제로 인식하도록 하는 가장 빠른 방법은 [Java Build Path] 설정

페이지의 [Order and Export] 탭에서 생성된 코드가 들어 있는 소스 디렉토리의 순서를 변경하는 것입니다 (예: 기본값을 `apt/java`로 설정할 수 있음).

HelloWorld 애플리케이션

Amazon SWF 애플리케이션이 구성되는 방식을 소개하기 위해 우리는 워크플로처럼 작동하지만 로컬에서 단일 프로세스로 실행되는 Java 애플리케이션을 생성할 것입니다. Amazon Web Services와의 연결은 필요하지 않습니다.

Note

HelloWorldWorkflow 예시는 이것을 기반으로 구축되며, Amazon SWF에 연결되어 워크플로 관리 작업을 처리합니다.

워크플로 애플리케이션은 다음과 같은 세 가지 기본 구성 요소로 이루어집니다.

- 활동 작업자는 일련의 활동을 지원하는데, 각 활동은 특정 작업을 독립적으로 수행하기 위해 실행하는 메서드입니다.
- 워크플로 작업자는 여러 활동의 실행을 조율하고 데이터 흐름을 관리합니다. 이는 워크플로 토폴로지를 프로그래밍 방식으로 실현한 것으로서, 기본적으로 다양한 활동이 실행되는 시점, 순차적으로 또는 동시적으로 실행할지 여부 등을 정의하는 순서도입니다.
- 워크플로 시작자는 실행이라는 이름의 워크플로 인스턴스를 시작하고, 실행 중에 이 인스턴스와 상호 작용할 수 있습니다.

HelloWorld는 세 가지 클래스 및 두 가지 관련 인터페이스로 구현됩니다. 이에 관해서는 다음 단원에서 설명합니다. 시작하기 전에 먼저 AWS Flow Framework for Java 설정에 설명된 것처럼 개발 환경을 설정하고 새 AWS Java 프로젝트를 생성해야 합니다. 다음 연습에 사용되는 패키지는 모두 `helloWorld.`*XYZ*라는 이름이 지정되어 있습니다. 이 이름을 사용하려면 다음과 같이 aop.xml에 `within` 속성을 설정하십시오.

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

HelloWorld를 구현하려면 `helloWorld.HelloWorld`라는 AWS SDK 프로젝트에 새 Java 패키지를 생성하고 다음 파일을 추가하십시오.

- `GreeterActivities.java`라는 이름의 인터페이스 파일
- 활동 작업자를 구현하는 `GreeterActivitiesImpl.java`라는 이름의 클래스 파일
- `GreeterWorkflow.java`라는 이름의 인터페이스 파일
- 워크플로 작업자를 구현하는 `GreeterWorkflowImpl.java`라는 이름의 클래스 파일
- 워크플로 시작자를 구현하는 `GreeterMain.java`라는 이름의 클래스 파일

세부 정보는 다음 단원에 설명되어 있으며 적절한 파일에 추가할 수 있는 각 구성 요소의 코드 전체가 포함됩니다.

HelloWorld 활동 구현

HelloWorld는 `"Hello World!"`라는 인사말을 콘솔에 출력하는 작업 전체를 세 가지 작업으로 나누는데, 각 작업은 활동 메서드에서 수행합니다. 활동 메서드는 `GreeterActivities` 인터페이스에 다음과 같이 정의되어 있습니다.

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

HelloWorld에는 활동 구현 한 개, 즉 `GreeterActivitiesImpl`이 있는데, 이 구현에서는 다음과 같이 `GreeterActivities` 메서드를 제공합니다.

```
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);
```

```
    }
}
```

활동은 서로 독립적이어서 여러 워크플로에서 자주 사용할 수 있습니다. 예를 들어 어느 워크플로에서도 say 활동을 사용하여 문자열을 콘솔에 출력할 수 있습니다. 또한 워크플로에는 각기 일련의 다른 작업을 수행하는 여러 개의 활동 구현이 있을 수 있습니다.

HelloWorld 워크플로 작업자

"Hello World!"를 콘솔에 출력하려면 활동 작업이 정확한 데이터로 정확한 순서에 따라 순차적으로 실행되어야 합니다. HelloWorld 워크플로 작업자는 단순한 선형 워크플로 토폴로지에 근거하여 활동의 실행을 조율합니다.

세 가지 활동은 순차적으로 실행되고, 데이터 한 활동에서 다음 활동으로 흘러갑니다.

HelloWorld 워크플로 작업자에는 다음과 같이 `GreeterWorkflow` 인터페이스에 정의된, 워크플로의 진입점인 메서드가 한 개 있습니다.

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

`GreeterWorkflowImpl` 클래스는 이 인터페이스를 다음과 같이 구현합니다.

```
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` 메서드는 `GreeterActivitiesImpl`의 인스턴스를 생성하고, 각 활동 메서드를 정확한 순서로 호출하고, 각 메서드에 적절한 데이터를 전달하여 HelloWorld 토폴로지를 구현합니다.

HelloWorld 워크플로 시작자

워크플로 시작자는 워크플로 실행을 시작하는 애플리케이션으로서, 실행 중에 워크플로와 상호 작용도 할 수 있습니다. `GreeterMain` 클래스에서는 HelloWorld 워크플로 시작자를 다음과 같이 구현합니다.

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

`GreeterMain`에서는 `GreeterWorkflowImpl`의 인스턴스를 생성하고 `greet`를 호출하여 워크플로 작업자를 실행합니다. `GreeterMain`을 Java 애플리케이션으로 실행하면 "Hello, World!"가 콘솔 출력에 보일 것입니다.

HelloWorldWorkflow 애플리케이션

기본 HelloWorld예제는 워크플로우처럼 구성되어 있지만 다음과 같은 몇 가지 주요 측면에서 Amazon SWF 워크플로와 다릅니다.

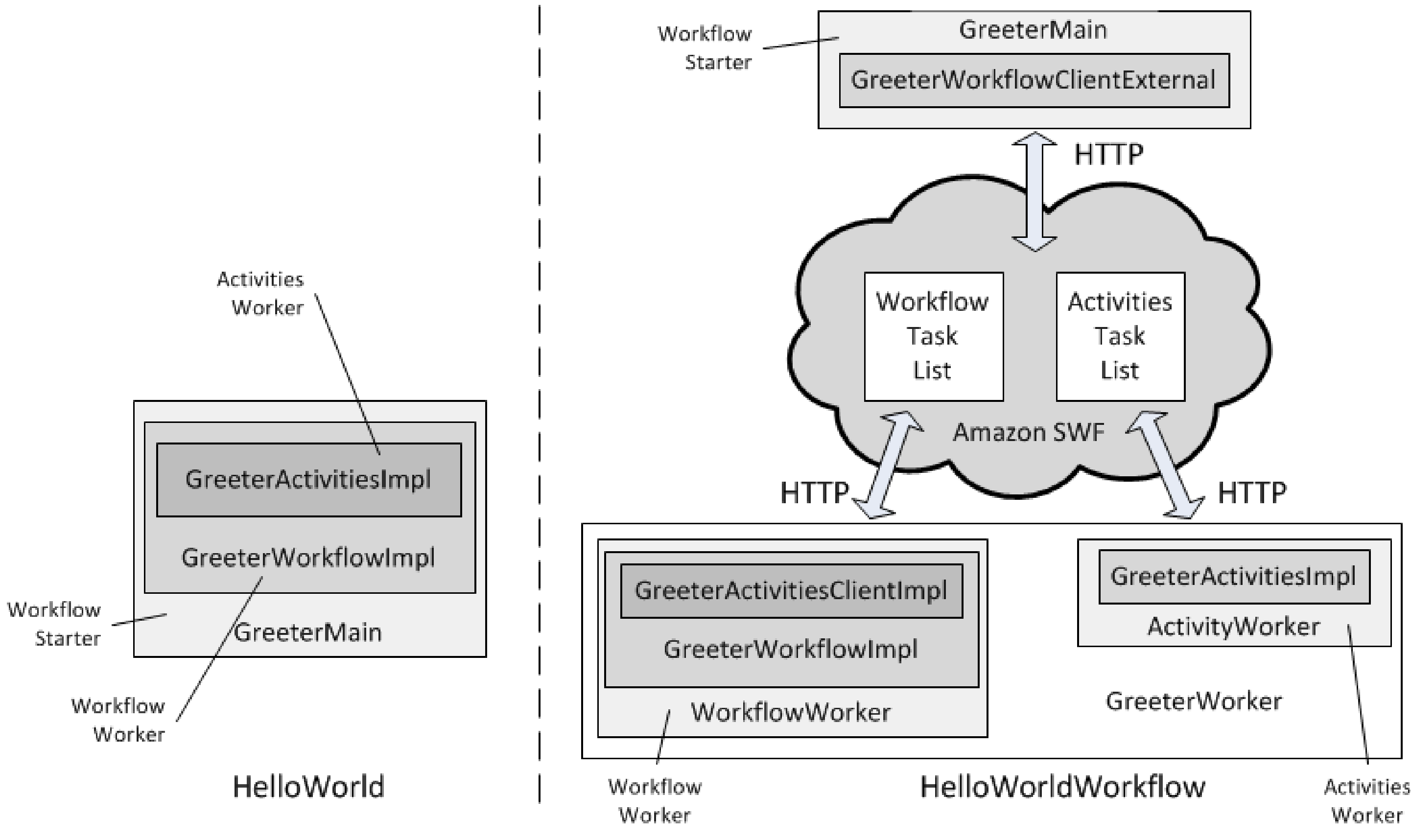
기존 및 Amazon SWF 워크플로 애플리케이션

HelloWorld	Amazon SWF 워크플로
로컬에서 단일 프로세스로 실행됩니다.	Amazon EC2 인스턴스, 프라이빗 데이터 센터, 클라이언트 컴퓨터 등 여러 시스템 전반에 분산할 수 있는 여러 프로세스를 실행합니다. 동일한 운영 체제를 실행할 필요조차 없습니다.
활동은 완료될 때까지 차단되는 동기식 메서드입니다.	활동은 즉시 반환하고 워크플로에서 다른 작업을 수행함과 동시에 활동이 완료되기를 기다릴 수 있게 해주는 비동기식 메서드로 대표됩니다.
워크플로 작업자는 적절한 메서드를 호출하여 활동 작업자와 상호 작용합니다.	워크플로 작업자는 중간 역할을 하는 Amazon SWF와 함께 HTTP 요청을 사용하여 활동 작업자와 상호 작용합니다.

HelloWorld	Amazon SWF 워크플로
워크플로 작업자는 적절한 메서드를 호출하여 워크플로 작업자와 상호 작용합니다.	워크플로 시작자는 중간 역할을 하는 Amazon SWF와 함께 HTTP 요청을 사용하여 워크플로 작업자와 상호 작용합니다.

사용자는 예를 들어 워크플로 작업자가 웹 서비스 호출을 통해 직접 활동 작업자와 상호 작용하게 함으로써 처음부터 분산형 비동기식 워크플로 애플리케이션을 구현할 수 있습니다. 그러나 이때 여러 활동의 비동기식 실행을 관리하고 데이터 흐름을 처리하는 등의 작업에 필요한 모든 복잡한 코드를 구현해야 합니다. Java 및 Amazon AWS Flow Framework SWF용 SWF는 이러한 모든 세부 사항을 처리하므로 사용자는 비즈니스 로직을 구현하는 데 집중할 수 있습니다.

HelloWorldWorkflow Amazon SWF 워크플로로 HelloWorld 실행되는 수정된 버전입니다. 다음 그림은 이 두 애플리케이션의 작동 방식을 요약한 것입니다.



HelloWorld 단일 프로세스로 실행되며 스타터, 워크플로 작업자 및 활동 작업자는 일반적인 메서드 호출을 사용하여 상호 작용합니다. `HelloWorldWorkflow`를 사용하면 시작자, 워크플로 작업자 및 활동 작업자는 HTTP 요청을 사용하여 Amazon SWF를 통해 상호 작용하는 분산 구성 요소입니다.

Amazon SWF는 워크플로 및 활동 작업 목록을 유지 관리하여 상호 작용을 관리하고, 각 구성 요소에 전달합니다. 이 섹션에서는 프레임워크의 작동 방식을 설명합니다 HelloWorldWorkflow.

HelloWorldWorkflow 는 AWS Flow Framework for Java API를 사용하여 구현됩니다. 이 API는 백그라운드에서 Amazon SWF와 상호 작용하는 복잡한 세부 사항을 처리하고 개발 프로세스를 상당히 단순화합니다. 이전에 사용했던 것과 동일한 프로젝트를 사용할 수 있습니다. 이 프로젝트는 이미 HelloWorld Java 애플리케이션용으로 AWS Flow Framework 구성되어 있습니다. 그러나 애플리케이션을 실행하려면 다음과 같이 Amazon SWF 계정을 설정해야 합니다.

- 아직 AWS 계정이 없는 경우 Amazon Web Services에서 계정을 등록하십시오.
- 계정의 액세스 ID 및 보안 ID를 AWS_ACCESS_KEY_ID 및 AWS_SECRET_KEY 환경 변수에 각각 지정합니다. 코드에 리터럴 키 값을 노출하지 않는 것이 좋습니다. 그 값을 환경 변수에 저장해두면 관리하기가 편합니다.
- Amazon Simple Workflow Service에서 Amazon SWF 계정에 가입하십시오.
- 에 AWS Management Console 로그인하고 Amazon SWF 서비스를 선택합니다.
- 상단 오른쪽 모서리에 있는 도메인 관리를 선택하고 새 Amazon SWF 도메인을 등록합니다. 도메인은 워크플로 및 활동 유형, 워크플로 실행과 같은 애플리케이션 리소스를 위한 논리적 컨테이너입니다. 편리한 도메인 이름은 아무거나 사용할 수 있지만, 안내에서는 ""를 사용합니다. helloWorldWalkthrough

를 구현하려면 HelloWorld의 복사본을 만드십시오. HelloWorldWorkflow HelloWorld 프로젝트 디렉터리에 패키지를 넣고 이름을 HelloWorld로 지정합니다. HelloWorldWorkflow. 다음 섹션에서는 AWS Flow Framework Java용 를 사용하고 Amazon SWF 워크플로 애플리케이션으로 실행하도록 원본 HelloWorld 코드를 수정하는 방법을 설명합니다.

HelloWorldWorkflow 액티비티 워커

HelloWorld 액티비티 워커를 단일 클래스로 구현했습니다. AWS Flow Framework for Java 액티비티 워커에는 다음과 같은 세 가지 기본 구성 요소가 있습니다.

- 실제 작업을 수행하는 활동 메서드는 인터페이스에서 정의되고 관련 클래스에서 구현됩니다.
- ActivityWorker클래스는 액티비티 메서드와 Amazon SWF 간의 상호 작용을 관리합니다.
- 활동 호스트 애플리케이션에서는 활동 작업자를 등록 및 시작하고 정리 작업을 처리합니다.

이 단원에서는 활동 메서드를 다룹니다. 다른 두 클래스는 나중에 설명합니다.

HelloWorldWorkflow 에서 `GreeterActivities` 다음과 같이 액티비티 인터페이스를 정의합니다.

```
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);
}
```

이 인터페이스가 꼭 필요한 것은 아니지만 Java HelloWorld 애플리케이션용으로는 AWS Flow Framework 사용할 수 있습니다. 인터페이스 정의 자체는 변결되지 않았다는 점에 유의하십시오. 하지만 Java 주석에는 두 개를, @ActivityRegistrationOptions 인터페이스 정의에는 두 개를 AWS Flow Framework 적용해야 합니다. @활동 주석은 구성 정보를 제공하고 AWS Flow Framework Java용 주석 프로세서가 인터페이스 정의를 사용하여 활동 클라이언트 클래스를 생성하도록 지시합니다. 이에 대해서는 나중에 설명합니다.

`@ActivityRegistrationOptions`활동의 동작을 구성하는 데 사용되는 여러 개의 명명된 값이 있습니다. HelloWorldWorkflow 두 개의 타임아웃을 지정합니다.

- `defaultTaskScheduleToStartTimeoutSeconds`에서는 활동 작업에서 작업이 얼마나 오래 대기 상태에 있을 수 있는지 지정하며 300초(5분)로 설정됩니다.
- `defaultTaskStartToCloseTimeoutSeconds`에서는 활동에서 작업을 수행하는 데 걸릴 수 있는 최대 시간을 지정하며 10초로 설정됩니다.

이 제한 시간을 통해 활동에서는 자체 작업을 적당한 시간 내에 완료할 수 있게 됩니다. 두 제한 시간 중 어느 하나라도 초과되면 프레임워크에서는 오류를 생성하고 워크플로 작업자는 이 문제를 어떻게 처리할지 결정해야 합니다. 그러한 오류를 처리하는 방법에 관한 설명은 오류 처리 단원을 참조하십시오.

`@Activities`에는 몇 가지 값이 있지만 일반적으로 활동의 버전 번호를 지정할 뿐입니다. 이를 통해 사용자는 활동 구현을 여러 세대에 걸쳐 추적할 수 있습니다. Amazon SWF에서 이 번호를 등록한 후에 `@ActivityRegistrationOptions` 값 변경을 비롯해 활동 인터페이스를 변경하는 경우에는 새 버전 번호를 사용해야 합니다.

HelloWorldWorkflow 에서 `GreeterActivitiesImpl` 다음과 같이 활동 메서드를 구현합니다.

```
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);
   }
}
```

코드가 HelloWorld 구현과 동일하다는 점에 유의하세요. 기본적으로 AWS Flow Framework 액티비티는 일부 코드를 실행하고 결과를 반환하는 메서드일 뿐입니다. 표준 애플리케이션과 Amazon SWF 워크플로 애플리케이션 간의 차이는 워크플로에서 활동을 실행하는 방식, 활동이 실행되는 위치 및 결과가 워크플로 작업자에게 반환되는 방식에 있습니다.

HelloWorldWorkflow 워크플로 워커

Amazon SWF 워크플로 작업자에는 세 가지 기본 구성 요소가 있습니다.

- 워크플로 관련 작업을 수행하는 클래스인 워크플로 구현
- 기본적으로 활동 클래스의 프록시이며 워크플로 구현에서 활동 메서드를 비동기식으로 실행하는 데 사용하는 활동 클라이언트 클래스
- 워크플로와 Amazon SWF 간의 상호 작용을 관리하는 WorkflowWorker클래스입니다.

이 단원에서는 워크플로 구현 및 활동 클라이언트를 다룹니다. `WorkflowWorker` 클래스는 나중에 설명합니다.

HelloWorldWorkflow 에서 `GreeterWorkflow` 다음과 같이 워크플로 인터페이스를 정의합니다.

```
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();
}
```

이 인터페이스도 Java 응용 프로그램에 꼭 필요한 것은 HelloWorld 아니지만 Java 응용 프로그램에는 AWS Flow Framework 필수적입니다. Java 주석에는 두 개를, @워크플로 워크플로 인터페이스 정의에는 두 개를 AWS Flow Framework 적용해야 합니다. @WorkflowRegistrationOptions 주석은 구성 정보를 제공하고 나중에 설명하겠지만 AWS Flow Framework Java용 주석 프로세서가 인터페이스를 기반으로 워크플로 클라이언트 클래스를 생성하도록 지시합니다.

`@Workflow`에는 DataConverter라는 선택적 매개 변수가 하나 있는데, 이 매개 변수는 종종 기본값과 함께 사용되는데 NullDataConverter, 이는 사용해야 함을 나타냅니다. JsonDataConverter

`@WorkflowRegistrationOptions`에는 워크플로 작업자를 구성하는 데 사용할 수 있는 여러 가지 선택적 파라미터도 있습니다. 여기서는 워크플로를 실행할 수 있는 시간을 지정하는 `defaultExecutionStartToCloseTimeoutSeconds`를 3600초(1시간)로 설정합니다.

`GreeterWorkflow`인터페이스 정의는 주석이라는 한 가지 중요한 HelloWorld 측면에서 다릅니다. @실행 워크플로 인터페이스에서는 워크플로 시작자와 같이 애플리케이션에서 호출할 수 있고 각기 특별한 역할이 있는 소수의 메서드로 제한된 메서드를 지정합니다. 프레임워크에서는 워크플로 인터페이스 메서드에 이름 또는 파라미터 목록을 지정하지 않습니다. 따라서 워크플로에 적합한 이름과 파라미터 목록을 사용하고 AWS Flow Framework for Java 주석을 첨부하여 메서드의 역할을 파악해야 합니다.

`@Execute`의 용도는 다음 두 가지입니다.

- `greet`를 워크플로의 진입점, 즉 워크플로 시작자가 워크플로를 시작하기 위해 호출하는 메서드로 식별합니다. 일반적으로 진입점에서는 파라미터를 한 개 이상 받아들이는데, 이를 통해 시작자는 워크플로를 초기화할 수 있습니다. 그러나 이 예시에서는 초기화할 필요가 없습니다.
- 워크플로의 버전 번호를 지정합니다. 이를 통해 사용자는 활동 구현을 여러 세대에 걸쳐 추적할 수 있습니다. Amazon SWF에서 이 번호를 등록한 후에 제한 시간 값 변경을 비롯해 워크플로 인터페이스를 변경하려면 새 버전 번호를 사용해야 합니다.

워크플로 인터페이스에 포함될 수 있는 기타 메서드에 대한 자세한 내용은 워크플로 및 활동 계약 단원을 참조하십시오.

HelloWorldWorkflow 다음과 같이 워크플로를 구현합니다. `GreeterWorkflowImpl`

```
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);
   }
}
```

코드는 과 HelloWorld 비슷하지만 두 가지 중요한 차이점이 있습니다.

- `GreeterWorkflowImpl`에서는 `GreeterActivitiesImpl` 대신에 활동 클라이언트인 `GreeterActivitiesClientImpl`의 인스턴스를 생성하고 클라이언트 객체에 있는 메서드를 호출하여 활동을 실행합니다.
- 이름과 인사 활동에서는 `String` 객체 대신에 `Promise<String>` 객체를 반환합니다.

HelloWorld 단일 프로세스로 로컬에서 실행되는 표준 Java 응용 프로그램이므로 간단히 인스턴스를 만들고 메서드를 순서대로 호출한 다음 한 액티비티의 반환 값을 다음 액티비티로 전달하여 워크플로 토폴로지를 구현할 `GreeterWorkflowImpl` 수 있습니다. `GreeterActivitiesImpl` Amazon SWF 워크플로에서 활동의 작업은 여전히 `GreeterActivitiesImpl`의 활동 메서드에서 수행합니다. 그러나 이 메서드는 워크플로와 동일한 프로세스로 실행될 필요가 없고 심지어는 동일 시스템에서 실행되지 않아도 됩니다. 워크플로에서는 활동을 비동기식으로 실행해야 합니다. 이러한 요구 사항에 따라 다음 문제가 제기됩니다.

- 다른 프로세스(다른 시스템일 수도 있음)에서 실행 중일 수 있는 활동 메서드를 실행하는 방법
- 활동 메서드를 비동기식으로 실행하는 방법
- 활동의 입력 및 반환 값을 관리하는 방법. 예를 들어 활동 A의 반환 값이 활동 B에게는 입력인 경우에는 활동 A가 완료될 때까지 활동 B가 실행되지 않게 해야 합니다.

활동 클라이언트 및 `Promise<T>`와 결합된 친숙한 Java 흐름 제어를 사용하여 애플리케이션의 제어를 통해 다양한 워크플로 토폴로지를 구현할 수 있습니다.

활동 클라이언트

GreeterActivitiesClientImpl은 기본적으로 GreeterActivitiesImpl의 프록시로서, 워크플로 구현에서 GreeterActivitiesImpl 메서드를 비동기식으로 실행할 수 있게 해줍니다.

GreeterActivitiesClient 및 GreeterActivitiesClientImpl 클래스는 사용자의 GreeterActivities 클래스에 첨부된 주석의 정보를 사용하여 자동으로 생성됩니다. 따라서 사용자는 이를 직접 구현하지 않아도 됩니다.

> **Note**
>
> Eclipse에서는 사용자가 프로젝트를 저장할 때 이 클래스를 생성합니다. 생성된 코드는 프로젝트 디렉터리의 .apt_generated 하위 디렉터리에서 볼 수 있습니다. GreeterWorkflowImpl 클래스의 컴파일 오류를 방지하려면 .apt_generated 디렉터리를 Java Build Path 대화 상자의 Order and Export 탭 상단으로 이동하는 것이 좋습니다.

워크플로 작업자는 해당되는 클라이언트 메서드를 호출하여 활동을 실행합니다. 이 메서드는 비동기식이며 Promise<T> 객체를 즉시 반환합니다. 여기에서 T는 활동의 반환 유형입니다. 반환된 Promise<T> 객체는 기본적으로 활동 메서드에서 최종적으로 반환할 값의 자리 표시자입니다.

- 활동 클라이언트 메서드에서 값을 반환하면 Promise<T> 객체는 초기에는 준비되지 않은 상태인데, 이는 아직 객체에서 유효한 반환 값을 표시하지 않고 있음을 뜻합니다.
- 상응하는 활동 메서드가 작업을 완료하고 값을 반환하면 프레임워크에서는 이 반환 값을 Promise<T> 객체에 할당하고 이 객체를 준비 상태로 둡니다.

Promise<T> 유형

Promise<T> 객체의 기본 목적은 비동기 구성 요소 간 데이터 흐름을 관리하고 이 구성 요소가 실행되는 시점을 제어하는 것입니다. 이 객체는 애플리케이션에서 동기화를 명시적으로 관리하거나 타이머와 같은 메커니즘에 의존해야 할 필요를 없애줌으로써 비동기 구성 요소가 조기에 실행되지 않도록 합니다. 활동 클라이언트 메서드를 호출하면 이 메서드에서는 즉시 값을 반환하지만 프레임워크에서는 입력 Promise<T> 객체 중 어느 하나라도 완료되어 유효한 데이터를 표시할 때까지 해당 활동 메서드 실행을 연기합니다.

GreeterWorkflowImpl의 관점에서 보면 세 가지 활동 클라이언트 메서드 모두 즉시 값을 반환합니다. GreeterActivitiesImpl의 관점에서 보면 프레임워크에서는 name이 완료될 때까지 getGreeting을 호출하지 않고 getGreeting이 완료될 때까지 say를 호출하지 않습니다.

HelloWorldWorkflow는 Promise<T>를 사용하여 한 활동에서 그다음 활동으로 데이터를 전달함으로써 활동 메서드에서 잘못된 데이터를 사용하지 않도록 할 뿐 아니라 활동에서 워크플로 토폴로지를 실행하고 암묵적으로 정의하는 시점을 제어합니다. 각 활동의 Promise<T> 반환 값을 그다음 활동에 전달하려면 활동이 순차적으로 실행되어 앞서 설명한 선형 토폴로지를 정의해야 합니다. Java의 경우 복잡한 토폴로지를 정의하는 데 특별한 모델링 코드를 사용할 필요가 없습니다. 표준 Java 흐름 제어 및 AWS Flow Framework 기능만 있으면 됩니다. Promise<T> 간단한 병렬 토폴로지를 구현하는 방법을 보여주는 예시는 HelloWorldWorkflowParallel 활동 작업자 단원을 참조하십시오.

> **Note**
>
> say와 같은 활동 메서드에서 값을 반환하지 않으면 상응하는 클라이언트 메서드에서 Promise<Void> 객체를 반환합니다. 객체에서는 데이터를 표시하지 않고 초기에는 준비되지 않은 상태였다가 활동이 완료되면 준비 상태가 됩니다. 따라서 Promise<Void> 객체를 다른 활동 클라이언트 메서드에 전달하여 이 메서드에서 원본 활동이 완료될 때까지 실행을 연기하도록 할 수 있습니다.

Promise<T>를 통해 워크플로 구현에서는 활동 클라이언트 메서드와 그 반환 값을 비동기식 메서드와 흡사하게 사용합니다. 그러나 Promise<T> 객체의 값에 액세스할 때는 주의해야 합니다. Java Future<T> 유형과 달리 이 프레임워크에서는 애플리케이션이 아닌 Promise<T>를 위해 동기화를 처리합니다. Promise<T>.get을 직접적으로 호출하였는데 객체가 준비 상태가 아니면 get에서 예외가 발생합니다. HelloWorldWorkflow에서는 Promise<T> 객체에 직접 액세스하지 않고 객체를 한 활동에서 그다음 활동으로 전달하기만 합니다. 객체가 준비 상태가 되면 프레임워크에서는 값을 추출하여 이를 활동 메서드에 표준 유형으로 전달합니다.

Promise<T> 객체는 비동기식 코드를 통해서만 액세스해야 합니다. 이때 프레임워크에서는 이 객체가 준비 상태가 되어 유효한 값을 표시하도록 보장합니다. HelloWorldWorkflow에서는 Promise<T> 객체를 활동 클라이언트 메서드로만 전달하여 이 문제를 처리합니다. 활동과 매우 유사하게 동작하는 비동기 워크플로 메서드에 객체를 전달하여 워크플로 구현에서 Promise<T> 객체 값에 액세스할 수 있습니다. 예는 HelloWorldWorkflowAsync 애플리케이션를 참조하세요.

HelloWorldWorkflow 워크플로우 및 활동 구현

워크플로우 및 활동 구현에는 관련 작업자 클래스가 ActivityWorker있으며 WorkflowWorker, 이 클래스에서는 작업에 대한 적절한 Amazon SWF 작업 목록을 폴링하고 각 작업에 대해 적절한 메서드를 실행하며 데이터 흐름을 관리하여 Amazon SWF와 활동 및 워크플로 구현 간 통신을 처리합니다. 자세한 내용은 AWS Flow Framework 기본 개념: 애플리케이션 구조을 참조하세요.

활동 및 워크플로 구현을 상응하는 작업자 객체에 연결하려면 다음 작업을 수행하는 작업자 애플리케이션을 한 개 이상 구현해야 합니다.

- Amazon SWF에 워크플로 또는 활동 등록
- 작업자 객체를 생성하여 이를 워크플로 또는 활동 작업자 구현에 연결
- 작업자 객체에게 Amazon SWF와의 통신을 시작하도록 지시

워크플로 및 활동을 별도 프로세스로 실행하고 싶다면 별도 워크플로 및 활동 작업자 호스트를 구현해야 합니다. 예는 HelloWorldWorkflowDistributed 애플리케이션를 참조하세요. 단순화를 위해 다음과 같이 동일한 프로세스에서 활동과 워크플로 작업자를 실행하는 단일 작업자 호스트를 HelloWorldWorkflow 구현합니다.

```
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();
    }
}
```

GreeterWorker에는 해당 클래스가 없으므로 프로젝트에 이름이 지정된 GreeterWorker Java 클래스를 추가하고 예제 코드를 해당 파일에 복사해야 합니다. HelloWorld

첫 번째 단계는 기본 Amazon SWF 서비스 메서드를 호출하는 AmazonSimpleWorkflowClient객체를 만들고 구성하는 것입니다. 이렇게 하려면 GreeterWorker에서 다음 작업을 수행해야 합니다.

1. ClientConfiguration객체를 생성하고 소켓 제한 시간을 70초로 지정합니다. 이 값을 통해 수립된 개방 연결을 통해 데이터가 전송될 때까지 기다리는 시간을 지정합니다. 이 시간이 지나면 소켓이 닫힙니다.
2. AWS 계정을 식별하는 Basic AWSCredentials 객체를 만들고 생성자에 계정 키를 전달합니다. 편의상, 또한 코드에서 이 키를 일반 텍스트로 노출하지 않도록 하기 위해 키는 환경 변수로 저장됩니다.
3. 워크플로를 나타내는 AmazonSimpleWorkflowClient객체를 만들고, BasicAWSCredentials 및 ClientConfiguration 객체를 생성자에 전달합니다.
4. 클라이언트 객체의 서비스 엔드포인트 URL을 설정합니다. Amazon SWF는 현재 모든 AWS 지역에서 사용할 수 있습니다.

편의상 GreeterWorker에서는 두 가지 문자열 상수를 정의합니다.

- domainAmazon SWF 계정을 설정할 때 만든 워크플로의 아마존 SWF 도메인 이름입니다. HelloWorldWorkflow워크플로를 "" helloWorldWalkthrough 도메인에서 실행한다고 가정합니다.
- taskListToPoll은 Amazon SWF에서 워크플로와 활동 작업자 간 통신을 관리하는 데 사용하는 작업 목록의 이름입니다. 사용자는 이 이름을 원하는 문자열로 자유롭게 설정할 수 있습니다. HelloWorldWorkflow 워크플로 및 활동 작업 목록 모두에 HelloWorldList ""를 사용합니다. 백그라운드에서는 이름이 여러 가지 네임스페이스가 되므로 두 작업 목록은 서로 구분됩니다.

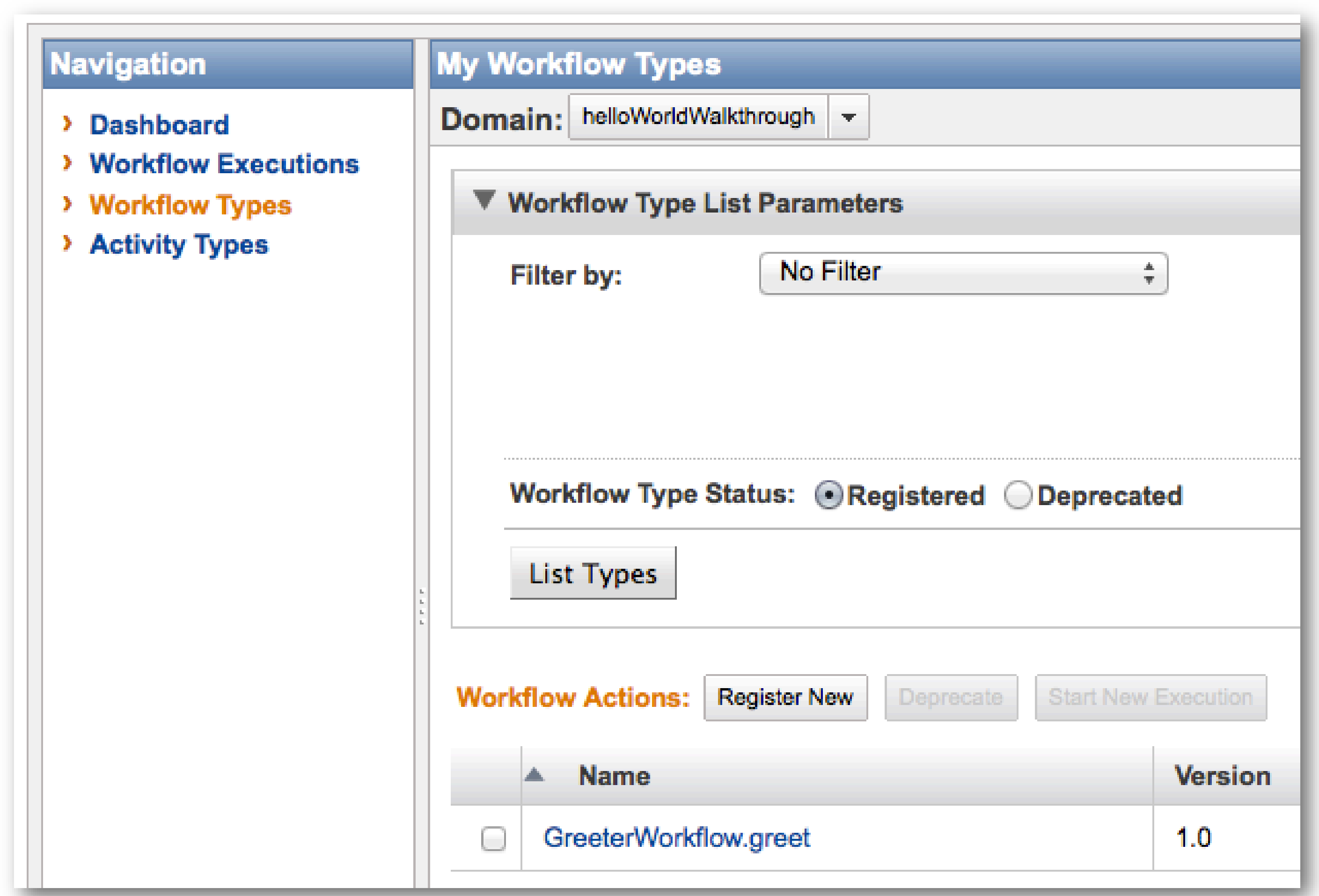
GreeterWorker문자열 상수와 객체를 사용하여 작업자 AmazonSimpleWorkflowClient객체를 생성하며, 이 작업자 객체는 활동과 작업자 구현 및 Amazon SWF 간의 상호 작용을 관리합니다. 특히 작업자 객체에서는 작업에 대해 적절한 작업 목록을 폴링하는 작업을 처리합니다.

GreeterWorker에서는 ActivityWorker 객체를 생성하고 새 클래스 인스턴스를 추가하여 이 객체에서 GreeterActivitiesImpl을 처리하도록 구성합니다. 그러면 GreeterWorker에서

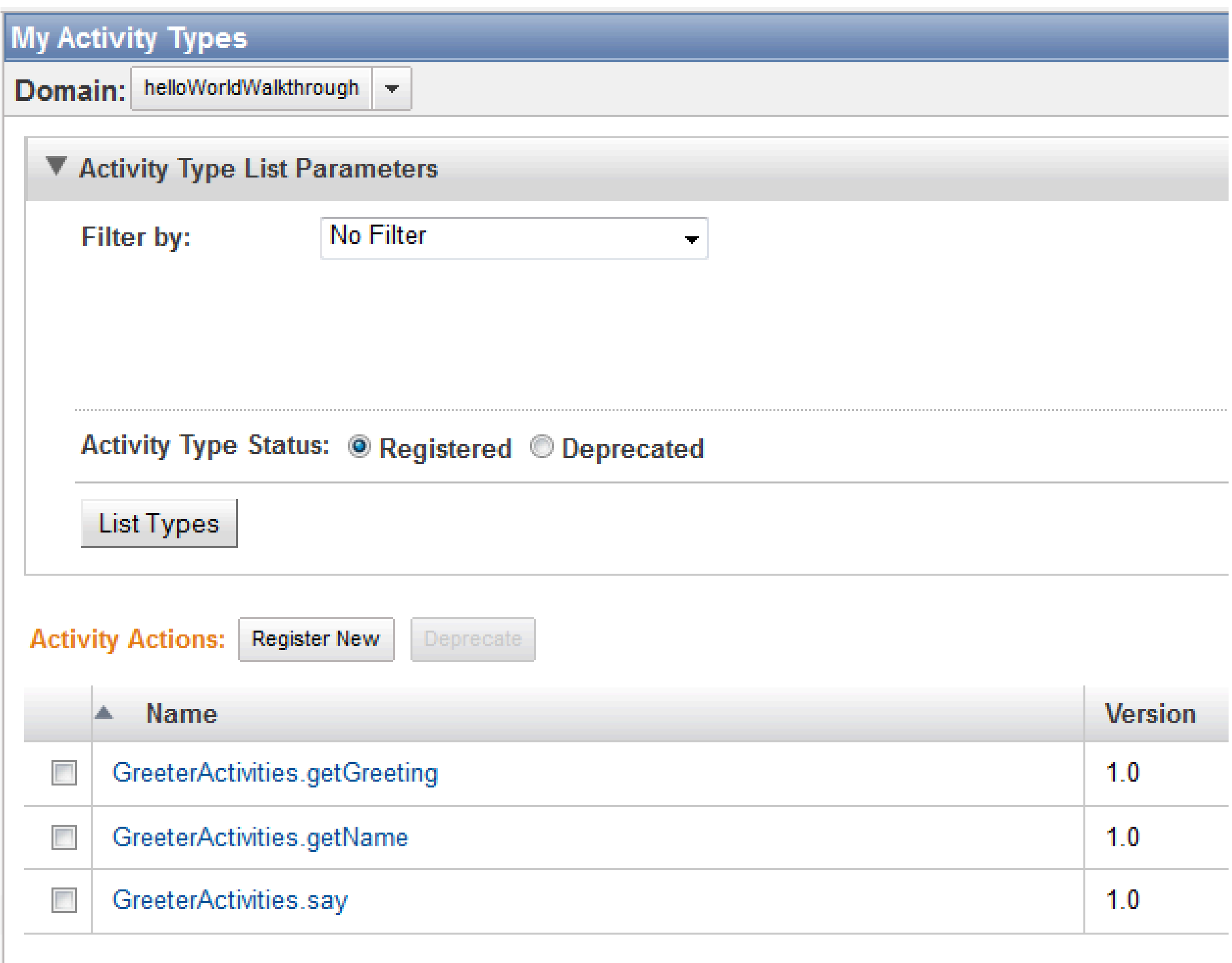
ActivityWorker 객체의 start 메서드를 호출하고, 이를 통해 객체에게 지정된 활동 작업 목록을 폴링하도록 지시합니다.

GreeterWorker에서는 WorkflowWorker 객체를 생성하고 GreeterWorkflowImpl.class라는 새 클래스 파일 이름을 추가하여 이 객체에서 GreeterWorkflowImpl을 처리하도록 구성합니다. 그러면 이 객체에서는 WorkflowWorker 객체의 start 메서드를 호출하고, 이를 통해 객체에게 지정된 워크플로 작업 목록을 폴링하도록 지시합니다.

이 시점에서 사용자는 GreeterWorker를 성공적으로 실행할 수 있습니다. 이를 실행하면 Amazon SWF에 워크플로 및 활동을 등록하고 각자에게 해당되는 작업 목록을 폴링하는 작업자 객체를 시작합니다. 이를 확인하려면 GreeterWorker를 실행하고 Amazon SWF 콘솔로 이동하여 도메인 목록에서 helloWorldWalkthrough를 선택합니다. 탐색 창에서 워크플로 유형을 선택하면 GreeterWorkflow.greet이 표시됩니다.



[Activity Types]를 선택하면 다음과 같이 GreeterActivities 메서드가 표시됩니다.



그러나 [Workflow Executions]를 선택하면 활성화된 실행이 표시되지 않습니다. 워크플로 및 활동 작업자가 작업에 대해 폴링 중이지만 아직 우리는 워크플로 실행을 시작한 것은 아닙니다.

HelloWorldWorkflow 스타터

퍼즐의 마지막 조각은 워크플로 실행을 시작하는 애플리케이션인 워크플로 시작자를 구현하는 것입니다. 실행 상태는 Amazon SWF에서 저장하므로 사용자는 그 내역과 실행 상태를 볼 수 있습니다. HelloWorldWorkflow 다음과 같이 `GreeterMain` 클래스를 수정하여 워크플로 스타터를 구현합니다.

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

public class GreeterMain {
```

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

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

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

      String domain = "helloWorldWalkthrough";

      GreeterWorkflowClientExternalFactory factory = new
  GreeterWorkflowClientExternalFactoryImpl(service, domain);
      GreeterWorkflowClientExternal greeter = factory.getClient("someID");
      greeter.greet();
    }
}
```

`GreeterMain`에서는 `GreeterWorker`와 동일한 코드를 사용하여 `AmazonSimpleWorkflowClient` 객체를 생성합니다. 그런 다음 `GreeterWorkflowClientImpl`에서 생성된 활동 클라이언트가 활동 메서드에 대해 프록시 역할을 수행하는 것과 흡사한 방식으로 워크플로에 대해 프록시 역할을 수행하는 `GreeterWorkflowClientExternal` 객체를 생성합니다. 사용자는 new를 사용해 워크플로 클라이언트 객체를 생성하는 대신에 다음 작업을 수행해야 합니다.

1. 외부 클라이언트 팩토리 객체를 생성하고 `AmazonSimpleWorkflowClient` 객체와 Amazon SWF 도메인 이름을 생성자에게 전달합니다. 클라이언트 팩토리 객체는 프레임워크의 주석 처리기에 의해 생성되며, 주석 처리기는 워크플로 인터페이스 이름에 ClientExternalFactoryImpl ""를 추가하기만 하면 객체 이름이 만들어집니다.
2. 팩토리 개체의 메서드를 호출하여 외부 클라이언트 개체를 만듭니다. 이 `getClient` 메서드는 워크플로 인터페이스 이름에 ClientExternal ""를 추가하여 개체 이름을 만듭니다. 사용자는 Amazon SWF에서 이러한 워크플로 인스턴스를 식별하는 데 사용할 문자열을 `getClient`에 선택적으로 전달할 수 있습니다. 또는 Amazon SWF에서는 생성된 GUID를 사용하여 워크플로 인스턴스를 표시합니다.

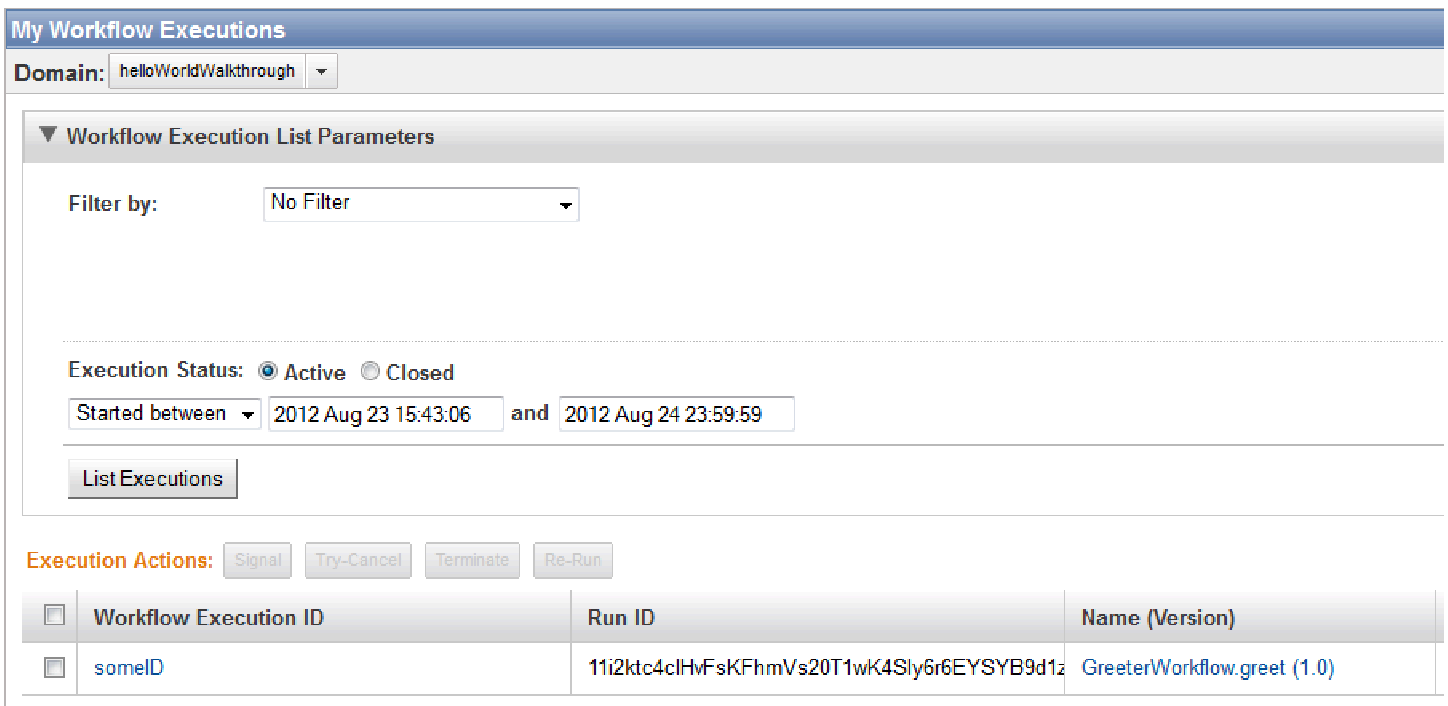
팩토리에서 반환된 클라이언트에서는 getClient 메서드로 전달된 문자열로 이름이 지정된 워크플로만 생성합니다(팩토리에서 반환된 클라이언트는 이미 Amazon SWF에 상태가 있음). 다른 ID로 워크플로를 실행하려면 팩토리로 돌아가 그 ID가 지정된 새 클라이언트를 생성해야 합니다.

`greet()`이 `@Execute` 주석이 지정된 메서드였으므로 워크플로 클라이언트에서는 워크플로를 시작하기 위해 `GreeterMain`에서 호출하는 `greet` 메서드를 노출합니다.

Note

또한 주석 프로세서에서는 하위 워크플로를 생성하는 데 사용되는 내부 클라이언트 팩토리 객체를 생성합니다. 자세한 내용은 하위 워크플로 실행 섹션을 참조하세요.

`GreeterWorker`가 여전히 실행 중인 경우 이를 잠시 종료하고 `GreeterMain`을 실행합니다. 이제 활성화된 워크플로 실행의 Amazon SWF 콘솔 목록에 다음과 같은 someID가 표시될 것입니다.



[`someID`]를 선택하고 [Events] 탭을 선택하면 다음과 같이 이벤트가 표시됩니다.

Workflow Execution: someID

Domain: helloWorldWalkthrough

Summary | **Events** | Activities

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

> **Note**
>
> 앞서 `GreeterWorker`를 시작하여 아직 실행 중이라면 사유를 짧막하게 설명한 더 긴 이벤트 목록이 표시됩니다. `GreeterWorker` 중지 후 `GreaterMain`을 다시 실행합니다.

[Events]에는 다음과 같이 이벤트 두 건만 표시됩니다.

- `WorkflowExecutionStarted`에서는 워크플로에서 실행이 시작되었음을 나타냅니다.
- `DecisionTaskScheduled`에서는 Amazon SWF가 첫 번째 결정 작업을 대기열에 추가했음을 나타냅니다.

첫 번째 결정 작업에서 워크플로가 차단된 이유는 워크플로가 `GreeterMain` 및 `GreeterWorker` 두 애플리케이션에 분산되었기 때문입니다. `GreeterMain`에서는 워크플로 실행을 시작하였으나 `GreeterWorker`는 실행되고 있지 않으므로 작업자는 목록 폴링 및 작업 수행을 하지 않고 있습니다. 두 애플리케이션 중 하나를 실행할 수 있지만 워크플로 실행이 첫 번째 결정 작업 단계를 넘어 계속 진행되도록 하려면 둘 다 필요합니다. 이제 `GreeterWorker`를 실행하면 워크플로 및 활동 작업자에서는 폴링이 시작되고 여러 작업이 신속하게 완료됩니다. 이제 `Events` 탭을 확인하면 이벤트의 첫 번째 배치가 표시됩니다.

Workflow Execution: someID

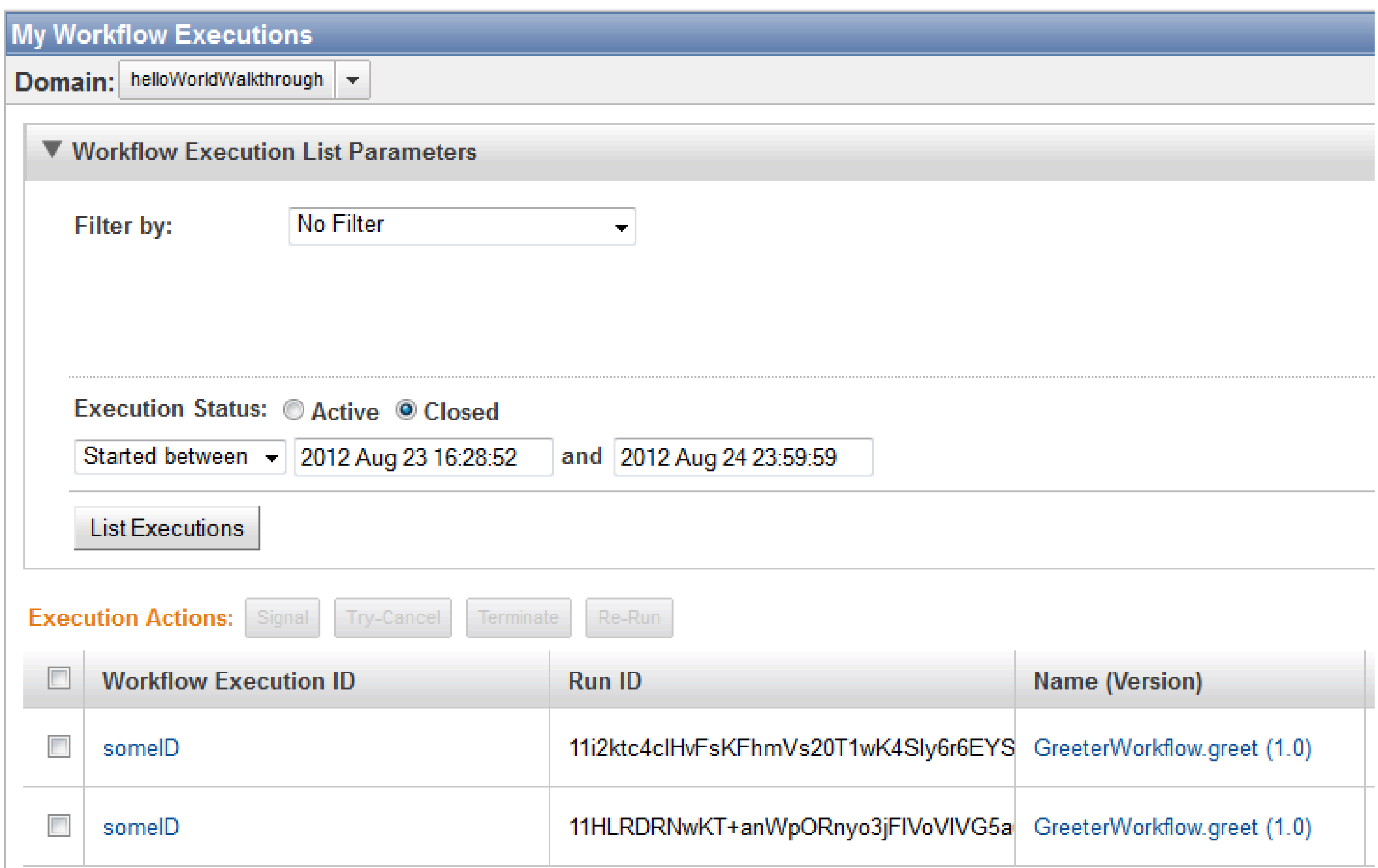
Domain: helloWorldWalkthrough

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

개별 이벤트를 선택하면 추가 정보를 얻을 수 있습니다. 확인을 마쳤을 때쯤에는 워크플로가 "Hello World!"를 콘솔에 출력했어야 합니다.

워크플로는 완료된 후에는 활성 실행 목록에 더 이상 표시되지 않습니다. 그러나 이를 다시 보고 싶다면 [Closed] 실행 상태 버튼을 선택한 후 [List Executions]를 선택하면 됩니다. 이렇게 하면 도메인을 생성할 때 지정한 보존 시간을 초과하지 않은, 지정 도메인(`helloWorldWalkthrough`) 내의 완료된 워크플로 인스턴스가 모두 표시됩니다.



각 워크플로 인스턴스에는 고유한 [Run ID] 값이 있다는 점에 유의하십시오. 다른 워크플로 인스턴스에 대해 동일한 실행 ID를 사용할 수 있습니다. 단, 한 번에 하나의 활성화된 실행에만 사용할 수 있습니다.

HelloWorldWorkflowAsync 애플리케이션

어떤 경우에는 활동을 사용하는 대신에 워크플로에서 특정 작업을 로컬로 수행하도록 하는 것이 더 좋습니다. 그러나 워크플로 작업에는 `Promise<T>` 객체에서 표시하는 값을 처리하는 과정이 수반되는 경우가 많습니다. `Promise<T>` 객체를 동기식 워크플로 메서드에 전달하면 이 메서드가 즉시 실행되지만 객체가 준비 상태가 될 때까지는 `Promise<T>` 객체의 값에 액세스할 수 없습니다. `Promise<T>.isReady`는 `true`를 반환할 때까지 폴링할 수 있지만, 이렇게 하는 것은 비능률적이므로 메서드가 장시간 차단할 수 있습니다. 더 좋은 방법은 비동기식 메서드를 사용하는 것입니다.

비동기 메서드는 표준 메서드(워크플로 구현 클래스의 멤버로 사용)와 매우 유사하게 구현되며 워크플로 구현 컨텍스트에서 실행됩니다. 사용자는 `@Asynchronous` 주석을 적용하여 워크플로 구현을 비동기식 메서드로 지정합니다. 이를 통해 프레임워크에게 워크플로 구현을 활동과 흡사하게 취급하도록 지시합니다.

- 워크플로 구현에서 비동기식 메서드를 호출하면 비동기식 메서드에서 즉시 반환합니다. 비동기식 메서드에서는 일반적으로 이 메서드가 완료될 때 준비 상태가 되는 `Promise<T>` 객체를 반환합니다.
- 비동기식 메서드에 `Promise<T>` 객체를 한 개 이상 전달하면 이 메서드는 모든 입력 객체가 준비 상태가 될 때까지 실행을 연기합니다. 따라서 비동기식 메서드에서는 입력 `Promise<T>` 값에 액세스할 때 예외가 반환될 위험이 없습니다.

> **Note**
> AWS Flow Framework for Java에서 워크플로를 실행하는 방식으로 인해 비동기식 메서드는 일반적으로 여러 번 실행되므로 사용자는 빠른 저 오버헤드 작업에만 비동기식 메서드를 사용해야 합니다. 활동은 대규모 컴퓨팅과 같이 시간이 오래 걸리는 작업을 수행할 때 사용해야 합니다. 자세한 내용은 AWS Flow Framework 기본 개념: 분산 실행 섹션을 참조하세요.

이 주제에서는 활동 중 하나를 비동기식 메서드로 대치하는 HelloWorldWorkflow의 수정 버전인 HelloWorldWorkflowAsync를 연습합니다. 애플리케이션을 구현하려면 프로젝트 디렉터리에 helloWorld.HelloWorldWorkflow 패키지의 사본을 생성하고 이 사본의 이름을 helloWorld.HelloWorldWorkflowAsync로 지정합니다.

> **Note**
> 이 주제는 HelloWorld 애플리케이션 및 HelloWorldWorkflow 애플리케이션 주제에 있는 개념과 파일을 기반으로 합니다. 계속하기 전에 그러한 주제에 있는 파일과 개념을 익히십시오.

다음 단원에서는 비동기식 메서드를 사용하기 위해 원본 HelloWorldWorkflow 코드를 수정하는 방법을 설명합니다.

HelloWorldWorkflowAsync 활동 구현

HelloWorldWorkflowAsync에서는 다음과 같이 `GreeterActivities`에 활동 작업자 인터페이스를 구현합니다.

```
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);
}
```

이 인터페이스는 다음 사항을 제외하면 HelloWorldWorkflow에서 사용하는 것과 비슷합니다.

- `getGreeting` 활동을 생략합니다. 이 작업은 현재 비동기식 메서드에서 처리합니다.
- 버전 번호가 2.0으로 설정됩니다. Amazon SWF에 활동 인터페이스를 등록한 후에는 버전 번호를 바꾸지 않는 한 수정할 수 없습니다.

나머지 활동 메서드 구현은 HelloWorldWorkflow와 동일합니다. `GreeterActivitiesImpl`에서 `getGreeting`만 삭제합니다.

HelloWorldWorkflowAsync 워크플로 구현

HelloWorldWorkflowAsync에서는 워크플로 인터페이스를 다음과 같이 정의합니다.

```
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();
}
```

이 인터페이스는 새 버전 번호 외에는 HelloWorldWorkflow와 동일합니다. 활동과 마찬가지로 등록된 워크플로를 변경하고 싶다면 버전을 변경해야 합니다.

HelloWorldWorkflowAsync에서는 워크플로를 다음과 같이 구현합니다.

```
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);
   }
}
```

HelloWorldWorkflowAsync에서는 `getGreeting` 활동을 `getGreeting` 비동기식 메서드로 대체하지만 `greet` 메서드는 다음과 같이 거의 같은 방식으로 작동합니다.

1. `getName` 활동을 실행하면 이름을 표시하는 `Promise<String>` 객체 `name`이 즉시 반환됩니다.
2. `getGreeting` 비동기식 메서드를 호출한 후 이 메서드에 `name` 객체를 전달합니다. `getGreeting`에서는 인사를 표시하는 `Promise<String>` 객체 `greeting`을 즉시 반환합니다.
3. `say` 활동을 실행한 후 이 활동에 `greeting` 객체를 전달합니다.
4. `getName`이 완료되면 `name`이 준비 상태가 되고 `getGreeting`에서는 자체 값을 사용해 인사를 구성합니다.
5. `getGreeting`이 완료되면 `greeting`이 준비 상태가 되고 `say`에서는 콘솔에 문자열을 출력합니다.

차이점은 인사에서 활동 클라이언트를 호출하여 `getGreeting` 활동을 실행하는 대신에 비동기식 `getGreeting` 메서드를 호출한다는 것입니다. 결과는 동일하지만, `getGreeting` 메서드는 `getGreeting` 활동과는 약간 다르게 작동합니다.

- 워크플로 작업자에서는 표준 함수 호출 의미론을 사용하여 `getGreeting`을 실행합니다. 그러나 활동의 비동기식 실행은 Amazon SWF에서 조정합니다.
- `getGreeting`은 워크플로 구현 프로세스 중에 실행됩니다.

- getGreeting에서는 String 객체가 아닌 Promise<String> 객체를 반환합니다. Promise이 보유한 문자열 값을 얻으려면 get() 메서드를 호출해야 합니다. 그러나 활동이 비동기식으로 실행 중이기 때문에 반환 값이 즉시 준비되지 않을 수 있습니다. 따라서 get()에서는 비동기식 메서드의 반환 값이 사용 가능해질 때까지 예외가 발생합니다.

  Promise 작동 방식에 대한 자세한 내용은 AWS Flow Framework 기본 개념: 활동과 워크플로 간 데이터 교환 단원을 참조하십시오.

getGreeting에서는 정적 Promise.asPromise 메서드에 인사 문자열을 전달하여 반환 값을 생성합니다. 이 메서드에서는 적절한 유형의 Promise<T> 객체를 생성한 후 값을 설정하여 이 값을 준비 상태에 입력합니다.

HelloWorldWorkflowAsync 워크플로 및 활동 호스트와 시작자

HelloWorldWorkflowAsync에서는 GreeterWorker를 워크플로 및 활동 구현에 대해 호스트 클래스로 구현합니다. 이것은 "HelloWorldAsyncList"로 설정된 taskListToPoll 이름 외에는 HelloWorldWorkflow 구현과 동일합니다.

```
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();
    }
}
```

HelloWorldWorkflowAsync에서는`GreeterMain`에 워크플로 시작자를 구현하는데, 이것은 HelloWorldWorkflow 구현과 동일합니다.

워크플로를 실행하려면 HelloWorldWorkflow와 마찬가지로 `GreeterWorker` 및 `GreeterMain`를 실행하십시오.

HelloWorldWorkflowDistributed 애플리케이션

HelloWorldWorkflow 및 HelloWorldWorkflowAsync의 경우 Amazon SWF가 워크플로 및 활동 구현 간 상호 작용을 조정하지만 로컬에서 단일 프로세스로 실행됩니다. `GreeterMain`은 별도 프로세스에 있지만 여전히 동일한 시스템에서 실행됩니다.

Amazon SWF의 주요 특징은 분산 애플리케이션을 지원한다는 것입니다. 예를 들어 사용자는 Amazon EC2 인스턴스에서 워크플로 작업자, 데이터 센터 컴퓨터에서 워크플로 시작자를, 그리고 클라이언트 데스크톱 컴퓨터에서 활동을 실행할 수 있습니다. 시스템마다 다른 활동을 실행하는 것도 가능합니다.

HelloWorldWorkflowDistributed 애플리케이션에서는 HelloWorldWorkflowAsync를 확장하여 두 시스템 및 세 가지 프로세스 전체에 걸쳐 애플리케이션을 분산합니다.

- 워크플로와 워크플로 시작자는 한 가지 시스템에서 별도 프로세스로 실행됩니다.
- 활동은 별도 시스템에서 실행됩니다.

애플리케이션을 구현하려면 프로젝트 디렉터리에 helloWorld.HelloWorldWorkflowAsync 패키지의 사본을 생성하고 이 사본의 이름을 helloWorld.HelloWorldWorkflowDistributed로 지정합니다. 다음 단원에서는 원본 HelloWorldWorkflowDistributedAsync 코드를 수정하여 두 시스템 및 세 가지 프로세스 전체에 걸쳐 애플리케이션을 분산하는 방법을 설명합니다.

별도 시스템에서 실행하기 위해 워크플로 또는 활동 구현을 변경하지 않아도 됩니다. 버전 번호도 변경할 필요가 없습니다. 또한 `GreeterMain`도 수정할 필요가 없습니다. 활동 및 워크플로 호스트만 변경하면 됩니다.

HelloWorldWorkflowAsync의 경우 단일 애플리케이션은 워크플로 및 활동 호스트의 역할을 합니다. 워크플로 및 활동 구현을 별도의 시스템에서 실행하려면 별도의 애플리케이션을 구현해야 합니다. 프로젝트에서 GreeterWorker를 삭제하고 새 클래스 파일 두 개, 즉 GreeterWorkflowWorker 및 GreeterActivitiesWorker를 추가합니다.

HelloWorldWorkflowDistributed에서는 다음과 같이 GreeterActivitiesWorker에서 활동 호스트를 구현합니다.

```
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();
   }
}
```

HelloWorldWorkflowDistributed에서는 다음과 같이 `GreeterWorkflowWorker`에서 워크플로 호스트를 구현합니다.

```
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();
   }
}
```

`GreeterActivitiesWorker`는 단지 `WorkflowWorker` 코드가 없는 `GreeterWorker`이고, `GreeterWorkflowWorker`는 단지 `ActivityWorker` 코드가 없는 `GreeterWorker`입니다.

워크플로 실행 방법:

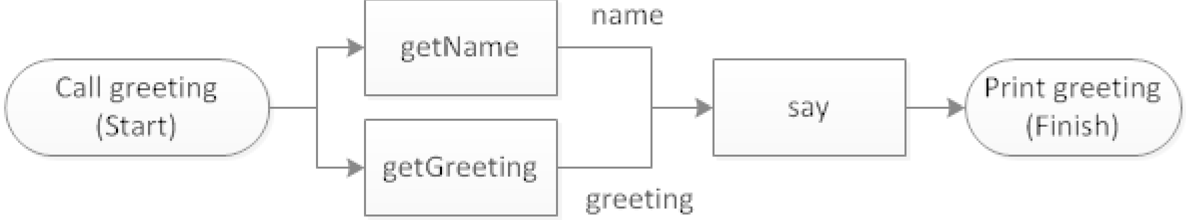
1. `GreeterActivitiesWorker`가 있는 실행 가능 JAR 파일을 진입점으로 생성합니다.
2. 1단계의 JAR 파일을 Java를 지원하는 운영 체제를 실행 중일 수 있는 다른 시스템으로 복사합니다.

3. 동일한 Amazon SWF 도메인에 액세스할 권한이 있는 AWS 보안 인증이 다른 시스템에서 사용 가능한지 확인합니다.
4. JAR 파일을 실행합니다.
5. 개발 시스템에서 이클립스를 사용하여 `GreeterWorkflowWorker` 및 `GreeterMain`을 실행합니다.

활동이 워크플로 작업자 및 워크플로 시작자가 아닌 시스템에서 실행된다는 사실 외에 워크플로는 HelloWorldAsync와 똑같은 방식으로 작동합니다. 그러나 콘솔에 "Hello World!"를 출력하는 `println` 직접 호출이 `say` 활동에 있으므로 출력은 활동 작업자를 실행 중인 시스템에 표시합니다.

HelloWorldWorkflowParallel 애플리케이션

Hello World!의 모든 이전 버전에서는 선형 워크플로 토폴로지를 사용합니다. 그러나 Amazon SWF는 선형 토폴로지에 국한되지 않습니다. 아래 그림에서와 같이 HelloWorldWorkflowParallel 애플리케이션은 병렬 토폴로지를 사용하는 HelloWorldWorkflow의 수정 버전입니다.



HelloWorldWorkflowParallel의 경우 `getName` 및 `getGreeting`은 병렬로 실행되며 각기 인사의 일부를 반환합니다. 그러면 `say`는 두 문자열을 인사로 병합한 후 이를 콘솔에 출력합니다.

애플리케이션을 구현하려면 프로젝트 디렉터리에 helloWorld.HelloWorldWorkflow 패키지의 사본을 생성하고 이 사본의 이름을 helloWorld.HelloWorldWorkflowParallel로 지정합니다. 다음 단원에서는 원본 HelloWorldWorkflow 코드를 수정하여 `getName` 및 `getGreeting`을 병렬로 실행하는 방법을 설명합니다.

HelloWorldWorkflowParallel 활동 작업자

HelloWorldWorkflowParallel 활동 인터페이스는 다음 예시와 같이 `GreeterActivities`에서 구현됩니다.

```
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);
}
```

이 인터페이스는 다음 사항을 제외하면 HelloWorldWorkflow와 유사합니다.

- `getGreeting`는 입력을 받아들이지 않고 인사 문자열만 반환합니다.
- `say`는 입력 문자열 두 개, 즉 인사 및 이름을 받아들입니다.
- 이 인터페이스에는 새 버전 번호가 있는데, 이 번호는 등록된 인터페이스를 변경할 때마다 필요합니다.

HelloWorldWorkflowParallel에서는 다음과 같이 `GreeterActivitiesImpl`에 활동을 구현합니다.

```
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` 및 `getGreeting`는 이제 인사 문자열의 절반만 반환합니다. `say`는 그 두 부분을 연결하여 완전한 구를 만든 후 이를 콘솔에 출력합니다.

HelloWorldWorkflowParallel 워크플로 작업자

HelloWorldWorkflowParallel 워크플로 인터페이스는 다음과 같이 `GreeterWorkflow`에 구현됩니다.

```
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();
}
```

이 클래스는 활동 작업자와 매칭하기 위해 버전 번호를 변경한 것 외에는 HelloWorldWorkflow 버전과 동일합니다.

워크플로는 `GreeterWorkflowImpl`에서 다음과 같이 구현됩니다.

```
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);
   }
}
```

한눈에 알 수 있듯이 이 구현은 HelloWorldWorkflow와 아주 유사하며, 세 가지 활동 클라이언트 메서드는 순차적으로 실행됩니다. 그러나 활동은 그렇지 않습니다.

- HelloWorldWorkflow는 `getGreeting`에 `name`을 전달하였습니다. `name`은 `Promise<T>` 객체였으므로 `getGreeting`에서는 `getName`이 완료될 때까지 활동 실행을 연기하였고, 이로 인해 두 활동은 순차적으로 실행되었습니다.

- HelloWorldWorkflowParallel은 입력 `getName` 또는 `getGreeting`을 전달하지 않습니다. 두 메서드 모두 실행을 연기하지 않고 연결된 활동 메서드에서는 즉시 병렬로 실행합니다.

`say` 활동은 `greeting` 및 `name` 모두를 입력 파라미터로 받아들입니다. 이 둘은 `Promise<T>` 객체이므로 `say`에서는 두 활동이 완료될 때까지 실행을 연기한 후 인사를 구성하여 출력합니다.

HelloWorldWorkflowParallel에서는 워크플로 토폴로지를 정의할 때 특수 모델링 코드를 사용하지 않습니다. 대신에 표준 Java 플로우 제어를 사용하고 `Promise<T>` 객체의 속성을 이용하여 묵시적으로 정의합니다. AWS Flow Framework for Java 애플리케이션에서는 기존 Java 제어 플로우 구성과 함께 `Promise<T>` 객체를 사용하는 간단한 방법으로 복잡한 토폴로지를 구현할 수 있습니다.

HelloWorldWorkflowParallel 워크플로 및 활동 호스트와 시작자

HelloWorldWorkflowParallel에서는 `GreeterWorker`를 워크플로 및 활동 구현에 대해 호스트 클래스로 구현합니다. 이것은 "HelloWorldParallelList"로 설정된 `taskListToPoll` 이름 외에는 HelloWorldWorkflow 구현과 동일합니다.

`HelloWorldWorkflowParallel`에서는 `GreeterMain`에 워크플로 시작자를 구현하는데, 이는 HelloWorldWorkflow 구현과 동일합니다.

워크플로를 실행하려면 `HelloWorldWorkflow`와 마찬가지로 `GreeterWorker` 및 `GreeterMain`을 실행하십시오.

AWS Flow Framework 자바의 작동 방식

AWS Flow Framework for Java를 Amazon SWF와 함께 사용하면 확장 가능하고 내결함성이 있는 애플리케이션을 쉽게 만들어 장기간 실행되거나 원격에서 실행되거나 둘 다일 수 있는 비동기 작업을 수행할 수 있습니다. “Hello World!” 의 예제에서는 를 사용하여 기본 워크플로 애플리케이션을 구현하는 방법에 대한 기본 사항을 AWS Flow Framework Java용이란 무엇입니까? 소개했습니다. AWS Flow Framework 이 섹션에서는 AWS Flow Framework 애플리케이션 작동 방식에 대한 개념적 정보를 제공합니다. 첫 번째 섹션에서는 응용 프로그램의 기본 구조를 요약하고 나머지 섹션에서는 AWS Flow Framework 응용 프로그램 작동 방식에 AWS Flow Framework 대해 자세히 설명합니다.

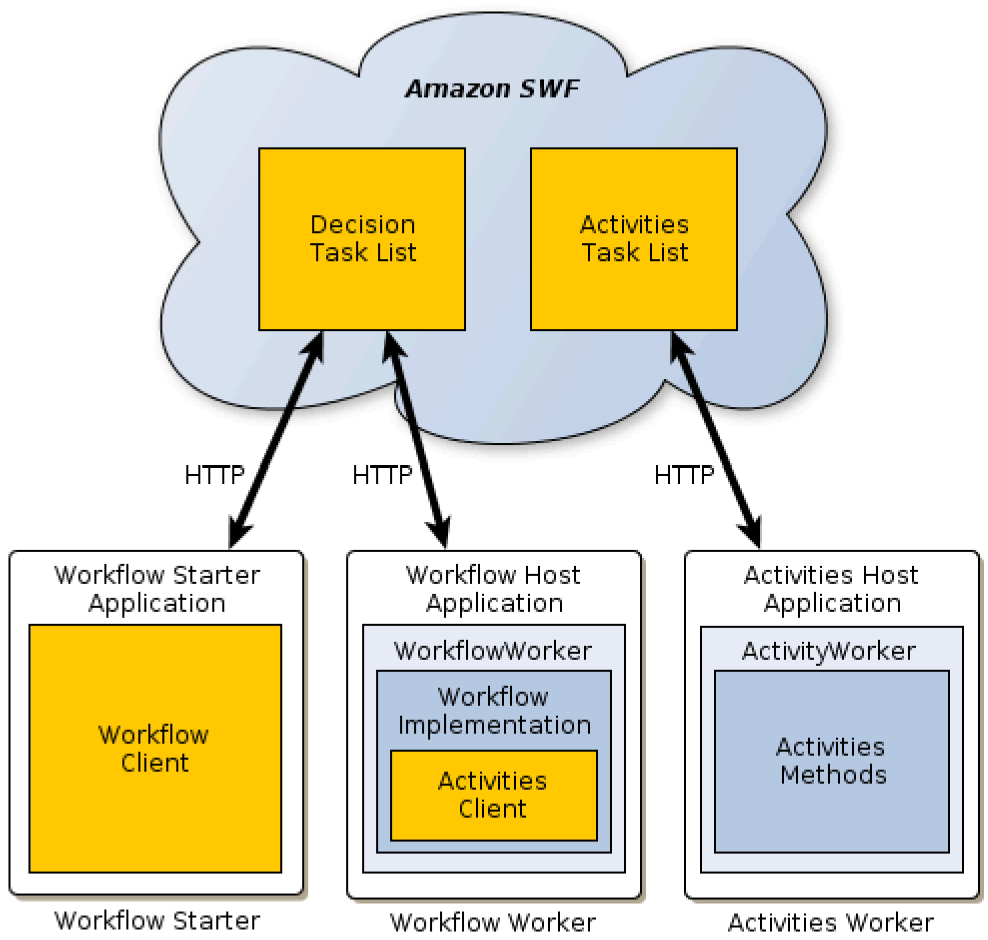
주제

- AWS Flow Framework 기본 개념: 애플리케이션 구조
- AWS Flow Framework 기본 개념: 안정적 실행
- AWS Flow Framework 기본 개념: 분산 실행
- AWS Flow Framework 기본 개념: 작업 목록 및 작업 실행
- AWS Flow Framework 기본 개념: 확장 가능 애플리케이션
- AWS Flow Framework 기본 개념: 활동과 워크플로 간 데이터 교환
- AWS Flow Framework 기본 개념: 애플리케이션과 워크플로 실행 간 데이터 교환
- Amazon SWF 제한 시간 유형

AWS Flow Framework 기본 개념: 애플리케이션 구조

개념상 AWS Flow Framework 애플리케이션은 세 가지 기본 구성 요소, 즉 워크플로 시작자, 워크플로 작업자 및 활동 작업자로 구성됩니다. 한 개 이상의 호스트 애플리케이션에서 작업자(워크플로 및 활동)를 Amazon SWF에 등록하고 작업자를 시작하고 정리하는 작업을 담당합니다. 작업자는 워크플로 실행 메커니즘을 처리하며 여러 호스트에서 구현될 수 있습니다.

다음 다이어그램은 기본 AWS Flow Framework 애플리케이션을 나타냅니다.



> **Note**
>
> 세 가지 애플리케이션에서 이들 구성 요소를 구현하는 것이 개념상 편리하지만, 애플리케이션을 생성하여 이 기능을 다양한 방식으로 구현할 수 있습니다. 예를 들어 활동 및 워크플로 작업에 대해 단일 호스트 애플리케이션을 사용하거나 별도 활동 및 워크플로 호트를 사용할 수 있습니다. 또한 각기 별도 호스트에서 일련의 다른 활동을 처리하는 등의 작업을 수행하는 여러 개의 활동 작업자를 보유할 수 있습니다.

세 가지 AWS Flow Framework 구성 요소는 요청을 관리하는 Amazon SWF로 HTTP 요청을 전송하여 간접적으로 상호 작용합니다. Amazon SWF는 다음을 수행합니다.

- 워크플로 작업자가 수행할 다음 작업을 결정하는 하나 이상의 결정 작업 목록을 유지 관리합니다.
- 활동 작업자가 수행할 작업을 결정하는 하나 이상의 활동 작업 목록을 유지 관리합니다.

- 상세한 단계별 워크플로 실행 내역을 유지 관리합니다.

AWS Flow Framework에서는 애플리케이션 코드가 HTTP 요청을 Amazon SWF에 전송하는 것과 같이 그림에 표시된 대부분의 세부 정보를 직접 처리할 필요가 없습니다. AWS Flow Framework 메서드를 호출하면 프레임워크가 이면에서 세부 정보를 처리합니다.

활동 작업자의 역할

활동 작업자는 워크플로에서 완수해야 하는 다양한 작업을 수행합니다. 작업자는 다음 요소로 구성되어 있습니다.

- 워크플로를 위해 특정 작업을 수행하는 일련의 활동 메서드가 포함된 활동 구현
- ActivityWorker 객체는 HTTP 긴 폴링 요청을 사용하여 수행할 활동 작업에 대해 Amazon SWF를 폴링하는 데 사용됩니다. 작업이 필요한 경우 Amazon SWF는 작업 수행에 필요한 정보를 전송하여 요청에 응답합니다. 그러면 ActivityWorker 객체에서 적절한 활동 메서드를 호출하여 그 결과를 Amazon SWF로 반환합니다.

워크플로 작업자의 역할

워크플로 작업자는 다양한 활동의 실행을 조율하고 데이터 흐름을 관리하며 실패한 활동을 처리합니다. 작업자는 다음 요소로 구성되어 있습니다.

- 활동 조율 로직을 포함하고 실패한 활동을 처리하는 등의 역할을 하는 워크플로 구현
- 활동 작업자의 프록시 역할을 하고 워크플로 작업자가 비동기식으로 실행할 활동을 예약할 수 있게 해주는 활동 클라이언트
- WorkflowWorker 객체는 HTTP 긴 폴링 요청을 사용하여 의사 결정 작업을 위해 Amazon SWF를 폴링합니다. 워크플로 작업 목록에 작업이 있는 경우 Amazon SWF는 작업 수행에 필요한 정보를 반환하여 요청에 응답합니다. 그러면 프레임워크에서는 워크플로를 실행하여 작업을 수행하고 그 결과를 Amazon SWF에 반환합니다.

워크플로 시작자의 역할

워크플로 시작자는 워크플로 실행이라고도 하는 워크플로 인스턴스를 시작하고, 워크플로 작업자에게 추가 데이터를 전송하거나 현재 워크플로 상태를 얻기 위해 실행 중에 인스턴스와 상호 작용할 수 있습니다.

워크플로 시작자는 워크플로 클라이언트를 사용하여 워크플로 실행을 시작하고, 실행 중에 필요에 따라 워크플로와 상호 작용하며, 정리 작업을 합니다. 워크플로 시작자는 로컬에서 실행되는 애플리케이션, 웹 애플리케이션, AWS CLI 또는AWS Management Console일 수 있습니다.

Amazon SWF가 애플리케이션과 상호 작용하는 방식

Amazon SWF는 워크플로 구성 요소 간의 상호 작용을 조정하고 상세한 워크플로 기록을 유지합니다. Amazon SWF는 구성 요소와의 통신을 시작하지 않고 구성 요소의 HTTP 요청을 기다린 다음 필요에 따라 요청을 관리합니다. 예:

- 요청이 작업자에게서 발신된 것이면 Amazon SWF는 사용 가능 작업에 대해 폴링하면서 작업이 사용 가능한 경우 작업자에게 직접 응답합니다. 폴링 작동 방식에 대한 자세한 내용은 Amazon Simple Workflow Service 개발자 안내서의 태스크 폴링을 참조하십시오.
- 요청이 활동 작업자가 보낸 작업 완료 알림인 경우 Amazon SWF는 실행 이력에 정보를 기록하고 작업을 결정 작업 리스트에 추가하여 워크플로 작업자에게 작업이 완료되었음을 알림으로써 다음 작업을 계속 진행하도록 합니다.
- 요청이 워크플로 작업자가 보낸 것으로서 활동을 실행하라는 알림인 경우 Amazon SWF는 실행 이력에 정보를 기록하고 작업을 활동 작업 리스트에 추가하여 활동 작업자에게 적절한 활동 메서드를 실행하도록 지시합니다.

이 접근 방식을 통해 작업자는 Amazon EC2 인스턴스, 기업 데이터 센터, 클라이언트 컴퓨터 등 인터넷이 연결된 모든 시스템에서 실행할 수 있습니다. 작업자는 동일한 운영 체제를 실행할 필요조차 없습니다. HTTP 요청은 작업자에서 시작되므로 외부에 보이는 포트가 필요 없습니다. 따라서 작업자는 방화벽 뒤에서 실행될 수 있습니다.

자세한 정보

Amazon SWF의 작동 방식에 대한 자세한 내용은 Amazon Simple Workflow Service 개발자 안내서를 참조하십시오.

AWS Flow Framework 기본 개념: 안정적 실행

비동기식 분산 애플리케이션에서는 기존 애플리케이션에서는 발생하지 않는 다음과 같은 안정성 문제에 대처해야 합니다.

- 원격 시스템의 장기 실행 구성 요소와 같은 비동기식 분산 구성 요소 간에 안정적 통신이 이루어지게 하는 방법

- 특히 장기 실행 애플리케이션에서 구성 요소가 실패하거나 연결이 끊어지는 경우 결과물이 분실되지 않도록 하는 방법
- 실패한 분산 구성 요소를 처리하는 방법

애플리케이션에서는 이러한 문제를 관리하기 위해 AWS Flow Framework 및 Amazon SWF에 의존할 수 있습니다. 사용자의 워크플로가 장기 실행 중이고 컴퓨터에서 인간 상호 작용을 통해 수행되는 비동기식 작업에 의존하는 경우에도 안정적이고 예측 가능한 방식으로 작동하는지 확인할 수 있는 메커니즘을 Amazon SWF에서 어떻게 제공하는지 살펴보겠습니다.

안정적 통신 제공

AWS Flow Framework는 Amazon SWF를 사용하여 분산된 활동 작업자에게 작업을 디스패치함으로써 워크플로 작업자와 활동 작업자 간에 안정적 통신이 이뤄질 수 있게 하고 그 결과를 워크플로 작업자에게 반환합니다. Amazon SWF는 다음과 같은 방법을 사용하여 작업자와 활동 간의 안정적인 커뮤니케이션을 보장합니다.

- Amazon SWF는 예약된 활동과 워크플로 작업을 내구성을 고려해 저장하고 최대 1회 수행되도록 보장합니다.
- Amazon SWF는 활동 작업이 성공적으로 완료되어 유효한 결과를 반환하거나 그렇지 않은 경우 워크플로 작업자에게 작업이 실패했음을 알리도록 보장합니다.
- Amazon SWF는 완료된 각 활동의 결과를 내구성을 고려해 저장하거나 실패한 활동의 경우 이와 관련된 오류 정보를 저장합니다.

그런 다음 AWS Flow Framework는 Amazon SWF의 활동 결과를 사용하여 워크플로 실행을 계속 진행할 방법을 확인합니다.

결과 분실 방지

워크플로 내역 보관

페타바이트 규모의 데이터에 관해 데이터 마이닝 작업을 수행하는 활동은 완료되는 데 몇 시간이 걸릴 수 있고, 인간 작업자에게 복잡한 작업을 수행하도록 지시하는 활동은 며칠, 심지어 몇 주가 걸릴 수도 있습니다.

이 같은 상황을 지원하기 위해 AWS Flow Framework 워크플로 및 활동은 완료하는 데 걸리는 시간을 임의로 길게 설정할 수 있습니다. 즉 워크플로 실행에 최대 1년의 한도를 설정할 수 있습니다. 장기 실행 프로세스를 안정적으로 실행하려면 워크플로의 실행 내역을 내구성을 고려해 지속적으로 저장할 수 있는 메커니즘이 필요합니다.

AWS Flow Framework는 각 워크플로 인스턴스의 실행 내역을 보관하는 Amazon SWF를 통해 이 문제를 해결합니다. 워크플로의 내역은 워크플로 진행에 관한 온전하고 신뢰할 수 있는 기록으로서, 예약되고 완료된 모든 워크플로 및 활동 작업과 완료되거나 실패한 활동에서 반환하는 정보가 포함되어 있습니다.

AWS Flow Framework 애플리케이션은 필요한 경우 워크플로 내역에 액세스할 수 있지만 일반적으로 이 내역과 직접 상호 작용할 필요가 없습니다. 대부분의 경우 애플리케이션에서는 프레임워크가 이면의 워크플로 내역과 상호 작용하도록 허용할 수 있습니다. 워크플로 기록에 대한 전체 논의는 Amazon Simple Workflow Service 개발자 안내서의 워크플로 기록을 참조하십시오.

상태 비저장 실행

실행 내역을 통해 워크플로 작업자는 상태 비저장 상태가 될 수 있습니다. 워크플로 또는 활동 작업자 인스턴스가 여러 개인 경우 모든 작업자가 모든 작업을 수행할 수 있습니다. 작업자는 Amazon SWF에서 작업을 수행하는 데 필요한 모든 상태 정보를 수신합니다.

이러한 접근 방식을 통해 워크플로는 더 안정화됩니다. 예를 들어 활동 작업자가 실패해도 워크플로를 다시 시작할 필요가 없습니다. 작업자를 다시 시작하기만 하면 실패가 발생한 시점에 상관없이 작업 리스트를 폴링하고 목록에 있는 모든 작업을 처리하기 시작합니다. 사용자는 별도 시스템 상에 있을 수 있는 워크플로 및 활동 작업자를 두 개 이상 사용하여 전체 워크플로가 내결함성을 갖도록 할 수 있습니다. 그러면 작업자 중 하나가 실패하면 다른 작업자가 계속해서 예약된 작업을 처리하므로 워크플로 진행이 중단되지 않습니다.

실패한 분산 구성 요소 처리

활동은 짧은 시간 동안 접속이 끊기는 등 일시적인 이유로 실패하는 경우가 많으므로, 실패한 활동을 처리하는 범용 전략은 활동을 재시도하는 것입니다. 애플리케이션에서는 복잡한 메시지 전달 전략을 시행하여 재시도 프로세스를 처리하는 대신에 AWS Flow Framework를 사용할 수 있습니다. 또한 실패한 활동을 재시도하기 위한 몇 가지 메커니즘과 워크플로 내에서 비동기식, 분산형 작업 실행을 통해 작동하는 내장 예외 처리 메커니즘을 제공합니다.

AWS Flow Framework 기본 개념: 분산 실행

워크플로 인스턴스는 기본적으로 실행 중인 활동 및 오케스트레이션 로직을 여러 원격 컴퓨터에 분산할 수 있는 가상 실행 스레드입니다. Amazon SWF와 AWS Flow Framework는 다음과 같은 방법으로 가상 CPU의 워크플로 인스턴스를 관리하는 운영 체제로서 기능합니다.

- 각 인스턴스의 실행 상태 유지
- 인스턴스 간 전환

- 전환된 시점에 인스턴스의 실행을 다시 시작

워크플로 다시 재생

활동은 장시간 동안 실행될 수 있기 때문에 활동이 완료될 때까지 단순히 워크플로를 차단하는 것은 바람직하지 않습니다. 그 대신에 AWS Flow Framework는 재생 메커니즘을 사용하여 워크플로 실행을 관리하는데, 이 메커니즘에서는 Amazon SWF에서 보관하는 워크플로 내역에 근거하여 에피소드에서 워크플로를 실행합니다.

각 에피소드에서는 각 활동을 한 번만 실행하는 방식으로 워크플로 로직을 다시 재생하고, 활동 및 비동기식 메서드는 이들의 Promise 객체가 준비 상태가 될 때까지 실행되지 않도록 합니다.

워크플로 시작자는 워크플로 실행을 시작할 때 첫 번째 다시 재생 에피소드를 시작합니다. 프레임워크에서는 워크플로의 진입점 메서드를 호출하고 다음 작업을 수행합니다.

1. 모든 활동 클라이언트 메서드를 호출하는 것을 비롯해 활동 완료에 의존하지 않는 모든 워크플로 작업을 실행합니다.
2. 실행을 위해 예약할 활동 작업 목록을 Amazon SWF에 제공합니다. 첫 번째 에피소드의 경우 이 목록은 Promise에 의존하지 않고 즉시 실행할 수 있는 활동으로만 구성됩니다.
3. 에피소드가 완료되었음을 Amazon SWF에 알립니다.

Amazon SWF는 활동 작업을 워크플로 내역에 저장하고 활동 작업 목록에 배치하여 실행 예약을 합니다. 활동 작업자는 작업 목록을 폴링하고 작업을 실행합니다.

활동 작업자는 작업을 완료하면 그 결과를 Amazon SWF에 반환합니다. 그러면 이를 워크플로 실행 내역에 기록하고 워크플로 작업 목록에 배치하여 워크플로 작업자를 위해 새 워크플로 작업을 예약합니다. 워크플로 작업자는 작업 목록을 폴링하고 작업을 받으면 다음과 같이 그다음 다시 재생 에피소드를 실행합니다.

1. 프레임워크에서는 워크플로의 진입점 메서드를 다시 실행하고 다음 작업을 수행합니다.
    - 모든 활동 클라이언트 메서드를 호출하는 것을 비롯해 활동 완료에 의존하지 않는 모든 워크플로 작업을 실행합니다. 그러나 프레임워크에서는 실행 내역을 점검하여 중복되는 활동 작업은 예약하지 않습니다.
    - 내역을 점검하여 어떤 활동 작업이 완료되었는지 확인하고 그러한 활동에 의존하는 모든 비동기식 워크플로 메서드를 실행합니다.
2. 실행할 수 있는 모든 워크플로 작업이 완료되면 다음과 같이 프레임워크에서는 Amazon SWF에 이를 보고합니다.

- 지난 에피소드 이후 입력 Promise<T> 객체가 준비 상태가 되어 실행 예약을 할 수 있는 모든 활동의 목록을 Amazon SWF에 제공합니다.
- 에피소드에서 추가 활동 작업을 생성하지 않았지만 여전히 완료되지 않은 활동이 있을 경우 프레임워크에서는 Amazon SWF에 에피소드가 완료되었음을 알립니다. 그다음에 다른 활동이 완료되길 기다렸다가 그다음 다시 재생 에피소드를 시작합니다.
- 에피소드에서 추가 활동 작업을 생성하지 않았고 모든 활동이 완료되었다면 프레임워크에서는 Amazon SWF에 워크플로 실행이 완료되었음을 알립니다.

다시 재생 동작의 예는 AWS Flow Framework for Java 다시 재생 동작 단원을 참조하십시오.

다시 재생 및 비동기식 워크플로 메서드

비동기식 워크플로 메서드는 모든 입력 Promise<T> 객체가 준비 상태가 될 때까지 실행을 연기하므로 활동과 흡사한 방식으로 사용되는 경우가 많습니다. 그러나 다시 재생 메커니즘에서는 활동과는 다른 방식으로 비동기식 메서드를 처리합니다.

- 다시 재생한다고 해서 비동기식 메서드가 한 번만 실행된다는 보장은 없습니다. 다시 재생에서는 입력 Promise 객체가 준비 상태가 될 때까지 비동기식 메서드에서 실행을 연기하지만, 그다음에는 모든 후속 에피소드에 대해 이 메서드를 실행합니다.
- 비동기식 메서드가 완료되면 새 에피소드를 시작하지 않습니다.

비동기식 워크플로 다시 재생의 한 가지 예가 AWS Flow Framework for Java 다시 재생 동작에 나와 있습니다.

다시 재생 및 워크플로 구현

대부분의 경우 다시 재생 메커니즘의 세부 정보를 일일이 신경 쓸 필요가 없습니다. 그것은 이면에서 기본적으로 일어나는 일일 뿐입니다. 그러나 다시 재생에는 워크플로 구현과 관련해 두 가지 중요한 의미가 있습니다.

- 워크플로 메서드를 사용해 장시간 실행되는 작업을 수행해서는 안 되는데, 그 이유는 다시 재생에서 작업을 여러 번 반복할 것이기 때문입니다. 심지어 비동기식 워크플로 메서드도 일반적으로 두 번 이상 실행됩니다. 그 대신에 장시간 실행되는 작업에 대해서는 활동을 사용하십시오. 그러면 다시 재생에서 활동을 한 번만 실행합니다.
- 워크플로 로직은 완전히 결정적인 것이어야 하고, 모든 에피소드에서는 동일한 제어 흐름 경로를 받아들여야 합니다. 예를 들어 제어 흐름 경로는 현재 시간에 의존해서는 안 됩니다. 다시 재생 및 결정 주의 요건에 관한 자세한 설명은 불확정성 단원을 참조하십시오.

AWS Flow Framework 기본 개념: 작업 목록 및 작업 실행

Amazon SWF는 워크플로와 활동 작업을 이름이 지정된 목록에 게시하는 방법으로 관리합니다. Amazon SWF는 워크플로 작업자 및 활동 작업자 각각에 대해 하나씩, 최소 두 가지 작업 목록을 보관합니다.

> **Note**
>
> 사용자는 목록마다 다른 작업자를 할당하여 필요한 개수 만큼 작업 목록을 지정할 수 있습니다. 작업 목록의 개수에는 제한이 없습니다. 일반적으로 작업자 객체를 생성할 때 작업자 호스트 애플리케이션에 작업자의 작업을 지정합니다.

다음은 `HelloWorldWorkflow` 호스트 애플리케이션에서 발췌한 것으로서, 새 활동 작업자를 생성하여 이를 `HelloWorldList` 활동 작업 목록에 할당합니다.

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

기본적으로 Amazon SWF는 `HelloWorldList` 목록의 작업자 작업을 예약합니다. 그런 다음 작업자는 작업에 대해 이 목록을 폴링합니다. 작업 목록에는 어떤 이름도 지정할 수 있습니다. 심지어 워크플로 및 활동 목록에 같은 이름을 사용할 수도 있습니다. 내부적으로 Amazon SWF는 워크플로 및 활동 작업 목록 이름을 여러 가지 네임스페이스에 배치하므로 두 목록은 서로 구분됩니다.

작업 목록을 지정하지 않는 경우 작업자가 Amazon SWF에 유형을 등록할 때 에서 기본 목록을 AWS Flow Framework 지정합니다. 자세한 설명은 [워크플로 및 활동 유형 등록](#) 섹션을 참조하세요.

특정 작업자 또는 작업자 그룹에서 특정 작업을 수행하도록 하는 것이 더 좋은 경우가 때로 있습니다. 예를 들면 이미지 처리 워크플로에서는 한 활동을 사용하여 이미지를 다운로드하고 다른 활동을 사용

하여 이미지를 처리할 수 있습니다. 동일 시스템에서 두 가지 작업을 수행하여 네트워크를 통해 대규모 파일을 전송하는 오버헤드를 피하는 것이 더 효율적입니다.

그러한 상황을 지원하기 위해 `schedulingOptions` 파라미터가 포함된 오버로드를 사용하여 활동 클라이언트 메서드를 호출할 때 작업 목록을 명시적으로 지정할 수 있습니다. 메서드에 적절하게 구성된 객체를 전달하여 작업 목록을 지정합니다. `ActivitySchedulingOptions`

예를 들어 `HelloWorldWorkflow` 애플리케이션의 `say` 활동을 `getName` 및 `getGreeting`과는 다른 활동 작업자가 호스팅한다고 가정합시다. 다음 예시에서는 `say`에서 `getName` 및 `getGreeting`이 처음부터 서로 다른 목록에 할당되었다 하더라도 이 둘과 동일한 작업 목록을 사용하도록 하는 방법을 보여줍니다.

```
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);
  }
}
```

비동기식 `runSay` 메서드는 클라이언트 객체로부터 `getGreeting` 작업 목록을 가져옵니다. 그런 다음 `say`에서 `getGreeting`과 동일한 작업 목록을 폴링하도록 하는 `ActivitySchedulingOptions` 객체를 생성 및 구성합니다.

> **Note**
>
> 사용자가 `schedulingOptions` 파라미터를 활동 클라이언트 메서드에 전달하면 이 메서드에서는 이 활동 실행에 대해서만 원본 작업 목록을 재정의합니다. 사용자가 작업 목록을 지정하

지 않고 활동 클라이언트 메서드를 다시 직접적으로 호출하면 Amazon SWF가 작업을 원본 목록에 할당하고 활동 작업자는 이 목록을 폴링합니다.

AWS Flow Framework 기본 개념: 확장 가능 애플리케이션

Amazon SWF에는 현재 로드를 처리하기 위해 워크플로 애플리케이션의 규모를 쉽게 조정할 수 있는 다음 두 가지 주요 기능이 있습니다.

- 상태 비저장 애플리케이션을 구현할 수 있게 해주는 전체 워크플로 실행 내역
- 작업 실행과 느슨하게 결합된 작업 일정 조정을 통해 현재 요구에 맞게 애플리케이션의 규모를 쉽게 조정할 수 있는 기능

Amazon SWF는 워크플로 및 활동 작업자와 직접 통신하는 것이 아니라 작업을 동적으로 할당된 작업 목록에 게시하는 방식으로 작업의 일정을 조정합니다. 그 대신에 작업자는 HTTP 요청을 사용하여 작업의 해당 목록을 폴링합니다. 이 접근 방식에서는 작업 일정 조정을 작업 실행에 느슨하게 결합하여 작업자가 Amazon EC2 인스턴스, 기업 데이터 센터, 클라이언트 컴퓨터 등 적합한 모든 시스템에서 실행될 수 있게 합니다. HTTP 요청은 작업자에서 시작되므로 외부에 보이는 포트가 필요 없습니다. 따라서 작업자는 방화벽 뒤에서도 실행될 수 있습니다.

작업자가 작업에 대한 폴링에 사용하는 긴 폴링 메커니즘을 사용하면 작업자가 오버로드되지 않습니다. 예약된 작업에 스파이크가 있다 하더라도 작업자는 자신의 고유 속도로 작업을 가져옵니다. 그러나 작업자는 상태 비저장이므로 추가 작업자 인스턴스를 시작하여 애플리케이션의 규모를 증가된 로드에 맞게 동적으로 조정할 수 있습니다. 각 인스턴스는 서로 다른 시스템에서 실행 중일지라도 동일한 작업 목록을 폴링하고 첫 번째 사용 가능 작업자 인스턴스에서는 작업자의 위치나 시작 시점에 상관없이 각 작업을 실행합니다. 로드가 감소하면 이에 따라 작업자의 수를 줄일 수 있습니다.

AWS Flow Framework 기본 개념: 활동과 워크플로 간 데이터 교환

비동기식 활동 클라이언트 메서드를 직접적으로 호출하면 이 메서드에서는 즉시 Promise(Future라고도 함) 객체를 반환하는데, 이 객체에서는 활동 메서드의 반환 값을 표시합니다. 처음에 이 Promise는 준비되지 않은 상태이고 반환 값은 정의되어 있지 않습니다. 활동 메서드에서 작업을 완료하고 값을 반환하면 프레임워크에서는 이 반환 값을 네트워크를 통해 워크플로 작업자에게 마샬링합니다. 그러면 이 작업자는 Promise에 값을 할당하고 객체를 준비 상태로 둡니다.

활동 메서드에 반환 값이 없다 하더라도 사용자는 여전히 워크플로 실행 관리에 Promise를 사용할 수 있습니다. 활동 클라이언트 메서드 또는 비동기식 워크플로 메서드에 반환된 Promise를 전달하면 해당 메서드에서는 객체가 준비 상태가 될 때까지 실행을 연기합니다.

Promise를 활동 클라이언트 메서드에 한 개 이상 전달하면 프레임워크에서는 작업을 대기열에 배치하기는 하지만 모든 객체가 준비 상태가 될 때까지 예약을 연기합니다. 그런 다음 각 Promise에서 데이터를 추출하여 이를 인터넷을 통해 활동 작업자에게 마샬링합니다. 그러면 이 작업자는 이를 활동 메서드에 표준 유형으로 전달합니다.

> **Note**
> 워크플로 및 활동 작업자 사이에 데이터를 대량 전송해야 하는 경우 선호되는 방식은 데이터를 편리한 위치에 저장하고 검색 정보를 전달하는 것입니다. 예를 들어 Amazon S3 버킷에 데이터를 저장하고 관련 URL을 전달할 수 있습니다.

Promise<T> 유형

`Promise<T>` 유형은 몇 가지 점에서 Java `Future<T>` 유형과 비슷합니다. 두 유형에서는 비동기식 메서드에서 반환하는 값을 표시하며 처음에는 정의되어 있지 않습니다. 사용자는 `get` 메서드를 직접적으로 호출하여 객체의 값에 액세스합니다. 이것 말고도 이 두 유형은 상당히 다르게 동작합니다.

- `Future<T>`는 동기화 구성으로서, 애플리케이션이 비동기식 메서드가 완료될 때까지 대기할 수 있게 해줍니다. `get`을 직접적으로 호출하였는데 객체가 준비 상태가 아니면 객체가 준비 상태가 될 때까지 차단합니다.
- `Promise<T>`의 경우 동기화는 프레임워크에서 처리됩니다. `get`을 직접적으로 호출하였는데 객체가 준비 상태가 아니면 `get`에서 예외가 발생합니다.

`Promise<T>`의 기본 목적은 한 활동에서 다른 활동으로의 데이터 흐름을 관리하는 것입니다. 이를 통해 입력 데이터가 유효할 때까지 활동이 실행되지 않게 합니다. 많은 경우 워크플로 작업자는 `Promise<T>` 객체에 직접 액세스할 필요가 없습니다. 즉 한 활동에서 다른 활동으로 객체를 전달하기만 하고 세부 정보는 프레임워크 및 활동 작업자가 처리하도록 합니다. 워크플로 작업자에 있는 `Promise<T>` 객체의 값에 액세스하려면 그 객체가 준비 상태인지 확인한 후에 `get` 메서드를 직접적으로 호출해야 합니다.

- 선호되는 방식은 `Promise<T>` 객체를 비동기식 워크플로 메서드에 전달하고 거기에서 값을 처리하는 것입니다. 비동기식 메서드에서는 모든 입력 `Promise<T>` 객체가 준비 상태가 될 때가 실행을 연기합니다. 이를 통해 사용자가 객체의 값에 안전하게 액세스할 수 있도록 보장합니다.

- Promise<T>에서는 객체가 준비 상태가 되면 true를 반환하는 isReady 메서드를 노출합니다. isReady를 사용하여 Promise<T> 객체를 폴링하는 것은 좋지 않지만, 특정 환경에서는 isReady가 유용합니다. 예제는 AWS Flow Framework 레시피를 참조하십시오.

AWS Flow Framework for Java는 Settable<T> 유형도 포함합니다. 이 유형은 Promise<T>에서 파생된 것으로서 동작이 유사합니다. 차이점은 프레임워크에서는 일반적으로 Promise<T> 객체의 값을 설정하고 워크플로 작업자는 Settable<T>의 값을 설정하는 작업을 담당합니다. 예제는 AWS Flow Framework 레시피를 참조하십시오.

어떤 상황에서는 워크플로 작업자가 Promise<T> 객체를 생성하고 그 값을 설정해야 합니다. 예를 들어 Promise<T> 객체를 반환하는 비동기식 메서드에서는 반환 값을 생성해야 합니다.

- 입력된 값을 표시하는 객체를 생성하려면 적절한 유형의 Promise<T> 객체를 생성한 후 값을 설정하고 이 값을 준비 상태에 입력하는 정적 Promise.asPromise 메서드를 직접적으로 호출합니다.
- Promise<Void> 객체를 생성하려면 정적 Promise.Void 메서드를 직접적으로 호출합니다.

> **Note**
>
> Promise<T>에서는 유효한 유형은 모두 표시할 수 있습니다. 그러나 데이터가 인터넷을 통해 마샬링되어야 하는 경우 그 유형은 데이터 변환기와 호환되어야 합니다. 자세한 내용은 다음 단원을 참조하십시오.

데이터 변환기 및 마샬링

AWS Flow Framework는 데이터 변환기를 사용하여 인터넷을 통해 데이터를 마샬링합니다. 기본적으로 프레임워크에서는 Jackson JSON 프로세서 기반 데이터 변환기를 사용합니다. 그러나 이 변환기에는 몇 가지 제한 사항이 있습니다. 예를 들면 문자열을 키로 사용하지 않는 맵은 마샬링할 수 없습니다. 기본 변환기로는 애플리케이션에 충분하지 않은 경우 사용자 지정 데이터 변환기를 구현할 수 있습니다. 자세한 내용은 DataConverters 섹션을 참조하십시오.

AWS Flow Framework 기본 개념: 애플리케이션과 워크플로 실행 간 데이터 교환

워크플로 진입점 메서드에는 파라미터가 한 개 이상 있을 수 있는데, 이를 통해 워크플로 시작자는 초기 데이터를 워크플로로 전달할 수 있습니다. 실행 중에 추가 데이터를 워크플로에 제공하는 것도 유용

할 수 있습니다. 예를 들어 고객이 자신의 배송지 주소를 변경하면 주문 처리 워크플로에 알려주어 적절히 변경할 수 있게 할 수 있습니다.

Amazon SWF를 통해 워크플로는 신호 메서드를 구현할 수 있습니다. 이로써 워크플로 시작자와 같은 애플리케이션에서는 데이터를 언제든지 워크플로로 전달할 수 있습니다. 신호 메서드에는 원하는 대로 이름과 파라미터를 지정할 수 있습니다. 사용자는 워크플로 인터페이스 정의를 포함하고 메서드 선언에 `@Signal` 주석을 붙여 신호 메서드로 지정합니다.

다음 예에서는 신호 메서드인 `changeOrder`를 선언하고 이를 통해 워크플로가 시작된 후에 워크플로 시작자가 원래 주문을 변경할 수 있게 해주는 주문 처리 워크플로 인터페이스를 보여줍니다.

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

프레임워크의 주석 프로세서는 신호 메서드와 동일한 이름의 워크플로 클라이언트 메서드를 생성하고, 워크플로 시작자는 클라이언트 메서드를 직접적으로 호출하여 데이터를 워크플로로 전달합니다. 예제는 AWS Flow Framework 레시피를 참조하십시오.

Amazon SWF 제한 시간 유형

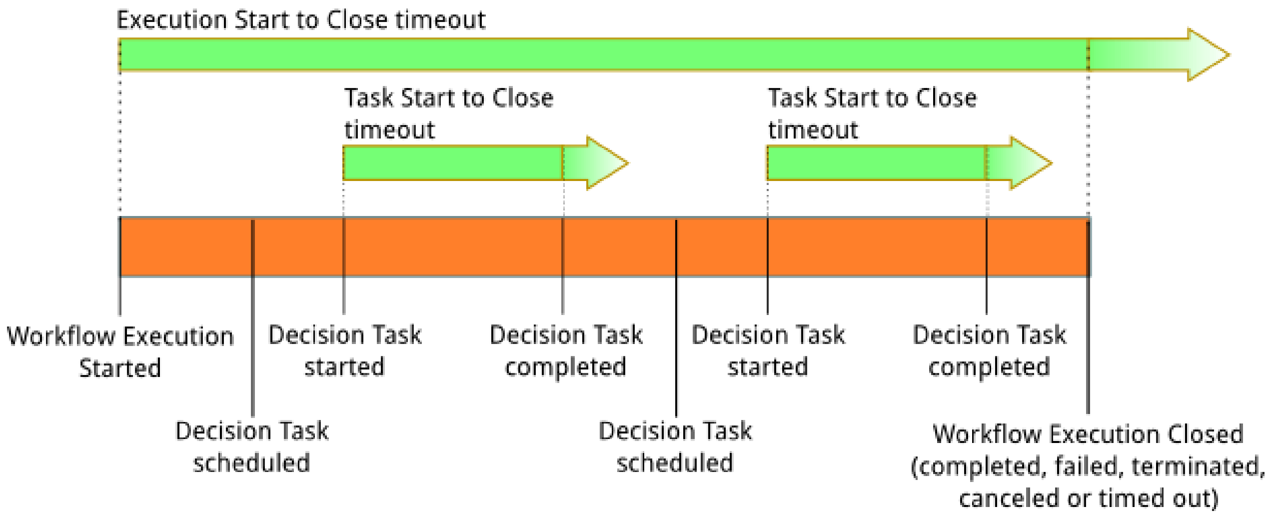
Amazon SWF는 다양한 유형의 제한 시간을 설정하여 워크플로 실행이 정확히 진행되도록 할 수 있습니다. 워크플로 전체를 얼마나 오래 실행할 수 있는지 지정하는 제한 시간도 있고, 활동 작업이 작업자에게 할당되기까지 걸리는 시간 및 예약 시점으로부터 완료되기까지 걸리는 시간을 지정하는 제한 시간도 있습니다. Amazon SWF API의 모든 제한 시간은 초 단위로 지정됩니다. Amazon SWF는 `NONE` 문자열을 제한 시간 값으로도 지원하며 이는 제한 시간이 없음을 나타냅니다.

결정 작업 및 활동 작업과 관련된 제한 시간의 경우, Amazon SWF는 워크플로 실행 내역에 이벤트를 추가합니다. 이러한 이벤트의 속성은 발생한 제한 시간의 유형과 해당하는 결정 작업 또는 활동 작업에 대한 정보를 제공합니다. Amazon SWF는 결정 작업을 예약합니다. 디사이더는 새 의사 결정 작업을 받으면 기록에서 타임아웃 이벤트를 확인하고 조치를 호출하여 적절한 조치를 취합니다.
RespondDecisionTaskCompleted

작업은 예약 시점부터 닫힐 때까지 열린 상태로 간주됩니다. 따라서 작업자가 작업을 처리하는 동안에는 열려 있는 상태로 보고됩니다. 작업자가 작업을 완료됨, 취소됨 또는 실패로 보고하면 작업이 닫힙니다. 제한 시간으로 인해 Amazon SWF에서 작업을 종료할 수도 있습니다.

워크플로 및 결정 작업의 제한 시간

다음 다이어그램은 워크플로 및 결정 제한 시간이 워크플로의 수명과 어떤 관계인지 보여줍니다.



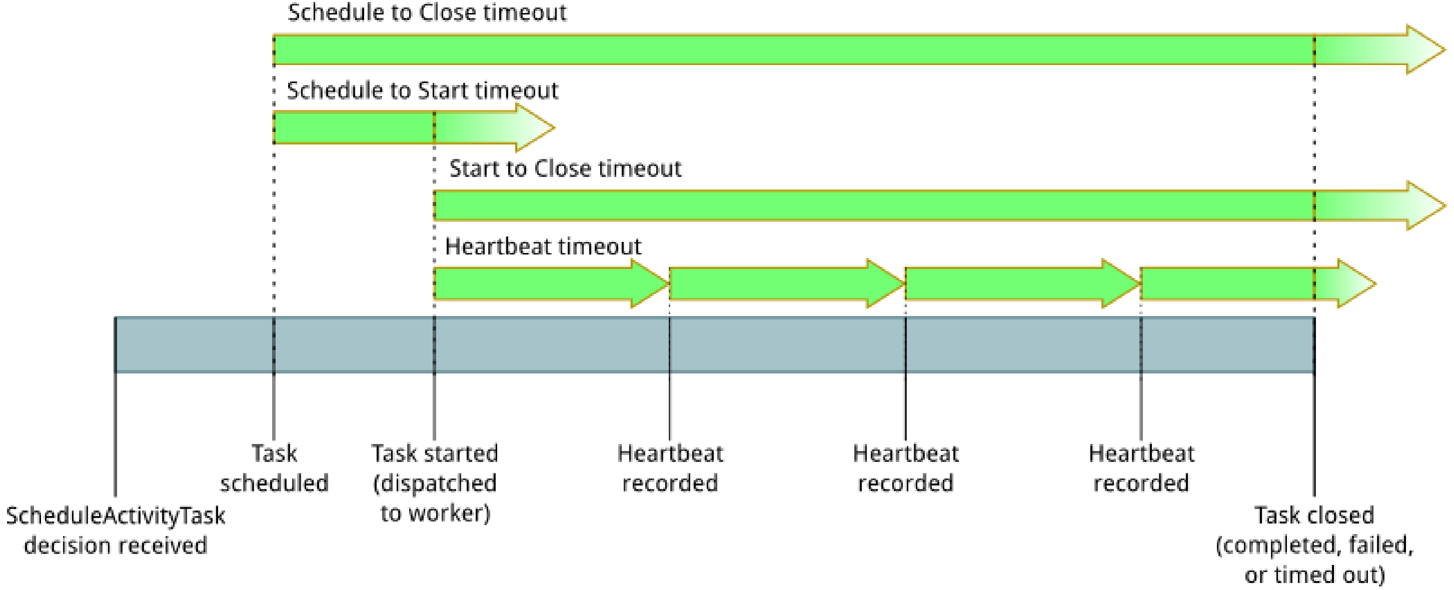
워크플로 및 결정 작업과 관련된 제한 시간 유형은 다음 두 가지입니다.

- 워크플로 시작-닫기(**`timeoutType: START_TO_CLOSE`**) – 이 제한 시간은 워크플로 실행이 완료되는 데 걸리는 최대 시간을 지정합니다. 워크플로 등록 과정에서 기본값으로 설정되지만 워크플로가 시작될 때 다른 값으로 재정의할 수 있습니다. 이 제한 시간을 초과하면 Amazon SWF는 워크플로 실행을 닫고 워크플로 실행 기록에 해당 WorkflowExecutionTimedOut유형의 이벤트를 추가합니다. `timeoutType`을 비롯한 이벤트 속성으로 이 워크플로 실행에 적용되는 `childPolicy`를 지정합니다. 하위 정책은 상위 워크플로 실행 시간이 초과되거나 그렇지 않고 종료되면 하위 워크플로 실행을 처리하는 방법을 지정합니다. 예를 들어, `childPolicy`를 TERMINATE로 설정하면 하위 워크플로 실행이 종료됩니다. 워크플로 실행이 시간 초과되면 가시성 호출 이외의 조치를 취할 수 없습니다.
- 결정 작업 시작-닫기(**`timeoutType: START_TO_CLOSE`**) – 이 제한 시간은 해당 결정자가 결정 작업을 완료하는 데 소요할 수 있는 최대 시간을 지정합니다. 이 값은 워크플로 유형 등록 중 설정합니다. 이 제한 시간을 초과하면 워크플로 실행 기록에 작업이 시간 초과로 표시되고 Amazon SWF는 워크플로 기록에 해당 DecisionTaskTimedOut유형의 이벤트를 추가합니다. 이벤트 속성에는 결정 작업이 예약되었을 때 해당하는 이벤트의 ID(`scheduledEventId`)와 결정 작업이 시작되었을 때 해당하는 이벤트의 ID(`startedEventId`)가 포함됩니다. Amazon SWF는 이벤트를 추가하는 한편 새 결정 작업을 예약해 결정자에게 이 결정 작업이 시간 초과되었음을 알립니다. 시간 초과가 발생한

후 `RespondDecisionTaskCompleted`를 사용해 시간 초과된 결정 작업을 완료하려고 하면 실패합니다.

활동 작업의 제한 시간

다음 다이어그램은 제한 시간이 활동 작업의 수명과 어떻게 관련이 있는지를 보여줍니다.



여기에는 활동 작업과 관련된 제한 시간 유형이 4개 있습니다.

- 활동 작업 시작-닫기(**`timeoutType: START_TO_CLOSE`**) – 이 제한 시간은 활동 작업자가 작업을 수신한 후 작업을 처리하기 위해 보낼 수 있는 최대 시간을 지정합니다. RespondActivityTaskCanceledRespondActivityTaskCompleted, 를 사용하여 제한 시간이 초과된 활동 작업을 종료하려고 하면 작업이 실패합니다. RespondActivityTaskFailed
- 활동 작업 하트비트(**`timeoutType: HEARTBEAT`**) – 이 제한 시간은 `RecordActivityTaskHeartbeat` 작업을 통해 진행 상황을 제공하기 전에 작업을 실행할 수 있는 최대 시간을 지정합니다.
- 활동 작업 예약-시작(**`timeoutType: SCHEDULE_TO_START`**) – 활동 작업을 수행할 작업자가 없을 때 Amazon SWF는 이 제한 시간만큼 기다렸다가 활동 작업을 시간 초과로 처리합니다. 시간이 초과로 만료된 작업은 다른 작업자에게 할당되지 않습니다.
- 활동 작업 예약-닫기(**`timeoutType: SCHEDULE_TO_CLOSE`**) – 이 제한 시간은 예약 시간부터 완료될 때까지 걸릴 수 있는 기간을 지정합니다. 가장 좋은 방법은 이 값이 작업 schedule-to-start 제한 시간과 작업 제한 시간의 합계보다 크지 않는 것입니다. start-to-close

Note

각 제한 시간 유형에는 기본값이 있는데, 일반적으로 NONE(무한정)으로 설정되어 있습니다. 그러나 활동 실행의 최대 시간은 1년으로 제한됩니다.

활동 유형 등록 중 활동에 대한 기본값을 설정하지만 활동 작업을 예약할 때 새 값으로 기본값을 재정의할 수 있습니다. 이러한 제한 시간 중 하나가 발생하면 Amazon SWF는 워크플로 기록에 해당 ActivityTaskTimedOut유형의 이벤트를 추가합니다. 이 이벤트의 `timeoutType` 값 속성은 어떤 제한 시간이 발생했는지 지정합니다. 각 제한 시간의 경우 `timeoutType`의 값은 괄호 안에 표시됩니다. 또한 이벤트 속성에는 활동 작업이 예약되었을 때 해당하는 이벤트의 ID(`scheduledEventId`)와 결정 작업이 시작되었을 때 해당하는 이벤트의 ID(`startedEventId`)가 포함됩니다. Amazon SWF는 이벤트를 추가하는 한편 새 결정 작업을 예약해 결정자에게 시간 초과가 발생했음을 알립니다.

모범 사례

AWS Flow Framework for Java를 최대한 활용하려면 이러한 모범 사례를 사용하십시오.

주제

- 결정자 코드 변경: 버전 관리 및 기능 플래그

결정자 코드 변경: 버전 관리 및 기능 플래그

이 단원에서는 다음과 같이 두 가지 메서드를 사용해 이전 버전과 호환되지 않는 결정자 변경을 방지하는 방법을 보여줍니다.

- 버전 관리에서는 기본 솔루션을 제공합니다.
- 기능 플래그를 통한 버전 관리는 버전 관리 솔루션을 바탕으로 구축됩니다. 새 버전의 워크플로는 도입되지 않으므로 버전 업데이트를 위해 새 코드를 푸시할 필요가 없습니다.

이 솔루션을 사용하기 전에 먼저 이전 버전과 호환되지 않는 결정자 변경의 원인과 결과를 설명하는 예제 시나리오 단원의 내용을 숙지하시기 바랍니다.

다시 재생 프로세스 및 코드 변경

AWS Flow Framework for Java 결정자 작업자는 결정 작업을 실행할 때 먼저 현재 실행 상태를 다시 빌드해야 단계를 추가할 수 있습니다. 결정자는 다시 재생이라는 프로세스를 사용하여 이를 수행합니다.

다시 재생 프로세스에서는 이미 발생한 이벤트 내역을 거치면서 이와 동시에 처음부터 결정자 코드를 다시 실행합니다. 이벤트 내역을 거침으로써 프레임워크에서는 신호 또는 작업 완료에 반응하고 코드에서 `Promise` 객체 차단을 해제할 수 있습니다.

프레임워크에서는 결정자 코드를 실행할 때 카운터를 늘려 각 예약 작업(활동, Lambda 함수, 타이머, 하위 워크플로 또는 발신 신호)에 ID를 할당합니다. 프레임워크에서는 이 ID를 Amazon SWF에 전달하고 `ActivityTaskCompleted`와 같은 내역 이벤트에 ID를 추가합니다.

다시 재생 프로세스의 성공에 중요한 것은 결정자 코드가 결정적이어야 하고 모든 워크플로 실행에서 모든 결정에 대해 동일한 순서로 동일한 작업을 예약해야 한다는 것입니다. 이 요구 사항을 준수하지 않으면 프레임워크에서 `ActivityTaskCompleted` 이벤트의 ID를 기존 `Promise` 객체에 매칭하지 못하는 등의 결과가 발생할 수 있습니다.

예제 시나리오

이전 버전과 호환되지 않는 것으로 간주되는 코드 변경 클래스가 있습니다. 이러한 변경에는 예약 작업의 번호, 유형 또는 순서를 수정하는 업데이트가 포함됩니다. 다음 예제를 검토하십시오.

결정자 코드를 작성하여 두 가지 타이머 작업을 예약합니다. 실행을 시작하고 결정을 실행합니다. 결과적으로 ID 1 및 2를 통해 두 가지 타이머 작업이 예약됩니다.

다음 결정이 실행되기 전에 결정자 코드를 업데이트하여 하나의 타이머만 예약하면 코드에서 생성한 타이머 작업과 ID 2가 매칭되지 않기 때문에 다음 결정 작업 중에 프레임워크에서 두 번째 `TimerFired` 이벤트를 다시 재생하지 못합니다.

시나리오 개요

다음 개요에서는 이 시나리오의 절차를 보여줍니다. 시나리오의 최종 목표는 타이머를 하나만 예약하면서도 마이그레이션 전에 시작된 실행에서 실패를 발생시키지 않는 시스템으로 마이그레이션하는 것입니다.

1. 초기 결정자 버전
   a. 결정자를 작성합니다.
   b. 결정자를 시작합니다.
   c. 결정자는 타이머 두 개를 예약합니다.
   d. 결정자는 실행 다섯 개를 시작합니다.
   e. 결정자를 중단합니다.
2. 이전 버전과 호환되지 않는 결정자 변경
   a. 결정자를 수정합니다.
   b. 결정자를 시작합니다.
   c. 결정자는 타이머 한 개를 예약합니다.
   d. 결정자는 실행 다섯 개를 시작합니다.

다음 단원에는 이 시나리오를 구현하는 방법을 보여주는 Java 코드 예시가 포함되어 있습니다. 솔루션 단원의 코드 예시에서는 이전 버전과 호환되지 않는 변경을 수정하는 다양한 방법을 보여줍니다.

Note

AWS SDK for Java 최신 버전을 사용하여 이 코드를 실행할 수 있습니다.

공통 코드

다음 Java 코드는 이 시나리오에 제시된 여러 가지 예시 내에서 변경되지 않습니다.

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").withWorkflowExecutionRetentic
        } 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;
    }

}
```

초기 결정자 코드 작성

다음은 결정자의 초기 Java 코드입니다. 이 코드는 1 버전으로 등록되고 5초 타이머 작업 두 개를 예약합니다.

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);
        }

    }
}
```

이전 버전과 호환되지 않는 변경 시뮬레이션

다음과 같이 결정자의 수정된 Java 코드는 이전 버전과 호환되지 않는 변경의 좋은 예입니다. 이 코드는 여전히 1 버전으로 등록되지만 타이머 한 개만 예약합니다.

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);
        }

    }
}
```

다음 Java 코드를 통해 수정된 결정자를 실행하여 이전 버전과 호환되지 않는 변경 문제를 시뮬레이션할 수 있습니다.

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();
    }

}
```

프로그램을 실행할 때 실패하는 세 가지 실행은 결정자의 초기 버전에서 시작되어 마이그레이션 후에도 계속된 것들입니다.

솔루션

다음 솔루션을 사용하여 이전 버전과 호환되지 않는 변경을 방지할 수 있습니다. 자세한 내용은 결정자 코드 변경 및 예제 시나리오 단원을 참조하십시오.

버전 관리 사용

이 솔루션에서는 결정자를 새 클래스에 복사하고 결정자를 수정한 다음, 새 워크플로 버전에서 결정자를 등록합니다.

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);
        }

    }

}
```

업데이트된 Java 코드에서 두번째 결정자 작업자는 워크플로의 두 버전을 모두 실행함으로써 진행 중인 실행이 버전 2의 변경 사항에 상관없이 독자적으로 계속 실행되게 합니다.

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();
    }

}
```

프로그램을 실행하면 모든 실행이 성공적으로 완료됩니다.

기능 플래그 사용

이전 버전과의 호환성 문제를 해결할 수 있는 또 다른 방법은 워크플로 버전이 아닌 입력 데이터에 근거해 코드를 분기하여 동일한 클래스에서 두 가지 구현을 지원하는 것입니다.

이러한 접근 방식을 택하는 경우에는 민감한 변경 사항을 도입할 때마다 필드를 입력 객체에 추가(또는 입력 객체의 기존 필드를 수정)합니다. 마이그레이션 이전에 시작하는 실행의 경우 입력 객체는 필드가 없습니다(또는 값이 다름). 따라서 버전 번호를 증가시킬 필요가 없습니다.

> **Note**
>
> 새 필드를 추가하는 경우 JSON 역직렬화 프로세스가 이전 버전과 호환되는지 확인합니다. 필드 도입 전에 직렬화된 객체는 마이그레이션 후에도 성공적으로 역직렬화되어야 합니다. 필드가 누락될 때마다 JSON에서는 `null` 값을 설정하므로 항상 상자 포장 유형(`boolean`이 아닌 `Boolean`)을 사용하고 값이 `null`인 경우를 처리해야 합니다.

`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);
            }
        }

    }
}
```

업데이트된 Java 코드에서 워크플로의 두 버전에 대한 코드는 여전히 1 버전에 등록됩니다. 그러나 마이그레이션 후 새 실행은 `true`로 설정된 입력 데이터의 `skipSecondTimer` 필드로 시작합니다.

`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();
    }

}
```

프로그램을 실행하면 모든 실행이 성공적으로 완료됩니다.

AWS Flow Framework for Java 프로그래밍 가이드

이 단원에서는 AWS Flow Framework for Java의 기능을 사용하여 워크플로 애플리케이션을 구현하는 방법에 관한 세부 정보를 제공합니다.

주제

- AWS Flow Framework를 사용하여 워크플로 애플리케이션 구현
- 워크플로 및 활동 계약
- 워크플로 및 활동 유형 등록
- 활동 및 워크플로 클라이언트
- 워크플로 구현
- 활동 구현
- AWS Lambda 작업 구현
- AWS Flow Framework for Java로 작성한 프로그램 실행
- 실행 콘텍스트
- 하위 워크플로 실행
- 연속 워크플로
- 작업 우선 순위 설정
- DataConverters
- 데이터를 비동기식 메서드로 전달
- 테스트 가능성 및 종속성 주입
- 오류 처리
- 실패한 활동 재시도
- 대몬(daemon) 작업
- AWS Flow Framework for Java 다시 재생 동작

AWS Flow Framework를 사용하여 워크플로 애플리케이션 구현

AWS Flow Framework를 사용하여 워크플로를 개발하는 작업에 수반되는 일반적인 단계는 다음과 같습니다.

1. 활동 및 워크플로 계약을 정의합니다. 애플리케이션의 요구 사항을 분석한 후 필요한 활동과 워크플로 토폴로지를 결정합니다. 활동에서는 필요한 처리 작업을 처리하고, 워크플로 토폴로지에서는 워크플로의 기본 구조 및 비즈니스 로직을 정의합니다.

   예를 들어 미디어 처리 애플리케이션에서는 파일을 다운로드하여 이를 처리한 후 처리한 파일을 Amazon Simple Storage Service(S3) 버킷에 업로드해야 할 수 있습니다. 이 작업은 다음과 같이 네 가지 활동 작업으로 구분할 수 있습니다.

   1. 서버에서 파일 다운로드
   2. 이 파일을 처리(예: 다른 미디어 형식으로 코드 변환)
   3. 이 파일을 S3 버킷에 업로드
   4. 로컬 파일을 삭제하여 정리 작업 수행

   이 워크플로에는 진입점 메서드가 있어 HelloWorldWorkflow 애플리케이션과 흡사하게 활동을 순차적으로 실행하는 단순한 선형 토폴로지를 구현합니다.

2. 활동 및 워크플로 인터페이스를 구현합니다. 워크플로 및 활동 계약을 Java 인터페이스에서 정의함으로써 SWF에서 호출 규칙을 예측할 수 있고 사용자는 워크플로 로직 및 활동 작업을 구현할 때 유연성을 발휘할 수 있습니다. 프로그램의 여러 부분은 각기 보유한 데이터의 소비자로 기능하면서도 다른 부분의 구현 세부 정보에 대해 많이 알 필요가 없습니다.

   예를 들어 `FileProcessingWorkflow` 인터페이스를 정의하고 비디오 인코딩, 압축, 썸네일 등에 여러 가지 워크플로 구현을 제공할 수 있습니다. 그러한 워크플로는 여러 가지 제어 플로우를 보유할 수 있고 다양한 활동 메서드를 호출할 수 있습니다. 따라서 워크플로 시작자는 이를 알 필요가 없습니다. 인터페이스를 사용하면 나중에 작업 코드와 교체할 수 있는 모의 구현을 사용하여 워크플로를 간단히 테스트할 수 있습니다.

3. 활동 및 워크플로를 생성합니다. AWS Flow Framework에서는 비동기식 실행 관리, HTTP 요청 전송, 데이터 마샬링 등에 관한 세부 정보를 구현할 필요가 없습니다. 대신에 워크플로 시작자는 워크플로 클라이언트에서 메서드를 호출하여 워크플로 인스턴스를 실행하고, 워크플로 구현에서는 활동 클라이언트에서 메서드를 호출하여 활동을 실행합니다. 프레임워크에서는 이러한 상호 작용에 관한 세부 정보를 배경에서 처리합니다.

   Eclipse를 사용 중이며 프로젝트를 구성한 경우 AWS Flow Framework for Java 설정 단원에서처럼 AWS Flow Framework 주석 프로세서에서는 인터페이스 정의를 사용하여 상응하는 인터페이스와 동일한 메서드 집합을 노출하는 워크플로 및 활동 클라이언트를 자동으로 생성합니다.

4. 활동 및 워크플로 호스트 애플리케이션을 구현합니다. 워크플로 및 활동 구현은 작업에 대해 Amazon SWF를 폴링하고 모든 데이터를 마샬링하며 적절한 구현 메서드를 직접적으로 호출하는

호스트 애플리케이션에 임베딩되어야 합니다. AWS Flow Framework for Java에는 호스트 애플리케이션 구현을 간단하고 편리하게 만들어주는 WorkflowWorker 및 ActivityWorker 클래스가 포함되어 있습니다.

5. 워크플로를 테스트합니다. AWS Flow Framework for Java는 워크플로를 인라인 및 로컬에서 테스트하는 데 사용할 수 있는 JUnit 통합을 제공합니다.
6. 작업자를 배포합니다. 작업자를 Amazon EC2 인스턴스 또는 데이터 센터의 컴퓨터 등에 적절하게 배포할 수 있습니다. 배포되어 시작된 작업자는 작업에 대해 Amazon SWF 폴링을 시작하고 필요에 따라 작업을 처리합니다.
7. 실행을 시작합니다. 애플리케이션에서는 워크플로 클라이언트를 사용하여 워크플로의 진입점을 호출함으로써 워크플로 인스턴스를 시작합니다. 또한 Amazon SWF 콘솔을 사용하여 워크플로를 시작할 수도 있습니다. 워크플로 인스턴스를 시작하는 방식에 상관없이 사용자는 Amazon SWF 콘솔을 사용하여 실행 중인 워크플로 인스턴스를 모니터링하고 실행 중, 완료 및 실패 인스턴스에 대한 워크플로 내역을 검토할 수 있습니다.

AWS SDK for Java에는 루트 디렉터리의 readme.html 파일에 있는 지침에 따라 검색하고 실행할 수 있는 일련의 AWS Flow Framework for Java 샘플이 포함되어 있습니다. 다양한 특정 프로그래밍 문제를 처리하는 방법을 보여주는 일련의 레시피(간단한 애플리케이션)가 있으며, 해당 레시피는 AWS Flow Framework 레시피에서 사용할 수 있습니다.

워크플로 및 활동 계약

워크플로 및 활동의 서명을 선언하는 데 Java 인터페이스를 사용합니다. 이 인터페이스는 워크플로(또는 활동)와 이 워크플로(또는 활동)의 클라이언트 사이의 계약을 형성합니다. 예를 들어 `MyWorkflow` 워크플로 유형은 다음과 같이 `@Workflow` 주석이 붙은 인터페이스를 사용하여 정의됩니다.

```
@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();
}
```

계약에는 구현별 설정이 없습니다. 이와 같은 구현 중립적 계약을 사용함으로써 클라이언트는 구현에서 분리될 수 있고, 따라서 클라이언트와 분리되지 않고도 구현 세부 정보를 변경할 수 있는 유연성을 얻게 됩니다. 반대로 사용자는 사용 중인 워크플로 또는 활동을 변경하지 않고도 클라이언트를 변경할 수 있습니다. 예를 들어 활동 구현을 변경하지 않고도 약속(`Promise<T>`)을 사용하여 비동기식으로 활동을 호출하도록 클라이언트를 수정할 수 있습니다. 마찬가지로, 활동 구현은 활동의 클라이언트를 변경할 필요 없이, 예를 들어 이메일을 보내는 사람에 의해 비동기적으로 완료되도록 변경할 수 있습니다.

위 예시에서 워크플로 인터페이스 `MyWorkflow`에는 새 실행을 시작하기 위한 `startMyWF` 메서드가 포함되어 있습니다. 이 메서드에는 `@Execute` 주석이 달려 있고 `void` 또는 `Promise<>` 반환 유형이 있어야 합니다. 특정 워크플로 인터페이스에는 최대 한 개의 메서드에 이 주석이 달릴 수 있습니다. 이 메서드는 워크플로 로직의 진입점이며, 프레임워크에서는 결정 작업을 수신하면 이 메서드를 호출하여 워크플로 로직을 실행합니다.

또한 워크플로 인터페이스에서는 워크플로로 전송할 수 있는 신호를 정합니다. 이 신호 메서드는 워크플로 실행에서 일치하는 이름이 있는 신호를 수신하면 호출됩니다. 예를 들어 `MyWorkflow` 인터페이스에서는 `@Signal` 주석이 붙은 `signal1` 신호 메서드를 선언합니다.

`@Signal` 주석은 신호 메서드에서 필요합니다. 신호 메서드의 반환 유형은 `void`이어야 합니다. 워크플로 인터페이스에는 0개 이상의 신호 메서드가 정의되어 있을 수 있습니다. 사용자는 `@Execute` 메서드 및 일부 `@Signal` 메서드 없이 워크플로 인터페이스를 선언함으로써 실행을 시작할 수는 없지만 실행 중인 실행에 신호를 전송할 수 있는 클라이언트를 생성할 수 있습니다.

`@Execute` 및 `@Signal` 주석이 달린 메서드에는 `Promise<T>` 또는 그 파생물이 아닌 유형의 파라미터가 몇 개든 있을 수 있습니다. 이를 통해 사용자는 워크플로 실행 시작 시 및 실행 중에 형식이 강력히 지정된 입력을 워크플로 실행에 전달할 수 있습니다. `@Execute` 메서드의 반환 유형은 `void` 또는 `Promise<>`이어야 합니다.

뿐만 아니라 사용자는 워크플로 인터페이스에서 메서드를 선언하여 워크플로 실행의 최신 상태를 보고할 수 있습니다(예: 앞서 본 예시의 `getState` 메서드). 이 상태는 워크플로의 전체 애플리케이션 상태가 아닙니다. 이 기능을 의도적으로 사용함으로써 사용자는 실행의 최신 상태를 나타낼 데이터를 최대 32KB까지 저장할 수 있습니다. 예를 들어 주문 처리 워크플로에서 사용자는 주문 접수, 처리 또는 취소를 나타내는 문자열을 저장할 수 있습니다. 이 메서드는 결정 작업이 완료될 때마다 최신 상태를 가져오기 위해 프레임워크의 호출을 받습니다. 이 상태는 Amazon Simple Workflow Service(SWF)에 저장되어 있어 생성된 외부 클라이언트를 사용하여 가져올 수 있습니다. 이를 통해 사용자는 워크플로 실행의 최신 상태를 확인할 수 있습니다. `@GetState`가 주석으로 달린 메서드에서는 인수를 받아

들여서는 안 되고 void 반환 유형이 있어서는 안 됩니다. 사용자는 이 메서드에서 필요에 따라 어떤 유형도 반환할 수 있습니다. 위 예시에서 MyWorkflowState의 객체(아래 정의 참조)는 문자열 상태 및 숫자 형식의 백분율로 표시되는 완료율을 저장하는 데 사용되는 메서드에서 반환합니다. 이 메서드는 워크플로 구현 객체의 읽기 전용 액세스를 수행할 것으로 예상되고 동기식으로 호출됩니다. 이로써 @Asynchronous라는 주석이 달린 메서드를 호출하는 것과 같은 모든 비동기식 작업의 사용을 허용하지 않습니다. 워크플로 인터페이스에 있는 메서드 중 최대 한 개에 @GetState 주석이 달릴 수 있습니다.

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

이와 마찬가지로 일련의 활동은 @Activities 주석이 붙은 인터페이스를 사용하여 정의됩니다. 인터페이스의 각 메서드는 활동에 해당합니다. 예를 들면 다음과 같습니다.

```
@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);
}
```

인터페이스를 통해 사용자는 관련된 일련의 활동을 함께 그룹화할 수 있습니다. 사용자는 활동 인터페이스 내에 개수에 상관없이 활동을 정의할 수 있고, 활동 인터페이스도 원하는 개수 만큼 정의할 수 있습니다. @Execute 및 @Signal 메서드와 마찬가지로 활동 메서드에서는 Promise<T> 또는 그 파생물이 아닌 유형의 인수를 몇 개든 받아들일 수 있습니다. 활동의 반환 유형은 Promise<T> 또는 그 파생물이어서는 안 됩니다.

워크플로 및 활동 유형 등록

Amazon SWF에서는 활동 및 워크플로 유형을 사용하려면 이를 등록해야 합니다. 프레임워크에서는 사용자가 작업자에 추가하는 구현에 워크플로 및 활동을 자동으로 등록합니다. 프레임워크에서는 워

크플로 및 활동을 구현하는 유형을 찾아 이를 Amazon SWF에 등록합니다. 기본적으로 프레임워크에서는 인터페이스 정의를 사용하여 워크플로 및 활동 유형의 등록 옵션을 유추합니다. 모든 워크플로 인터페이스에는 `@WorkflowRegistrationOptions` 주석 또는 `@SkipRegistration` 주석이 있어야 합니다. 워크플로 작업자는 `@WorkflowRegistrationOptions` 주석이 있는 모든 워크플로 유형을 등록합니다. 이와 마찬가지로 모든 활동 메서드는 `@ActivityRegistrationOptions` 주석 또는 `@SkipRegistration` 주석이 추가되거나 이 두 주석 중 하나가 `@Activities` 인터페이스에 표시되어야 합니다. 활동 작업자는 `@ActivityRegistrationOptions` 주석이 적용되는 모든 활동 유형을 등록합니다. 작업자 중 하나를 시작하면 자동으로 등록이 이루어집니다. `@SkipRegistration` 주석이 있는 워크플로 및 활동 유형은 등록되지 않습니다. `@ActivityRegistrationOptions`및 `@SkipRegistration` 주석에는 재정의 의미론이 있고 가장 구체적인 것이 활동 유형에 적용됩니다.

Amazon SWF에서는 한 번 등록된 유형은 재등록하거나 수정할 수 없다는 점에 유의하십시오. 프레임워크에서는 모든 유형을 등록하려고 할 것입니다. 그러나 이미 등록된 유형의 경우 재등록되지 않고 오류가 보고되지 않습니다.

등록된 설정을 수정해야 하는 경우 해당 유형의 새 버전을 등록해야 합니다. 새 실행을 시작하거나 생성된 클라이언트를 사용하는 활동을 호출할 때 등록된 설정을 재정의할 수도 있습니다.

등록하려면 유형 이름과 몇 가지 다른 등록 옵션이 필요합니다. 기본적인 구현에서는 이를 다음과 같이 결정합니다.

워크플로 유형 이름 및 버전

프레임워크에서는 워크플로 인터페이스에서 워크플로 유형의 이름을 정합니다. 기본 워크플로 유형 이름의 형식은 {*prefix*}{*name*}입니다. {*prefix*}는 `@Workflow` 인터페이스의 이름으로 설정되고 뒤에 '.'가 추가되며 {*name*}은 `@Execute` 메서드의 이름으로 설정됩니다. 앞의 예에서 워크플로 유형의 기본 이름은 `MyWorkflow.startMyWF`입니다. `@Execute` 메서드의 이름 파라미터를 사용하여 기본 이름을 재정의할 수 있습니다. 예시에서 워크플로 유형의 기본 이름은 `startMyWF`입니다. 이 이름은 빈 문자열이어서는 안 됩니다. `@Execute`를 사용하여 이름을 재정의하는 경우 프레임워크에서는 이름에 접두사를 자동으로 붙이지 않는다는 점에 유의하십시오. 고유한 이름 지정 체계를 자유롭게 사용할 수 있습니다.

워크플로 버전은 `@Execute` 주석의 `version` 파라미터를 사용해 지정됩니다. `version`의 기본값은 없고 명시적으로 지정되어야 합니다. `version`은 자유 형식 문자열로서, 고유한 버전 관리 체계를 자유롭게 사용할 수 있습니다.

신호 이름

신호의 이름은 @Signal 주석의 이름 파라미터를 사용해 지정할 수 있습니다. 지정하지 않는 경우 기본값인 신호 메서드의 이름이 지정됩니다.

활동 유형 이름 및 버전

프레임워크에서는 활동 인터페이스에서 활동 유형의 이름을 정합니다. 기본 활동 유형 이름의 형식은 {*prefix*}{*name*}입니다. {*prefix*}는 @Activities 인터페이스의 이름으로 설정되고 뒤에 '.'가 추가되며 {*name*}은 메서드 이름으로 설정됩니다. 기본 {*prefix*}는 활동 인터페이스의 @Activities 주석에서 재정의할 수 있습니다. 또한 활동 메서드의 @Activity 주석을 사용하여 활동 유형 이름을 지정할 수도 있습니다. @Activity를 사용하여 이름을 재정의하는 경우 프레임워크에서는 이름에 접두사를 자동으로 붙이지 않는다는 점에 유의하십시오. 고유한 이름 지정 체계를 자유롭게 사용할 수 있습니다.

활동 버전은 @Activities 주석의 버전 파라미터를 사용해 지정됩니다. 이 버전은 인터페이스에 정의된 모든 활동에 대해 기본값으로 사용되며 @Activity 주석을 사용해 활동별로 재정의할 수 있습니다.

기본 작업 목록

기본 작업 목록은 @WorkflowRegistrationOptions 및 @ActivityRegistrationOptions 주석을 사용하고 defaultTaskList 파라미터를 설정하여 구성할 수 있습니다. 기본적으로 USE_WORKER_TASK_LIST로 설정됩니다. 이것은 활동 또는 워크플로 유형 등록에 사용되는 작업자 객체에 구성된 작업 목록을 사용하도록 프레임워크에게 지시하는 특수 값입니다. 또한 이 주석을 사용해 기본 작업 목록을 NO_DEFAULT_TASK_LIST로 설정하여 기본 작업 목록을 등록하지 않는 쪽을 선택할 수도 있습니다. 이 방법은 작업 목록이 런타임에 지정되도록 하고 싶은 경우에 사용할 수 있습니다. 기본 작업 목록이 등록되지 않은 경우 생성된 클라이언트의 해당되는 메서드 오버로드에 관한 StartWorkflowOptions 및 ActivitySchedulingOptions 파라미터를 사용하여 워크플로를 시작하거나 활동 메서드를 시작할 때 작업 목록을 지정해야 합니다.

기타 등록 옵션

Amazon SWF API에서 허용하는 모든 워크플로 및 활동 유형 등록 옵션은 프레임워크를 통해 지정할 수 있습니다.

워크플로 등록 옵션의 전체 목록은 다음을 참조하십시오.

- @워크플로

- @실행
- @WorkflowRegistrationOptions
- @신호

활동 등록 옵션의 전체 목록은 다음을 참조하십시오.

- @활동
- @활동
- @ActivityRegistrationOptions

유형 등록을 완벽히 제어하고 싶다면 작업자 확장성 단원을 참조하십시오.

활동 및 워크플로 클라이언트

워크플로 및 활동 클라이언트는 `@Workflow` 및 `@Activities` 인터페이스 기반 프레임워크에서 생성합니다. 클라이언트에서만 의미가 있는 메서드와 설정이 포함된 별도의 클라이언트 인터페이스가 생성됩니다. Eclipse를 사용하여 개발 중인 경우 이 작업은 사용자가 적절한 인터페이스가 포함된 파일을 저장할 때마다 Amazon SWF Eclipse 플러그인에서 수행합니다. 생성된 코드는 인터페이스와 동일한 패키지에 있는 프로젝트의 생성된 소스 디렉터리에 배치됩니다.

> **Note**
>
> Eclipse에서 사용하는 기본 디렉터리 이름은 apt_generated라는 점에 유의하십시오. Eclipse에서는 Package Explorer에 이름이 '.'으로 시작하는 디렉터리를 표시하지 않습니다. Project Explorer에서 생성된 파일을 보고 싶다면 다른 디렉터리 이름을 사용하십시오. Eclipse에서 Package Explorer에 있는 패키지를 마우스 오른쪽 버튼으로 클릭한 후 [Properties], [Java Compiler], [Annotation processing]를 선택하고 [Generate source directory] 설정을 수정합니다.

워크플로 클라이언트

워크플로에 대해 생성된 아티팩트에는 클라이언트 측 인터페이스 세 가지와 이를 구현하는 클래스가 포함되어 있습니다. 생성된 클라이언트에 포함된 것은 다음과 같습니다.

- 워크플로 실행을 시작하고 신호를 전송하기 위해 비동기식 메서드를 제공하는 워크플로 구현 내에서 소비될 비동기식 클라이언트

- 실행을 시작하고 신호를 전송하며 워크플로 구현 범위 밖에서 워크플로 상태를 가져오는 데 사용할 수 있는 외부 클라이언트
- 연속 워크플로를 생성하는 데 사용할 수 있는 자체 클라이언트

예를 들어 예시 `MyWorkflow` 인터페이스를 위한 생성된 클라이언트 인터페이스는 다음과 같습니다.

```
//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);
```

인터페이스에는 사용자가 선언한 `@Workflow` 인터페이스에 있는 각 메서드에 상응하는 오버로드된 메서드가 있습니다.

외부 클라이언트에서는 `StartWorkflowOptions`를 받아들이는 `@Execute` 메서드의 추가 오버로드 하나로 `@Workflow` 인터페이스에 있는 메서드를 미러링합니다. 사용자는 새 워크플로 실행을 시작할 때 이 오버로드를 사용하여 추가 옵션을 전달할 수 있습니다. 이 옵션을 통해 기본 작업 목록와 제한 시간 설정을 재정의하고 태그를 워크플로 실행에 연결할 수 있습니다.

한편 비동기식 클라이언트에는 `@Execute` 메서드의 비동기식 호출을 허용하는 메서드가 있습니다. 다음 메서드 오버로드는 워크플로 인터페이스에 있는 `@Execute` 메서드에 대해 클라이언트 인터페이스에서 생성됩니다.

1. 원본 인수를 있는 그대로 받아들이는 오버로드. 이 오버로드의 반환 유형은 원본 메서드에서 `void`를 반환한 경우 `Promise<Void>`가 됩니다. 그렇지 않은 경우에는 원본 메서드에서 선언된 `Promise<>`가 됩니다. 예:

   원본 메서드:

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

   생성된 메서드:

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

   이 오버로드는 워크플로의 모든 인수가 사용 가능하여 대기할 필요가 없을 때 사용해야 합니다.

2. 원본 인수를 있는 그대로 받아들이고 `Promise<?>` 유형의 추가 가변 인수를 받아들이는 오버로드. 이 오버로드의 반환 유형은 원본 메서드에서 `void`를 반환한 경우 `Promise<Void>`가 됩니다. 그렇지 않은 경우에는 원본 메서드에서 선언된 `Promise<>`가 됩니다. 예:

   원본 메서드:

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

   생성된 메서드:

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

이 오버로드는 워크플로의 모든 인수가 사용 가능하여 대기할 필요가 없지만 사용자는 일부 다른 약속이 준비 상태가 되기를 기다리고 싶은 경우에 사용해야 합니다. 가변 인수는 이처럼 인수로 선언되지 않았지만 사용자가 호출이 실행될 때까지 기다리기 원하는 `Promise<?>` 객체를 전달하는 데 사용할 수 있습니다.

3. 원본 인수를 있는 그대로 받아들이고 `StartWorkflowOptions` 유형의 추가 인수 및 `Promise<?>` 유형의 추가 가변 인수를 받아들이는 오버로드. 이 오버로드의 반환 유형은 원본 메서드에서 `void`를 반환한 경우 `Promise<Void>`가 됩니다. 그렇지 않은 경우에는 원본 메서드에서 선언된 `Promise<>`가 됩니다. 예:

   원본 메서드:

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

   생성된 메서드:

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

   이 오버로드는 워크플로의 모든 인수가 사용 가능하여 대기할 필요가 없는 경우, 워크플로 실행을 시작하는 데 사용되는 기본 설정을 재정의하고 싶은 경우, 또는 일부 다른 약속이 준비 상태가 되기를 기다리고 싶은 경우에 사용해야 합니다. 가변 인수는 이처럼 인수로 선언되지 않았지만 사용자가 호출이 실행될 때까지 기다리기 원하는 `Promise<?>` 객체를 전달하는 데 사용할 수 있습니다.

4. 원본 메서드의 각 인수가 `Promise<>` 래퍼로 대치된 오버로드. 이 오버로드의 반환 유형은 원본 메서드에서 `void`를 반환한 경우 `Promise<Void>`가 됩니다. 그렇지 않은 경우에는 원본 메서드에서 선언된 `Promise<>`가 됩니다. 예:

   원본 메서드:

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

   생성된 메서드:

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

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

이 오버로드는 워크플로 실행으로 전달될 인수가 비동기식으로 평가되어야 하는 경우에 사용해야 합니다. 이 메서드 오버로드에 대한 호출은 전달된 모든 인수가 준비 상태가 될 때까지는 실행되지 않습니다.

일부 인수가 이미 준비 상태가 되었다면 이 인수를 `Promise.asPromise(value)` 메서드를 통해 이미 준비 상태가 된 `Promise`로 변환합니다. 예:

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

5. 원본 메서드의 각 인수가 `Promise<>` 래퍼로 대치된 오버로드. 또한 이 오버로드에는 `Promise<?>` 유형의 추가 가변 인수가 있습니다. 이 오버로드의 반환 유형은 원본 메서드에서 `void`를 반환한 경우 `Promise<Void>`가 됩니다. 그렇지 않은 경우에는 원본 메서드에서 선언된 `Promise<>`가 됩니다. 예:

   원본 메서드:

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

   생성된 메서드:

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

   이 오버로드는 워크플로 실행으로 전달되어야 할 인수가 비동기식으로 평가되어야 하고, 아울러 사용자는 일부 다른 약속이 준비 상태가 되기를 기다리고 싶은 경우에 사용해야 합니다. 이 메서드 오버로드에 대한 호출은 전달된 모든 인수가 준비 상태가 될 때까지는 실행되지 않습니다.

6. 원본 메서드의 각 인수가 `Promise<?>` 래퍼로 대치된 오버로드. 또한 이 오버로드에는 `StartWorkflowOptions` 유형의 추가 인수와 `Promise<?>` 유형의 가변 인수가 있습니다. 이 오버로드의 반환 유형은 원본 메서드에서 `void`를 반환한 경우 `Promise<Void>`가 됩니다. 그렇지 않은 경우에는 원본 메서드에서 선언된 `Promise<>`가 됩니다. 예:

   원본 메서드:

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

생성된 메서드:

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

이 오버로드는 워크플로 실행으로 전달되어야 할 인수가 비동기식으로 평가되고 사용자는 워크플로 실행을 시작하는 데 사용되는 기본 설정을 재정의하고 싶은 경우에 사용합니다. 이 메서드 오버로드에 대한 호출은 전달된 모든 인수가 준비 상태가 될 때까지는 실행되지 않습니다.

워크플로 인터페이스의 각 신호에 해당하는 메서드도 생성됩니다. 예를 들면 다음과 같습니다.

원본 메서드:

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

생성된 메서드:

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

비동기식 클라이언트에는 원본 인터페이스의 `@GetState`라는 주석이 달린 메서드에 상응하는 메서드가 포함되어 있지 않습니다. 상태를 조회하려면 웹 서비스 호출이 필요하므로 워크플로 내에서 사용하기에는 적합하지 않습니다. 따라서 외부 클라이언트를 통해서만 제공됩니다.

자체 클라이언트는 현재 실행이 완료되면 새 실행을 시작하는 용도로 워크플로 내에서 사용하도록 마련된 것입니다. 이 클라이언트의 메서드는 비동기식 클라이언트에 있는 메시드와 비슷하지만 `void`를 반환합니다. 이 클라이언트에는 `@Signal` 및 `@GetState`라는 주석이 달린 메서드에 상응하는 메서드가 없습니다. 자세한 내용은 [연속 워크플로](#) 단원을 참조하십시오.

생성된 클라이언트는 기본 인터페이스인 `WorkflowClient` 및 `WorkflowClientExternal`에서 각각 파생됩니다. 이 인터페이스에서는 워크플로 실행을 취소 또는 종료하는 데 사용할 수 있는 메서드를 제공합니다. 이러한 인터페이스에 관한 자세한 내용은 AWS SDK for Java 설명서를 참조하십시오.

생성된 클라이언트를 통해 사용자는 형식이 강력히 지정된 방식으로 워크플로 실행과 상호 작용할 수 있습니다. 생성된 클라이언트의 인스턴스는 생성된 후에 특정 워크플로 실행에 묶여 그 실행에 대해서만 사용할 수 있게 됩니다. 뿐만 아니라 프레임워크에서도 워크플로 유형 또는 실행에 고유하지 않은 동적 클라이언트를 제공합니다. 생성된 클라이언트는 보이지 않는 이면에서는 이 클라이언트에 의존합니다. 사용자는 이 클라이언트를 직접 사용할 수도 있습니다. 동적 클라이언트 단원을 참조하십시오.

또한 프레임워크에서는 형식이 강력히 지정된 클라이언트를 생성하기 위한 팩토리를 생성합니다. 예시 `MyWorkflow` 인터페이스를 위한 생성된 클라이언트 팩토리는 다음과 같습니다.

```
//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);
}
```

`WorkflowClientFactory` 기본 인터페이스는 다음과 같습니다.

```
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);
}
```

클라이언트의 인스턴스를 생성하려면 이 팩토리를 사용해야 합니다. 팩토리를 사용하면 일반 클라이언트(사용자 지정 클라이언트 구현을 제공하는 데 일반 클라이언트를 사용해야 함)와 클라이언트가 데이터를 마샬링하는 데 사용하는 `DataConverter` 및 워크플로 실행을 시작하는 데 사용되는 옵션을 구성할 수 있습니다. 자세한 내용은 DataConverters 및 하위 워크플로 실행 단원을 참조하십시오. `StartWorkflowOptions`에는 등록 시 지정된 기본값(예: 시간 초과)을 재정의하는 데 사용할 수 있는 설정이 포함되어 있습니다. `StartWorkflowOptions` 클래스에 관한 자세한 내용은 AWS SDK for Java 설명서를 참조하십시오.

외부 클라이언트는 워크플로 범위 밖에서 워크플로 실행을 시작하는 데 사용할 수 있는 반면, 비동기식 클라이언트는 워크플로 내에서 코드로부터 워크플로 실행을 시작하는 데 사용할 수 있습니다. 실행을 시작하려면 생성된 클라이언트를 사용하여 워크플로 인터페이스에서 `@Execute`라는 주석이 붙은 메서드에 상응하는 메서드를 호출하기만 하면 됩니다.

또한 프레임워크에서는 클라이언트 인터페이스에 대해 구현 클래스를 생성합니다. 이 클라이언트에서는 Amazon SWF에 대한 요청을 생성 및 전송하여 적절한 작업을 수행합니다. `@Execute` 메서드의 클라이언트 버전에서는 새 워크플로 실행을 시작하거나 Amazon SWF API를 사용하여 하위 워크플로 실행을 생성합니다. 이와 유사하게 `@Signal` 메서드의 클라이언트 버전에서는 Amazon SWF API를 사용하여 신호를 전송합니다.

> **Note**
>
> 외부 워크플로 클라이언트는 Amazon SWF 클라이언트 및 도메인으로 구성해야 합니다. 이를 파라미터로 받아들이는 클라이언트 팩토리 생성자를 사용하거나 이미 Amazon SWF 클라이언트 및 도메인으로 구성된 일반 클라이언트 구현에서 전달할 수 있습니다.
> 프레임워크는 워크플로 인터페이스의 유형 계층 구조를 탐색하고, 또한 상위 워크플로 인터페이스에 대해 클라이언트 인터페이스를 생성하며 그로부터 파생됩니다.

활동 클라이언트

워크플로 클라이언트와 유사하게 클라이언트는 @Activities라는 주석이 붙은 각 인터페이스에 대해 생성됩니다. 생성된 아티팩트에는 클라이언트 측 인터페이스와 클라이언트 클래스가 포함되어 있습니다. 위의 예시 @Activities 인터페이스를 위한 생성된 인터페이스(MyActivities)는 다음과 같습니다.

```
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);
}
```

인터페이스에는 @Activities 인터페이스에 있는 각 활동 메서드에 상응하는 일련의 오버로드된 메서드가 포함되어 있습니다. 이 오버로드는 편의상 제공되는 것이며 활동을 비동기식으로 호출할 수 있게 해줍니다. @Activities 인터페이스의 각 활동 메서드에 대해 다음과 같은 메서드 오버로드가 클라이언트 인터페이스에 생성됩니다.

1. 원본 인수를 있는 그대로 받아들이는 오버로드. 이 오버로드의 반환 유형은 Promise<*T*>이며, 여기에서 *T*는 원본 메서드의 반환 유형입니다. 예:

   원본 메서드:

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

   생성된 메서드:

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

이 오버로드는 워크플로의 모든 인수가 사용 가능하여 대기할 필요가 없을 때 사용해야 합니다.

2. 원본 인수를 있는 그대로 받아들이고 `ActivitySchedulingOptions` 유형의 인수 및 `Promise<?>` 유형의 추가 가변 인수를 받아들이는 오버로드. 이 오버로드의 반환 유형은 `Promise<T>`이며, 여기에서 *T*는 원본 메서드의 반환 유형입니다. 예:

   원본 메서드:

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

   생성된 메서드:

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

   이 오버로드는 워크플로의 모든 인수가 사용 가능하여 대기할 필요가 없는 경우, 기본 설정을 재정의하고 싶은 경우, 또는 추가 `Promise`가 준비 상태가 되기를 기다리고 싶은 경우에 사용해야 합니다. 가변 인수는 이처럼 인수로 선언되지 않았지만 사용자가 호출이 실행될 때까지 기다리기 원하는 추가 `Promise<?>` 객체를 전달하는 데 사용할 수 있습니다.

3. 원본 메서드의 각 인수가 `Promise<>` 래퍼로 대치된 오버로드. 이 오버로드의 반환 유형은 `Promise<T>`이며, 여기에서 *T*는 원본 메서드의 반환 유형입니다. 예:

   원본 메서드:

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

   생성된 메서드:

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

   이 오버로드는 활동으로 전달될 인수가 비동기식으로 평가되는 경우에 사용해야 합니다. 이 메서드 오버로드에 대한 호출은 전달된 모든 인수가 준비 상태가 될 때까지는 실행되지 않습니다.

4. 원본 메서드의 각 인수가 `Promise<>` 래퍼로 대치된 오버로드. 또한 이 오버로드에는 `ActivitySchedulingOptions` 유형의 추가 인수와 `Promise<?>` 유형의 가변 인수가 있습니다. 이 오버로드의 반환 유형은 `Promise<T>`이며, 여기에서 *T*는 원본 메서드의 반환 유형입니다. 예:

   원본 메서드:

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

   생성된 메서드:

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

   이 오버로드는 활동으로 전달될 인수가 비동기식으로 평가되는 경우, 해당 유형으로 등록된 기본 설정을 재정의하고 싶은 경우, 또는 추가 `Promise`가 준비 상태가 되기를 기다리고 싶은 경우에 사용해야 합니다. 이 메서드 오버로드에 대한 호출은 전달된 모든 인수가 준비 상태가 될 때까지는 실행되지 않습니다. 생성된 클라이언트 클래스에서는 이 인터페이스를 구현합니다. 각 인터페이스 메서드의 구현에서는 Amazon SWF에 대한 요청을 생성 및 전송하여 Amazon SWF API를 사용하는 적절한 유형의 활동 작업을 예약합니다.

5. 원본 인수를 있는 그대로 받아들이고 `Promise<?>` 유형의 추가 가변 인수를 받아들이는 오버로드. 이 오버로드의 반환 유형은 `Promise<T>`이며, 여기에서 *T*는 원본 메서드의 반환 유형입니다. 예:

   원본 메서드:

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

   생성된 메서드:

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

   이 오버로드는 활동의 모든 인수가 사용 가능하여 대기할 필요가 없지만 사용자는 `Promise` 객체가 준비 상태가 되기를 기다리고 싶은 경우에 사용해야 합니다.

6. 원본 메서드에 있는 각 인수가 `Promise` 래퍼로 교체되고 `Promise<?>` 유형의 추가 가변 인수가 있는 오버로드. 이 오버로드의 반환 유형은 `Promise<T>`이며, 여기에서 *T*는 원본 메서드의 반환 유형입니다. 예:

    원본 메서드:

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

    생성된 메서드:

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

    이 오버로드는 활동의 모든 인수가 비동기식으로 대기되고 사용자도 일부 기타 `Promise`가 준비 상태가 되기를 기다리고 싶은 경우에 사용해야 합니다. 이 메서드 오버로드에 대한 호출은 전달된 모든 `Promise` 객체가 준비 상태가 될 때까지 비동기식으로 실행됩니다.

또한 생성된 활동 클라이언트에는 모든 활동 오버로드가 호출하는 `{`*`activity method name`*`}Impl()`이라는, 각 활동 메서드에 상응하는 보호 받는 메서드가 있습니다. 사용자는 이 메서드를 재정의하여 모의 클라이언트 구현을 생성할 수 있습니다. 이 메서드에서 받아들이는 인수: `Promise<>` 래퍼의 원본 메서드에 대한 모든 인수, `ActivitySchedulingOptions`, `Promise<?>` 유형의 가변 인수. 예:

원본 메서드:

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

생성된 메서드:

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

예약 옵션

생성된 활동 클라이언트를 통해 ActivitySchedulingOptions를 인수로 전달할 수 있습니다. ActivitySchedulingOptions 구조에는 프레임워크가 Amazon SWF에서 예약하는 활동 작업의 구성을 결정하는 설정이 포함되어 있습니다. 이 설정에서는 등록 옵션으로 지정된 기본값을 재정의합니다. 예약 옵션을 동적으로 지정하려면 ActivitySchedulingOptions 객체를 생성하고 이를 원하는 대로 구성한 후 활동 메서드에 전달합니다. 다음 예에서는 활동 작업에 사용해야 할 작업 목록을 지정하였습니다. 이렇게 하면 이 활동 호출에 대해 기본 등록 작업 목록이 재정의됩니다.

```
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);
    }
}
```

동적 클라이언트

프레임워크는 생성된 클라이언트 외에도 워크플로 실행을 동적으로 시작하고, 신호를 보내고, 활동을 예약하는 데 사용할 수 있는 범용 클라이언트(DynamicWorkflowClient 및 DynamicActivityClient)도 제공합니다. 예를 들어 설계 시에는 그 유형이 알려지지 않은 활동을 예약하고 싶은 경우가 있을 수 있습니다. 이때는 그러한 활동 작업을 예약하기 위해 DynamicActivityClient를 사용할 수 있습니다. 이와 유사한 방법으로 DynamicWorkflowClient를 사용하여 하위 워크플로 실행을 동적으로 예약할 수 있습니다. 다음 예

에서 워크플로는 다음과 같이 데이터베이스에서 활동을 검색하여 이를 예약하기 위해 동적 활동 클라이언트를 사용합니다.

```
//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);
}
```

자세한 내용은 AWS SDK for Java 설명서를 참조하십시오.

워크플로 실행에 신호 전송 및 워크플로 실행 취소

생성된 워크플로 클라이언트에는 워크플로로 전송할 수 있는 각 신호에 상응하는 메서드가 있습니다. 워크플로에서 이 메서드를 사용하여 다른 워크플로 실행으로 신호를 보낼 수 있습니다. 이로써 신호 전송에 사용할 수 있는 형식 지정 메커니즘을 얻을 수 있습니다. 하지만 신호 이름을 메시지로 수신하는 경우와 같이 신호 이름을 동적으로 결정해야 하는 경우도 있습니다. 동적 워크플로 클라이언트를 사용하여 어떤 워크플로 실행에도 동적으로 신호를 보낼 수 있습니다. 이와 유사한 방법으로 클라이언트를 사용하여 다른 워크플로 실행의 취소를 요청할 수 있습니다.

다음 예에서 워크플로는 데이터베이스에서 신호를 전송할 실행을 검색하여 동적 워크플로 클라이언트를 사용해 동적으로 신호를 전송합니다.

```
//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);
}
```

워크플로 구현

워크플로를 구현하려면 원하는 `@Workflow` 인터페이스를 구현하는 클래스를 작성합니다. 인스턴스의 경우 예시 워크플로 인터페이스(`MyWorkflow`)는 다음과 같이 구현될 수 있습니다.

```
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);
  }
}
```

이 클래스의 @Execute 메서드는 워크플로 로직의 진입점입니다. 결정 작업이 처리되어야 하는 시점에 프레임워크에서는 다시 재생을 사용하여 객체 상태를 재구성하므로 각 결정 작업마다 새 객체가 생성됩니다.

Promise<*T*>를 파라미터로 사용하는 것은 @Workflow 인터페이스 내 @Execute 메서드에서 허용되지 않습니다. 이는 비동기식 호출이 순전히 호출자의 결정이기 때문입니다. 워크플로 구현 자체는 호출이 동기식이었는지 비동기식이었는지 여부에 의존하지 않습니다. 따라서 생성된 클라이언트 인터페이스에는 이 메서드가 비동기식으로 호출될 수 있도록 Promise<*T*> 파라미터를 받아들이는 오버로드가 있습니다.

@Execute 메서드의 반환 유형은 void 또는 Promise<*T*>만 가능합니다. 해당하는 외부 클라이언트의 반환 유형은 Promise<>가 아닌 void임에 유의하십시오. 외부 클라이언트는 비동기식 코드에서 사용하기 위한 것이 아니므로 외부 클라이언트에서는 Promise 객체를 반환하지 않습니다. 외부로 표시된 워크플로 실행의 결과를 얻으려면 워크플로가 활동을 통해 외부 데이터 스토어에서 상태를 업데이트하도록 설계할 수 있습니다. 또한 Amazon SWF의 시각화 API는 진단 목적으로 워크플로의 결과를 가져오는 데 사용할 수 있습니다. 워크플로 실행의 결과를 가져오는 데 시각화 API를 사용하는 것을 일반적인 관례로 삼지 않는 것이 좋습니다. 왜냐하면 이러한 API 직접 호출은 Amazon SWF에서 제한이 발생할 수 있기 때문입니다. 시각화 API를 사용하려면 WorkflowExecution 구조를 사용하는 워크플로 실행을 식별해야 합니다. 이 구조는 생성된 워크플로 클라이언트에서 getWorkflowExecution 메서드를 호출하여 얻을 수 있습니다. 그러면 이 메서드에서는 클라이언트가 묶인 워크플로 실행에 상응하는 WorkflowExecution 구조를 반환합니다. 시각화 API에 대한 자세한 내용은 Amazon Simple Workflow Service API 참조를 참조하십시오.

워크플로 구현에서 활동을 호출할 때는 생성된 활동 클라이언트를 사용해야 합니다. 이와 마찬가지로 신호를 전송하려면 생성된 워크플로 클라이언트를 사용합니다.

결정 컨텍스트

프레임워크에서는 워크플로 코드가 프레임워크에서 실행될 때마다 주변 컨텍스트를 제공합니다. 이 컨텍스트에서는 타이머 생성과 같은 워크플로 구현에서 액세스할 수 있는 컨텍스트 고유 기능을 제공합니다. 자세한 내용은 실행 콘텍스트 단원을 참조하십시오.

실행 상태 노출

Amazon SWF를 통해 사용자는 워크플로 내역에서 사용자 지정 상태를 추가할 수 있습니다. 워크플로 실행에서 보고하는 최신 상태는 Amazon SWF 서비스에 대한 시각화 호출을 통해 Amazon SWF 콘솔에서 사용자에게 반환됩니다. 예를 들어 주문 처리 워크플로에서는 '주문 접수', '주문 배송' 등 여러 단계에서 주문 상태를 보고할 수 있습니다. AWS Flow Framework for Java에서는 이 작업이 @GetState 주석이 붙은 워크플로 인터페이스에서 메서드를 통해 수행됩니다. 결정자는 결정 작업 처리를 마치면

이 메서드를 호출하여 워크플로 구현에서 최신 상태를 조회합니다. 상태는 시각화 호출 외에도 생성된 외부 클라이언트(내부적으로는 시각화 API를 사용함)를 사용하여 조회할 수 있습니다.

다음 예시에서는 실행 컨텍스트를 설정하는 방법을 보여줍니다.

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

생성된 외부 클라이언트는 언제든지 워크플로 실행의 최신 상태를 조회하는 데 사용할 수 있습니다.

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

위 예시에서 실행 상태는 여러 단계에서 보고됩니다. 워크플로 인스턴스가 시작되면 `periodicWorkflow`에서는 초기 상태를 '방금 시작됨'으로 보고합니다. 그다음에는

callPeriodicActivity에 대한 각 호출을 통해 워크플로 상태를 업데이트합니다. activity1이 100번 호출되었다면 메서드가 반환하고 워크플로 인스턴스가 완료됩니다.

워크플로 로컬

때로 워크플로 구현에서 정적 변수를 사용해야 하는 경우가 있을 수 있습니다. 예를 들어 워크플로 구현의 여러 위치(여러 클래스일 수 있음)에서 액세스해야 하는 카운터를 저장하고 싶을 수 있습니다. 하지만 정적 변수가 스레드 전체에 공유되므로 워크플로에서 정적 변수에 의존할 수는 없습니다. 작업자가 다른 스레드에 있는 다른 결정 작업을 동시에 처리할 수 있기 때문에 이것은 문제가 됩니다. 또는 워크플로 구현에서 필드에 그러한 상태를 저장할 수 있지만, 이때 구현 객체를 두루 전달해야 합니다. 이러한 필요에 대처하기 위해 프레임워크에서는 WorkflowExecutionLocal<?> 클래스를 제공합니다. 의미론과 같은 정적 변수가 있어야 하는 모든 상태는 WorkflowExecutionLocal<?>을 사용하는 인스턴스 로컬로 보관되어야 합니다. 사용자는 이 유형의 정적 변수를 선언하고 사용할 수 있습니다. 예를 들어 다음 코드 조각에서 WorkflowExecutionLocal<String>은 사용자 이름을 저장하는 데 사용됩니다.

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

활동 구현

활동은 `@Activities` 인터페이스의 구현을 제공함으로써 구현됩니다. AWS Flow Framework for Java에서는 작업자에 구성된 활동 구현 인스턴스를 사용하여 실행 시간에 활동 작업을 처리합니다. 작업자는 적절한 유형의 활동 구현을 자동으로 검색합니다.

사용자는 속성 및 필드를 사용하여 데이터베이스 연결과 같은 활동 인스턴스로 리소스를 전달할 수 있습니다. 활동 구현 객체는 여러 스레드에서 액세스할 수 있기 때문에 공유 리소스는 스레드 세이프여야 합니다.

활동 구현에서는 `Promise<>` 유형의 파라미터를 받아들이거나 이 유형의 객체를 반환하지 않는다는 점에 유의하십시오. 이는 활동의 구현이 간접적으로 호출된 방식(동기식 또는 비동기식)에 의존해서는 안 되기 때문입니다.

앞서 설명한 활동 인터페이스는 다음과 같이 구현될 수 있습니다.

```
public class MyActivitiesImpl implements MyActivities {

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

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

스레드 로컬 컨텍스트는 사용 중인 작업 객체, 데이터 변환기 객체 등을 가져오는 데 사용할 수 있는 활동 구현에 대해 사용할 수 있습니다. 현재 컨텍스트는 `ActivityExecutionContextProvider.getActivityExecutionContext()`를 통해 액세스할

수 있습니다. 자세한 내용은 AWS SDK for Java 설명서에서 ActivityExecutionContext 및 실행 콘텍스트 단원을 참조하십시오.

활동을 수동으로 완료

위 예시의 @ManualActivityCompletion 주석은 선택 사항입니다. 이 주석은 활동을 구현하는 메서드에서만 허용되며 활동 메서드에서 결과가 반환될 때 활동이 자동으로 완료되지 않도록 구성하는 데 사용됩니다. 이는 활동을 비동기적으로 완료하려는 경우(예: 사람의 작업이 완료된 후 수동으로 완료하려는 경우)에 유용할 수 있습니다.

활동 메서드에서 결과를 반환하면 프레임워크에서는 기본적으로 완료된 활동을 고려합니다. 이는 활동 작업자가 활동 작업 완료를 Amazon SWF에 보고하고 결과를 제공함을 뜻합니다(있는 경우). 그러나 활동 메서드에서 결과가 반환될 때 활동 작업이 완료됨으로 표시되는 것을 사용자가 원하지 않는 사용 사례가 있습니다. 이 기능은 인간 작업을 모델링할 때 특히 유용합니다. 예를 들어 활동 메서드에서는 활동 작업이 완료되기 전에 일부 작업을 완료해야 하는 사람에게 이메일을 전송할 수 있습니다. 이러한 경우 사용자는 활동 메서드에 @ManualActivityCompletion 주석을 붙여 활동 작업자에게 활동을 자동으로 완료해서는 안 된다고 알릴 수 있습니다. 활동을 수동으로 완료하기 위해 프레임워크에서 제공하는 ManualActivityCompletionClient를 사용하거나 Amazon SWF SDK에서 제공하는 Amazon SWF Java 클라이언트에 있는 RespondActivityTaskCompleted 메서드를 사용할 수 있습니다. 자세한 내용은 AWS SDK for Java 설명서를 참조하십시오.

활동 작업을 완료하려면 작업 토큰을 제공해야 합니다. 작업 토큰은 Amazon SWF에서 작업을 고유하게 식별하는 데 사용됩니다. 활동 구현의 ActivityExecutionContext에서 이 토큰에 액세스할 수 있습니다. 이 토큰을 작업을 완료할 책임이 있는 당사자에게 전달해야 합니다. 이 토큰은 ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken()을 호출하여 ActivityExecutionContext에서 가져올 수 있습니다.

다음과 같이 Hello World 예시의 getName 활동을 구현하여 누군가에게 인사말 메시지를 제공하도록 요청하는 이메일을 보낼 수 있습니다.

```
@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";
}
```

다음 코드 조각은 인사말을 제공하고 ManualActivityCompletionClient를 사용하여 작업을 종료하는 데 사용할 수 있습니다. 또는 다음과 같이 작업을 실패 처리할 수도 있습니다.

```
public class CompleteActivityTask {

    public void completeGetNameActivity(String taskToken) {

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

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

AWS Lambda 작업 구현

주제

- AWS Lambda 정보
- Lambda 작업 사용의 이점 및 제한 사항
- AWS Flow Framework for Java 워크플로에서 Lambda 작업 사용
- HelloLambda 샘플 보기

AWS Lambda 정보

AWS Lambda는 사용자 지정 코드로 생성된 이벤트에 대한 응답으로 또는 Amazon S3, DynamoDB, Amazon Kinesis, Amazon SNS 및 Amazon Cognito 등 다양한 AWS 서비스에서 코드를 실행하는 완전 관리형 컴퓨팅 서비스입니다. Lambda에 대한 자세한 내용은 AWS Lambda 개발자 가이드를 참조하세요.

Amazon Simple Workflow Service는 Lambda 작업을 제공하므로 기존 Amazon SWF 활동 대신 또는 이러한 활동과 함께 Lambda 함수를 실행할 수 있습니다.

Important

Amazon SWF에서 자동으로 실행한 Lambda 실행(요청)에 대한 비용이 AWS 계정에 청구됩니다. Lambda 요금에 대한 자세한 내용은 https://aws.amazon.com/lambda/pricing/을 참조하십시오.

Lambda 작업 사용의 이점 및 제한 사항

일반적인 Amazon SWF 활동 대신 Lambda 작업을 사용하면 다음과 같은 여러 가지 이점이 있습니다.

- Lambda 작업은 Amazon SWF 활동 유형처럼 등록하거나 버전을 관리할 필요가 없습니다.
- 워크플로에 이미 정의해 둔 기존 Lambda 함수는 어느 것이나 사용할 수 있습니다.
- Lambda 함수는 Amazon SWF에서 직접 호출하기 때문에 일반적인 활동에 대해 수행해야 하는 것처럼 작업을 실행할 작업자 프로그램을 구현할 필요가 없습니다.
- Lambda에서는 함수 실행을 추적 및 분석할 수 있도록 측정치 및 로그를 제공합니다.

또한 Lambda 작업과 관련해서는 반드시 알고 있어야 하는 여러 가지 제한 사항이 있습니다.

- Lambda 작업은 Lambda를 지원하는 AWS 리전에서만 실행할 수 있습니다. 현재 Lambda를 지원하는 리전에 대한 자세한 내용은 Amazon Web Services 일반 참조의 Lambda 리전 및 엔드포인트를 참조하십시오.
- Lambda 작업은 현재 기본 SWF HTTP API 및 Java용 AWS Flow Framework에서만 지원됩니다. Ruby용 AWS Flow Framework에서는 현재 Lambda 작업이 지원되지 않습니다.

AWS Flow Framework for Java 워크플로에서 Lambda 작업 사용

AWS Flow Framework for Java 워크플로에서 Lambda 작업을 사용하는 것과 관련해 다음과 같은 세 가지 요구 사항이 있습니다.

- 실행할 Lambda 함수입니다. 정의한 Lambda 함수를 자유롭게 사용할 수 있습니다. Lambda 함수를 생성하는 방법에 대한 자세한 내용은 AWS Lambda 개발자 안내서를 참조하십시오.
- Amazon SWF 워크플로에서 Lambda 함수를 실행할 수 있도록 액세스 권한을 제공하는 IAM 역할.
- 워크플로 내에서 Lambda 작업을 예약할 수 있는 코드.

IAM 역할 설정

Amazon SWF에서 Lambda 함수를 간접적으로 호출하려면 Amazon SWF에서 Lambda에 대한 액세스를 제공하는 IAM 역할을 제공해야 합니다. 다음 작업 중 하나를 수행할 수 있습니다.

- 사전 정의된 역할인 AWSLambdaRole을 선택해 워크플로에 계정과 연결된 모든 Lambda 함수를 간접적으로 호출할 수 있는 권한을 부여합니다.
- 자체 정책 및 연결된 역할을 정의해 Amazon 리소스 이름(ARN)으로 지정된 특정 Lambda 함수를 간접적으로 호출하는 워크플로 권한을 부여합니다.

IAM 역할에 대한 권한 제한

리소스 신뢰 정책의 `SourceArn` 및 `SourceAccount` 컨텍스트 키를 사용하여 Amazon SWF에 제공하는 IAM 역할에 대한 권한을 제한할 수 있습니다. 이러한 키는 지정된 도메인 ARN에 속하는 Amazon Simple Workflow Service 실행에서만 사용하도록 IAM 정책의 사용을 제한합니다. 두 글로벌 조건 컨텍스트 키를 모두 사용하는 경우 `aws:SourceAccount` 값과 `aws:SourceArn` 값에서 참조되는 계정은 동일한 정책 문에서 사용할 경우 동일한 계정 ID를 사용해야 합니다.

다음 신뢰 정책 예에서는 `SourceArn` 컨텍스트 키를 사용하여 IAM 서비스 역할이 `123456789012` 계정의 `someDomain`에 속하는 Amazon Simple Workflow Service 실행에서만 사용되도록 제한합니다.

```
{
  "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"
        }
       }
     }
   ]
}
```

다음 신뢰 정책 예에서는 `SourceAccount` 컨텍스트 키를 사용하여 IAM 서비스 역할을 123456789012 계정의 Amazon Simple Workflow Service 실행에서만 사용하도록 제한합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
     {
       "Sid": "",
       "Effect": "Allow",
       "Principal": {
         "Service": "swf.amazonaws.com"
       },
       "Action": "sts:AssumeRole",
       "Condition": {
         "StringLike": {
           "aws:SourceAccount": "123456789012"
       }
      }
     }
   ]
}
```

Amazon SWF에 Lambda 역할을 간접 호출하기 위한 액세스 권한 제공

사전 정의된 역할인 AWSLambdaRole을 사용해 Amazon SWF 워크플로에 계정과 연결된 모든 Lambda 함수를 간접적으로 호출할 수 있는 권한을 부여합니다.

AWSLambdaRole을 사용해 Amazon SWF에 Lambda 함수를 간접적으로 호출할 수 있는 액세스 권한을 부여하려면

1. [Amazon IAM 콘솔](https://console.aws.amazon.com/iam/)을 엽니다.

2. [Roles]를 선택한 다음 [Create New Role]을 선택합니다.
3. `swf-lambda`와 같이 역할에 이름을 지정한 다음 [Next Step]을 선택합니다.
4. AWS 서비스 역할에서 Amazon SWF를 선택하고 다음 단계를 선택합니다.
5. [Attach Policy] 화면의 목록에서 [AWSLambdaRole]을 선택합니다.
6. 역할을 검토한 후 [Next Step]을 선택하고 [Create Role]을 선택합니다.

특정 Lambda 함수를 간접 호출할 액세스 권한을 제공하는 IAM 역할 정의

워크플로에서 특정 Lambda 함수를 간접적으로 호출하는 액세스 권한을 제공하려는 경우 자체 IAM 정책을 정의해야 합니다.

특정 Lambda 함수에 대한 액세스 권한을 부여하는 IAM 정책을 생성하려면

1. Amazon IAM 콘솔을 엽니다.
2. [Policies]를 선택한 다음 [Create Policy]를 선택합니다.
3. [Copy an AWS Managed Policy]를 선택하고 목록에서 [AWSLambdaRole]을 선택합니다. 정책이 자동으로 생성됩니다. 경우에 따라 필요에 맞춰 정책 이름 및 설명을 편집합니다.
4. 정책 문서의 리소스 필드에 Lambda 함수의 ARN을 추가합니다. 예:

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

> Note
>
> IAM 역할에서 리소스를 지정하는 방법에 대한 전체 설명은 IAM 사용의 IAM 정책 개요를 참조하십시오.

5. [Create policy]를 선택하여 정책 생성을 마칩니다.

그런 다음 새 IAM 역할을 생성할 때 이 정책을 선택하고 해당 역할을 사용해 Amazon SWF 워크플로에 간접 호출 액세스 권한을 부여합니다. 이 절차는 AWSLambdaRole 정책을 사용해 역할을 생성하는 것과 매우 유사합니다. 대신 역할을 생성할 때 고유한 정책을 선택합니다.

Lambda 정책을 사용하여 Amazon SWF 역할을 생성하려면

1. Amazon IAM 콘솔을 엽니다.
2. [Roles]를 선택한 다음 [Create New Role]을 선택합니다.
3. `swf-lambda-function`와 같이 역할에 이름을 지정한 다음 [Next Step]을 선택합니다.
4. AWS 서비스 역할에서 Amazon SWF를 선택하고 다음 단계를 선택합니다.
5. 정책 연결 화면의 목록에서 Lambda 함수 관련 정책을 선택합니다.
6. 역할을 검토한 후 [Next Step]을 선택하고 [Create Role]을 선택합니다.

실행을 위해 Lambda 작업을 예약하려면

Lambda 함수를 간접적으로 호출할 수 있게 해주는 IAM 역할을 정의했다면 이 함수가 워크플로의 일부로 실행되도록 예약할 수 있습니다.

> Note
>
> 이 프로세스는 AWS SDK for Java의 HelloLambda 샘플을 통해 완전히 입증되었습니다.

실행을 위해 Lambda 작업을 예약하려면

1. 워크플로 구현에서 `DecisionContext` 인스턴스의 `getLambdaFunctionClient()`를 직접적으로 호출하여 `LambdaFunctionClient`의 인스턴스를 얻습니다.

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

2. `LambdaFunctionClient`의 `scheduleLambdaFunction()` 메서드를 사용하여 작업을 예약하고 생성한 Lambda 함수의 이름과 Lambda 작업을 위한 모든 입력 데이터를 전달합니다.

```
// 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. 워크플로 실행 시작자에서 `StartWorkflowOptions.withLambdaRole()`을 사용하여 IAM Lambda 역할을 기본 워크플로 옵션에 추가한 후 이 워크플로를 시작할 때 옵션을 전달합니다.

```
// 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);
```

HelloLambda 샘플 보기

Lambda 작업을 사용하는 워크플로의 구현을 제공하는 샘플은 AWS SDK for Java에서 제공됩니다. 이 샘플을 보거나 실행하려면 소스를 다운로드하십시오.

HelloLambda 샘플을 빌드 및 실행하는 방법에 관한 자세한 설명은 AWS Flow Framework for Java 샘플과 함께 제공되는 README 파일에 있습니다.

AWS Flow Framework for Java로 작성한 프로그램 실행

주제

- WorkflowWorker
- ActivityWorker
- 작업자 스레딩 모델
- 작업자 확장성

프레임워크는 AWS Flow Framework for Java 런타임을 초기화하고 Amazon SWF와 통신하기 위한 작업자 클래스를 제공합니다. 워크플로 또는 활동 작업자를 구현하려면 작업자 클래스의 인스턴스를 생성하여 시작해야 합니다. 이 작업자 클래스는 진행 중인 비동기식 작업을 관리하고 차단 해제된 비동기식 메서드를 호출하며 Amazon SWF와 통신하는 역할을 담당합니다. 이 클래스는 워크플로 및 활동 구현, 스레드 수, 폴의 작업 목록 등으로 구성할 수 있습니다.

프레임워크에는 작업자 클래스 두 개가 함께 제공되는데, 하나는 활동을 위한 것이고 하나는 워크플로를 위한 것입니다. 워크플로 로직을 실행하려면 `WorkflowWorker` 클래스를 사용합니다. 이와 마찬가지로 활동에 대해서는 `ActivityWorker` 클래스가 사용됩니다. 이 클래스에서는 활동 작업에 대해 Amazon SWF를 자동으로 폴링하고 구현에 있는 적절한 메서드를 간접적으로 호출합니다.

다음 예에서는 `WorkflowWorker`를 인스턴스화하고 작업에 대해 폴링을 시작하는 방법을 보여줍니다.

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

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

`ActivityWorker`의 인스턴스를 생성하는 기본 절차와 작업에 대해 폴링을 시작하는 방법은 다음과 같습니다.

```
AmazonSimpleWorkflow swfClient
      = new AmazonSimpleWorkflowClient(awsCredentials);
```

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

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

활동 또는 결정자를 종료하고 싶다면 사용자의 애플리케이션에서 사용 중인 작업자 클래스의 인스턴스뿐 아니라 Amazon SWF Java 클라이언트 인스턴스를 종료해야 합니다. 이렇게 하면 작업자 클래스에서 사용하는 모든 리소스가 적절히 해제됩니다.

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

실행을 시작하려면 생성된 외부 클라이언트의 인스턴스를 생성하고 `@Execute` 메서드를 호출하기만 하면 됩니다.

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

WorkflowWorker

이름에서 암시하듯이 이 작업자 클래스는 워크플로 구현에서 사용하기 위한 것으로서, 작업자 목록 및 워크플로 구현 유형으로 구성됩니다. 이 작업자 클래스에서는 루프를 실행하여 지정된 작업 목록에서 결정 작업에 대한 폴링 작업을 수행합니다. 또한 결정 작업이 수신되면 워크플로 구현의 인스턴스를 생성하고 작업을 처리할 `@Execute` 메서드를 호출합니다.

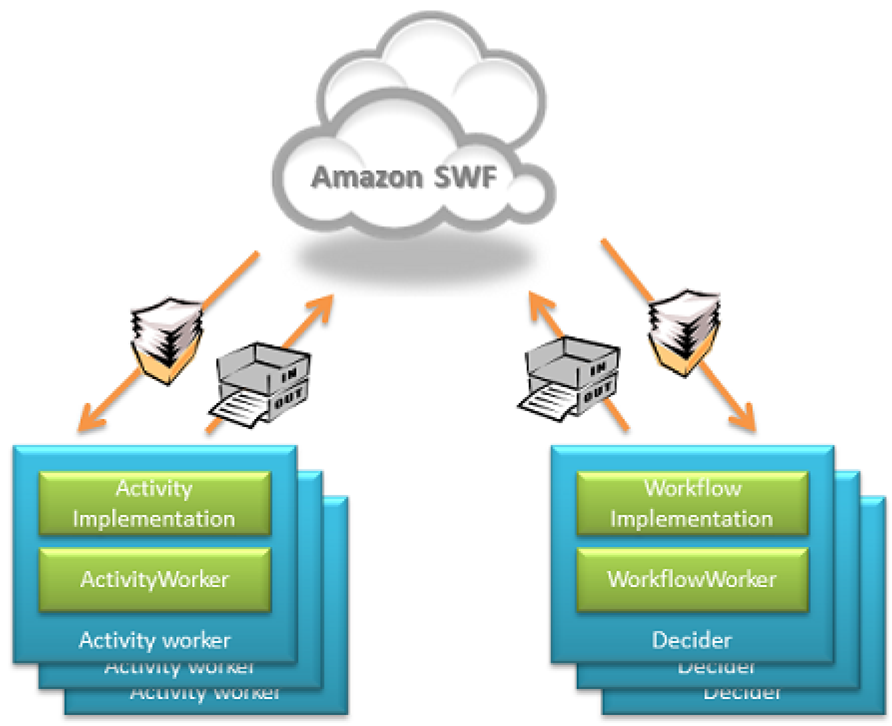
ActivityWorker

활동 작업자 구현을 위해서는 `ActivityWorker` 클래스를 사용하여 활동 작업에 대해 작업 목록을 간편하게 폴링할 수 있습니다. 사용자는 활동 구현 객체로 활동 작업자를 구성합니다. 이 작업자 클래스에서는 루프를 실행하여 지정된 작업 목록에서 활동 작업에 대한 폴링 작업을 수행합니다. 또한 활동 작업이 수신되면 사용자가 제공한 적절한 구현을 검색하고 작업을 처리할 활동 메서드를 호출합니다. 각 결정 작업에 대해 새 인스턴스를 생성하기 위해 팩토리를 호출하는 `WorkflowWorker`와 달리 `ActivityWorker`에서는 사용자가 제공한 객체를 사용합니다.

`ActivityWorker` 클래스에서는 AWS Flow Framework for Java 주석을 사용하여 등록 및 실행 옵션을 결정합니다.

작업자 스레딩 모델

AWS Flow Framework for Java에서 활동 또는 결정자의 구현은 작업자 클래스의 인스턴스입니다. 사용자의 애플리케이션은 작업자의 역할을 수행해야 할 각 머신 및 프로세스에서 작업자 객체를 구성하고 인스턴스화하는 작업을 담당합니다. 그러면 작업자 객체는 Amazon SWF에서 작업을 수신하여 이를 사용자의 활동 또는 워크플로 구현에 디스패치하고 그 결과를 Amazon SWF에 보고합니다. 워크플로 인스턴스 하나가 여러 작업자에 걸쳐 적용될 수 있습니다. Amazon SWF에 보류 중인 활동 작업이 한 개 이상인 경우, 한 작업을 첫 번째로 사용 가능한 작업자에게 할당하고 그다음 작업은 그다음 사용 가능 작업자에게 할당하는 식으로 진행합니다. 이를 통해 동일한 워크플로 인스턴스에 속한 작업이 다른 작업자에서 동시에 처리될 수 있습니다.

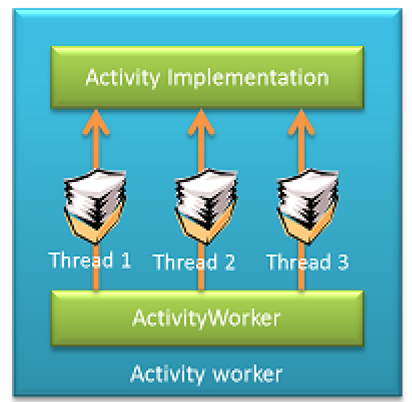


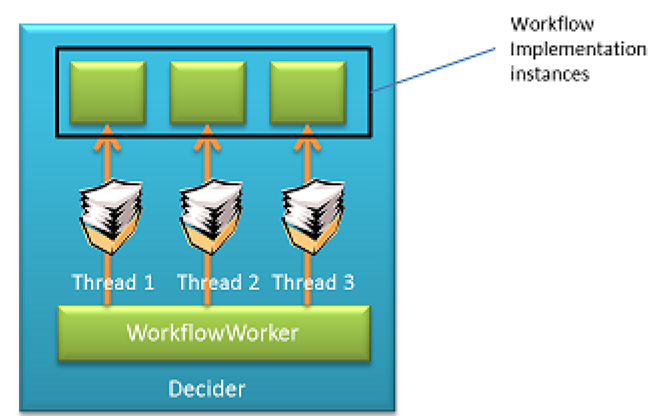
게다가 각 작업자가 여러 스레드에서 작업을 처리하도록 구성할 수 있습니다. 이는 작업자가 하나밖에 없더라도 워크플로 인스턴스의 활동 작업이 동시에 실행될 수 있음을 뜻합니다.

결정 작업은 Amazon SWF에서 특정 워크플로 실행에 대해 한 번에 한 결정만 실행하도록 보장한다는 점을 제외하고 이와 마찬가지로 동작합니다. 워크플로 실행 하나에는 일반적으로 결정 작업 여러 개가 필요하므로 결국 여러 프로세스 및 스레드에서 실행될 수도 있습니다. 결정자는 워크플로 구현의 유형으로 구성됩니다. 또한 결정자에서 결정 작업을 수신하면 워크플로 구현의 인스턴스(객체)를 생성합니

다. 프레임워크에서는 이러한 인스턴스를 생성하기 위한 확장 가능 팩토리 패턴을 제공합니다. 기본 워크플로 팩토리에서는 매번 새 객체를 생성합니다. 이 동작을 재정의할 사용자 지정 팩토리를 제공할 수 있습니다.

워크플로 구현 유형을 사용해 구성하는 결정자와 반대로 활동 작업자는 활동 구현의 인스턴스(객체)를 사용하여 구성합니다. 또한 활동 작업자에서 활동 작업을 수신하면 이 작업은 적절한 활동 구현 객체로 디스패치됩니다.





워크플로 작업자에서는 스레드 풀 하나를 유지 관리하고 해당 작업에 대해 Amazon SWF를 폴링하는 데 사용된 것과 동일한 스레드에서 워크플로를 실행합니다. 활동은 장시간 동안 실행되므로(최소한 워크플로 로직과 비교했을 때) 활동 작업자 클래스에서는 스레드 풀 두 개를 유지 관리합니다. 하나는 활동 작업에 대한 Amazon SWF 폴링을 위한 것이고, 다른 하나는 활동 구현을 실행하여 작업을 처리하기 위한 것입니다. 이를 통해 사용자는 작업을 실행할 스레드의 수와 별도로 작업에 대해 폴링할 스레드의 수를 구성할 수 있습니다. 예를 들어 사용자는 폴링할 스레드는 적게, 작업을 실행할 스레드는 많이 보유할 수 있습니다. 활동 작업자 클래스에서는 작업을 처리할 자유 스레드뿐 아니라 자유 폴링 스레드도 있는 경우에만 작업에 대해 Amazon SWF를 폴링합니다.

이러한 스레딩 및 인스턴스화 동작은 다음을 뜻합니다.

1. 활동 구현은 상태 비저장이어야 합니다. 사용자는 인스턴스 변수를 사용하여 활동 객체에 애플리케이션 상태를 저장해서는 안 됩니다. 하지만 필드를 사용하여 데이터베이스 연결과 같은 리소스를 저장할 수는 있습니다.
2. 활동 구현은 스레드 세이프이어야 합니다. 동일한 인스턴스를 사용하여 서로 다른 스레드에서 작업을 동시에 처리할 수 있으므로 활동 코드에서의 공유 리소스 액세스는 동기화되어야 합니다.
3. 워크플로 구현은 상태 저장일 수 있고, 인스턴스 변수를 사용하여 상태를 저장할 수 있습니다. 각 결정 작업을 처리하기 위해 워크플로 구현의 새 인스턴스를 생성한다 해도 프레임워크에서는 상태가 다시 적절하게 생성되도록 합니다. 하지만 워크플로 구현은 확정적이어야 합니다. 자세한 내용은 Under the Hood 단원을 참조하십시오.

4. 워크플로 구현은 기본 팩토리 사용 시 스레드 세이프일 필요는 없습니다. 기본 구현에서는 한 스레드에서 한 번에 워크플로 구현의 인스턴스를 하나만 사용하도록 합니다.

작업자 확장성

또한 AWS Flow Framework for Java에는 사용자에게 세부적인 제어 기능 및 확장성을 부여하는 하위 수준 작업자 클래스가 포함되어 있습니다. 이 클래스를 사용하면 워크플로 및 활동 유형 등록을 완전히 사용자 지정할 수 있고 구현 객체 생성을 위한 팩토리를 설정할 수 있습니다. 이 작업자는 `GenericWorkflowWorker`와 `GenericActivityWorker`입니다.

`GenericWorkflowWorker`는 워크플로 정의 팩토리 생성을 위한 팩토리를 사용하여 구성할 수 있습니다. 워크플로 정의 팩토리에서는 워크플로 구현의 인스턴스를 생성하고 등록 옵션과 같은 구성 설정을 제공하는 작업을 담당합니다. 정상 조건에서 사용자는 `WorkflowWorker` 클래스를 직접 사용해야 합니다. 이 클래스에서는 프레임워크에 제공된 `POJOWorkflowDefinitionFactoryFactory` 및 `POJOWorkflowDefinitionFactory` 팩토리의 구현을 자동으로 생성하고 구성할 수 있습니다. 이 팩토리의 경우 워크플로 구현 클래스에 인수 생성자가 없어야 합니다. 이 생성자는 실행 시간에 워크플로 객체의 인스턴스를 생성하는 데 사용됩니다. 팩토리에서는 워크플로 인터페이스 및 구현에서 사용한 주석을 검색하여 적절한 등록 및 실행 옵션을 생성합니다.

사용자는 `WorkflowDefinitionFactory`, `WorkflowDefinitionFactoryFactory` 및 `WorkflowDefinition`을 구현하여 팩토리의 구현을 제공할 수 있습니다. `WorkflowDefinition` 클래스는 작업자 클래스에서 결정 작업 및 신호를 디스패치하는 데 사용됩니다. 사용자는 이 기본 클래스를 구현하여 팩토리뿐 아니라 워크플로 구현으로 요청을 디스패치하는 작업을 완전히 사용자 지정할 수 있습니다. 예를 들어 이 확장성 포인트를 사용하여 자신의 주석에 근거해 워크플로를 작성하거나 프레임워크에서 사용하는 코드 우선 접근 방식 대신에 WSDL에서 생성할 수 있는 사용자 지정 프로그래밍 모델을 제공할 수 있습니다. 사용자 지정 팩토리를 사용하려면 `GenericWorkflowWorker` 클래스를 사용해야 합니다. 이 클래스에 대한 자세한 내용은 AWS SDK for Java 설명서를 참조하십시오.

이와 마찬가지로 `GenericActivityWorker`를 통해 사용자는 사용자 지정 활동 구현 팩토리를 제공할 수 있습니다. `ActivityImplementationFactory` 및 `ActivityImplementation` 클래스를 구현하여 사용자는 활동 인스턴스화를 완전히 제어하고 등록 및 실행 옵션을 사용자 지정할 수 있습니다. 이 클래스에 대한 자세한 내용은 AWS SDK for Java 설명서를 참조하십시오.

실행 콘텍스트

주제

- 결정 컨텍스트

- [활동 실행 컨텍스트]

프레임워크에서는 워크플로 및 활동 구현에 주변 컨텍스트를 제공합니다. 이 컨텍스트는 처리 중인 작업에 고유한 것으로서, 사용자가 자신의 구현에 사용할 수 있는 약간의 유틸리티를 제공합니다. 컨텍스트 객체는 작업자가 새 작업을 처리할 때마다 생성됩니다.

결정 컨텍스트

결정 작업이 실행될 때 프레임워크에서는 `DecisionContext` 클래스를 통해 워크플로 구현에 컨텍스트를 제공합니다. `DecisionContext`에서는 워크플로 실행의 실행 ID와 클록 및 타이머 기능과 같이 컨텍스트에 맞는 정보를 제공합니다.

워크플로 구현의 DecisionContext에 액세스

`DecisionContextProviderImpl` 클래스를 사용하여 워크플로 구현의 `DecisionContext`에 액세스합니다. 또는 "테스트 가능성 및 종속성 주입" 단원에 나와 있는 것처럼 Spring을 사용하여 워크플로 구현의 필드 또는 속성에 컨텍스트를 주입할 수 있습니다.

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

클록 및 타이머 생성

`DecisionContext`에는 타이머 및 클록 기능을 제공하는 `WorkflowClock` 유형의 속성이 포함되어 있습니다. 워크플로 로직은 결정적이어야 하므로 워크플로 구현에서 시스템 클록을 직접 사용해서는 안 됩니다. `WorkflowClock`의 `currentTimeMills` 메서드에서는 처리 중인 결정의 시작 이벤트 시각을 반환합니다. 이를 통해 다시 재생 중에 동일한 시각을 가져올 수 있고, 따라서 워크플로 로직은 결정적인 것이 됩니다.

`WorkflowClock`에도 지정된 간격이 지나면 준비 상태가 되는 `Promise` 객체를 반환하는 `createTimer` 메서드가 있습니다. 이 값을 다른 비동기식 메서드에 파라미터로 사용하여 지정된 시간까지 실행을 연기할 수 있습니다. 이로써 비동기식 메서드 또는 활동을 나중에 실행하기 위해 효과적으로 예약할 수 있습니다.

다음 목록의 예시에서는 활동을 주기적으로 호출하는 방법을 보여줍니다.

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

위 목록에서 callPeriodicActivity 비동기식 메서드는 activity1를 호출한 후 현재 AsyncDecisionContext를 사용하여 타이머를 생성합니다. 이 메서드는 반환된 Promise를 자신에 대한 재귀적 호출에 인수로 전달합니다. 이 재귀적 호출은 타이머가 소진될 때까지(이 예시에서는 1시간) 대기했다가 실행됩니다.

활동 실행 컨텍스트

DecisionContext에서 결정 작업이 처리 중일 때 컨텍스트 정보를 제공하는 것과 마찬가지로 ActivityExecutionContext에서는 활동 작업이 처리 중일 때 이와 유사한 컨텍스트 정보를 제공합니다. 이 컨텍스트는 ActivityExecutionContextProviderImpl 클래스를 통해 사용자의 활동 코드에 제공됩니다.

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

ActivityExecutionContext를 사용하여 다음 작업을 수행할 수 있습니다.

장시간 실행 활동에 하트비트 사용

활동이 장시간 실행되는 경우 Amazon SWF에 진행 상황을 주기적으로 보고하여 작업이 여전히 진행 중임을 알려야 합니다. 그러한 하트비트가 없는 경우 작업 하트비트 제한 시간이 활동 유형 등록 시 또는 활동 예약 중에 설정되었다면 작업이 제한 시간을 초과할 수 있습니다. 하트비트를 전송하려면 ActivityExecutionContext의 recordActivityHeartbeat 메서드를 사용할 수 있습니다. 또한 하트비트에서는 지속적인 활동을 취소할 수 있는 메커니즘을 제공합니다. 추가 세부 정보 및 예시는 오류 처리 단원을 참조하십시오.

활동 작업에 관한 세부 정보 얻기

원하는 경우 실행기가 작업을 가져왔을 때 Amazon SWF에서 전달한 활동 작업에 관한 모든 세부 정보를 얻을 수 있습니다. 이 정보에는 작업, 작업 유형, 작업 토큰 등에 대한 입력에 관한 정보

가 포함되어 있습니다. 예를 들어 사람의 작업에 의해 수동으로 완료되는 활동을 구현하려는 경우 ActivityExecutionContext를 사용하여 작업 토큰을 검색하고 이를 활동 작업을 완료하는 프로세스에 전달해야 합니다. 자세한 내용은 활동을 수동으로 완료 단원을 참조하십시오.

실행기에서 사용 중인 Amazon SWF 클라이언트 객체 가져오기

실행기에서 사용 중인 Amazon SWF 클라이언트 객체는 ActivityExecutionContext의 getService 메서드를 직접적으로 호출하여 가져올 수 있습니다. 이 기능은 Amazon SWF 서비스를 직접 호출하려는 경우에 유용합니다.

하위 워크플로 실행

지금까지의 예시에서는 워크플로 실행을 애플리케이션에서 직접 시작하였습니다. 그러나 생성된 클라이언트에 있는 워크플로 진입점 메서드를 호출하는 방식으로 워크플로 내에서 워크플로 실행을 시작할 수 있습니다. 워크플로 실행이 다른 워크플로 실행의 맥락에서 시작되는 경우 이를 가리켜 하위 워크플로 실행이라고 합니다. 이를 통해 복잡한 워크플로를 더 작은 유닛으로 리팩토링한 후 이를 서로 다른 워크플로에서 공유할 수 있습니다. 예를 들어 결제 처리 워크플로를 생성한 후 이를 주문 처리 워크플로에서 호출할 수 있습니다.

의미상 하위 워크플로 실행은 다음 차이점을 제외하고는 독립 실행형 워크플로와 동일하게 작동합니다.

1. 사용자의 명시적인 작업(예: TerminateWorkflowExecution Amazon SWF API 호출 또는 시간 초과로 인해 종료)으로 인해 상위 워크플로가 종료되는 경우 하위 워크플로 실행은 하위 정책에 따라 결정됩니다. 이 하위 정책에서 하위 워크플로 실행을 종료, 취소 또는 중단(계속 실행)하도록 설정할 수 있습니다.
2. 하위 워크플로의 결과(진입점 메서드의 반환 값)는 비동기식 메서드에서 반환하는 Promise<T>와 마찬가지로 상위 워크플로 실행에서 사용할 수 있습니다. 이것은 애플리케이션이 Amazon SWF API를 사용하여 출력을 얻어야 하는 독립 실행형 실행과는 다릅니다.

다음 예시의 OrderProcessor 워크플로에서는 PaymentProcessor 하위 워크플로를 생성합니다.

```
@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);

}
```

연속 워크플로

일부 사용 사례에서는 영구적으로 또는 장기간 실행되는 워크플로(예: 서버 집합의 상태를 모니터링하는 워크플로)가 필요할 수 있습니다.

> **Note**
>
> Amazon SWF에서는 워크플로 실행 내역 전체를 보관하므로 이 내역은 시간이 지남에 따라 계속 늘어납니다. 프레임워크에서는 재생을 수행할 때 Amazon SWF에서 이 내역을 검색하는데, 내역의 규모가 너무 크면 비용이 많이 듭니다. 이처럼 장시간 실행 또는 연속되는 워크플로에서는 현재 실행을 주기적으로 종료하고 새로운 수행을 시작하여 처리가 계속되도록 해야 합니다.

이것은 워크플로 실행의 논리적 연속입니다. 생성된 자체 클라이언트를 이러한 목적으로 사용할 수 있습니다. 워크플로 구현에서 자체 클라이언트의 `@Execute` 메서드를 호출합니다. 현재 실행이 완료되면 프레임워크에서 동일한 워크플로 Id를 사용하여 새 실행을 시작합니다.

또한 현재 `DecisionContext`에서 가져올 수 있는 `GenericWorkflowClient`의 `continueAsNewOnCompletion` 메서드를 호출하여 실행을 계속할 수도 있습니다. 예를 들어 다음 워크플로 구현에서는 타이머를 설정하여 하루가 지나면 자체 진입점을 호출하여 새 실행이 시작되게 합니다.

```
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();
    }
}
```

워크플로에서 재귀적으로 자신을 호출하면 프레임워크에서는 모든 대기 중 작업이 완료되었을 때 현재 워크플로를 종료하고 새 워크플로 실행을 시작합니다. 대기 중인 작업이 있는 한 현재 워크플로 실행은 종료되지 않는다는 점에 유의하십시오. 새 실행은 원본 실행으로부터 자동으로 내역이나 데이터를 전혀 상속하지 않습니다. 일부 상태를 새 실행으로 넘기고 싶다면 이를 명시적으로 입력으로 전달해야 합니다.

작업 우선 순위 설정

기본적으로 작업 목록의 작업은 도착 시간에 따라 제공됨: 가능한 한 먼저 예약된 작업이 일반적으로 먼저 실행됩니다. 선택적 작업 우선 순위를 설정해 특정 작업에 우선 순위를 부여할 수 있음: Amazon SWF는 작업 목록에서 우선 순위가 높은 작업을 우선 순위가 낮은 작업보다 먼저 제공하려고 합니다.

워크플로 및 활동 둘 다에 대해 작업 우선 순위를 설정할 수 있습니다. 워크플로의 작업 우선 순위는 워크플로가 예약한 활동 작업의 우선 순위에 영향을 미치지 않고 워크플로가 시작한 하위 워크플로에도 영향을 미치지 않습니다. 활동 또는 워크플로의 기본 우선 순위는 등록 중(사용자 또는 Amazon SWF가) 설정하지만 활동을 예약하거나 워크플로 실행을 시작하는 동안 재정의하지 않는 한 항상 등록된 작업 우선 순위가 사용됩니다.

작업 우선 순위 값의 범위는 "-2147483648" ~ "2147483647"일 수 있으며 숫자가 클수록 우선 순위가 높음을 나타냅니다. 활동 또는 워크플로에 대해 작업 우선 순위를 설정하지 않으면 우선 순위 0이 할당됩니다.

주제

- 워크플로의 작업 우선 순위 설정
- 활동의 작업 우선 순위 설정

워크플로의 작업 우선 순위 설정

워크플로를 등록 또는 시작할 때 워크플로의 작업 우선 순위를 설정할 수 있습니다. 워크플로 실행 설정 시 재정의되지 않는 한 워크플로 유형 등록 시 설정한 작업 우선 순위가 해당 형의 워크플로 실행에 대해 기본값으로 사용됩니다.

기본 작업 우선 순위로 워크플로 유형을 등록하려면 선언 시 WorkflowRegistrationOptions에서 defaultTaskPriority 옵션을 설정합니다.

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

또한 워크플로를 시작할 때 워크플로의 taskPriority를 설정할 수 있습니다. 이로써 등록된(기본) 작업 우선 순위는 재정의됩니다.

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

PriorityWorkflowClientExternalFactory cf
    = new PriorityWorkflowClientExternalFactoryImpl(swfService, domain);

priority_workflow_client = cf.getClient();

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

이외에도 하위 워크플로를 시작하거나 워크플로를 새로 진행할 때 작업 우선 순위를 설정할 수 있습니다. 예를 들어, ContinueAsNewWorkflowExecutionParameters 또는 StartChildWorkflowExecutionParameters에서 taskPriority 옵션을 설정할 수 있습니다.

활동의 작업 우선 순위 설정

활동을 등록하거나 예약할 때 활동에 대한 작업 우선 순위를 설정할 수 있습니다. 활동 예약 재정의하지 않는 한 작업 유형 등록 시 설정한 작업 우선 순위가 활동 실행 시 기본 우선 순위로 사용됩니다.

기본 작업 우선 순위로 활동 유형을 등록하려면 선언 시 ActivityRegistrationOptions에서 defaultTaskPriority 옵션을 설정합니다.

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

또한 활동을 예약할 때 활동의 taskPriority를 설정할 수 있습니다. 이로써 등록된(기본) 작업 우선 순위는 재정의됩니다.

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

ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();

activityClient.doSomethingImportant(activityOptions);
```

DataConverters

워크플로 구현에서 원격 활동을 호출하면 이 활동에 전달된 입력과 활동 실행의 결과를 직렬화하여 연결을 통해 전송될 수 있도록 해야 합니다. 프레임워크에서는 이를 위해 DataConverter 클래스를 사용합니다. 이것은 사용자가 자신의 고유한 직렬 변환기를 제공하기 위해 구현할 수 있는 추상 클래스입니다. 기본 Jackson 직렬 변환기 기반 구현인 `JsonDataConverter`가 프레임워크에서 제공됩니다. 자세한 내용은 AWS SDK for Java 설명서를 참조하십시오. Jackson이 직렬화를 수행하는 방식과 직렬화에 영향을 미치기 위해 사용할 수 있는 Jackson 주석에 대한 자세한 내용은 Jackson JSON Processor 설명서를 참조하십시오. 사용되는 연결 형식은 계약의 일부로 간주됩니다. 따라서 사용자는

@Activities 및 @Workflow 주석의 DataConverter 속성을 설정하여 자신의 활동 및 워크플로 인터페이스에서 DataConverter를 지정할 수 있습니다.

프레임워크에서는 사용자가 @Activities 주석에서 지정한 DataConverter 유형의 객체를 생성하여 활동에 대한 입력을 직렬화하고 그 결과를 역직렬화합니다. 이와 마찬가지로 사용자가 @Workflow 주석에서 지정하는 DataConverter 유형의 객체는 사용자가 워크플로에 전달하는 파라미터를 직렬화하는 데 사용되고 하위 워크플로의 경우에는 그 결과를 역직렬화하는 데 사용됩니다. 입력 외에도 프레임워크는 추가 데이터(예: 예외 세부 정보)를 Amazon SWF에 전달합니다. 워크플로 직렬 변환기는 이 데이터를 직렬화하는 데에도 사용됩니다.

또한 사용자는 프레임워크에서 DataConverter의 인스턴스를 자동으로 생성하도록 하고 싶지 않으면 이를 직접 제공할 수도 있습니다. 생성된 클라이언트에는 DataConverter를 받아들이는 생성자 오버로드가 있습니다.

DataConverter 유형을 지정하지 않고 DataConverter 객체를 전달하지 않으면 기본적으로 JsonDataConverter가 사용됩니다.

데이터를 비동기식 메서드로 전달

주제

- 모음 및 맵을 비동기식 메서드로 전달
- Settable<T>
- @NoWait
- Promise<Void>
- AndPromise 및 OrPromise

Promise<*T*>에 관해서는 앞 단원에서 설명하였습니다. 여기에서는 Promise<*T*>의 고급 사용 사례 몇 가지를 다룹니다.

모음 및 맵을 비동기식 메서드로 전달

프레임워크에서는 어레이, 모음 및 맵을 비동기식 메서드에 Promise 유형으로 전달할 수 있도록 지원합니다. 예를 들어 비동기식 메서드에서는 다음 목록과 같이 Promise<ArrayList<String>>를 인수로 받아들일 수 있습니다.

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

```
        activityClient.printActivity(s);
    }
}
```

의미상 이것은 기타 Promise 형식 파라미터로 작동하고 비동기식 메서드는 모음이 사용 가능해질 때까지 기다렸다가 실행됩니다. 모음의 멤버가 Promise 객체인 경우 사용자는 모든 멤버가 다음 코드 조각과 같이 준비 상태가 될 때까지 프레임워크가 대기하도록 할 수 있습니다. 이렇게 하면 비동기식 메서드는 모음의 각 멤버가 준비 상태가 될 때까지 대기합니다.

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

@Wait 주석을 파라미터에 사용하여 파라미터에 Promise 객체가 포함되어 있음을 나타내야 한다는 점에 유의하십시오.

printActivity 활동에서는 String 인수를 받아들이지만 생성된 클라이언트의 매칭 메서드는 Promise<String>을 사용한다는 점도 유의하십시오. 우리는 클라이언트에 있는 메서드를 호출하고 있는 것이지 활동 메서드를 직접 호출하고 있는 것은 아닙니다.

Settable<T>

Settable<*T*>는 Promise의 값을 수동으로 설정할 수 있게 해주는 설정된 메서드를 제공하는 Promise<T>의 파생 유형입니다. 예를 들어 다음 워크플로는 신호 메서드에서 설정되는 Settable<?>를 기다림으로써 신호가 수신되기를 기다립니다.

```
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;
    }
}
```

또한 Settable<?>는 한 번에 다른 하나의 약속에 묶일 수 있습니다. AndPromise 및 OrPromise를 사용하여 약속을 그룹화 수 있습니다. unchain() 메서드를 호출하여 묶인 Settable을 풀 수 있습니다. Settable<?>는 묶여 있을 때 묶여 있는 약속이 준비 상태가 되면 자동으로 준비 상태가 됩니다. 묶는 기능은 프로그램 다른 부분의 doTry() 범위 내에서 반환된 약속을 사용하고 싶은 경우 특히 유용합니다. TryCatchFinally가 중첩 클래스로 사용되기 때문에 상위 범위에서 Promise<>를 선언하고 이를 doTry()에서 설정할 수는 없습니다. 이는 Java에서는 변수가 최종으로 표시되기 위해 상위 범위에서 선언되고 중첩 클래스에서 사용되어야 하기 때문입니다. 예:

```
@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;
}
```

Settable은 한 번에 한 약속에 묶일 수 있습니다. unchain() 메서드를 호출하여 묶인 Settable을 풀 수 있습니다.

@NoWait

Promise를 비동기식 메서드에 전달하면 기본적으로 프레임워크에서는 Promise(들)이 준비 상태가 될 때까지 기다렸다가 메서드를 실행합니다(모음 유형 제외). 사용자는 비동기식 메서드의 선언에서 @NoWait 주석 또는 파라미터를 사용하여 이 동작을 재정의할 수 있습니다. 이 방법은 비동기식 메서드 자체에서 설정할 Settable<T>에서 전달하는 경우 유용합니다.

Promise<Void>

비동기식 메서드의 종속성은 한 메서드에서 다른 메서드로 인수로 반환되는 Promise를 전달함으로써 구현됩니다. 그러나 메서드에서 void를 반환하고 싶지만 완료된 후 여전히 다른 비동기식 메서드가 실행되기를 원하는 경우가 있을 수 있습니다. 이러한 경우에는 Promise<Void>를 메서드의 반환 유형으로 사용할 수 있습니다. Promise 클래스에서는 Promise<Void> 객체를 생성하는 데 사용할 수 있는 정적 Void 메서드를 제공합니다. 이 Promise는 비동기식 메서드가 실행을 마치면 준비 상태가 됩니다. 이 Promise를 기타 Promise 객체와 마찬가지로 다른 비동기식 메서드로 전달할 수 있습니다. Settable<Void>을 사용 중이라면 그 위에 null로 설정된 메서드를 호출하여 준비 상태로 만듭니다.

AndPromise 및 OrPromise

AndPromise 및 OrPromise를 통해 Promise<> 객체 여러 개를 논리적 약속 하나로 그룹화할 수 있습니다. 이를 구성하는 데 사용된 모든 약속이 준비 상태가 되면 AndPromise가 준비 상태가 됩니다. 이를 구성하는 데 사용된 약속 모음의 약속 중 어느 하나라도 준비 상태가 되면 OrPromise가 준비 상태가 됩니다. AndPromise 및 OrPromise에서 getValues()을 호출하여 구성 약속의 값 목록을 가져올 수 있습니다.

테스트 가능성 및 종속성 주입

주제

- Spring 통합
- JUnit 통합

이 프레임워크는 제어 반전(IoC)에 적합하게 설계되었습니다. 활동 및 워크플로 구현뿐 아니라 프레임워크에서 제공하는 작업자 및 컨텍스트 객체는 Spring과 같은 컨테이너를 사용하여 구성 및 인스턴스

화할 수 있습니다. 즉시 사용 가능한 이 프레임워크는 Spring Framework와 통합됩니다. 뿐만 아니라 단위 테스트 워크플로 및 활동 구현을 위해 JUnit과의 통합도 지원합니다.

Spring 통합

com.amazonaws.services.simpleworkflow.flow.spring 패키지에는 사용자 애플리케이션에서 Spring 프레임워크를 쉽게 사용할 수 있게 해주는 클래스가 포함되어 있습니다. 이 클래스에는 사용자 지정 Scope 및 Spring 인식 활동 및 워크플로 작업자인 `WorkflowScope`, `SpringWorkflowWorker` 및 `SpringActivityWorker`이 포함됩니다. 이 클래스를 통해 사용자는 전적으로 Spring을 통해 작업자뿐 아니라 워크플로 및 활동 구현을 구성할 수 있습니다.

WorkflowScope

`WorkflowScope`는 프레임워크에서 제공하는 사용자 지정 Spring Scope 구현입니다. 이 범위를 통해 사용자는 수명 범위가 결정 작업의 수명으로 지정된 Spring 컨테이너에 객체를 생성할 수 있습니다. 이 범위의 빈(bean)은 작업자가 새 결정 작업을 수신할 때마다 인스턴스화됩니다. 사용자는 워크플로 구현 빈과 이 구현에서 의존하는 기타 빈에 대해 이 범위를 사용해야 합니다. 프레임워크에서는 각 결정 작업에 대해 새 빈을 생성할 것을 요구하기 때문에 Spring에서 제공하는 singleton과 프로토타입 범위를 워크플로 구현 빈에 사용해서는 안 됩니다. 이를 준수하지 않으면 예상치 못한 동작이 발생합니다.

다음 예시에서는 `WorkflowScope`를 등록한 후 이를 사용해 워크플로 구현 빈 및 활동 클라이언트 빈을 구성하는 Spring 구성의 코드 조각을 보여줍니다.

```
<!-- 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>
```

구성 관련 행: `workflowImpl` 빈의 구성에서 사용되는 `<aop:scoped-proxy proxy-target-class="false" />`가 필요한 이유는 `WorkflowScope`에서 CGLIB를 사용한 프록시 설정을 지원하지 않기 때문입니다. 사용자는 다른 범위에 있는 다른 빈에 연결된 `WorkflowScope`에 있는 모든 빈에 대해 이 구성을 사용해야 합니다. 이 경우 `workflowImpl` 빈은 singleton 범위 내에 있는 워크플로 작업자 빈에 연결해야 합니다(아래의 전체 예시 참조).

Spring Framework 설명서에서 사용자 지정 범위 사용에 관한 자세한 내용을 학습하실 수 있습니다.

Spring 인식 작업자

Spring 사용 시 사용자는 프레임워크에서 제공하는 Spring 인식 작업자 클래스인 `SpringWorkflowWorker` 및 `SpringActivityWorker`를 사용해야 합니다. 이 작업자는 다음 예시와 같이 Spring을 사용하여 애플리케이션에 주입될 수 있습니다. Spring 인식 작업자는 Spring의 `SmartLifecycle` 인터페이스를 구현하고, Spring 컨텍스트가 시작될 때 작업에 대해 폴링을 자동으로 시작하도록 기본 설정되어 있습니다. 사용자는 작업자의 `disableAutoStartup` 속성을 `true`로 설정하여 이 기능을 비활성화할 수 있습니다.

다음 예시에서는 결정자를 구성하는 방법을 보여줍니다. 이 예시에서는 `MyActivities` 및 `MyWorkflow` 인터페이스(여기에는 표시되지 않음)와 그에 상응하는 구현인 `MyActivitiesImpl` 및 `MyWorkflowImpl`를 사용합니다. 생성된 클라이언트 인터페이스 및 구현은 `MyWorkflowClient`/`MyWorkflowClientImpl`과 `MyActivitiesClient`/`MyActivitiesClientImpl`입니다(여기에는 표시되지 않음).

활동 클라이언트는 다음과 같이 Spring의 자동 연결 기능을 사용하여 워크플로 구현에 주입됩니다.

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

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

결정자에 대한 Spring 구성은 다음과 같습니다.

```
<?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>
```

`SpringWorkflowWorker`는 Spring에서 완전히 구성되고 Spring 컨텍스트가 시작될 때 폴링을 자동으로 시작하므로 결정자에 대한 호스트 프로세스는 다음과 같이 간단합니다.

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

이와 마찬가지로 활동 작업자는 다음과 같이 구성할 수 있습니다.

```
<?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>
```

다음과 같이 활동 작업자 호스트 프로세스는 결정자와 유사합니다.

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

결정 컨텍스트 주입

워크플로 구현에서 컨텍스트 객체에 의존하는 경우 사용자는 역시 Spring을 통해 이 객체를 쉽게 주입할 수 있습니다. 프레임워크에서는 Spring 컨테이너에 컨텍스트 관련 빈을 자동으로 등록합니다. 예를 들어 다음 코드 조각에서는 여러 컨텍스트 객체가 자동 연결되었습니다. 컨텍스트 객체의 다른 Spring 구성은 필요하지 않습니다.

```
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();
    }
}
```

Spring XML 구성을 통해 워크플로 구현에서 컨텍스트 객체를 구성하고 싶다면 com.amazonaws.services.simpleworkflow.flow.spring 패키지의 `WorkflowScopeBeanNames` 클래스에 선언된 빈(bean) 이름을 사용하면 됩니다. 예:

```
<!-- 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>
```

또는 `DecisionContextProvider`를 워크플로 구현에 주입하고 이를 사용하여 컨텍스트를 생성할 수 있습니다. 이 방법은 공급자 및 컨텍스트의 사용자 지정 구현을 제공하고 싶은 경우에 유용할 수 있습니다.

리소스 활동에 주입

제어 반전(IoC) 컨테이너를 사용하여 활동 구현을 인스턴스화하고 구성하며 데이터베이스 연결과 같은 리소스를 활동 구현 클래스의 속성으로 선언하여 주입할 수 있습니다. 그러한 리소스는 일반적으로 그 범위가 singleton으로 지정됩니다. 활동 구현은 여러 스레드에서 활동 작업자가 호출한다는 점에 유의하십시오. 따라서 공유된 리소스에 대한 액세스는 동기화되어야 합니다.

JUnit 통합

이 프레임워크에서는 JUnit으로 단위 테스트를 작성하고 실행하는 데 사용할 수 있는, 테스트 클록과 같은 컨텍스트 객체의 테스트 구현뿐 아니라 JUnit 확장도 제공합니다. 사용자는 이러한 확장을 사용해 워크플로 구현을 로컬로 인라인에서 테스트할 수 있습니다.

간단한 단위 테스트 작성

해당 워크플로에 대해 테스트를 작성하려면 com.amazonaws.services.simpleworkflow.flow.junit 패키지에 있는 `WorkflowTest` 클래스를 사용하십시오. 이 클래스는 프레임워크에 고유한 JUnit `MethodRule` 구현이며 해당 워크플로 코드를 로컬에서 실행하여 Amazon SWF를 통하는 대신에 인라인에서 활동을 직접적으로 호출합니다. 이를 통해 사용자는 요금을 발생시키지 않고도 원하는 횟수 만큼 테스트를 실행할 수 있는 유연성을 얻게 됩니다.

이 클래스를 사용하려면 `WorkflowTest` 유형의 필드를 선언하고 여기에 `@Rule` 주석을 붙이기만 하면 됩니다. 테스트를 실행하기 전에 먼저 새 `WorkflowTest` 객체를 생성한 후 이 객체에 활동 및 워크플로 구현을 추가하십시오. 그러면 생성된 워크플로 클라이언트 팩토리를 사용하여 클라이언트를 생성하고 워크플로 실행을 시작할 수 있습니다. 이 프레임워크에서는 사용자 지정 JUnit 실행자인 `FlowBlockJUnit4ClassRunner`도 제공하는데, 이 실행자는 워크플로 테스트에 사용해야 합니다. 예:

```
@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);
    }
}
```

또한 사용자는 자신이 WorkflowTest에 추가하는 각 활동 구현에 별도의 작업 목록을 지정할 수 있습니다. 예를 들어 호스트 고유 작업 목록에 활동을 예약하는 워크플로 구현이 있다면 다음과 같이 각 호스트의 작업 목록에 활동을 등록할 수 있습니다.

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

@Test의 코드는 비동기식이라는 점에 유의하십시오. 따라서 실행을 시작하려면 비동기식 워크플로 클라이언트를 사용해야 합니다. 테스트 결과를 확인하기 위해 AsyncAssert 헬프 클래스가 제공됩니다. 이 클래스를 통해 사용자는 약속이 준비 상태가 되길 기다렸다가 결과를 확인할 수 있습니다. 이 예시에서 우리는 워크플로 실행의 결과가 준비될 때까지 기다렸다가 테스트 결과를 확인합니다.

Spring을 사용하는 경우 WorkflowTest 클래스 대신에 SpringWorkflowTest 클래스를 사용할 수 있습니다. SpringWorkflowTest는 Spring 구성을 통해 활동 및 워크플로 구현을 쉽게 구성할 때 사용할 수 있는 속성을 제공합니다. Spring 인식 작업자와 마찬가지로 사용자는 WorkflowScope를 사용하여 워크플로 구현 빈을 구성해야 합니다. 이렇게 하면 각 결정 작업에 대해 새 워크플로 구현 빈이 하나씩 생성되도록 보장할 수 있습니다. 범위 지정된 프록시 프록시 대상 클래스 설정이 false로 설정되도록 이 빈을 구성해야 합니다. 자세한 내용은 Spring 통합 단원을 참조하십시오. Spring 통합 단원에 있는 예시 Spring 구성은 다음과 같이 SpringWorkflowTest를 사용하는 워크플로를 테스트하도록 변경할 수 있습니다.

```
<?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>
```

활동 구현 모의

테스트 중에 실제 활동 구현을 사용할 수 있지만, 단지 워크플로 로직을 단위 테스트하고 싶다면 활동을 모의해야 합니다. 이는 `WorkflowTest` 클래스에 활동 인터페이스의 모의 구현을 제공하는 방법으로 수행할 수 있습니다. 예:

```
@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);
    }
}
```

또는 활동 클라이언트의 모의 구현을 제공하고 이를 워크플로 구현에 주입할 수 있습니다.

컨텍스트 객체 테스트

워크플로 구현이 프레임워크 컨텍스트 객체(예: `DecisionContext`)에 의존하는 경우 이러한 워크플로를 테스트하기 위해 특별한 작업을 수행하지 않아도 됩니다. 테스트가 `WorkflowTest`를 통해 실행되면 테스트 컨텍스트 객체를 자동으로 주입합니다. 워크플로 구현에서 컨텍스트 객체(예: `DecisionContextProviderImpl` 사용)에 액세스하면 테스트 구현을 진행합니다. 사용자는 테스트 코드(`@Test` 메서드)에서 이러한 테스트 컨텍스트 객체를 조작하여 흥미로운 테스트 사례를 만들 수 있습니다. 예를 들어 해당 워크플로에서 타이머를 생성하는 경우 사용자는 `WorkflowTest` 클래스에 있는 `clockAdvanceSeconds` 메서드를 호출하여 타이머를 활성화시켜 클록이 시간상 앞으로 이동하도록 할 수 있습니다. 또한 `WorkflowTest`의 `ClockAccelerationCoefficient` 속성을 사용해 클록 속도를 높여 타이머가 평상시보다 더 일찍 활성화되게 할 수 있습니다. 예를 들어 해당 워크플로에서 한 시간짜리 타이머를 생성하는 경우 사용자는 `ClockAccelerationCoefficient`를 60으로 설정하여 타이머를 1분 내에 활성화시킬 수 있습니다. `ClockAccelerationCoefficient`는 1로 기본 설정됩니다.

com.amazonaws.services.simpleworkflow.flow.test 및 com.amazonaws.services.simpleworkflow.flow.junit 패키지에 관한 자세한 내용은 AWS SDK for Java 설명서를 참조하십시오.

오류 처리

주제

- TryCatchFinally 의미론
- 취소
- 중첩 TryCatchFinally

Java의 `try/catch/finally` 구성체는 오류 처리를 간소화하며 유비쿼터스 방식으로 사용됩니다. 이를 통해 사용자는 오류 처리기를 코드 블록에 연결할 수 있습니다. 내부적으로 이 작업은 호출 스택에서 오류 처리기에 관한 추가 메타데이터를 스터핑하는 방식으로 수행됩니다. 예외가 발생하면 실행 시간에서는 호출 스택에서 연결된 오류 처리기를 검색하여 호출하고, 적절한 오류 처리기를 찾을 수 없으면 예외를 호출 체인 상위로 전파합니다.

이 작업은 동기식 코드에서는 잘 수행되지만, 비동기 및 분산 프로그램에서 오류를 처리하면 추가적인 문제가 발생합니다. 비동기식 호출에서는 값을 즉시 반환하므로 비동기식 호출이 실행될 때 호출자는 호출 스택에 있지 않습니다. 이는 비동기식 코드의 미처리 예외는 호출자가 통상적인 방식으로는 처리할 수 없음을 뜻합니다. 일반적으로 비동기식 코드에서 시작되는 예외는 오류 상태를 비동기식 메서드

로 전달되는 콜백으로 전달하여 처리합니다. 또는 Future<?>가 사용되고 있다면 사용자가 이 코드에 액세스하려 할 때 오류를 보고합니다. 이것은 바람직한 상태는 아닌데, 그 이유는 예외를 수신하는 코드(Future<?>를 사용하는 콜백 또는 코드)에는 원본 호출의 컨텍스트가 없고 예외를 적당히 처리하지 못할 수 있기 때문입니다. 뿐만 아니라 분산 비동기 시스템에서는 동시에 실행되는 구성 요소에서 두 개 이상의 오류가 동시에 발생할 수 있습니다. 이러한 오류의 유형과 심각도는 여러 가지일 수 있으므로 그에 맞게 처리해야 합니다.

비동기식 호출 후에 리소스를 정리하는 것 역시 까다롭습니다. 동기식 코드와 달리 사용자는 리소스를 정리하기 위해 호출 코드에서 try/catch/finally를 사용할 수는 없는데, 그 이유는 try 블록에서 시작된 작업이 finally 블록이 실행될 때에도 여전히 지속되고 있을 수 있기 때문입니다.

프레임워크에서는 분산 비동기식 코드에서 오류를 처리하는 방식을 Java의 try/catch/finally와 유사하게 만드는, 그리고 그와 거의 같은 수준으로 간소화하는 메커니즘을 제공합니다.

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

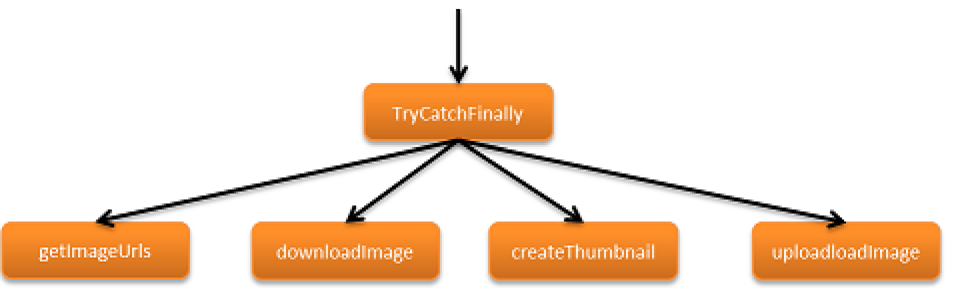
TryCatchFinally 클래스와 그 변형인 TryFinally 및 TryCatch는 Java의 try/catch/finally와 유사한 방식으로 작동합니다. 사용자를 이를 사용해 오류 처리기를 비동기식 및 원격 작업으로 실행될 수 있는 워크플로 코드 블록에 연결할 수 있습니다. doTry() 메서드는 논리적으로 try 블록과 같습니다. 프레임워크에서는 코드를 doTry()에서 자동으로 실행합니다. Promise 객체 목록은 TryCatchFinally의 생성자로 전달될 수 있습니다. doTry 메서드는 생성자로 전달된 모든 Promise 객체가 준비 상태가 될 때까지 실행됩니다. doTry() 내에서 비동기식으로 호출된 코드에서 예외를 제기하면 doTry()의 모든 대기 중 작업이 취소되고 예외 처리를 위해 doCatch()가 호출됩니다. 예를 들어 위 목록에서 downloadImage에 예외가 발생하면 createThumbnail 및 uploadImage는 취소됩니다. 끝으로 모든 비동기식 작업이 완료되면(완료, 실패 또는 취소) doFinally()가 호출됩니다. 이 메서드는 리소스 정리에 사용할 수 있습니다. 또한 사용자는 필요에 맞게 이 클래스를 중첩할 수 있습니다.

doCatch()에서 예외가 보고되면 프레임워크에서는 비동기 및 원격 호출을 포함하는 완결된 논리 호출 스택을 제공합니다. 이 스택은 디버깅할 때 유용하며, 특히 사용자에게 다른 비동기식 메서드를 호출하는 비동기식 메서드가 있는 경우에 유용합니다. 예를 들어 downloadImage에서 발생한 예외에서는 다음과 같은 예외를 생성합니다.

```
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 의미론

AWS Flow Framework for Java 프로그램 실행은 동시 실행 브랜치의 트리로 시각화할 수 있습니다. 비동기식 메서드, 활동 및 TryCatchFinally 자체를 호출하여 이 실행 트리에 새 브랜치를 생성할 수 있습니다. 예를 들어 이미지 처리 워크플로는 다음 그림의 트리와 같은 모양일 수 있습니다.



실행의 브랜치 하나에서 오류가 발생하면 해당 브랜치가 해제되는데, 이는 예외로 인해 Java 프로그램에서 호출 스택이 해제되는 것과 마찬가지입니다. 해제는 오류가 처리되거나 트리의 루트에 도달할 때까지(이때 워크플로 실행이 종료됨) 실행 브랜치 위쪽으로 계속해서 이동합니다.

프레임워크에서는 발생하는 오류를 보고함과 동시에 작업을 예외로 처리합니다. 또한 `TryCatchFinally`에 정의된 예외 처리기(`doCatch()` 메서드)를 상응하는 `doTry()`에서 코드가 생성한 모든 작업에 연결합니다. 시간 초과나 처리되지 않은 예외로 인해 작업이 실패하면 적절한 예외가 발생하고 해당 `doCatch()`가 간접적으로 호출되어 이를 처리합니다. 이 작업을 완료하기 위해 프레임워크에서는 Amazon SWF와 협력하여 원격 오류를 전파하고 이 오류를 호출자의 컨텍스트에서 예외로 다시 생성합니다.

취소

동기식 코드에서 예외가 발생하면 컨트롤에서는 `try` 블록에 남아 있는 모든 코드를 무시하고 직접 `catch` 블록으로 건너뜁니다. 예:

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

이 코드에서 `b()`에 예외가 발생하면 `c()`는 호출되지 않습니다. 이를 워크플로에 비유하면 다음과 같습니다.

```
new TryCatch() {

    @Override
    protected void doTry() throws Throwable {
        activityA();
```

```
        activityB();
        activityC();
    }

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

이 경우 activityA, activityB 및 activityC를 호출하면 모두 성공적으로 값을 반환하고, 그 결과 비동기식으로 실행될 세 가지 작업이 생성됩니다. 나중에 activityB의 작업에 오류가 발생한다고 가정하면 Amazon SWF에서는 이 오류를 내역에 기록합니다. 이를 처리하기 위해 프레임워크에서는 먼저 동일한 doTry()의 범위 내에서 시작된 다른 모든 작업(이 경우에는 activityA 및 activityC)을 취소하려고 할 것입니다. 그러한 작업이 모두 완료되면(취소, 실패 또는 성공적으로 완료) 적절한 doCatch() 메서드가 호출되어 오류를 처리합니다.

c()가 실행되지 않은 동기식 예시와 달리 activityC가 호출되었고 작업이 실행 예약되었으므로 프레임워크에서는 이를 취소하려고 하겠지만, 취소된다는 보장은 없습니다. 취소를 보장할 수 없는 이유는 활동이 이미 완료되었을 수 있거나 취소 요청을 무시할 수 있거나 오류로 인해 실패할 수 있기 때문입니다. 하지만 프레임워크에서는 상응하는 doTry()에서 시작된 모든 작업이 완료된 후에만 doCatch()가 호출되도록 보장합니다. 또한 doTry() 및 doCatch()에서 시작된 모든 작업이 완료된 후에만 doFinally()가 호출되도록 보장합니다. 예를 들어 위 예시의 활동이 서로에게 의존한다면, 즉 activityB에서 activityA를 의존하고 activityC에서 activityB를 의존한다면 다음과 같이 activityC는 activityB가 완료될 때까지 Amazon SWF에 예약되어 있지 않기 때문에 즉시 취소됩니다.

```
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();
    }
```

```
};
```

활동 하트비트

AWS Flow Framework for Java의 협력적 취소 메커니즘을 통해 진행 중인 활동 작업이 정상적으로 취소될 수 있습니다. 취소가 트리거되면 차단되거나 작업자에게 할당되기를 기다리던 작업은 자동으로 취소됩니다. 그러나 작업이 이미 작업자에게 할당된 경우 프레임워크에서는 활동에게 취소를 요청합니다. 활동 구현에서는 그러한 취소 요청을 명시적으로 처리해야 합니다. 이렇게 하려면 활동의 하트비트를 보고하면 됩니다.

하트비트 보고를 통해 활동 구현에서는 진행 중인 활동 작업의 진행 상황을 보고할 수 있습니다. 이는 모니터링에 유용하며 이를 통해 활동에서는 취소 요청을 확인할 수 있습니다. 취소 요청이 이루어지면 `recordActivityHeartbeat` 메서드에서는 `CancellationException` 예외가 발생합니다. 활동 구현에서는 이 예외를 포착하고 취소 요청에 따라 작업을 수행하거나 예외를 삼켜 요청을 무시할 수 있습니다. 취소 요청을 준수하기 위해서는 활동에서 원하는 대로 정리 작업을 수행한 후(해당되는 경우) `CancellationException` 예외를 다시 제기합니다. 이 예외가 활동 구현에서 발생되었다면 프레임워크에서는 이 활동 작업이 취소됨 상태로 완료되었다고 기록합니다.

다음 예시에서는 이미지를 다운로드하고 처리하는 활동을 보여줍니다. 이 활동은 각 이미지를 처리한 후 하트비트하고 취소가 요청되면 정리한 후 예외를 다시 제기하여 취소를 인정합니다.

```
@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;
        }
    }
}
```

보고 활동 하트비트는 필요하지 않지만, 해당 활동이 장시간 실행되거나 오류 조건 하에서는 취소되길 원하는 고비용 작업을 수행할 수 있는 경우에는 사용하는 것이 좋습니다. 사용자는 활동 구현에서 `heartbeatActivityTask`를 주기적으로 호출해야 합니다.

활동에서 제한 시간을 초과하면 `ActivityTaskTimedOutException` 예외가 발생하고 예외 객체의 `getDetails`에서는 상응하는 활동 작업을 위해 `heartbeatActivityTask`에 대한 마지막 성공적 호출로 전달된 데이터를 반환합니다. 워크플로 구현에서는 이 정보를 사용하여 제한 시간을 초과하기 전까지 활동 작업이 얼마나 진행되었는지 확인할 수 있습니다.

> **Note**
>
> Amazon SWF에서는 하트비트 요청을 제한할 수 있으므로 너무 자주 하트비트하는 것은 좋지 않습니다. Amazon SWF에서 설정한 제한에 대해서는 Amazon Simple Workflow Service 개발자 안내서를 참조하십시오.

명시적으로 작업 취소

오류 조건 외에도 사용자가 명시적으로 작업을 취소할 수 있는 경우가 있습니다. 예를 들어 신용 카드 결제를 처리하는 활동은 사용자가 주문을 취소한 경우 취소되어야 합니다. 프레임워크를 통해 사용자는 `TryCatchFinally` 범위에서 생성된 작업을 명시적으로 취소할 수 있습니다. 다음 예시에서는 결제가 처리되고 있는 도중에 신호가 수신되면 결제 작업이 취소됩니다.

```
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);
        }
    }
}
```

취소된 작업에 대한 알림 수신

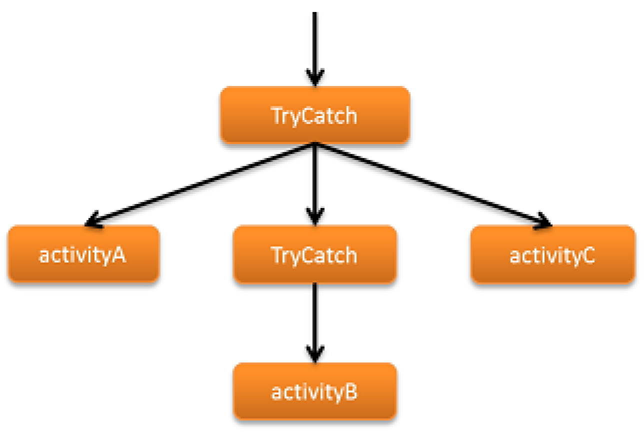
작업이 취소됨 상태로 완료되면 프레임워크에서는 `CancellationException` 예외를 제기하여 워크플로 로직에 이를 알립니다. 활동이 취소됨 상태로 완료되면 내역에 레코드가 생성되고 프레임워크에서는 `CancellationException`으로 적절한 `doCatch()`를 호출합니다. 앞의 예시와 같이 결제 처리 작업이 취소되면 워크플로에서는 `CancellationException`을 수신합니다.

미처리 `CancellationException`은 기타 예외와 마찬가지로 상위 실행 브랜치로 전파됩니다. 그러나 `doCatch()` 메서드에서는 해당 범위에 다른 예외가 없는 경우에만 `CancellationException`을 수신하고, 다른 예외는 취소보다 높은 우선 순위를 부여받습니다.

중첩 TryCatchFinally

사용자는 필요에 따라 `TryCatchFinally`를 중첩할 수 있습니다. 각 `TryCatchFinally`에서는 실행 트리에 새 브랜치를 생성하므로 사용자는 중첩된 범위를 생성할 수 있습니다. 상위 범위에 예외가 발생하면 그 안에 있는 중첩 `TryCatchFinally`에서 시작된 모든 작업에 대해 취소 시도가 이루어집니다. 그러나 중첩 `TryCatchFinally`에서 발생한 예외는 상위로 자동 전파되지는 않습

니다. 예외를 중첩 TryCatchFinally에서 이를 포함하는 TryCatchFinally로 전파하고 싶다면 이 예외를 doCatch()에서 다시 제기해야 합니다. 바꿔 말하면 Java의 try/catch와 마찬가지로 미처리 예외만 표시됩니다. 취소 메서드를 호출하여 중첩 TryCatchFinally를 취소하면 중첩 TryCatchFinally가 취소되지만 이를 포함하는 TryCatchFinally는 자동으로 취소되지 않습니다.




```
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);
    }
};
```

실패한 활동 재시도

잠시 연결이 끊기는 등 일시적인 이유로 활동이 실패하는 경우가 가끔 있습니다. 딴 때는 활동이 성공할 수 있으므로 많은 경우 활동 실패를 처리하는 적절한 방법은 활동을 재시도하는 것입니다. 시도 횟수는 여러 번이 될 수 있습니다.

활동 재시도와 관련해 다양한 전략이 있지만, 워크플로의 세부 내용에 따라 가장 좋은 전략이 무엇인지 결정됩니다. 그러한 전략은 다음과 같은 세 가지 기본 범주로 구분됩니다.

- 성공할 때까지 재시도하는 전략은 활동이 완료될 때까지 계속해서 재시도하는 것입니다.
- 기하급수적 재시도 전략에서는 활동이 완료되거나 프로세스가 최대 시도 횟수와 같이 지정된 중단 지점에 이를 때까지 재시도 사이의 시간 간격을 기하급수적으로 늘립니다.
- 사용자 지정 재시도 전략에서는 시도가 실패한 후에 활동을 재시도할지 여부, 재시도한다면 재시도 방법을 결정합니다.

다음 단원에서는 이러한 전략을 구현하는 방법에 관해 설명합니다. 예시 워크플로 작업자는 모두 단일 활동인 `unreliableActivity`를 사용하는데, 이 활동에서는 임의로 다음 중 하나를 수행합니다.

- 즉시 완료
- 의도적으로 제한 시간 값을 초과하여 실패
- 의도적으로 `IllegalStateException`을 발생시켜 실패

성공할 때까지 재시도하는 전략

가장 단순한 재시도 전략은 결국 성공할 때까지 활동 재시도가 실패할 때마다 계속 재시도하는 것입니다. 기본 패턴은 다음과 같습니다.

1. 워크플로의 진입점 메서드에서 중첩된 `TryCatch` 또는 `TryCatchFinally` 클래스 구현합니다.
2. `doTry`에서 활동을 실행합니다.
3. 이 활동이 실패한 경우, 프레임워크에서 `doCatch`를 호출하면 진입점 메서드가 다시 실행됩니다.
4. 활동이 성공적으로 완료될 때까지 2-3단계를 반복합니다.

다음 워크플로에서는 성공할 때까지 재시도하는 전략을 구현합니다. 이 워크플로 인터페이스는 `RetryActivityRecipeWorkflow`에서 구현되고 이 워크플로의 진입점인 `runUnreliableActivityTillSuccess`라는 메서드가 한 개 있습니다. 워크플로 작업자는 `RetryActivityRecipeWorkflowImpl`에서 다음과 같이 구현됩니다.

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

워크플로는 다음과 같이 작동합니다.

1. `runUnreliableActivityTillSuccess`에서는 활동이 실패했는지 여부와 재시도할지 여부를 나타내는 데 사용되는 `retryActivity`라는 `Settable<Boolean>` 객체를 생성합니

다. Settable<T>는 Promise<T>에서 파생된 것으로서 거의 동일한 방식으로 작동하지만, Settable<T> 객체의 값은 사용자가 수동으로 설정해 줍니다.

2. runUnreliableActivityTillSuccess에서는 익명의 중첩된 TryCatch 클래스를 구현하여 unreliableActivity 활동에서 발생하는 모든 예외를 처리합니다. 비동기식 코드에서 발생하는 예외를 처리하는 방법에 관한 자세한 내용은 오류 처리 단원을 참조하십시오.
3. doTry에서는 activityRanSuccessfully라는 Promise<Void> 객체를 반환하는 unreliableActivity 활동을 실행합니다.
4. doTry에서는 다음과 같이 파라미터가 두 개인 비동기식 setRetryActivityToFalse 메서드를 호출합니다.
   - activityRanSuccessfully에서는 unreliableActivity 활동에서 반환하는 Promise<Void> 객체를 받아들입니다.
   - retryActivity에서는 retryActivity 객체를 받아들입니다.

   unreliableActivity가 완료되면 activityRanSuccessfully가 준비 상태가 되고 setRetryActivityToFalse에서는 retryActivity를 false로 설정합니다. 완료되지 않으면 activityRanSuccessfully는 결코 준비 상태가 되지 않고 setRetryActivityToFalse는 실행되지 않습니다.
5. unreliableActivity에서 예외가 발생하면 프레임워크에서는 doCatch를 호출하여 예외 객체를 전달합니다. doCatch는 retryActivity를 true로 설정합니다.
6. runUnreliableActivityTillSuccess에서는 비동기식 restartRunUnreliableActivityTillSuccess 메서드를 호출한 후 이 메서드에 retryActivity 객체를 전달합니다. retryActivity이 Promise<T> 유형이므로 restartRunUnreliableActivityTillSuccess에서는 retryActivity가 준비 상태가 될 때까지 실행을 연기합니다. 실행은 TryCatch가 완료된 후에 이뤄집니다.
7. retryActivity가 준비 상태가 되면 restartRunUnreliableActivityTillSuccess에서 값을 추출합니다.
   - 값이 false이면 재시도가 성공한 것입니다. restartRunUnreliableActivityTillSuccess는 아무것도 하지 않고 재시도 시퀀스는 종료됩니다.
   - 값이 true이면 재시도가 실패한 것입니다. restartRunUnreliableActivityTillSuccess에서 runUnreliableActivityTillSuccess를 호출하여 다시 활동을 실행합니다.
8. unreliableActivity가 완료될 때까지 1-7단계가 반복됩니다.

Note

doCatch에서는 예외를 처리하지 않고 단지 retryActivity 객체를 true로 설정하여 활동이 실패하였음을 나타냅니다. 재시도는 비동기식 restartRunUnreliableActivityTillSuccess 메서드에서 처리하는데, 이 메서드에서는 TryCatch가 완료될 때까지 실행을 연기합니다. 이 접근 방식을 사용하는 이유는 doCatch에서 활동을 재시도하면 이를 취소할 수 없기 때문입니다. restartRunUnreliableActivityTillSuccess에서 활동을 재시도하면 사용자는 취소 가능한 활동을 실행할 수 있습니다.

기하급수적 재시도 전략

기하급수적 재시도 전략의 경우 지정된 시간, 즉 N초 후 프레임워크에서 실패한 활동을 다시 실행합니다. 이 시도가 실패하면 프레임워크에서는 2N초 후에, 이어서 4N초 후에 실행하는 식으로 계속해서 활동을 다시 실행합니다. 대기 시간이 상당히 길어질 수 있으므로 일반적으로 무한정 재시도를 계속하는 것보다는 일정 시점에서 중단하는 것이 좋습니다.

프레임워크에서는 다음과 같이 기하급수적 재시도 전략을 구현할 수 있는 세 가지 방식을 제공합니다.

- @ExponentialRetry 주석은 가장 단순한 접근 방식이지만, 컴파일 시간에 재시도 구성 옵션을 설정해야 합니다.
- RetryDecorator 클래스를 통해 런타임에 재시도 구성을 설정하고 필요에 따라 이를 변경할 수 있습니다.
- AsyncRetryingExecutor 클래스를 통해 런타임에 재시도 구성을 설정하고 필요에 따라 이를 변경할 수 있습니다. 뿐만 아니라 프레임워크에서는 사용자 구현 AsyncRunnable.run 메서드를 호출하여 매번 재시도를 실행합니다.

모든 접근 방식에서는 다음 구성 옵션을 지원합니다. 이 옵션에서 시간 값은 초 단위입니다.

- 최초 재시도 대기 시간
- 다음과 같이 재시도 간격을 계산하는 데 사용되는 백오프 계수

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

  기본값은 2.0입니다.

- 최대 재시도 횟수. 기본값은 무제한입니다.
- 최대 재시도 간격. 기본값은 무제한입니다.
- 만료 시간. 프로세스의 총 시간이 이 값을 초과하면 재시도가 중단됩니다. 기본값은 무제한입니다.
- 재시도 프로세스를 트리거할 예외. 기본적으로 모든 예외가 재시도 프로세스를 트리거합니다.
- 재시도를 트리거하지 않을 예외. 기본적으로 어떤 예외도 배제되지 않습니다.

다음 단원에서는 기하급수적 재시도 전략을 구현하는 다양한 방법에 관해 설명합니다.

@ExponentialRetry을 이용한 기하급수적 재시도

활동에 대해 기하급수적 재시도 전략을 구현하는 가장 간단한 방법은 인터페이스 정의에서 활동에 대해 `@ExponentialRetry` 주석을 붙이는 것입니다. 활동이 실패하면 프레임워크에서는 지정된 옵션 값에 따라 재시도 프로세스를 자동으로 처리합니다. 기본 패턴은 다음과 같습니다.

1. `@ExponentialRetry`를 적절한 활동에 붙이고 재시도 구성을 지정합니다.
2. 주석이 달린 활동이 실패하면 프레임워크에서는 주석의 인수에서 지정한 구성에 따라 활동을 자동으로 재시도합니다.

`ExponentialRetryAnnotationWorkflow` 워크플로 작업자는 `@ExponentialRetry` 주석을 사용하여 기하급수적 재시도 전략을 구현합니다. 이 작업자는 다음과 같이 인터페이스 정의가 `ExponentialRetryAnnotationActivities`에 구현되는 `unreliableActivity` 활동을 사용합니다.

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

`@ExponentialRetry`옵션에서는 다음과 같은 전략을 지정합니다.

- 활동에서 `IllegalStateException`이 발생한 경우에만 재시도합니다.

- 초기 대기 시간 5초를 사용합니다.
- 재시도를 5회 이내로 제한합니다.

이 워크플로 인터페이스는 `RetryWorkflow`에서 구현되고 이 워크플로의 진입점인 `process`라는 메서드가 한 개 있습니다. 워크플로 작업자는 `ExponentialRetryAnnotationWorkflowImpl`에서 다음과 같이 구현됩니다.

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

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

워크플로는 다음과 같이 작동합니다.

1. `process`에서는 동기식 `handleUnreliableActivity` 메서드를 실행합니다.
2. `handleUnreliableActivity`에서는 `unreliableActivity` 활동을 실행합니다.

`IllegalStateException` 예외가 발생하여 활동이 실패하면 프레임워크에서는 `ExponentialRetryAnnotationActivities`에 지정된 재시도 전략을 자동으로 실행합니다.

RetryDecorator 클래스를 이용한 기하급수적 재시도

`@ExponentialRetry`는 사용하기 간편합니다. 그러나 구성이 정적이고 컴파일 시에 설정되므로 프레임워크에서는 활동이 실패할 때마다 동일한 재시도 전략을 사용합니다. 런타임에 구성을 지정하고 필요에 따라 이를 변경할 수 있게 해주는 `RetryDecorator` 클래스를 사용하여 더 유연한 기하급수적 재시도 전략을 구현할 수 있습니다. 기본 패턴은 다음과 같습니다.

1. 재시도 구성을 지정하는 `ExponentialRetryPolicy` 객체를 만들고 구성합니다.
2. `RetryDecorator` 객체를 만들고 1단계의 `ExponentialRetryPolicy` 객체를 생성자에게 전달합니다.
3. 활동 클라이언트의 클래스 이름을 `RetryDecorator` 객체의 장식 메서드에 전달하여 장식자 객체를 활동에 추가합니다.

4. 활동을 실행합니다.

활동이 실패하면 프레임워크에서는 `ExponentialRetryPolicy` 객체의 구성에 따라 활동을 재시도합니다. 이 객체를 수정하여 필요에 따라 재시도 구성을 변경할 수 있습니다.

> **Note**
>
> `@ExponentialRetry`주석 및 `RetryDecorator` 클래스는 상호 배타적입니다. `@ExponentialRetry` 주석에서 지정한 재시도 정책을 동적으로 재정의하는 데 `RetryDecorator`를 사용할 수는 없습니다.

다음 워크플로 구현에서는 `RetryDecorator` 클래스를 사용하여 기하급수적 재시도 전략을 구현하는 방법을 보여줍니다. 이 구현에서는 `@ExponentialRetry` 주석이 없는 `unreliableActivity` 활동을 사용합니다. 이 워크플로 인터페이스는 `RetryWorkflow`에서 구현되고 이 워크플로의 진입점인 `process`라는 메서드가 한 개 있습니다. 워크플로 작업자는 `DecoratorRetryWorkflowImpl`에서 다음과 같이 구현됩니다.

```
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();
  }
}
```

워크플로는 다음과 같이 작동합니다.

1. `process`에서는 다음 방법으로 `ExponentialRetryPolicy` 객체를 생성 및 구성합니다.
   - 초기 재시도 간격을 생성자에게 전달

- 객체의 withMaximumAttempts 메서드를 호출하여 최대 시도 횟수를 5로 설정. ExponentialRetryPolicy에서는 다른 구성 옵션을 지정하는 데 사용할 수 있는 다른 with 객체를 노출합니다.

2. process에서는 retryDecorator이라는 RetryDecorator 객체를 만들고 1단계의 ExponentialRetryPolicy 객체를 생성자에게 전달합니다.
3. process에서는 retryDecorator.decorate 메서드를 호출한 후 이 메서드에 활동 클라이언트의 클래스 이름을 전달하여 활동에 장식자를 추가합니다.
4. handleUnreliableActivity에서는 활동을 실행합니다.

활동이 실패하면 프레임워크에서는 1단계에서 지정한 구성에 따라 활동을 재시도합니다.

Note

ExponentialRetryPolicy 클래스의 with 메서드 중 몇 가지, 즉 setBackoffCoefficient, setMaximumAttempts, setMaximumRetryIntervalSeconds 및 setMaximumRetryExpirationIntervalSeconds는 사용자가 호출하여 언제든지 해당 구성 옵션을 변경할 수 있는 해당 set 메서드가 있습니다.

AsyncRetryingExecutor 클래스를 이용한 기하급수적 재시도

RetryDecorator 클래스에서는 재시도 프로세스 구성과 관련해 @ExponentialRetry보다 더 많은 유연성을 제공하지만, 프레임워크에서는 ExponentialRetryPolicy 객체의 현재 구성에 따라 여전히 재시도를 자동으로 실행합니다. 더 유연한 접근 방식은 AsyncRetryingExecutor 클래스를 사용하는 것입니다. 런타임에 재시도 프로세스를 구성할 수 있게 해줄 뿐 아니라 프레임워크에서는 단순히 활동을 실행하는 대신에 사용자 구현 AsyncRunnable.run 메서드를 호출하여 각 재시도를 실행합니다.

기본 패턴은 다음과 같습니다.

1. 재시도 구성을 지정할 ExponentialRetryPolicy 객체를 만들고 구성합니다.
2. AsyncRetryingExecutor 객체를 생성하여 이 객체에 ExponentialRetryPolicy 객체와 워크플로 클록의 인스턴스를 전달합니다.
3. 익명의 중첩된 TryCatch 또는 TryCatchFinally 클래스를 구현합니다.

4. 익명의 AsyncRunnable 클래스를 구현하고 run 메서드를 재정의하여 활동 실행을 위한 사용자 지정 코드를 구현합니다.
5. doTry를 재정의하여 AsyncRetryingExecutor 객체의 execute 메서드를 호출한 후 이 메서드에 단계 4의 AsyncRunnable 클래스를 전달합니다. AsyncRetryingExecutor 객체에서는 AsyncRunnable.run을 호출하여 활동을 실행합니다.
6. 활동이 실패하면 AsyncRetryingExecutor 객체에서는 1단계에서 지정한 재시도 정책에 따라 AsyncRunnable.run 메서드를 다시 호출합니다.

다음 워크플로에서는 AsyncRetryingExecutor 클래스를 사용하여 기하급수적 재시도 전략을 구현하는 방법을 보여줍니다. 이 워크플로에서는 앞서 설명한 DecoratorRetryWorkflow 워크플로 동일한 unreliableActivity 활동을 사용합니다. 이 워크플로 인터페이스는 RetryWorkflow에서 구현되고 이 워크플로의 진입점인 process라는 메서드가 한 개 있습니다. 워크플로 작업자는 AsyncExecutorRetryWorkflowImpl에서 다음과 같이 구현됩니다.

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

워크플로는 다음과 같이 작동합니다.

1. process에서는 handleUnreliableActivity 메서드를 호출한 후 이 메서드에 구성 설정을 전달합니다.
2. handleUnreliableActivity에서는 1단계의 구성 설정을 사용하여 ExponentialRetryPolicy인 retryPolicy를 만듭니다.
3. handleUnreliableActivity에서는 AsyncRetryExecutor 객체인 executor를 생성하여 2 단계의 ExponentialRetryPolicy 객체와 워크플로 클록의 인스턴스를 생성자에게 전달합니다.
4. handleUnreliableActivity에서는 익명의 중첩된 TryCatch 클래스를 구현하고 doTry 및 doCatch 메서드를 재정의하여 재시도를 실행하고 모든 예외를 처리합니다.
5. doTry에서는 익명의 AsyncRunnable 클래스를 구현하고 run 메서드를 재정의하여 unreliableActivity 실행을 위한 사용자 지정 코드를 구현합니다. 간소화를 위해 run에서는 활동을 수행할 뿐이지만 사용자는 상황에 따라 적절하게 더 정교한 접근 방식을 구현할 수 있습니다.
6. doTry에서는 executor.execute를 호출한 후 이 메서드에 AsyncRunnable 객체를 전달합니다. execute에서는 AsyncRunnable 객체의 run 메서드를 호출하여 활동을 실행합니다.
7. 활동이 실패하면 실행기에서는 retryPolicy 객체 구성에 따라 run을 다시 호출합니다.

TryCatch 클래스를 사용하여 오류를 처리하는 방법에 관한 자세한 내용은 AWS Flow Framework for Java 예외 단원을 참조하십시오.

사용자 지정 재시도 전략

실패한 활동을 재시도하기 위한 가장 유연한 접근 방식은 사용자 지정 전략으로서, 성공할 때까지 재시도하는 전략과 흡사하게 이 전략에서는 재시도를 실행하는 비동기식 메서드를 반복적으로 호출합니다. 그러나 사용자는 단순히 활동을 다시 실행하는 대신에 연속적인 재시도 실행 여부 및 방법을 결정하는 사용자 지정 로직을 구현합니다. 기본 패턴은 다음과 같습니다.

1. 활동이 실패했는지 여부를 나타내는 데 사용되는 Settable<T> 상태 객체를 생성합니다.

2. 중첩된 `TryCatch` 또는 `TryCatchFinally` 클래스를 구현합니다.
3. `doTry`에서는 활동을 실행합니다.
4. 활동이 실패하면 `doCatch`에서는 상태 객체가 활동이 실패했음을 나타내도록 설정합니다.
5. 비동기식 실패 처리 메서드를 호출한 후 이 메서드에 상태 객체를 전달합니다. 이 메서드에서는 `TryCatch` 또는 `TryCatchFinally`가 완료될 때까지 실행을 연기합니다.
6. 실패 처리 메서드에서는 활동을 재시도할지 여부, 그리고 재시도한다면 언제할지 그 시점을 결정합니다.

다음 워크플로에서는 사용자 지정 재시도 전략을 구현하는 방법을 보여줍니다. 이 워크플로에서는 `DecoratorRetryWorkflow` 및 `AsyncExecutorRetryWorkflow` 워크플로와 동일한 `unreliableActivity` 활동을 사용합니다. 이 워크플로 인터페이스는 `RetryWorkflow`에서 구현되고 이 워크플로의 진입점인 `process`라는 메서드가 한 개 있습니다. 워크플로 작업자는 `CustomLogicRetryWorkflowImpl`에서 다음과 같이 구현됩니다.

```
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;
    }
}
```

워크플로는 다음과 같이 작동합니다.

1. process에서는 비동기식 callActivityWithRetry 메서드를 실행합니다.
2. callActivityWithRetry에서는 활동이 실패했는지 여부를 나타내는 데 사용되는 실패라는 이름의 Settable<Throwable> 객체를 생성합니다. Settable<T>는 Promise<T>에서 파생된 것으로서 거의 동일한 방식으로 작동하지만, Settable<T> 객체의 값은 사용자가 수동으로 설정해 줍니다.
3. callActivityWithRetry에서는 익명의 중첩된 TryCatchFinally 클래스를 구현하여 unreliableActivity에서 발생하는 모든 예외를 처리합니다. 비동기식 코드에서 발생하는 예외를 처리하는 방법에 관한 자세한 내용은 AWS Flow Framework for Java 예외 단원을 참조하십시오.
4. doTry에서는 unreliableActivity를 실행합니다.
5. unreliableActivity에서 예외가 발생하면 프레임워크에서는 doCatch를 호출하여 예외 객체를 전달합니다. doCatch에서는 failure를 예외 객체로 설정합니다. 이로써 활동이 실패하였음을 나타내고 객체를 준비 상태로 둡니다.
6. doFinally에서는 failure가 준비 상태인지를 점검합니다. 준비 상태는 failure를 doCatch에서 설정한 경우에만 true가 됩니다.
   - failure가 준비 상태인 경우 doFinally는 아무 것도 하지 않습니다.
   - failure가 준비 상태가 아니면 완료된 활동과 doFinally에서는 실패를 null로 설정합니다.
7. callActivityWithRetry에서는 비동기식 retryOnFailure 메서드를 호출한 후 이 메서드에 실패를 전달합니다. 실패는 Settable<T> 유형이므로 callActivityWithRetry에서는 실패가 준비 상태가 될 때까지 실행을 연기합니다. 실행은 TryCatchFinally가 완료된 후에 이루어집니다.
8. retryOnFailure에서는 실패에서 값을 가져옵니다.
   - 실패가 null로 설정되어 있으면 재시도가 성공한 것입니다. retryOnFailure는 아무것도 하지 않고, 이로써 재시도 프로세스는 종료됩니다.

- 실패가 예외 객체로 설정되고 shouldRetry에서 true를 반환하면 retryOnFailure에서는 callActivityWithRetry를 호출하여 활동을 재시도합니다.

  shouldRetry에서는 실패한 활동을 재시도할지 여부를 결정하는 사용자 지정 로직을 구현합니다. 간소화를 위해 shouldRetry에서는 항상 true를 반환하고 retryOnFailure에서는 즉시 활동을 실행하지만 사용자는 필요에 따라 더 정교한 로직을 구현할 수 있습니다.

9. unreliableActivity가 완료되거나 shouldRetry에서 프로세스를 중단하기로 결정할 때까지 2~8단계가 반복됩니다.

> **Note**
>
> doCatch에서는 재시도 프로세스를 처리하지 않고 단지 활동이 실패하였음을 나타내도록 실패를 설정합니다. 재시도 프로세스는 비동기식 retryOnFailure 메서드에서 처리하는데, 이 메서드에서는 TryCatch가 완료될 때까지 실행을 연기합니다. 이 접근 방식을 사용하는 이유는 doCatch에서 활동을 재시도하면 이를 취소할 수 없기 때문입니다. retryOnFailure에서 활동을 재시도하면 사용자는 취소 가능한 활동을 실행할 수 있습니다.

대몬(daemon) 작업

AWS Flow Framework for Java를 통해 특정 작업을 daemon으로 표시할 수 있습니다. 이렇게 하면 다른 모든 작업이 완료되면 취소되어야 하는 일부 배경 작업을 수행하는 작업을 생성할 수 있습니다. 예를 들어 상태 모니터링 작업은 나머지 워크플로가 완료되면 취소되어야 합니다. 이를 위해서는 비동기식 메서드 또는 TryCatchFinally의 인스턴스에 daemon 플래그를 설정하면 됩니다. 다음 예시에서는 비동기식 메서드인 monitorHealth()가 daemon으로 표시됩니다.

```
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();
  }
}
```

위 예시에서 doUsefulWorkActivity가 완료되면 monitoringHealth는 자동으로 취소됩니다. 그러면 이 비동기식 메서드에 근간을 둔 실행 분기 전체가 취소됩니다. 취소의 의미는 TryCatchFinally에서의 의미와 같습니다. 이와 마찬가지로 부울 플래그를 생성자로 전달하여 TryCatchFinally 대몬(daemon)을 표시할 수 있습니다.

```
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
            }
        };
    }
}
```

TryCatchFinally 내에서 시작된 대몬(daemon) 작업은 해당 작업이 생성된 컨텍스트로 범위가 지정됩니다. 즉doTry(), doCatch() 또는 doFinally() 메서드로 범위가 지정됩니다. 예를 들면 다음 예시에서 startMonitoring 비동기식 메서드는 대몬(daemon)으로 표시되고 doTry()의 호출을 받습니다. 이에 대해 생성된 작업은 doTry() 내에서 시작된 다른 작업(이 경우에는 doUsefulWorkActivity)이 완료되자마자 취소됩니다.

```
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 for Java 다시 재생 동작

이 주제에서는 AWS Flow Framework Java용이란 무엇입니까? 단원의 예시를 사용하여 다시 재생 동작의 예시를 다룹니다. 동기식 및 비동기식 시나리오 둘 다 설명합니다.

예 1: 동기식 다시 재생

동기식 워크플로에서 다시 재생이 작동하는 방식의 예를 보려면 다음과 같이 해당 구현 내에 `println` 호출을 추가하여 HelloWorldWorkflow 워크플로와 액티비티 구현을 수정합니다.

```
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);
   }
}
```

코드에 대한 자세한 내용은 HelloWorldWorkflow 애플리케이션 단원을 참조하십시오. 다음은 출력을 편집한 버전으로서, 각 다시 재생 에피소드의 시작을 나타내는 설명이 포함되어 있습니다.

```
//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
```

이 예시의 다시 재생 프로세스는 다음과 같이 작동합니다.

- 첫 번째 에피소드에서는 종속성이 없는 `getName` 활동 작업을 예약합니다.
- 두 번째 에피소드에서는 `getName`에 의존하는 `getGreeting` 활동 작업을 예약합니다.
- 세 번째 에피소드에서는 `getGreeting`에 의존하는 `say` 활동 작업을 예약합니다.
- 마지막 에피소드에서는 추가 작업을 예약하지 않고 미완료 활동을 검색하지 않습니다. 이로써 워크플로 실행은 종료됩니다.

> **Note**
> 세 가지 활동 클라이언트 메서드는 각 에피소드에 대해 한 번 호출됩니다. 그러나 그러한 호출 중 하나만 활동 작업을 결과로 얻으므로 각 작업은 한 번만 수행됩니다.

예 2: 비동기식 다시 재생

동기식 다시 재생 예시와 마찬가지로 사용자는 HelloWorldWorkflowAsync 애플리케이션을 수정하여 비동기식 다시 재생이 어떻게 작동하는지 확인할 수 있습니다. 이렇게 하면 다음과 같이 출력됩니다.

```
//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
```

두 가지 활동만 있으므로 HelloWorldAsync에서는 다시 재생 에피소드를 세 개 사용합니다. `getGreeting` 활동은 getGreeting 비동기식 워크플로 메서드로 대치되었습니다. 이 메서드는 완료될 때 다시 재생 에피소드를 시작하지 않습니다.

첫 번째 에피소드는 이름 활동의 완료 여부에 따라 달라지므로 `getGreeting`을 직접적으로 호출하지 않습니다. 그러나 getName이 완료된 후에 다시 재생에서는 이어지는 각 에피소드에 대해 getGreeting을 한 번 호출합니다.

참고 항목

- AWS Flow Framework 기본 개념: 분산 실행

Under the Hood

주제

- 태스크
- 실행 순서
- 워크플로 실행
- 불확정성

태스크

AWS Flow Framework for Java에서 비동기식 코드의 실행을 관리하는 데 사용하는 기본 프리미티브는 Task 클래스입니다. Task 유형의 객체는 비동기식으로 수행해야 할 작업을 나타냅니다. 사용자가 비동기식 메서드를 호출하면 프레임워크에서는 이 메서드에 코드를 실행할 Task를 생성하고 나중에 실행될 것에 대비해 이를 목록에 배치합니다. 이와 마찬가지로 Activity를 호출하면 이에 대해 Task가 생성됩니다. 이 작업이 완료된 후에 메서드 호출에서는 결과를 반환하는데, 일반적으로 호출의 향후 결과로 Promise<T>를 반환합니다.

Task 클래스는 퍼블릭이므로 직접 사용할 수 있습니다. 예를 들어 우리는 Hello World 예시를 다시 작성하여 비동기식 메서드 대신에 Task를 사용할 수 있습니다.

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

Task의 생성자로 전달된 모든 Promise가 준비 상태가 되면 프레임워크에서는 doExecute() 메서드를 호출합니다. Task 클래스에 관한 자세한 내용은 AWS SDK for Java 설명서를 참조하십시오.

또한 프레임워크에는 Promise<T>이기도 한 Task를 나타내는 Functor라는 클래스가 포함되어 있습니다. Task가 완료되면 Functor 객체는 준비 상태가 됩니다. 다음 예시에서는 Functor가 생성되어 인사말 메시지를 가져옵니다.

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

실행 순서

상응하는 비동기식 메서드 또는 활동으로 전달된 모든 `Promise<T>` 유형의 파라미터가 준비 상태가 될 때만 작업은 실행 가능해집니다. 실행할 준비가 된 `Task`는 논리적으로 준비 상태인 대기열로 이동합니다. 바꿔 말하면 실행을 위해 예약됩니다. 작업자 클래스에서는 사용자가 비동기식 메서드의 본문에 작성한 코드를 간접적으로 호출하거나 활동 메서드의 경우 Amazon Simple Workflow Service(AWS)의 활동 작업을 예약하여 작업을 실행합니다.

작업이 실행되고 결과가 산출되면 이로 인해 다른 작업이 준비 상태가 되고 프로그램 실행은 계속해서 진행됩니다. 프레임워크에서 작업을 실행하는 방식은 사용자의 비동기식 코드의 실행 순서를 이해하는 데 중요합니다. 프로그램에 순차적으로 표시되는 코드는 실제로는 그러한 순서로 실행되지 않을 수 있습니다.

```
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!");
}
```

위 목록에 있는 코드는 다음과 같이 출력됩니다.

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

이것은 예상치 못한 것일 수 있지만 다음과 같이 비동기식 메서드에 대한 작업이 실행된 방식을 짚어보면 쉽게 설명될 수 있습니다.

1. `getUserName`을 호출하면 `Task`가 생성됩니다. 이것을 `Task1`이라고 합시다. `getUserName`에서는 파라미터를 받아들이지 않으므로 `Task1`는 즉시 준비 상태인 대기열에 배치됩니다.
2. 그다음에 `printHelloName`을 호출하면 `getUserName`의 결과를 기다려야 하는 `Task`가 생성됩니다. 이것을 `Task2`이라고 합시다. 필요한 값이 아직 준비 상태가 아니므로 `Task2`는 대기 목록에 배치됩니다.
3. 그다음에 `printHelloWorld`에 대한 작업이 생성되어 준비 상태인 대기열에 추가됩니다. 이것을 `Task3`이라고 합시다.
4. `println` 문은 "Hello, Amazon!"을 콘솔에 출력합니다.
5. 이 시점에서 `Task1` 및 `Task3`는 준비 상태인 대기열에 있고 `Task2`는 대기 목록에 있습니다.
6. 작업자는 `Task1`을 실행하고 그 결과로 인해 `Task2`가 준비 상태가 됩니다. `Task2`는 준비 상태인 대기열에서 `Task3`의 뒤에 추가됩니다.
7. 그러면 `Task3` 및 `Task2`는 이 순서대로 실행됩니다.

활동의 실행에서는 동일한 패턴을 따릅니다. 사용자가 활동 클라이언트에서 메서드를 직접적으로 호출하면 이 메서드는 실행되자마자 Amazon SWF에 활동을 예약하는 `Task`를 생성합니다.

프레임워크에서는 메서드 호출을 활동 호출 및 비동기식 작업으로 변환하기 위한 로직을 사용자 프로그램에 주입하기 위해 코드 생성 및 동적 프록시와 같은 기능에 의존합니다.

워크플로 실행

또한 워크플로 구현의 실행은 작업자 클래스에서 관리합니다. 워크플로 클라이언트에서 메서드를 직접적으로 호출하면 이 메서드는 Amazon SWF를 직접적으로 호출하여 워크플로 인스턴스를 생성합니다. Amazon SWF의 작업을 프레임워크의 작업과 혼동해서는 안 됩니다. Amazon SWF의 작업은 활동 작업 또는 결정 작업입니다. 활동 작업의 실행은 간단합니다. 활동 작업자 클래스에서는 Amazon SWF에서 활동 작업을 수신하고, 사용자의 구현에 있는 적절한 활동 메서드를 간접적으로 호출하여 그 결과를 Amazon SWF에 반환합니다.

결정 작업의 실행은 더 많은 것이 수반됩니다. 워크플로 작업자는 Amazon SWF에서 결정 작업을 수신합니다. 결정 작업은 결과적으로 워크플로 로직에게 그다음에 수행할 작업이 무엇인지 묻는 요청입니다. 첫 번째 결정 작업은 워크플로 클라이언트를 통해 시작될 때 워크플로 인스턴스에 대해 생성됩니다. 이 결정 작업을 수신하자마자 프레임워크에서는 `@Execute`이라는 주석이 붙은 워크플로 메서드에 있는 코드를 실행하기 시작합니다. 이 메서드에서는 활동을 예약하는 조정 로직을 실행합니다. 워크플로 인스턴스의 상태가 변경되면 (예: 활동이 완료될 때) 추가 결정 작업이 예약됩니다. 이 시점에 워크플로 로직에서는 활동의 결과에 따라 수행할 작업을 결정할 수 있습니다. 예를 들면 다른 활동을 예약하기로 결정할 수 있습니다.

프레임워크에서는 결정 작업을 워크플로 로직으로 매끄럽게 번역하여 이 모든 세부 정보를 개발자가 볼 수 없도록 숨깁니다. 개발자의 관점에서 이 코드는 정규 프로그램처럼 보입니다. 이면에서 프레임워크는 Amazon SWF에서 보관하는 내역을 사용하여 이를 Amazon SWF 및 결정 작업에 대한 직접 호출과 매핑합니다. 결정 작업이 도착하면 프레임워크에서는 그때까지 완료된 활동의 결과를 플러그인하는 프로그램 실행을 다시 재생합니다. 이 결과를 기다리던 비동기식 메서드 및 활동은 차단이 해제되고 프로그램 실행은 계속 진행됩니다.

예시 이미지 처리 워크플로의 실행과 그에 상응하는 내역이 다음 표에 나와 있습니다.

썸네일 워크플로의 실행

워크플로 프로그램 실행	Amazon SWF에서 보관하는 내역
초기 실행	
1. 디스패치 루프 2. getImageUrls 3. downloadImage 4. createThumbnail(대기열에 있는 작업) 5. uploadImage(대기열에 있는 작업) 6. <다음번 루프 반복>	1. 워크플로 인스턴스가 시작됨, id="1" 2. downloadImage 예약됨
다시 재생	
1. 디스패치 루프 2. getImageUrls 3. downloadImage 이미지 경로="foo" 4. createThumbnail 5. uploadImage(대기열에 있는 작업)	1. 워크플로 인스턴스가 시작됨, id="1" 2. downloadImage 예약됨 3. downloadImage 완료, 반환="foo" 4. createThumbnail 예약됨

워크플로 프로그램 실행	Amazon SWF에서 보관하는 내역
6. <다음번 루프 반복>	
다시 재생	
1. 디스패치 루프 2. getImageUrls 3. downloadImage 이미지 경로="foo" 4. createThumbnail 썸네일 경로="bar" 5. uploadImage 6. <다음번 루프 반복>	1. 워크플로 인스턴스가 시작됨, id="1" 2. downloadImage 예약됨 3. downloadImage 완료, 반환="foo" 4. createThumbnail 예약됨 5. createThumbnail 완료, 반환="bar" 6. uploadImage 예약됨
다시 재생	
1. 디스패치 루프 2. getImageUrls 3. downloadImage 이미지 경로="foo" 4. createThumbnail 썸네일 경로="bar" 5. uploadImage 6. <다음번 루프 반복>	1. 워크플로 인스턴스가 시작됨, id="1" 2. downloadImage 예약됨 3. downloadImage 완료, 반환="foo" 4. createThumbnail 예약됨 5. createThumbnail 완료, 반환="bar" 6. uploadImage 예약됨 7. uploadImage 완료됨 ...

`processImage`에 대한 직접 호출이 실행되면 프레임워크에서는 Amazon SWF에 새 워크플로 인스턴스를 생성합니다. 이것은 시작되려는 워크플로 인스턴스의 내구성 있는 기록입니다. 프로그램은 `downloadImage` 활동이 직접적으로 호출될 때까지 실행되고, 이 활동에서는 Amazon SWF에 활동 예약을 요청합니다. 워크플로는 계속 실행되어 후속 활동을 위한 작업을 생성하지만 `downloadImage` 활동이 완료될 때까지는 실행할 수 없습니다. 따라서 이 재생 에피소드는 종료됩니다. Amazon SWF는 실행을 위해 `downloadImage` 활동에 대한 작업을 디스패치하고, 작업이 완료되면 결과와 함께 기록에 레코드가 생성됩니다. 워크플로는 이제 계속 진행될 준비가 되고 결정 작업이 Amazon SWF에서 생성됩니다. 프레임워크에서는 결정 작업을 수신하고 다운로드된 이미지의 결과를 내역에 기록된 대로 플러그인하는 워크플로를 다시 재생합니다. 이로써 `createThumbnail`에 대한 작업이 차단 해제되고 프로그램의 실행은 Amazon SWF에서 `createThumbnail` 활동 작업을 예약함으로써 계속 진행됩니

다. 동일한 프로세스가 uploadImage에 대해 반복됩니다. 프로그램의 실행은 워크플로에서 모든 이미지를 처리하여 보류 중인 작업이 없을 때까지 이러한 방식으로 계속 진행됩니다. 로컬에서는 실행 상태가 저장되지 않으므로 각 결정 작업은 다른 머신에서 잠재적으로 실행될 수 있습니다. 이를 통해 사용자는 내결함성이 있고 쉽게 확장 가능한 프로그램을 쉽게 작성할 수 있습니다.

불확정성

프레임워크는 다시 재생에 의존하므로 중요한 것은 조율 코드(활동 구현을 제외한 모든 워크플로 코드)가 확정적이어야 한다는 것입니다. 예를 들어 사용자 프로그램의 제어 흐름은 임의의 숫자 또는 현재 시간에 의존해서는 안 됩니다. 이러한 것들은 호출할 때마다 변경되므로 다시 재생에서는 조율 로직을 통해 동일한 경로를 따르지 않을 수 있습니다. 이는 결국 예상치 못한 결과 또는 오류로 이어질 수 있습니다. 프레임워크에서는 현재 시간을 확정적인 방식으로 얻는 데 사용할 수 있는 WorkflowClock을 제공합니다. 자세한 내용은 실행 콘텍스트 단원을 참조하십시오.

> **Note**
>
> 워크플로 구현 객체의 부정확한 Spring 연결 역시 불확정성으로 귀결될 수 있습니다. 워크플로 구현 빈과 이 구현에서 의존하는 빈은 워크플로 범위 내에 있어야 합니다(WorkflowScope). 예를 들어 워크플로 구현 빈을 상태를 유지하고 글로벌 컨텍스트에 있는 빈에 연결하면 예상치 못한 동작이 발생합니다. 자세한 내용은 Spring 통합 단원을 참조하십시오.

문제 해결 및 디버깅 팁

주제

- 컴파일 오류
- 알 수 없는 리소스 장애
- 약속에서 get()을 호출할 때 발생하는 예외
- 불확정적 워크플로
- 버전 관리로 인한 문제
- 워크플로 실행 관련 문제 해결 및 디버깅
- 손실된 작업

이 섹션에서는 Java를 사용하여 워크플로를 개발하는 동안 발생할 수 있는 몇 가지 일반적인 함정에 대해 설명합니다. AWS Flow Framework 또한 문제를 진단하고 디버깅하는 데 도움이 되는 팁도 제공합니다.

컴파일 오류

AspectJ 컴파일 시간 위빙 옵션을 사용 중이라면 컴파일러가 워크플로 및 활동에서 생성된 클라이언트 클래스를 찾지 못하는 컴파일 시간 오류를 겪을 수 있습니다. 이러한 컴파일 오류는 컴파일 도중 AspectJ 빌더에서 생성된 클라이언트를 무시하는 경우가 원인인 경우가 많습니다. 사용자는 프로젝트에서 AspectJ 기능을 제거했다가 다시 활성화하는 방법으로 이 문제를 해결할 수 있습니다. 워크플로 또는 활동 인터페이스가 변경될 때마다 이 작업을 해주어야 한다는 점에 유의하십시오. 이러한 번거로움 때문에 이 방법 대신 로드 시간 위빙 옵션을 사용하는 것이 좋습니다. 자세한 내용은 AWS Flow Framework for Java 설정 단원을 참조하십시오.

알 수 없는 리소스 장애

Amazon SWF는 사용자가 사용할 수 없는 리소스에 대해 작업을 수행하려 할 때 알 수 없는 리소스 장애를 반환합니다. 이 장애의 흔한 원인은 다음과 같습니다.

- 사용자가 존재하지 않는 도메인을 사용해 작업자를 구성하기 때문입니다. 이를 수정하려면 먼저 Amazon SWF 콘솔 또는 Amazon SWF 서비스 API를 사용하여 도메인을 등록합니다.
- 사용자가 등록되지 않은 유형의 워크플로 실행 또는 활동 작업을 생성하려 하기 때문입니다. 이러한 장애는 작업자가 실행되기 전에 사용자가 워크플로 실행을 생성하려 하는 경우 발생할 수 있습니다.

작업자는 처음 실행될 때 자신의 유형을 등록하므로 사용자는 실행을 시작하려 하기 전에 최소 한 번 작업자를 실행(또는 콘솔 또는 서비스 API를 사용하여 유형을 수동으로 등록)해야 합니다. 유형이 등록되면 사용자는 실행 중인 작업자가 없더라도 실행을 생성할 수 있습니다.

- 작업자가 이미 제한 시간을 초과한 작업을 완료하려 하기 때문입니다. 예를 들어 작업자가 작업을 처리하는 데 너무 오래 걸리고 제한 시간을 초과하면 작업을 완료하거나 실패하려고 할 때 UnknownResource 오류가 발생합니다. AWS Flow Framework 작업자는 계속해서 Amazon SWF를 조사하고 추가 작업을 처리할 것입니다. 그러나 제한 시간 조정을 고려해야 합니다. 제한 시간을 조정하려면 새 버전의 활동 유형을 등록해야 합니다.

약속에서 get()을 호출할 때 발생하는 예외

Java Future와 달리 `Promise`는 비차단형 구성으로서, 아직 준비 상태가 아닌 `Promise`에서 `get()`을 호출하면 차단하는 대신 예외를 발생시킵니다. 올바른 `Promise` 사용 방법은 이를 비동기식 메서드(또는 작업)에 전달하고 비동기식 메서드에 있는 값에 액세스하는 것입니다. AWS Flow Framework for Java에서는 비동기식 메서드가 이에 전달된 모든 `Promise` 인수가 준비 상태가 된 경우에만 직접적으로 호출되도록 합니다. 코드가 올바르다고 생각되거나 AWS Flow Framework 샘플 중 하나를 실행하는 동안 이 문제가 발생하는 경우 AspectJ가 제대로 구성되지 않았기 때문일 가능성이 큽니다. 자세한 내용은 AWS Flow Framework for Java 설정 단원을 참조하십시오.

불확정적 워크플로

불확정성 단원에서 설명한 것처럼 워크플로의 구현은 확정적이어야 합니다. 불확정성으로 이어질 수 있는 흔한 실수로는 시스템 클록 사용, 임의 숫자 사용 및 GUID 생성이 있습니다. 이러한 구성에서는 다른 시각에 다른 값을 반환할 수 있으므로 워크플로의 제어 흐름은 실행될 때마다 다른 경로를 취할 수 있습니다(자세한 내용은 AWS Flow Framework 기본 개념: 분산 실행 및 Under the Hood 단원 참조). 프레임워크에서 워크플로 실행 중에 불확정성을 감지하면 예외가 발생합니다.

버전 관리로 인한 문제

새 버전의 워크플로 또는 활동을 구현하는 경우(예: 새 기능을 추가할 때) `@Workflow`, `@Activites` 또는 `@Activity`과 같은 적절한 주석을 사용하여 유형의 버전을 증가 시켜야 합니다. 새 버전의 워크플로가 배포되면 이미 실행 중인 기존 버전의 실행을 보유하게 되는 경우가 많습니다. 따라서 적절한 버전의 워크플로 및 활동을 지닌 작업자가 작업을 받도록 해야 합니다. 이렇게 하려면 각 버전에 다른 작업 목록 세트를 사용하면 됩니다. 예를 들어 작업 목록의 이름에 버전 번호를 추가할 수 있습니다. 이렇게 하면 서로 다른 버전의 워크플로 및 활동에 속하는 작업이 적절한 작업자에게 할당됩니다.

워크플로 실행 관련 문제 해결 및 디버깅

워크플로 실행과 관련된 문제를 해결하는 첫 번째 단계는 Amazon SWF 콘솔을 사용하여 워크플로 내역을 검색하는 것입니다. 워크플로 내역은 워크플로 실행의 실행 상태를 변경한 모든 이벤트에 대한 완전하고 신뢰할 수 있는 기록입니다. 이 내역은 Amazon SWF에서 유지 관리하며 문제를 진단하는 데 중요한 역할을 합니다. Amazon SWF 콘솔을 통해 사용자는 워크플로 실행을 검색하고 개별 내역 이벤트로 드릴다운할 수 있습니다.

AWS Flow Framework 워크플로 실행을 로컬에서 재생하고 디버깅하는 데 사용할 수 있는 `WorkflowReplayer` 클래스를 제공합니다. 사용자는 이 클래스를 사용하여 종료되었거나 실행 중인 워크플로 실행을 디버깅할 수 있습니다. `WorkflowReplayer`에서는 Amazon SWF에 저장된 내역에 의존하여 다시 재생을 수행합니다. 사용자는 자신의 Amazon SWF 계정에서 워크플로 실행을 가리키도록 하거나 내역 이벤트를 제공합니다(예: Amazon SWF에서 내역을 가져와 이를 나중에 사용할 수 있도록 로컬에서 직렬화할 수 있습니다). `WorkflowReplayer`를 사용하여 워크플로 실행을 다시 재생하면 사용자의 계정에서 실행 중인 워크플로 실행에 영향을 미치지 않습니다. 다시 재생은 클라이언트 상에서 완료됩니다. 사용자는 평상시와 같이 디버깅 도구를 사용하여 워크플로를 디버깅하고 중단점을 생성하며 코드를 한 단계씩 실행할 수 있습니다. Eclipse를 사용하는 경우 필터 패키지에 단계 필터를 추가하는 것을 고려해 보십시오. AWS Flow Framework

예를 들어 다음 코드 조각은 워크플로 실행을 다시 재생하는 데 사용할 수 있습니다.

```
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 또한 워크플로 실행의 비동기 스레드 덤프를 가져올 수 있습니다. 이 스레드 덤프에서는 사용자에게 모든 개방 비동기 작업의 호출 스택을 부여합니다. 이 정보는 실행에서 어떤 작업이 보류 중이고 멈춰 있을 수 있는지 판별하는 데 유용할 수 있습니다. 예:

```
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);
}
```

손실된 작업

때로 작업자를 종료하고 이어서 재빨리 새 작업자를 시작했는데 해당 작업이 이전 작업자에게 전달되어버리는 경우가 있을 수 있습니다. 이러한 장애는 몇 가지 프로세스에 분산되어 있는 시스템의 경합 조건으로 인해 발생할 수 있습니다. 이 문제는 긴밀한 루프에서 단위 테스트를 실행하는 경우 생길 수도 있습니다. Eclipse에서 테스트를 중단하는 것 역시 때로 이 문제의 원인일 수 있는데, 그 이유는 종료 핸들러가 호출되지 않을 수 있기 때문입니다.

문제가 사실은 작업을 받는 이전 작업자로 인한 것인지 확인하려면 워크플로 내역을 검토하여 새 작업자가 수신하기를 기대한 작업을 어떤 프로세스에서 수신했는지 판별해야 합니다. 예를 들어 내역의 `DecisionTaskStarted` 이벤트에는 작업을 수신한 워크플로 작업자의 ID가 들어 있습니다. Flow Framework에서 사용하는 ID의 형식은 {*processId*}@{*### ##*}입니다. 예를 들어 다음은 샘플 실행에 대한 Amazon SWF 콘솔 내 `DecisionTaskStarted` 이벤트에 관한 세부 정보입니다.

이벤트 타임스탬프	Mon Feb 20 11:52:40 GMT-800 2012
자격 증명	2276@ip-0A6C1DF5
예약된 이벤트 Id	33

이러한 상황을 방지하려면 각 테스트에 대해 서로 다른 작업 목록을 사용하십시오. 또한 이전 작업자 종료와 새 작업자 시작 사이에 지연을 추가하는 것을 고려하십시오.

AWS Flow Framework for Java 참조

주제

- AWS Flow Framework for Java 주석
- AWS Flow Framework for Java 예외
- AWS Flow Framework for Java 패키지

AWS Flow Framework for Java 주석

주제

- @활동
- @활동
- @ActivityRegistrationOptions
- @비동기식
- @실행
- @ExponentialRetry
- @GetState
- @ManualActivityCompletion
- @신호
- @SkipRegistration
- @Wait 및 @NoWait
- @워크플로
- @WorkflowRegistrationOptions

@활동

인터페이스에서 이 주석을 사용하여 일련의 활동 유형을 선언할 수 있습니다. 이 주석이 달린 인터페이스의 각 메서드는 활동 유형을 나타냅니다. 한 인터페이스에 `@Workflow` 및 `@Activities` 주석이 둘 다 있을 수는 없습니다.

이 주석에는 다음 파라미터를 지정할 수 있습니다.

`activityNamePrefix`

인터페이스에서 선언된 활동 유형의 이름에 붙일 접두사를 지정합니다. 기본값인 빈 문자열로 설정된 경우 '.' 앞에 있는 인터페이스 이름이 접두사로 사용됩니다.

`version`

인터페이스에서 선언된 활동 유형의 기본 버전을 지정합니다. 기본값은 `1.0`입니다.

`dataConverter`

이 활동 유형의 작업 및 그 결과 생성 시 데이터 직렬화/역직렬화에 사용할 `DataConverter`의 유형을 지정합니다. `NullDataConverter`를 기본값으로 설정합니다. 이 값은 `JsonDataConverter`를 사용해야 함을 뜻합니다.

@활동

이 주석은 `@Activities`가 주석으로 달린 인터페이스 내의 메서드에서 사용할 수 있습니다.

이 주석에는 다음 파라미터를 지정할 수 있습니다.

`name`

활동 유형의 이름을 지정합니다. 기본값은 빈 문자열입니다. 이는 활동 유형의 이름을 정할 때 기본 접두사 및 활동 메서드 이름을 사용해야 함을 뜻합니다({접두사}{이름} 형식). `@Activity` 주석에서 이름을 지정하는 경우 프레임워크에서는 그 이름에 접두사를 자동으로 붙이지 않는다는 점에 유의하십시오. 고유한 이름 지정 체계를 자유롭게 사용할 수 있습니다.

`version`

활동 유형의 버전을 지정합니다. 이렇게 하면 포함된 인터페이스의 `@Activities` 주석에 지정된 기본 버전이 재정의됩니다. 기본값은 빈 문자열입니다.

@ActivityRegistrationOptions

활동 유형의 등록 옵션을 지정합니다. 이 주석은 `@Activities`가 주석으로 달린 인터페이스 또는 이 인터페이스 내의 메서드에서 사용할 수 있습니다. 두 군데에서 모두 지정하면 메서드에서 사용되는 주석이 적용됩니다.

이 주석에는 다음 파라미터를 지정할 수 있습니다.

`defaultTasklist`

이 활동 유형에 대해 Amazon SWF에 등록할 기본 작업 목록을 지정합니다. 이 기본값은 `ActivitySchedulingOptions` 파라미터를 사용하는 생성된 클라이언트의 활동 메서드를 호출할 때 재정의할 수 있습니다. `USE_WORKER_TASK_LIST`를 기본값으로 설정합니다. 이 값은 등록을 수행 중인 작업자가 사용하는 작업 목록을 사용해야 함을 나타내는 특수 값입니다.

`defaultTaskScheduleToStartTimeoutSeconds`

이 활동 유형에 대해 Amazon SWF에 등록할 defaultTaskScheduleToStartTimeout을 지정합니다. 이것은 이 활동 유형의 작업이 작업자에게 할당될 때까지 대기하도록 허용되는 최대 시간입니다. 자세한 내용은 Amazon Simple Workflow Service API 참조를 참조하십시오.

`defaultTaskHeartbeatTimeoutSeconds`

이 활동 유형에 대해 Amazon SWF에 등록될 `defaultTaskHeartbeatTimeout`을 지정합니다. 활동 작업자는 이 시간 내에 하트비트를 제공해야 합니다. 그러지 않으면 작업이 제한 시간을 초과하게 됩니다. 이 제한 시간을 비활성화해야 함을 암시하는 특수 값인 -1이 기본값으로 설정됩니다. 자세한 내용은 Amazon Simple Workflow Service API 참조를 참조하십시오.

`defaultTaskStartToCloseTimeoutSeconds`

이 활동 유형에 대해 Amazon SWF에 등록할 defaultTaskStartToCloseTimeout을 지정합니다. 이 제한 시간을 통해 작업자가 이 유형의 활동 작업을 처리하는 데 허용되는 최대 시간이 결정됩니다. 자세한 내용은 Amazon Simple Workflow Service API 참조를 참조하십시오.

`defaultTaskScheduleToCloseTimeoutSeconds`

이 활동 유형에 대해 Amazon SWF에 등록될 `defaultScheduleToCloseTimeout`을 지정합니다. 이 제한 시간을 통해 작업이 개방 상태에 머물 수 있는 총 시간이 결정됩니다. 이 제한 시간을 비활성화해야 함을 암시하는 특수 값인 -1이 기본값으로 설정됩니다. 자세한 내용은 Amazon Simple Workflow Service API 참조를 참조하십시오.

@비동기식

워크플로 조정 로직에 있는 메서드에서 사용되는 경우 메서드를 비동기식으로 실행해야 함을 나타냅니다. 메서드를 호출하면 즉시 결과를 반환하지만, 실제 실행은 메서드로 전달된 모든 Promise<> 파라미터가 준비 상태가 될 때 비동기식으로 이루어집니다. @Asynchronous라는 주석이 달린 메서드에는 Promise<> 또는 void 반환 유형이 있어야 합니다.

daemon

비동기식 메서드에 대해 생성된 작업이 대몬(daemon) 작업인지 여부를 나타냅니다. 기본적으로 `False`로 설정됩니다.

@실행

`@Workflow` 주석이 붙은 인터페이스의 메서드에서 사용되는 경우 워크플로의 진입점을 식별합니다.

Important

인터페이스에 있는 메서드 하나만 `@Execute`로 장식할 수 있습니다.

이 주석에는 다음 파라미터를 지정할 수 있습니다.

`name`

워크플로 유형의 이름을 지정합니다. 설정되지 않은 경우 이름은 {접두사}{이름}으로 기본 설정됩니다. 여기에서 {접두사}는 '.' 앞에 있는 워크플로 인터페이스의 이름이고 {이름}은 워크플로에 있는 `@Execute` 장식 메서드의 이름입니다.

`version`

워크플로 유형의 버전을 지정합니다.

@ExponentialRetry

활동 또는 비동기식 메서드에서 사용되는 경우 메서드에서 미처리 예외가 발생하면 기하급수적 재시도 정책을 설정합니다. 재시도는 시도 회수의 제곱으로 계산되는 백오프 기간 이후에 이루어집니다.

이 주석에는 다음 파라미터를 지정할 수 있습니다.

`intialRetryIntervalSeconds`

첫 번째 재시도가 이루어질 때까지 대기할 시간을 지정합니다. 이 값은 `maximumRetryIntervalSeconds` 및 `retryExpirationSeconds`보다 커서는 안 됩니다.

`maximumRetryIntervalSeconds`

재시도 사이의 최대 시간 간격을 지정합니다. 최대 시간에 이르면 재시도 간격은 이 값으로 제한됩니다. 무제한 시간을 뜻하는 -1이 기본값으로 설정됩니다.

`retryExpirationSeconds`

기하급수적 재시도가 중단되는 시간을 지정합니다. 만료가 없음을 뜻하는 -1이 기본값으로 설정됩니다.

`backoffCoefficient`

재시도 간격을 계산하는 데 사용되는 계수를 지정합니다. 기하급수적 재시도 전략 섹션을 참조하세요.

`maximumAttempts`

기하급수적 재시도가 중단되는 시도 회수를 지정합니다. 재시도 횟수에 제한이 없음을 뜻하는 -1이 기본값으로 설정됩니다.

`exceptionsToRetry`

재시도를 트리거해야 하는 예외 유형의 목록을 지정합니다. 이 유형의 미처리 예외는 더 전파되지 않고, 메서드는 계산된 재시도 간격이 지나면 재시도됩니다. 기본적으로 목록에는 `Throwable`이 포함됩니다.

`excludeExceptions`

재시도를 트리거해서는 안 되는 예외 유형의 목록을 지정합니다. 이 유형의 미처리 예외가 전파되도록 허용됩니다. 목록은 기본적으로 비어 있습니다.

@GetState

`@Workflow` 주석이 붙은 인터페이스의 메서드에서 사용되는 경우 이 메서드가 최신 워크플로 실행 상태를 조회하는 데 사용되는지 식별합니다. `@Workflow`라는 주석이 달린 인터페이스에는 이 주석이 달린 메서드가 최대 한 개 있을 수 있습니다. 이 주석이 달린 메서드에서는 파라미터를 받아들여서는 안 되고 `void`가 아닌 반환 유형이 있어야 합니다.

@ManualActivityCompletion

활동 메서드에서 이 주석을 사용하여, 메서드에서 결과가 반환될 때 활동 작업이 완료되어서는 안 됨을 나타낼 수 있습니다. 활동 작업은 자동으로 완료되지 않으므로 Amazon SWF API를 사용하여 직접 수동으로 완료해야 합니다. 이 기능은 활동 작업이 자동화되지 않은 일부 외부 시스템에 위임되거나 완료를 위해 인간의 개입이 필요한 경우의 사용 사례로 유용합니다.

@신호

`@Workflow` 주석이 붙은 인터페이스의 메서드에서 사용되는 경우 인터페이스에서 선언한 워크플로 유형의 실행에서 수신할 수 있는 신호를 식별합니다. 이 주석을 사용하려면 신호 메서드를 정의해야 합니다.

이 주석에는 다음 파라미터를 지정할 수 있습니다.

`name`

신호 이름의 이름 부분을 지정합니다. 지정하지 않을 경우 메서드의 이름이 사용됩니다.

@SkipRegistration

`@Workflow` 주석이 붙은 인터페이스에서 사용되는 경우 워크플로 유형이 Amazon SWF 에 등록되어서는 안 됨을 나타냅니다. `@Workflow`라는 주석이 달린 인터페이스에서는 `@WorkflowRegistrationOptions` 및 `@SkipRegistrationOptions` 주석 둘 다가 아닌 둘 중 하나를 사용해야 합니다.

@Wait 및 @NoWait

이 주석을 `Promise<>` 유형의 파라미터에서 사용하여 AWS Flow Framework for Java가 이 파라미터가 준비 상태가 될 때까지 기다렸다가 메서드를 실행할지 여부를 나타낼 수 있습니다. 기본적으로 `@Asynchronous` 메서드로 전달되는 `Promise<>` 파라미터는 메서드가 실행되기 전에 준비 상태가 되어야 합니다. 어떤 상황에서는 이 기본 동작을 재정의해야 합니다. `@Asynchronous` 메서드로 전달되고 `@NoWait`라는 주석이 달린 `Promise<>` 파라미터는 대기 대상이 아닙니다.

`List<Promise<Int>>`와 같이 약속이 포함된 모음 파라미터(또는 약속의 하위 클래스)는 `@Wait`라는 주석이 달려 있어야 합니다. 기본적으로 프레임워크에서는 모음의 멤버를 기다리지 않습니다.

@워크플로

인터페이스에서 워크플로 유형을 선언할 때 이 주석이 사용됩니다. 워크플로에 대해 진입점을 선언하려면 이 주석으로 장식된 인터페이스에 @실행 주석으로 장식된 메서드가 정확히 한 개 포함되어 있어야 합니다.

Note

인터페이스에는 동시에 선언된 `@Workflow` 및 `@Activities` 주석이 둘 다 있을 수 없습니다. 이 둘은 상호 배타적입니다.

이 주석에는 다음 파라미터를 지정할 수 있습니다.

`dataConverter`

이 워크플로 유형의 워크플로 실행에 요청을 전송하고 그로부터 결과를 수신할 때 사용할 `DataConverter`를 지정합니다.

기본값은 모든 요청 및 응답 데이터를 JavaScript Object Notation(JSON)으로 처리하기 위해 결국 `JsonDataConverter`로 폴백하는 `NullDataConverter`입니다.

예

```
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

`@Workflow` 주석이 달린 인터페이스에서 사용되는 경우 워크플로 유형 등록 시 Amazon SWF에서 사용한 기본 설정을 제공합니다.

Note

`@Workflow`라는 주석이 달린 인터페이스에서는 `@WorkflowRegistrationOptions` 또는 `@SkipRegistrationOptions` 주석을 사용해야 합니다. 둘 다 지정할 수는 없습니다.

이 주석에는 다음 파라미터를 지정할 수 있습니다.

설명

워크플로 유형에 대한 텍스트 설명(선택 사항)입니다.

`defaultExecutionStartToCloseTimeoutSeconds`

워크플로 유형에 대해 Amazon SWF에 등록될 `defaultExecutionStartToCloseTimeout`을 지정합니다. 이것은 이 유형의 워크플로 실행이 완료되는 데 걸리는 최대 시간입니다.

워크플로 제한 시간에 대한 자세한 내용은 [Amazon SWF 제한 시간 유형](#) 단원을 참조하십시오.

`defaultTaskStartToCloseTimeoutSeconds`

워크플로 유형에 대해 Amazon SWF에 등록될 `defaultTaskStartToCloseTimeout`을 지정합니다. 이것은 이 유형의 워크플로 실행에 대한 결정 작업 하나가 완료되는 데 허용되는 시간입니다.

`defaultTaskStartToCloseTimeout`을 지정하지 않으면 30초로 기본 설정됩니다.

워크플로 제한 시간에 대한 자세한 내용은 [Amazon SWF 제한 시간 유형](#) 단원을 참조하십시오.

`defaultTaskList`

이 워크플로 유형의 실행에 대한 결정 작업에 사용되는 기본 작업 목록입니다. 여기에 설정된 기본값은 워크플로 실행을 시작할 때 `StartWorkflowOptions`를 사용하여 재정의할 수 있습니다.

`defaultTaskList`를 지정하지 않으면 `USE_WORKER_TASK_LIST`로 기본 설정됩니다. 이 값은 워크플로 등록을 수행 중인 작업자가 사용하는 작업 목록을 사용해야 함을 나타냅니다.

`defaultChildPolicy`

이 유형의 실행이 종료된 경우 하위 워크플로에 사용할 정책을 지정합니다. 기본값은 `ABANDON`입니다. 가능한 값은 다음과 같습니다.

- `ABANDON` – 하위 워크플로 실행이 계속 실행되도록 합니다
- `TERMINATE` – 하위 워크플로 실행을 종료합니다
- `REQUEST_CANCEL` – 하위 워크플로 실행의 취소를 요청합니다

AWS Flow Framework for Java 예외

AWS Flow Framework for Java에서는 다음과 같은 예외가 사용됩니다. 이 단원에서는 예외에 대한 개요를 제공합니다. 자세한 내용은 개별 예외에 관한 AWS SDK for Java 설명서를 참조하십시오.

주제

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

ActivityFailureException

이 예외는 프레임워크 내부에서 활동 실패를 전달하는 데 사용됩니다. 미처리 예외로 인해 활동이 실패하면 이 예외는 `ActivityFailureException`에 포함되어 Amazon SWF에 보고됩니다. 활동 작업자 확장성 포인트를 사용하는 경우에만 이 예외를 처리해야 합니다. 사용자의 애플리케이션 코드에서는 이 예외를 처리할 필요가 없습니다.

ActivityTaskException

이것은 활동 작업 실패 예외인 `ScheduleActivityTaskFailedException`, `ActivityTaskFailedException`, `ActivityTaskTimedoutException`의 기본 클래스입니다. 이 클래스에는 실패한 작업의 작업 ID와 활동 유형이 들어 있습니다. 사용자는 워크플로 구현에서 이 예외를 포착하여 일반적인 방식으로 활동 실패를 처리할 수 있습니다.

ActivityTaskFailedException

활동의 미처리 예외는`ActivityTaskFailedException`을 발생시키는 방식으로 워크플로 구현에 보고됩니다. 원본 예외는 이 예외의 원인 속성에서 가져올 수 있습니다. 또한 이 예외에서는 내역의 고유 활동 식별자와 같이 디버깅에 유용한 기타 정보를 제공합니다.

프레임워크에서는 활동 작업자에서 원본 예외를 직렬화하여 원격 예외를 제공할 수 있습니다.

ActivityTaskTimedOutException

이 예외는 Amazon SWF에서 활동이 제한 시간을 초과한 경우에 발생합니다. 이러한 예외는 활동 작업이 요구 시간 내에 작업자에게 할당될 수 없거나 요구 시간 내에 작업자가 완료할 수 없는 경우에 발생할 수 있습니다. 사용자는 이러한 제한 시간을 활동 메서드 호출 시 `@ActivityRegistrationOptions` 주석 또는 `ActivitySchedulingOptions` 파라미터를 사용하여 활동에 설정할 수 있습니다.

ChildWorkflowException

하위 워크플로 실행의 실행을 보고하는 데 사용되는 예외의 기본 클래스. 이 예외에는 하위 워크플로 실행의 ID뿐 아니라 이 워크플로의 유형도 들어 있습니다. 사용자는 이 예외를 포착하여 하위 워크플로 실행 실패를 일반적인 방식으로 처리할 수 있습니다.

ChildWorkflowFailedException

하위 워크플로의 미처리 예외는`ChildWorkflowFailedException`을 발생시키는 방식으로 상위 워크플로 구현에 보고됩니다. 원본 예외는 이 예외의 `cause` 속성에서 가져올 수 있습니다. 또한 이 예외에서는 하위 실행의 고유 식별자와 같이 디버깅에 유용한 기타 정보를 제공합니다.

ChildWorkflowTerminatedException

이 예외는 상위 워크플로 실행에서 발생하여 하위 워크플로 실행의 종료를 보고합니다. 하위 워크플로의 종료를 처리하고 싶다면(예: 정리 또는 보상 작업 수행) 사용자는 이 예외를 포착해야 합니다.

ChildWorkflowTimedOutException

이 예외는 상위 워크플로 실행에서 발생하여 Amazon SWF에서 하위 워크플로 실행이 시간 초과되어 종료되었음을 보고합니다. 하위 워크플로의 강제 종료를 처리하고 싶다면(예: 정리 또는 보상 작업 수행) 사용자는 이 예외를 포착해야 합니다.

DataConverterException

프레임워크에서는 `DataConverter` 구성 요소를 사용하여 네트워크를 통해 전송된 데이터를 마샬링 및 마샬링 취소합니다. 이 예외는 `DataConverter`에서 데이터 마샬링 또는 마샬링 취소에 실패한 경우에 발생합니다. 이 예외는 다양한 이유로 발생할 수 있는데, 데이터 마샬링 및 마샬링 취소에 사용 중인 `DataConverter` 구성 요소의 불일치를 예로 들 수 있습니다.

DecisionException

이것은 Amazon SWF에서 결정을 적용하는 데 실패했음을 나타내는 예외의 기본 클래스입니다. 사용자는 이 예외를 포착하여 그러한 예외를 일반적인 방식으로 처리할 수 있습니다.

ScheduleActivityTaskFailedException

이 예외는 Amazon SWF에서 활동 작업 예약에 실패한 경우에 발생합니다. 이는 활동이 더 이상 사용되지 않거나 계정의 Amazon SWF 한도에 도달한 경우 등 다양한 이유로 인해 발생할 수 있습니다. 예외의 `failureCause` 속성에서는 활동 예약에 실패한 정확한 원인을 지정합니다.

SignalExternalWorkflowException

이 예외는 다른 워크플로 실행을 알리라는 해당 워크플로 실행의 요청을 Amazon SWF에서 처리하는 데 실패한 경우 발생합니다. 이는 대상 워크플로 실행을 찾을 수 없는 경우, 즉 지정한 워크플로 실행이 존재하지 않거나 닫힌 상태인 경우 발생합니다.

StartChildWorkflowFailedException

이 예외는 Amazon SWF에서 하위 워크플로 실행을 시작하는 데 실패한 경우에 발생합니다. 이는 지정된 하위 워크플로 유형이 더 이상 사용되지 않거나 계정의 Amazon SWF 한도에 도달한 경우 등 다양한 이유로 인해 발생할 수 있습니다. 예외의 `failureCause` 속성에서는 하위 워크플로 실행을 시작하는 데 실패한 정확한 원인을 지정합니다.

StartTimerFailedException

이 예외는 Amazon SWF에서 워크플로 실행이 요청한 타이머를 시작하는 데 실패한 경우에 발생합니다. 이 예외는 지정된 타이머 ID가 이미 사용 중이거나 사용자 계정에 설정된 Amazon SWF 한도에 도달한 경우에 발생할 수 있습니다. 예외의 `failureCause` 속성에서는 실패의 정확한 원인을 지정합니다.

TimerException

이것은 타이머와 관련된 예외의 기본 클래스입니다.

WorkflowException

이 예외는 프레임워크 내부에서 워크플로 실행의 실패를 보고하는 데 사용됩니다. 워크플로 작업자 확장성 포인트를 사용하는 경우에만 이 예외를 처리해야 합니다.

AWS Flow Framework for Java 패키지

이 단원에서는 AWS Flow Framework for Java에 포함된 패키지에 대한 개요를 제공합니다. 각 패키지에 대한 자세한 내용은 AWS SDK for Java API 참조의 com.amazonaws.services.simpleworkflow.flow를 참조하십시오.

com.amazonaws.services.simpleworkflow.flow

Amazon SWF와 통합되는 구성 요소가 포함되어 있습니다.

com.amazonaws.services.simpleworkflow.flow.annotations

AWS Flow Framework for Java 프로그래밍 모델에서 사용하는 주석이 포함되어 있습니다.

com.amazonaws.services.simpleworkflow.flow.aspectj

@비동기식 및 @ExponentialRetry와 같은 기능에 필요한 AWS Flow Framework for Java 구성 요소가 포함되어 있습니다.

com.amazonaws.services.simpleworkflow.flow.common

프레임워크 정의 상수와 같은 범용 유틸리티가 포함되어 있습니다.

com.amazonaws.services.simpleworkflow.flow.core

`Task` 및 `Promise`와 같은 핵심 기능이 포함되어 있습니다.

com.amazonaws.services.simpleworkflow.flow.generic

일반 클라이언트와 같은 핵심 구성 요소가 포함되어 있습니다. 이 구성 요소를 기반으로 다른 기능이 구축됩니다.

com.amazonaws.services.simpleworkflow.flow.interceptors

`RetryDecorator`를 비롯한 프레임워크 제공 장식자의 구현이 포함되어 있습니다.

com.amazonaws.services.simpleworkflow.flow.junit

Junit 통합을 제공하는 구성 요소가 포함되어 있습니다.

com.amazonaws.services.simpleworkflow.flow.pojo

주석 기반 프로그래밍 모델에 대해 활동 및 워크플로 정의를 구현하는 클래스가 포함되어 있습니다.

com.amazonaws.services.simpleworkflow.flow.spring

Spring 통합을 제공하는 구성 요소가 포함되어 있습니다.

com.amazonaws.services.simpleworkflow.flow.test

단위 테스팅 워크플로 구현에 대해 `TestWorkflowClock`와 같은 헬퍼 클래스가 포함되어 있습니다.

com.amazonaws.services.simpleworkflow.flow.worker

활동 및 워크플로 작업자의 구현이 포함되어 있습니다.

문서 기록

다음 표에서는 AWS Flow Framework for Java 개발자 가이드 최근 릴리스가 나온 이후에 이 문서에서 변경된 중요 사항에 대해 설명합니다.

- API 버전: 2012-01-25
- 설명서 최종 업데이트: 2018년 6월 25일

변경 사항	설명	변경 날짜
업데이트	`backoffCoefficient` 의 `@ExponentialRetry` 에 대한 설명에 있는 오류를 수정했습니다. @ExponentialRetry 섹션을 참조하세요.	2018년 6월 25일
업데이트	이 가이드 전체의 코드 샘플을 정리했습니다.	2017년 6월 5일
업데이트	이 가이드의 구성 및 내용을 간소화하고 개선했습니다.	2017년 5월 19일
업데이트	결정자 코드 변경: 버전 관리 및 기능 플래그 단원을 간소화하고 개선했습니다.	2017년 10월 4일
업데이트	결정자 코드 변경 사항에 관한 새 지침이 담긴 모범 사례 단원을 새로 추가했습니다.	2017년 3월 3일
새로운 기능	워크플로 내 기존 활동 작업 이외에 Lambda 작업을 지정할 수 있습니다. 자세한 내용은 AWS Lambda 작업 구현 섹션을 참조하세요.	2015년 7월 21일
새로운 기능	Amazon SWF는 우선 순위가 높은 작업을 우선 순위가 낮은 작업보다 먼저 실행하기 위해 작업 목록에서 작업 우선 순위를 설정할 수 있는 기능을 지원합니다. 자세한 내용은 작업 우선 순위 설정 섹션을 참조하세요.	2014년 12월 17일
업데이트	내용을 업데이트 및 수정하였습니다.	2013년 8월 1일

변경 사항	설명	변경 날짜
업데이트	• Eclipse 4.3 및 AWS SDK for Java 1.4.7의 설정 지침 업데이트를 포함하여 업데이트 및 수정이 완료되었습니다. • 시작자 시나리오 구축을 위한 일련의 자습서를 추가하였습니다.	2013년 6월 28일
새로운 기능	AWS Flow Framework for Java 최초 릴리스.	2012년 2월 27일

AWS 용어집

최신 AWS 용어는 AWS 용어집 참조서의 AWS 용어집을 참조하세요.

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.