開發人員指南

AWS Flow Framework 適用於 Java 的

API 版本 2021-04-28

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Flow Framework 適用於 Java 的: 開發人員指南

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商標和商業外觀不得用於任何非 Amazon 的產品或服務，也不能以任何可能造成客戶混淆、任何貶低或使 Amazon 名譽受損的方式使用 Amazon 的商標和商業外觀。所有其他非 Amazon 擁有的商標均為其各自擁有者的財產，這些擁有者可能隸屬於 Amazon，或與 Amazon 有合作關係，或由 Amazon 贊助。

Table of Contents

什麼是適用於 Java AWS Flow Framework 的 ? ... 1
本指南內容 ... 1
開始使用 ... 2
設定框架 ... 2
使用 Maven 新增流程架構 ... 2
HelloWorld 應用程式 ... 3
HelloWorld 活動實作 ... 4
HelloWorld 工作流程工作者 ... 5
HelloWorld 工作流程啟動者 ... 5
HelloWorldWorkflow 應用程式 ... 6
HelloWorldWorkflow 活動工作者 ... 8
HelloWorldWorkflow 工作流程工作者 ... 9
HelloWorldWorkflow 工作流程和活動實作 ... 13
HelloWorldWorkflow 啟動者 ... 17
HelloWorldWorkflowAsync 應用程式 ... 22
HelloWorldWorkflowAsync 活動實作 ... 23
HelloWorldWorkflowAsync 工作流程實作 ... 24
HelloWorldWorkflowAsync 工作流程和活動主機與啟動者 ... 26
HelloWorldWorkflowDistributed 應用程式 ... 27
HelloWorldWorkflowParallel 應用程式 ... 29
HelloWorldWorkflowParallel 活動工作者 ... 30
HelloWorldWorkflowParallel 工作流程工作者 ... 31
HelloWorldWorkflowParallel 工作流程和活動主機與啟動者 ... 32
了解 AWS Flow Framework ... 33
應用程式結構 ... 33
活動工作者的角色 ... 35
工作流程工作者的角色 ... 35
工作流程啟動者的角色 ... 35
Amazon SWF 如何與您的應用程式互動 ... 35
如需詳細資訊 ... 36
可靠的執行 ... 36
提供可靠的通訊 ... 36
確保結果不遺失 ... 37
處理失敗的分散式元件 ... 37

分散式執行 ... 38
重新執行工作流程 ... 38
重新執行和非同步的工作流程方法 ... 39
重新執行與工作流程實作 ... 39
任務清單和任務執行 ... 39
可擴展應用程式 ... 41
活動與工作流程之間的資料交換 ... 42
Promise<T> 類型 ... 42
資料轉換器和封送處理 ... 43
應用程式與工作流程執行之間的資料交換 ... 44
逾時類型 ... 44
工作流程和決策任務的逾時 ... 45
活動任務的逾時 ... 46
了解任務 ... 48
任務 ... 48
執行順序 ... 49
工作流程執行 ... 50
不確定性 ... 52
程式設計指南 ... 53
實作工作流程應用程式 ... 53
工作流程和活動合約 ... 55
工作流程和活動類型註冊 ... 57
工作流程類型名稱和版本 ... 57
訊號名稱 ... 58
活動類型名稱和版本 ... 58
預設任務清單 ... 58
其他註冊選項 ... 58
活動和工作流程用戶端 ... 59
工作流程用戶端 ... 59
活動用戶端 ... 67
排程選項 ... 70
動態用戶端 ... 71
工作流程實作 ... 73
決策內容 ... 74
公開執行狀態 ... 74
工作流程區域變數 ... 76

活動實作 ... 77
手動完成活動 ... 78
實作 Lambda 任務 ... 79
關於 AWS Lambda ... 80
使用 Lambda 任務的優點和限制 ... 80
在適用於 Java AWS Flow Framework 的 工作流程中使用 Lambda 任務 ... 80
檢視 HelloLambda 範例 ... 85
執行使用 AWS Flow Framework 適用於 Java 的 編寫的程式 ... 85
WorkflowWorker ... 87
ActivityWorker ... 87
工作者執行緒模型 ... 87
工作者可擴充性 ... 89
執行內容 ... 90
決策內容 ... 90
活動執行內容 ... 92
子工作流程執行 ... 93
持續的工作流程 ... 95
設定任務優先順序 ... 96
為工作流程設定任務優先順序 ... 97
為活動設定任務優先順序 ... 97
DataConverters ... 98
將資料傳遞給非同步方法 ... 99
將集合和對應傳遞給非同步方法 ... 99
Settable<T> ... 100
@NoWait ... 101
Promise<Void> ... 101
AndPromise 和 OrPromise ... 101
可試性與相依性插入 ... 102
Spring 整合 ... 102
JUnit 整合 ... 108
錯誤處理 ... 114
TryCatchFinally 語意 ... 116
取消 ... 116
巢狀 TryCatchFinally ... 120
重試失敗的活動 ... 121
重試到成功為止策略 ... 122

指數重試策略 124
自訂重試策略 130
協助程式任務 133
重新執行行為 135
範例 1：同步重新執行 135
範例 2：非同步重新執行 137
另請參閱 138
最佳實務 139
變更決策者程式碼 139
重播程序和程式碼變更 139
範例藍本 139
解決方案 146
故障診斷 152
編譯錯誤 152
未知的資源錯誤 152
在 Promise 上呼叫 get() 時的例外狀況 153
非確定性工作流程 153
版本控制引起的問題 153
對工作流程執行進行故障診斷和偵錯 153
遺失的任務 155
由於 API 參數長度限制導致的驗證失敗 155
參考資料 157
註釋 157
@Activities 157
@Activity 158
@ActivityRegistrationOptions 158
@異步 159
@Execute 159
@ExponentialRetry 160
@GetState 161
@ManualActivityCompletion 161
@Signal 161
@SkipRegistration 161
@Wait 和 @NoWait 161
@工作流程 162
@WorkflowRegistrationOptions 163

例外狀況 164
ActivityFailureException 164
ActivityTaskException 165
ActivityTaskFailedException 165
ActivityTaskTimedOutException 165
ChildWorkflowException 165
ChildWorkflowFailedException 165
ChildWorkflowTerminatedException 165
ChildWorkflowTimedOutException 166
DataConverterException 166
DecisionException 166
ScheduleActivityTaskFailedException 166
SignalExternalWorkflowException 166
StartChildWorkflowFailedException 166
StartTimerFailedException 166
TimerException 167
WorkflowException 167
套件 167
文件歷史記錄 169
..... clxxi

什麼是適用於 Java AWS Flow Framework 的 ？

使用 AWS Flow Framework，您可以專注於實作工作流程邏輯。在幕後，架構會使用 Amazon SWF 的排程、路由和狀態管理功能來管理工作流程的執行，並使其可擴展、可靠和可稽核。 AWS Flow Framework型工作流程高度並行。工作流程可以分散到多個元件，這些元件可以在不同的電腦上以不同的程序執行，並獨立擴展。如果其任何元件正在執行，應用程式可以繼續執行，使其具有高度的容錯能力。

本指南內容

本指南提供如何安裝、設定和使用 AWS Flow Framework 來建置 Amazon SWF 應用程式的相關資訊。

AWS Flow Framework 適用於 Java 的 入門

如果您剛開始使用 AWS Flow Framework 適用於 Java 的 ，請閱讀 AWS Flow Framework 適用於 Java 的 入門一節。它將引導您下載並安裝 AWS Flow Framework 適用於 Java 的 、如何設定開發環境，並引導您完成建立工作流程的簡單範例。

了解適用於 Java AWS Flow Framework 的

介紹基本 Amazon SWF 和 AWS Flow Framework 概念，說明應用程式的基本結構 AWS Flow Framework ，以及如何在分散式工作流程的各部分之間交換資料。

AWS Flow Framework for Java 程式設計指南

本章提供使用適用於 Java AWS Flow Framework 的 開發工作流程應用程式的基本程式設計指導，包括如何註冊活動和工作流程類型、實作工作流程用戶端、建立子工作流程、處理錯誤等。

了解 AWS Flow Framework 適用於 Java 的 中的任務

本章提供適用於 Java AWS Flow Framework 的 運作方式的更深入介紹，為您提供有關非同步工作流程執行順序和標準工作流程執行邏輯逐步解說的其他資訊。

適用於 Java AWS Flow Framework 的 疑難排解和偵錯秘訣

本章提供常見錯誤的相關資訊，能讓您對工作流程進行故障診斷，或者可從中學習如何「避免常見錯誤」。

AWS Flow Framework for Java 參考

本章參考 AWS Flow Framework 適用於 Java 的 新增至適用於 Java 的 SDK 的註釋、例外狀況和套件。

AWS Flow Framework 適用於 Java 的 入門

本節透過 AWS Flow Framework 引導您完成一系列簡單的範例應用程式，介紹基本程式設計模型和 API。範例應用程式是以介紹 C 和相關程式設計語言所使用的標準 Hello World 應用程式為根據。以下是 Hello World 的一般 Java 實作：

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

以下是範例應用程式的簡短說明。它們包含完整的來源碼，所以您可以自行實作並執行應用程式。開始之前，您應該先設定開發環境並建立 AWS Flow Framework 適用於 Java 的 專案，如 中的 設定 AWS Flow Framework 適用於 Java 的。

- HelloWorld 應用程式 將 Hello World 實作為標準的 Java 應用程式，但結構類似工作流程應用程式，來介紹工作流程應用程式。
- HelloWorldWorkflow 應用程式 使用 AWS Flow Framework for Java 將 HelloWorld 轉換為 Amazon SWF 工作流程。
- HelloWorldWorkflowAsync 應用程式 修改 `HelloWorldWorkflow` 以使用 asynchronous workflow 方法。
- HelloWorldWorkflowDistributed 應用程式 修改 `HelloWorldWorkflowAsync`，以便工作流程和活動工作者可以在不同的系統中執行。
- HelloWorldWorkflowParallel 應用程式 修改 `HelloWorldWorkflow` 以平行執行兩項活動。

設定 AWS Flow Framework 適用於 Java 的

AWS Flow Framework 適用於 Java 的 隨附於 適用於 Java 的 AWS SDK。如果您尚未設定 適用於 Java 的 AWS SDK，請參閱《 適用於 Java 的 AWS SDK 開發人員指南》中的入門，以取得安裝和設定 SDK 本身的相關資訊。

使用 Maven 新增流程架構

Amazon SWF 建置工具是開放原始碼，若要檢視或下載程式碼或自行建置工具，請造訪位於 https：//https://github.com/aws/aws-swf-build-tools 的儲存庫。

Amazon 在 Maven Central Repository 中提供 Amazon SWF 建置工具。

若要設定 Maven 的流程框架，請將下列相依性新增至專案的 `pom.xml` 檔案：

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

HelloWorld 應用程式

為了介紹 Amazon SWF 應用程式結構的方式，我們將建立 Java 應用程式，其行為與工作流程相似，但會在本機以單一程序執行。不需要連線到 Amazon Web Services。

> **Note**
>
> HelloWorldWorkflow 範例以這個範例為基礎，連線至 Amazon SWF 來處理工作流程的管理。

由三個基本元件組成的工作流程應用程式：

- 「活動工作者」支援一組「活動」，它們每一個都是執行特定任務的獨立執行方法。
- 「工作流程工作者」協調活動執行並管理資料流程。其以程式設計方式具體化「工作流程拓撲」，基本上是定義各種活動執行的時機、是依序還是同時執行等等的流程圖。
- 「工作流程啟動者」會啟動稱為「執行」的工作流程執行個體，並在執行期間與之互動。

HelloWorld 實作為三種類別和兩個相關的界面，會在下列各節中說明。開始之前，您應該設定開發環境並建立新的 AWS Java 專案，如 中所述設定 AWS Flow Framework 適用於 Java 的。下列演練所使用的套件全都名為 `helloWorld.`*`XYZ`*。若要使用這些名稱，請在 aop.xml 中如下設定 `within` 屬性：

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

若要實作 HelloWorld，請在名為 的 AWS SDK 專案中建立新的 Java 套件，`helloWorld.HelloWorld`並新增下列檔案：

- 名為 GreeterActivities.java 的界面檔案
- 名為 GreeterActivitiesImpl.java 的類別檔案，其實作活動工作者。
- 名為 GreeterWorkflow.java 的界面檔案。
- 名為 GreeterWorkflowImpl.java 的類別檔案，實作工作流程工作者。
- 名為 GreeterMain.java 的類別檔案，其實作工作流程啟動者。

下列各節會詳細討論，並包含各元件的完整程式碼，您可將它們新增至適當的檔案。

HelloWorld 活動實作

HelloWorld 會將在主控台中列印 "Hello World!" 歡迎語的完整任務分割成三項任務，每一個都由「活動方法」執行。活動方法在 GreeterActivities 界面中定義，如下所示。

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

HelloWorld 會實作一項活動 GreeterActivitiesImpl，這會提供 GreeterActivities 方法，如下所示：

```
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);
   }
}
```

活動彼此互不影響，且通常為不同的工作流程所使用。例如，任何工作流程皆可使用 `say` 活動在主控台中列印字串。工作流程還可以實作多項活動，每一項都執行不同的任務集。

HelloWorld 工作流程工作者

列印「Hello World！」 到 主控台，活動任務必須以正確的順序搭配正確的資料依序執行。HelloWorld 工作流程工作者會根據簡易的「線性工作流程拓撲」協調活動執行，如下圖所示。

依序執行三項活動，資料即會依照程序由一項活動流動至下一項活動。

HelloWorld 工作流程工作者只有一個方法 (工作流程的進入點)，其在 `GreeterWorkflow` 界面中定義，如下所示：

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

`GreeterWorkflowImpl` 類別實作此界面，如下所示：

```
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` 方法實作 HelloWorld 拓撲的方法是：建立 `GreeterActivitiesImpl` 執行個體、依正確的順序呼叫各活動方法，然後將適當的資料傳遞到各方法。

HelloWorld 工作流程啟動者

「工作流程啟動者」是開始執行工作流程的應用程式，也可能會與執行中的工作流程互動。`GreeterMain` 類別會實作 HelloWorld 工作流程啟動者，如下所示：

```
public class GreeterMain {
```

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

`GreeterMain` 會建立 `GreeterWorkflowImpl` 執行個體並呼叫 `greet` 執行工作流程工作者。`GreeterMain` 以 Java 應用程式執行，您應該會看到「Hello World！」 在主控台輸出中。

HelloWorldWorkflow 應用程式

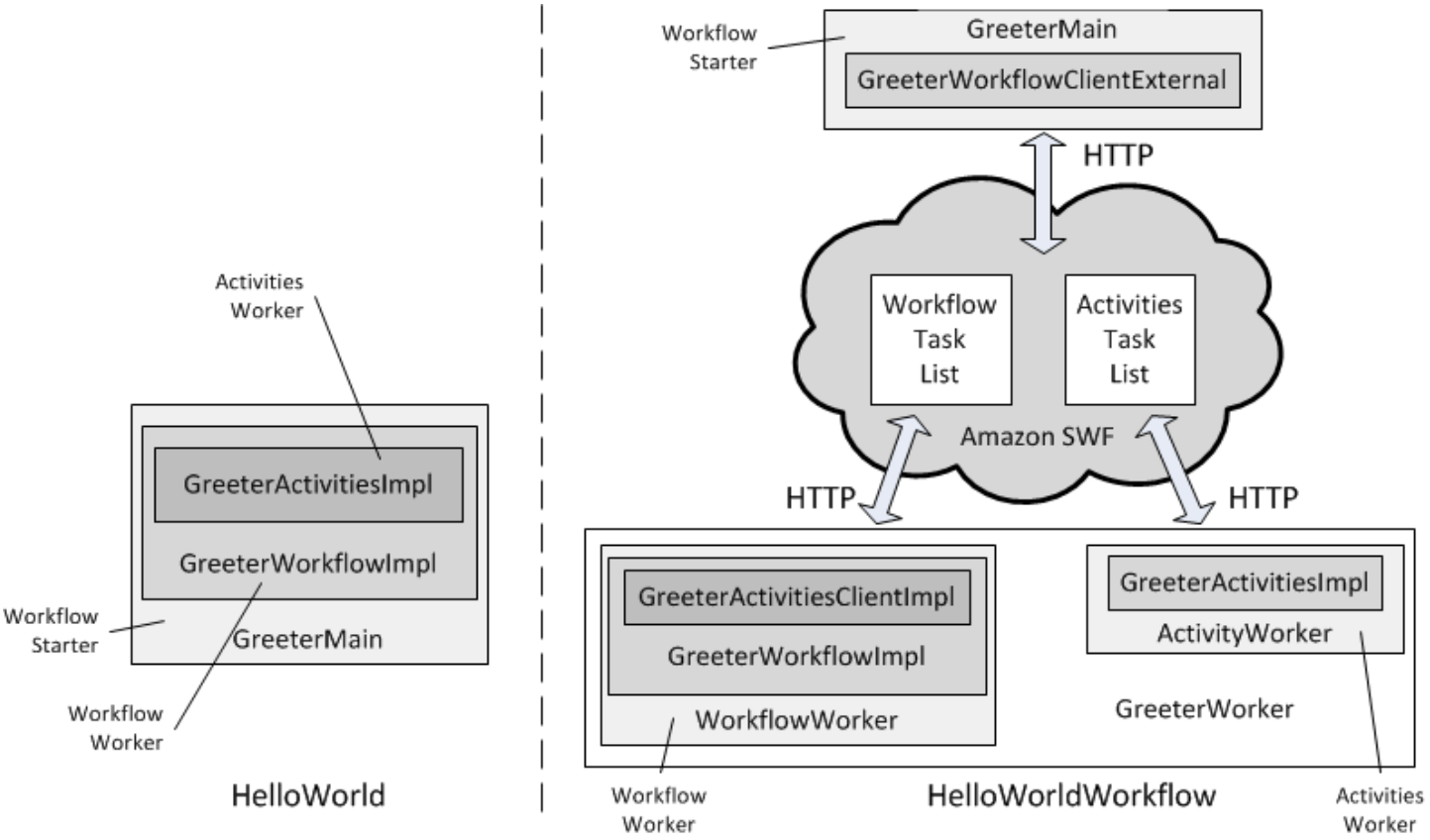
雖然基本 HelloWorld 範例的結構類似於工作流程，但它在幾個關鍵方面與 Amazon SWF 工作流程不同：

傳統和 Amazon SWF 工作流程應用程式

HelloWorld	Amazon SWF 工作流程
以單一程序在本機執行。	以多個程序的形式執行，這些程序可以分散到多個系統，包括 Amazon EC2 執行個體、私有資料中心、用戶端電腦等。甚至不必執行相同的作業系統。
活動為同步的方法，完成後才予以解鎖。	活動由非同步方法代表，其會立即傳回，並於等待活動完成期間，允許工作流程執行其他任務。
工作流程工作者透過呼叫合適的方法與活動工作者互動。	工作流程工作者使用 HTTP 請求與活動工作者互動，Amazon SWF 充當中介裝置。
工作流程啟動者透過呼叫合適的方法與工作流程工作者互動。	工作流程啟動者使用 HTTP 請求與工作流程工作者互動，Amazon SWF 做為中介裝置。

您可以從頭開始實作分散式非同步工作流程應用程式，例如讓您的工作流程工作者透過 Web 服務呼叫，直接與活動工作者互動。但是，您必須接著實作所有必要的複雜程式碼，藉以管理多項活動的非同步執行、處理資料流程等。 AWS Flow Framework 適用於 Java 和 Amazon SWF 的 會處理所有這些詳細資訊，讓您專注於實作商業邏輯。

HelloWorldWorkflow 是 HelloWorld 的修改版本，作為 Amazon SWF 工作流程執行。下圖摘要說明這兩個應用程式的運作方式。



HelloWorld 以單一程序執行，且啟動者、工作流程工作者與活動工作者使用傳統的方法呼叫互動。透過 `HelloWorldWorkflow`，入門、工作流程工作者和活動工作者是使用 HTTP 請求透過 Amazon SWF 互動的分散式元件。Amazon SWF 會透過維護工作流程和活動任務清單來管理互動，而這些任務會分派至個別元件。本節說明 HelloWorldWorkflow 的框架運作方式。

HelloWorldWorkflow 是使用 AWS Flow Framework 適用於 Java 的 API 實作的，該 API 可處理在背景中與 Amazon SWF 互動時有時複雜的詳細資訊，並大幅簡化開發程序。您可以使用與 HelloWorld 相同的專案，HelloWorld 已針對 Java 應用程式設定 AWS Flow Framework 。不過，若要執行應用程式，您必須設定 Amazon SWF 帳戶，如下所示：

- 如果您還沒有 AWS 帳戶，請在 Amazon Web Services 註冊帳戶。
- 將您帳戶的存取 ID 和密鑰 ID 分別指派給 AWS_ACCESS_KEY_ID 和 AWS_SECRET_KEY 環境變數。這是不在程式碼中公開文字金鑰值的良好做法。將之存放在環境變數中會是處理此問題的便利方法。
- 在 Amazon Simple Workflow Service 註冊 Amazon SWF 帳戶。 https://aws.amazon.com/swf/
- 登入 AWS Management Console ，然後選取 Amazon SWF 服務。

- 選擇右上角的管理網域並註冊新的 Amazon SWF 網域。「網域」是應用程式資源的邏輯容器，例如工作流程和活動類型及工作流程執行。您可使用任何便利的網域名稱，在演練時會使用 "helloWorldWalkthrough"。

若要實作 HelloWorldWorkflow，請在您的專案目錄中建立 helloWorld.HelloWorld 套件的副本，並命名為 helloWorld.HelloWorldWorkflow。下列各節說明如何修改原始 HelloWorld 程式碼，以使用 AWS Flow Framework 適用於 Java 的 ，並以 Amazon SWF 工作流程應用程式的形式執行。

HelloWorldWorkflow 活動工作者

HelloWorld 實作其活動工作者為單一類別。for AWS Flow Framework Java 活動工作者有三個基本元件：

- 執行實際任務的活動方法在 界面中定義，並在相關類別中實作。
- ActivityWorker 類別會管理活動方法和 Amazon SWF 之間的互動。
- 「活動主機」應用程式會註冊活動工作者並予以啟動，且會處理清理。

本節將討論活動方法，其他兩個類別待後文討論。

HelloWorldWorkflow 在 `GreeterActivities` 中定義活動界面，如下所示：

```
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);
}
```

HelloWorld 不需要此界面，但適用於 Java AWS Flow Framework 應用程式的 。請注意，界面定義本身並未變更。不過，您必須將兩個 AWS Flow Framework 用於 Java 註釋，@ActivityRegistrationOptions以及 @Activities套用至界面定義。註釋提供組態資訊，並指示 AWS Flow Framework for Java 註釋處理器使用界面定義來產生活動用戶端類別，稍後會進行討論。

`@ActivityRegistrationOptions` 有數個具名值，用以設定活動行為。HelloWorldWorkflow 指定兩種逾時：

- `defaultTaskScheduleToStartTimeoutSeconds` 指定任務在活動任務清單中排入佇列的時間，設為 300 秒 (5 分鐘)。
- `defaultTaskStartToCloseTimeoutSeconds` 指定活動執行任務可使用的時間上限，設為 10 秒。

這些逾時確保活動在合理的時間內完成其任務。如果超過任一種逾時，框架會產生錯誤，而工作流程工作者必須決定如何處理此問題。若需如何處理這類錯誤的討論，請參閱「錯誤處理」。

`@Activities` 有數個值，但通常僅指出活動的版本編號，能夠讓您追蹤所產生的不同活動實作。如果您在向 Amazon SWF 註冊活動界面之後變更活動界面，包括變更`@ActivityRegistrationOptions`值，則必須使用新的版本編號。

HelloWorldWorkflow 會在 `GreeterActivitiesImpl` 中實作活動方法，如下所示：

```
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);
   }
}
```

請注意，程式碼與 HelloWorld 實作相同。活動本身 AWS Flow Framework 只是執行一些程式碼，而且可能傳回結果的方法。標準應用程式與 Amazon SWF 工作流程應用程式之間的差異在於工作流程執行活動的方式、活動執行的位置，以及如何將結果傳回工作流程工作者。

HelloWorldWorkflow 工作流程工作者

Amazon SWF 工作流程工作者有三個基本元件。

- 「工作流程實作」類別，會執行工作流程相關的任務。

- 「活動用戶端」類別，基本上為活動類別的代理，而工作流程實作會用以非同步執行活動方法。
- WorkflowWorker 類別，可管理工作流程與 Amazon SWF 之間的互動。

本節討論工作流程實作和活動用戶端，`WorkflowWorker` 類別留待後文討論。

HelloWorldWorkflow 在 `GreeterWorkflow` 中定義工作流程界面，如下所示：

```
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();
}
```

HelloWorld 也不需要此界面，但對於 AWS Flow Framework 適用於 Java 的 應用程式而言至關重要。您必須將兩個 AWS Flow Framework 用於 Java 註釋，@工作流程以及 @WorkflowRegistrationOptions套用至工作流程界面定義。註釋提供組態資訊，並指示 AWS Flow Framework for Java 註釋處理器根據界面產生工作流程用戶端類別，如稍後所述。

`@Workflow` 有一個選用參數 dataConverter，通常與其預設值 NullDataConverter 搭配使用，這表示應該使用 JsonDataConverter。

`@WorkflowRegistrationOptions` 也有數個選擇性參數，可用以設定工作流程工作者。在這裡，我們設定 `defaultExecutionStartToCloseTimeoutSeconds`- 指定工作流程可以執行的時間長度 - 到 3600 秒 (1 小時）。

`GreeterWorkflow` 界面定義和 HelloWorld 有一項重大差異：@Execute 註釋。工作流程界面指定可由應用程式呼叫的方法 (例如工作流程啟動者)，並限定在少數幾個方法，各有特定的角色。框架不會指定工作流程界面方法的名稱或參數清單；您可以使用適合您工作流程的名稱和參數清單，並套用 AWS Flow Framework 適用於 Java 的 註釋來識別方法的角色。

`@Execute` 有兩個用途：

- 將 `greet` 識別為工作流程的進入點，即為工作流程啟動者呼叫以啟動工作流程的方法。一般而言，進入點可採用一或多個參數，讓啟動者初始化工作流程，但此範例不需要初始化。

- 指出工作流程的版本編號，能夠讓您追蹤所產生的不同工作流程實作。若要在向 Amazon SWF 註冊工作流程界面後變更工作流程界面，包括變更逾時值，您必須使用新的版本編號。

如需可包含在工作流程界面中之其他方法的資訊，請參閱「工作流程和活動合約」。

HelloWorldWorkflow 在 `GreeterWorkflowImpl` 中實作工作流程，如下所示：

```
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);
   }
}
```

程式碼類似 HelloWorld，但有兩項重要差異。

- `GreeterWorkflowImpl` 建立 `GreeterActivitiesClientImpl` 的執行個體 (活動用戶端)，不是 `GreeterActivitiesImpl` 的執行個體，並在用戶端物件上呼叫方法來執行活動。
- 名稱和歡迎活動傳回 `Promise<String>` 物件，不是 `String` 物件。

HelloWorld 為在本機以單一程序執行的標準 Java 應用程式，所以 `GreeterWorkflowImpl` 可以僅透過建立 `GreeterActivitiesImpl` 執行個體、依序呼叫方法，並將傳回值從一個活動傳遞到下一個活動，藉此實作工作流程拓撲。使用 Amazon SWF 工作流程時，活動的任務仍會由來自 的活動方法執行`GreeterActivitiesImpl`。不過，方法不一定會在和工作流程相同的程序中執行 (甚至可能不在同一個系統中執行)，且工作流程需非同步執行活動。這些要求會引起下列問題：

- 如何執行可能在不同程序，甚或是不同系統中執行的活動方法。
- 如何非同步執行活動方法。
- 如何管理活動的輸入和傳回值。例如，如果活動 A 的傳回值是活動 B 的輸入，您必須確保活動 B 不會在活動 A 完成前執行。

您可以使用熟悉的 Java 流程控制，結合活動用戶端與 `Promise<T>`，藉由應用程式的控制流程，實作各種工作流程拓撲。

活動用戶端

GreeterActivitiesClientImpl 基本上是 GreeterActivitiesImpl 的代理，允許工作流程實作非同步執行 GreeterActivitiesImpl 方法。

GreeterActivitiesClient 和 GreeterActivitiesClientImpl 類別會使用套用到您 GreeterActivities 類別之註釋中提供的資訊，自動為您產生。您無須自行實作。

> **Note**
>
> Eclipse 會在您儲存專案時產生這些類別。您可在您專案目錄的 .apt_generated 子目錄中檢視所產生的程式碼。
> 為了避免GreeterWorkflowImpl類別中的編譯錯誤，最佳實務是將.apt_generated目錄移至 Java Build Path 對話方塊的 Order and Export 索引標籤頂端。

工作流程工作者透過呼叫對應的用戶端方法執行活動。方法是非同步的，並會立即傳回 Promise<T> 物件，其中 T 是活動的傳回類型。傳回的 Promise<T> 物件基本上是活動方法最終傳回值的預留位置。

- 當活動用戶端方法傳回時，Promise<T> 物件一開始是「未就緒狀態」，表示物件尚未能代表有效的傳回值。
- 當對應的活動方法完成其任務並傳回之後，框架會將傳回值指派給 Promise<T> 物件，讓物件進入「就緒狀態」。

Promise<T> 類型

Promise<T> 物件的主要用途為管理非同步元件間的資料流程，並控制其執行的時機。此物件讓您的應用程式不必明確管理同步，或依賴計時器等機制確保非同步元件不提前執行。當您呼叫活動用戶端方法時，其會立即傳回，但框架會延遲執行對應的活動方法，直到任何輸入 Promise<T> 物件就緒且代表有效的資料為止。

就 GreeterWorkflowImpl 而言，這三個活動用戶端方法都會立即傳回。就 GreeterActivitiesImpl 而言，框架在 name 完成前不會呼叫 getGreeting，在 getGreeting 完成前不會呼叫 say。

使用 Promise<T> 將資料從一項活動傳遞到下一項活動，HelloWorldWorkflow 不僅能確保活動方法不嘗試使用無效的資料，還能控制何時執行活動及暗示定義工作流程拓撲。將每項活動

的 Promise<T> 傳回值傳遞到下一項活動，需要活動依序執行，定義前文討論過的線性拓撲。使用 AWS Flow Framework for Java，您不需要使用任何特殊建模程式碼來定義甚至複雜的拓撲，只需要標準 Java 流程控制和 Promise<T>。如需如何實作簡易平行拓撲的範例，請參閱「HelloWorldWorkflowParallel 活動工作者」。

> **Note**
>
> 當 say 等活動方法不傳回值時，對應的用戶端方法會傳回 Promise<Void> 物件。物件不代表資料，但其在一開始時未就緒，在活動完成時即變為就緒。因此，您可將 Promise<Void> 物件傳遞到其他活動用戶端方法，確保其在原始活動完成前延遲執行。

Promise<T> 允許工作流程實作使用活動用戶端方法及傳回值，近似於同步方法。不過，您必須謹慎存取 Promise<T> 物件的值。不像 Java Future<T> 類型，框架會處理 Promise<T> 的同步，而非應用程式同步。如果您呼叫 Promise<T>.get，而且物件未就緒，則 get 會拋出例外狀況。請注意，HelloWorldWorkflow 從不直接存取 Promise<T> 物件，而只會將物件從一項活動傳遞到下一項活動。當物件變成就緒後，框架會擷取值並將之以標準類型傳遞到活動方法。

Promise<T> 物件應僅由非同步程式碼存取，其中框架保證非同步程式碼的物件已就緒且代表有效值。HelloWorldWorkflow 只將 Promise<T> 物件傳遞給活動用戶端方法，藉以處理此問題。您可以透過將Promise<T>物件傳遞至與活動相似的非同步工作流程方法，來存取工作流程實作中的物件值。如需範例，請參閱「HelloWorldWorkflowAsync 應用程式」。

HelloWorldWorkflow 工作流程和活動實作

工作流程和活動實作有相關聯的工作者類別：ActivityWorker 和 WorkflowWorker。它們透過輪詢任務的適當 Amazon SWF 任務清單、執行每個任務的適當方法，以及管理資料流程，來處理 Amazon SWF 與活動和工作流程實作之間的通訊。如需詳細資訊，請參閱AWS Flow Framework 基本概念：應用程式結構

若要建立活動和工作流程實作與對應之工作者物件的關聯，您要實作可執行下列作業的一或多個工作者應用程式：

- 向 Amazon SWF 註冊工作流程或活動。
- 建立工作者物件並建立其與工作流程或活動工作者實作的關聯。
- 引導工作者物件開始與 Amazon SWF 通訊。

如果您想要將工作流程和活動執行為不同的程序，您必須實作不同的工作流程和活動工作者主機。如需範例，請參閱「HelloWorldWorkflowDistributed 應用程式」。為簡化起見，HelloWorldWorkflow 實作的單一工作者主機，會在同一個程序中執行活動和工作流程工作者，如下所示：

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

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

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

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

     String domain = "helloWorldWalkthrough";
     String taskListToPoll = "HelloWorldList";

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

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

`GreeterWorker` 沒有 HelloWorld 相應實物，所以您必須將名為 `GreeterWorker` 的 Java 類別新增至專案，並將範例程式碼複製到檔案。

第一步是建立和設定 AmazonSimpleWorkflowClient 物件，這會叫用基礎 Amazon SWF 服務方法。若要這麼做，`GreeterWorker` 會：

1. 建立 ClientConfiguration 物件並指定 70 秒的插槽逾時。此值指定在關閉插槽前，透過已建立的開放連線傳輸資料的等待時間。
2. 建立 BasicAWSCredentials 物件以識別 AWS 帳戶並將帳戶金鑰傳遞給建構函數。為方便起見，也為了避免在程式碼中以純文字公開它們，金鑰會以環境變數存放。
3. 建立 AmazonSimpleWorkflowClient 物件代表工作流程，將 `BasicAWSCredentials` 和 `ClientConfiguration` 物件傳遞到建構函數。
4. 設定用戶端物件的服務端點 URL。Amazon SWF 目前適用於所有 AWS 區域。

為方便起見，`GreeterWorker` 定義兩個字串常數。

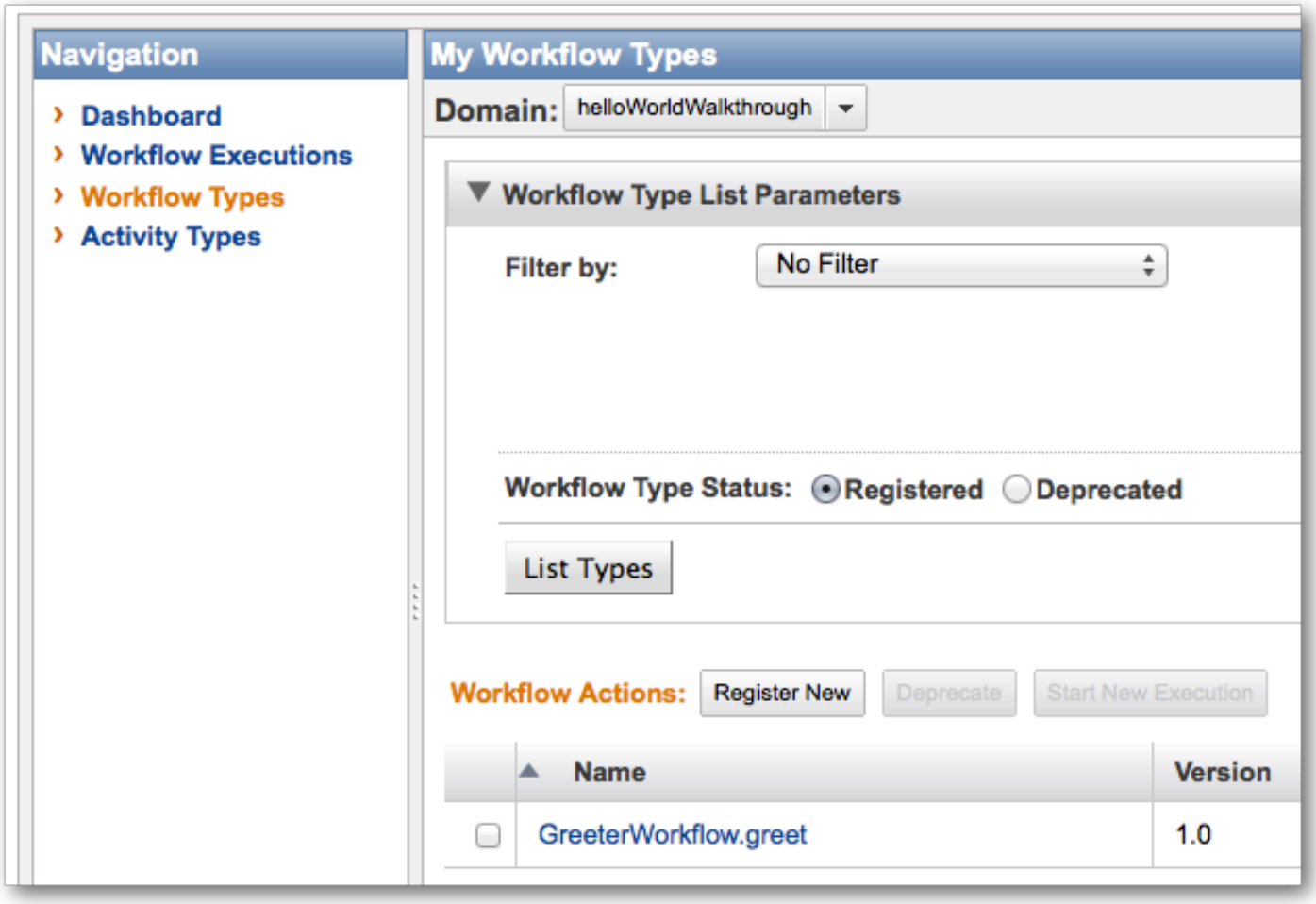
- `domain` 是您設定 Amazon SWF 帳戶時建立的工作流程的 Amazon SWF 網域名稱。`HelloWorldWorkflow` 假設您在「helloWorldWalkthrough」網域中執行工作流程。
- `taskListToPoll` 是 Amazon SWF 用來管理工作流程和活動工作者之間通訊的任務清單名稱。您可將名稱設成任何方便的字串。HelloWorldWorkflow 的工作流程和活動任務清單都使用 "HelloWorldList"。在幕後，名稱會以不同的命名空間作結，因此這兩份清單是截然不同的。

`GreeterWorker` 使用字串常數和 AmazonSimpleWorkflowClient 物件來建立工作者物件，以管理活動和工作者實作與 Amazon SWF 之間的互動。尤其工作者物件處理的任務為向合適的任務清單輪詢任務。

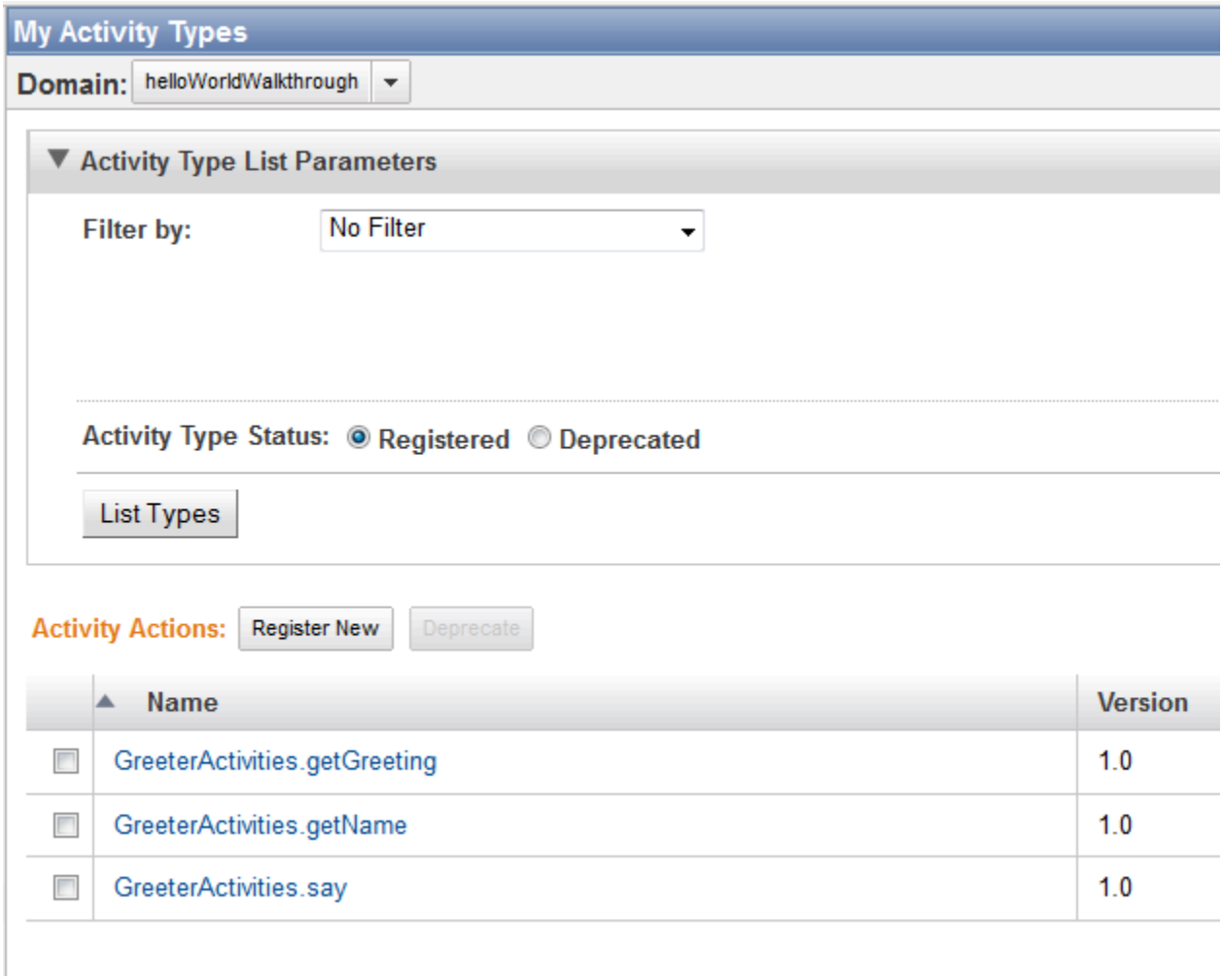
`GreeterWorker` 建立 `ActivityWorker` 物件，並新增新的類別執行個體以設定它處理 `GreeterActivitiesImpl`。然後，`GreeterWorker` 呼叫 `ActivityWorker` 物件的 `start` 方法，指引物件開始輪詢指定的活動任務清單。

`GreeterWorker` 建立 `WorkflowWorker` 物件，並新增類別檔案名稱 `GreeterWorkflowImpl.class` 以設定它處理 `GreeterWorkflowImpl`。然後，它呼叫 `WorkflowWorker` 物件的 `start` 方法，指引物件開始輪詢指定的工作流程任務清單。

此時您可以順利執行 `GreeterWorker`。它會向 Amazon SWF 註冊工作流程和活動，並啟動輪詢其個別任務清單的工作者物件。若要驗證這一點，請執行 `GreeterWorker` 並前往 Amazon SWF 主控台，然後從網域helloWorldWalkthrough清單中選取 。如果您在導覽窗格中選擇工作流程類型，您應該會看到 `GreeterWorkflow.greet`：



如果您選擇 Activity Types (活動類型)，即會顯示 `GreeterActivities` 方法：



但如果您選擇 Workflow Executions (工作流程執行)，會看到沒有任何作用中的執行。雖然工作流程和活動工作者正在輪詢任務，但我們尚未啟動工作流程執行。

HelloWorldWorkflow 啟動者

最後一塊拼圖便是實作工作流程啟動者，其為起始工作流程執行的應用程式。執行狀態由 Amazon SWF 存放，因此您可以檢視其歷史記錄和執行狀態。HelloWorldWorkflow 透過修改 `GreeterMain` 類別來實作工作流程啟動者，如下所示：

```
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` 使用和 `GreeterWorker` 一樣的程式碼建立 `AmazonSimpleWorkflowClient` 物件。然後，它建立 `GreeterWorkflowClientExternal` 物件，作用如同工作流程的代理，非常類似在 `GreeterWorkflowClientImpl` 中建立的活動用戶端，作用如同活動方法的代理。不要使用 `new` 建立工作流程用戶端物件，您必須：

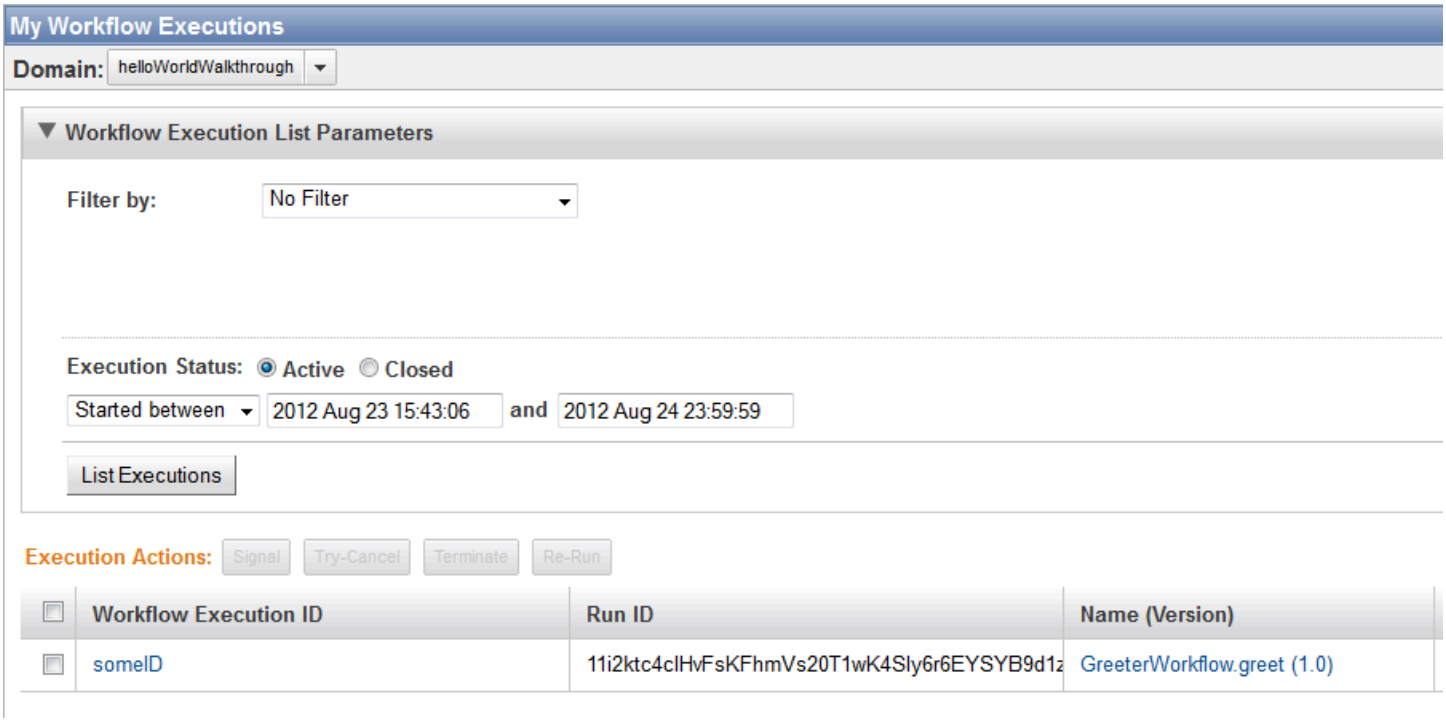
1. 建立外部用戶端原廠物件，並將`AmazonSimpleWorkflowClient`物件和 Amazon SWF 網域名稱傳遞給建構函數。用戶端 factory 物件是由框架的註釋處理器所建立，只要將 "ClientExternalFactoryImpl" 附加至工作流程界面名稱後面即可建立物件名稱。
2. 透過呼叫 factory 物件的 `getClient` 方法來建立外部用戶端物件，只要將 "ClientExternal" 附加至工作流程界面名稱後面即可建立物件名稱。您可以選擇性地傳遞`getClient`字串，Amazon SWF 將使用該字串來識別工作流程的此執行個體。否則，Amazon SWF 會使用產生的 GUID 代表工作流程執行個體。

從工廠傳回的用戶端只會建立名為 的工作流程，並將字串傳遞至 getClient 方法（從工廠傳回的用戶端在 Amazon SWF 中已有狀態）。若要使用不同的 ID 執行工作流程，您需要回到 factory 並使用另外指定的 ID 建立新的用戶端。

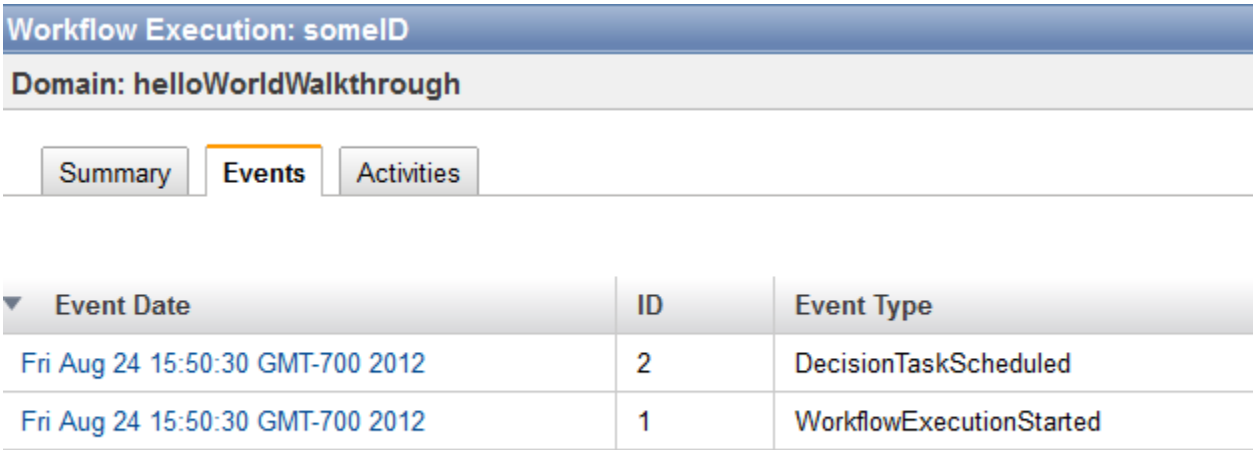
工作流程用戶端公開 `GreeterMain` 呼叫以開始工作流程的 `greet` 方法，因為 `greet()` 是以 `@Execute` 註釋指定的方法。

> **Note**
>
> 註釋處理器也會建立內部用戶端 factory 物件，用以建立子工作流程。如需詳細資訊，請參閱子工作流程執行。

如果 `GreeterWorker` 還在執行，請暫時予以關機，然後執行 `GreeterMain`。您現在應該會在 Amazon SWF 主控台的作用中工作流程執行清單中看到 someID：。



如果您選擇 `someID` 和選擇 Events (事件) 標籤，即會顯示事件：



> **Note**
>
> 如果之前已啟動 GreeterWorker，且其還在執行，您會看到一份較長的事件清單，內含最近討論過的原因。停止 GreeterWorker 並再次嘗試執行 GreaterMain。

Events (事件) 標籤只顯示兩個事件：

- WorkflowExecutionStarted 表示工作流程已開始執行。
- DecisionTaskScheduled 表示 Amazon SWF 已將第一個決策任務排入佇列。

在第一個決策任務封鎖工作流程的原因是，工作流程分散到兩個應用程式 GreeterMain 和 GreeterWorker。GreeterMain 已啟動工作流程執行，但 GreeterWorker 未執行，所以工作者未輪詢清單與執行任務。應用程式 您可單獨執行任一應用程式，但工作流程執行需要兩者，才能在第一項決策任務後繼續進行。如果您現在執行 GreeterWorker，則工作流程和活動工作者會開始輪詢，各種任務會迅速完成。如果您現在勾選 Events (事件) 標籤，即會顯示第一批次的事件。

Workflow Execution: someID

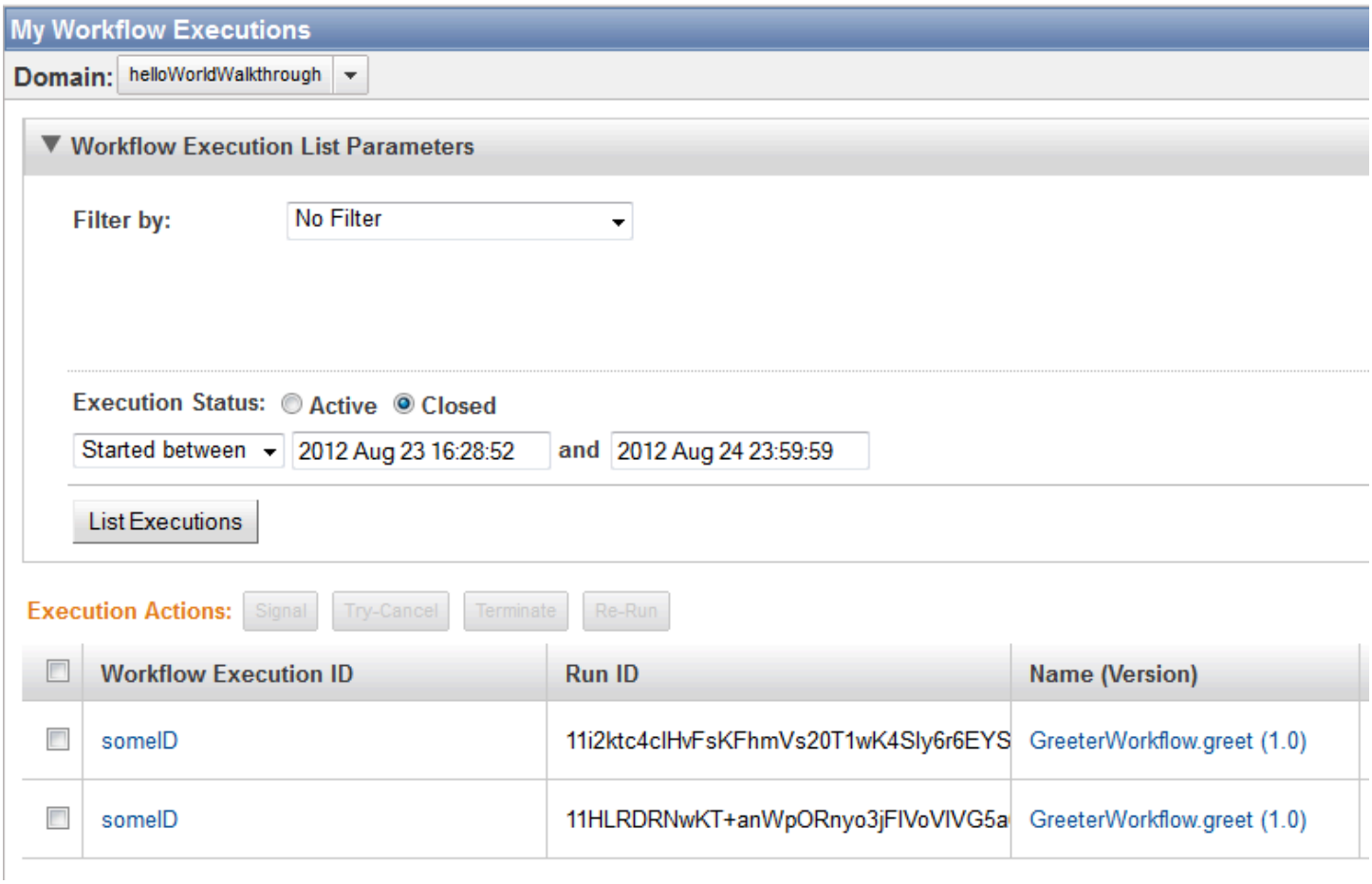
Domain: helloWorldWalkthrough

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

您可以選擇個別事件以取得詳細資訊。當您完成查看時，工作流程應該已列印「Hello World！」 您的主控台。

工作流程完成後，即不會出現在作用中的執行清單上。但您若想予以檢閱，請選擇 Closed (已結束) 執行狀態按鈕，然後選擇 List Executions (列出執行)。這會顯示指定網域 (`helloWorldWalkthrough`) 中所有已完成但未超過保留期的工作流程執行個體，其中保留期已於您建立網域時指定。



請注意，每個工作流程執行個體都有唯一的 Run ID (執行 ID) 值。您可以針對不同的工作流程執行個體使用相同的工作流程 ID，但一次只能執行一個作用中的執行。

HelloWorldWorkflowAsync 應用程式

在某些時候會偏好讓工作流程在本機執行某些任務，而不是使用活動。不過，工作流程任務通常包含處理 `Promise<T>` 物件所代表的值。如果您將 `Promise<T>` 物件傳遞給同步工作流程方法，則會立即執行方法，但無法存取 `Promise<T>` 物件值，直到物件就緒為止。您可以輪詢 `Promise<T>.isReady`，直到傳回 `true` 為止，但此效率較顯不彰，而且方法可能會被阻擋很長一段時間。較佳的方式是使用「非同步方法」。

非同步方法的實作方式非常類似標準方法，通常是工作流程實作類別的成員，並在工作流程實作的內容中執行。套用 `@Asynchronous` 註釋 (其指示框架將之視為活動)，您即可將之指定為非同步方法。

- 工作流程實作呼叫非同步方法時，會立即予以傳回。非同步方法一般會傳回 `Promise<T>` 物件，而此物件會在方法完成時就緒。

- 如果您將一或多個 `Promise<T>` 物件傳遞給非同步方法，則會延遲執行，直到所有輸入物件都就緒為止。因此，非同步方法可以存取其輸入 `Promise<T>` 值，而不會發出例外狀況。

> **Note**
>
> 由於 AWS Flow Framework 適用於 Java 的 執行工作流程的方式，非同步方法通常會執行多次，因此您應該僅將其用於快速的低額外負荷任務。您應該使用活動來執行大型運算這類冗長任務。如需詳細資訊，請參閱AWS Flow Framework 基本概念：分散式執行。

本主題是 HelloWorldWorkflowAsync (為 HelloWorldWorkflow 的修改版本，將其中一個活動取代為非同步方法) 的演練。若要實作應用程式，請在您的專案目錄中建立 helloWorld.HelloWorldWorkflow 套件的副本，並命名為 helloWorld.HelloWorldWorkflowAsync。

> **Note**
>
> 本主題是建置在 HelloWorld 應用程式 和 HelloWorldWorkflow 應用程式 主題所呈現的概念和檔案。熟悉這些主題呈現的檔案和概念再繼續。

下列各節說明如何修改原始 HelloWorldWorkflow 程式碼以使用非同步方法。

HelloWorldWorkflowAsync 活動實作

HelloWorldWorkflowAsync 會在 `GreeterActivities` 中實作其活動工作者界面，如下所示：

```
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);
}
```

此界面與 HelloWorldWorkflow 所使用的界面類似，相異之處如下：

- 此界面會省略 `getGreeting` 活動；非同步方法現在會處理該任務。
- 版本編號設定為 2.0。向 Amazon SWF 註冊活動界面之後，除非您變更版本編號，否則無法修改它。

其餘的活動方法實作與 HelloWorldWorkflow 相同。只需要從 `GreeterActivitiesImpl` 中刪除 `getGreeting`。

HelloWorldWorkflowAsync 工作流程實作

HelloWorldWorkflowAsync 定義工作流程界面，如下所示：

```
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();
}
```

除了新的版本編號之外，界面會與 HelloWorldWorkflow 相同。與活動相同，如果您想要變更已註冊的工作流程，則必須變更其版本。

HelloWorldWorkflowAsync 實作工作流程，如下所示：

```
import com.amazonaws.services.simpleworkflow.flow.annotations.Asynchronous;
import com.amazonaws.services.simpleworkflow.flow.core.Promise;

public class GreeterWorkflowImpl implements GreeterWorkflow {
   private GreeterActivitiesClient operations = new GreeterActivitiesClientImpl();

   @Override
   public void greet() {
      Promise<String> name = operations.getName();
      Promise<String> greeting = getGreeting(name);
      operations.say(greeting);
```

```
    }

    @Asynchronous
    private Promise<String> getGreeting(Promise<String> name) {
       String returnString = "Hello " + name.get() + "!";
       return Promise.asPromise(returnString);
    }
}
```

HelloWorldWorkflowAsync 會將 getGreeting 活動取代為 getGreeting 非同步方法，但 greet 方法的運作方式極為相同：

1. 執行 getName 活動，其會立即傳回代表名稱的 Promise<String> 物件 name。
2. 呼叫 getGreeting 非同步方法，並將 name 物件遞給它。getGreeting 會立即傳回代表問候語的 Promise<String> 物件 greeting。
3. 執行 say 活動，並將 greeting 物件傳遞給它。
4. getName 完成時，name 會就緒，且 getGreeting 會使用其值來建構問候語。
5. getGreeting 完成時，greeting 會就緒，且 say 會將字串列印至主控台。

差異在於以問候語呼叫非同步 getGreeting 方法，而不是呼叫活動用戶端來執行 getGreeting 活動。最後的結果會相同，但 getGreeting 方法的運作方式與 getGreeting 活動有些不同。

- 工作流程工作者會使用標準函數呼叫語意來執行 getGreeting。不過，活動的非同步執行是由 Amazon SWF 媒介。
- getGreeting 會在工作流程實作程序中執行。
- getGreeting 會傳回 Promise<String> 物件，而不是 String 物件。若要取得 Promise 保留的 String 值，您可以呼叫其 get() 方法。不過，由於活動是以非同步方式執行，因此其傳回值可能不會立即就緒； get()將引發例外狀況，直到非同步方法的傳回值可用為止。

  如需 Promise 運作方式的詳細資訊，請參閱「AWS Flow Framework 基本概念：活動和工作流程之間的資料交換」。

getGreeting 透過將問候語字串傳遞給靜態 Promise.asPromise 方法，來建立傳回值。此方法會建立適當類型的 Promise<T> 物件，並設定值，然後讓使之進入就緒狀態。

HelloWorldWorkflowAsync 工作流程和活動主機與啟動者

HelloWorldWorkflowAsync 將 `GreeterWorker` 實作為工作流程和活動實作的主機類別。此實作與 HelloWorldWorkflow 實作相同，差別在於 `taskListToPoll` 的名稱設為 "`HelloWorldAsyncList`"。

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

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

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

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

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

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

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

HelloWorldWorkflowAsync 會在 `GreeterMain` 中實作工作流程啟動者；與 HelloWorldWorkflow 實作相同。

若要執行工作流程，請執行 `GreeterWorker` 和 `GreeterMain`，如同 HelloWorldWorkflow。

HelloWorldWorkflowDistributed 應用程式

使用 HelloWorldWorkflow 和 HelloWorldWorkflowAsync，Amazon SWF 會調解工作流程和活動實作之間的互動，但它們會以單一程序在本機執行。 `GreeterMain` 處於不同的程序，但仍在相同的系統上執行。

Amazon SWF 的主要功能是支援分散式應用程式。例如，您可以在 Amazon EC2 執行個體上執行工作流程工作者、在資料中心電腦上執行工作流程入門，以及在用戶端桌上型電腦上執行活動。您甚至可以在不同的系統中執行不同的活動。

HelloWorldWorkflowDistributed 應用程式擴展 HelloWorldWorkflowAsync，將應用程式分散到兩個系統和三項程序。

- 在一個系統中，工作流程和工作流程啟動者執行為不同的程序。
- 在不同系統中執行的活動。

若要實作應用程式，請在您的專案目錄中建立 helloWorld.HelloWorldWorkflowAsync 套裝服務的複本，並命名為 helloWorld.HelloWorldWorkflowDistributed。以下各節說明如何修改原始的 HelloWorldWorkflowAsync 程式碼，將應用程式分散到兩個系統和三項程序。

您不必變更工作流程或活動實作，甚至不用變更版本編號，就可在不同的系統中執行它們。您也不必修改 `GreeterMain`。您只需要變更活動和工作流程主機。

使用 HelloWorldWorkflowAsync，單一應用程式的作用如同工作流程和活動主機。若要在不同的系統中執行工作流程和活動實作，您必須實作不同的應用程式。從專案中刪除 GreeterWorker，然後新增兩個新的類別檔案 GreeterWorkflowWorker 和 GreeterActivitiesWorker。

HelloWorldWorkflowDistributed 會在 GreeterActivitiesWorker 中實作它的活動主機，如下所示：

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

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

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

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

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

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

HelloWorldWorkflowDistributed 會在 `GreeterWorkflowWorker` 中實作它的工作流程主機，如下所示：

```
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();
   }
}
```

請注意，`GreeterActivitiesWorker` 只是沒有 `WorkflowWorker` 程式碼的 `GreeterWorker`，而 `GreeterWorkflowWorker` 只是沒有 `ActivityWorker` 程式碼的 `GreeterWorker`。

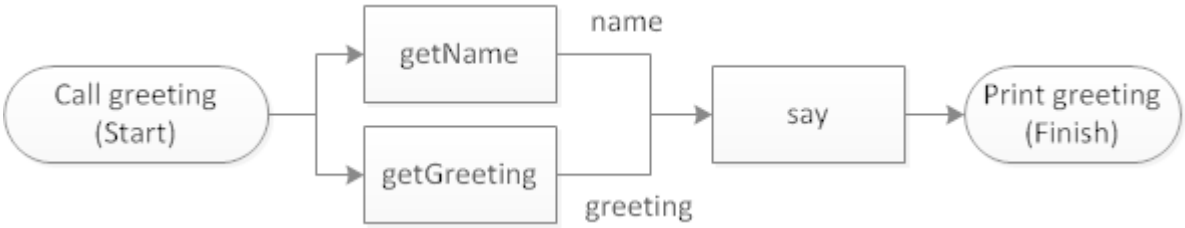
執行工作流程：

1. 以 `GreeterActivitiesWorker` 做為進入點來建立可執行的 JAR 檔案。
2. 將步驟 1 的 JAR 檔案複製到另一個系統，只要支援 Java，任何作業系統皆可。
3. 確保對相同 Amazon SWF 網域具有存取權的 AWS 登入資料可在其他系統上使用。
4. 執行 JAR 檔案。
5. 在您的開發系統中，使用 Eclipse 執行 `GreeterWorkflowWorker` 和 `GreeterMain`。

除了活動和工作流程工作者及工作流程啟動者在不同的系統中執行之外，工作流程運作的方式和 HelloWorldAsync 完全一致。不過，因為 `println`呼叫會列印「Hello World！」 至 主控台位於`say`活動中，輸出會出現在執行活動工作者的系統上。

HelloWorldWorkflowParallel 應用程式

先前的 Hello World! 版本皆使用線性工作流程拓撲。不過，Amazon SWF 不限於線性拓撲。HelloWorldWorkflowParallel 應用程式即為 HelloWorldWorkflow 的修改版本，使用平行拓撲，如下圖所示。



使用 HelloWorldWorkflowParallel，`getName` 和 `getGreeting` 會平行執行問候語的每個傳回部分。`say` 接著會將兩個字串合併到問候語，並將之列印至主控台。

若要實作應用程式，請在您的專案目錄中建立 helloWorld.HelloWorldWorkflow 套件的副本，並命名為 helloWorld.HelloWorldWorkflowParallel。下列各節說明如何修改原始 HelloWorldWorkflow 程式碼以平行執行 `getName` 和 `getGreeting`。

HelloWorldWorkflowParallel 活動工作者

HelloWorldWorkflowParallel 活動界面在 `GreeterActivities` 中實作，如下列範例所示。

```
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);
}
```

此界面與 HelloWorldWorkflow 類似，例外狀況如下：

- `getGreeting` 不會採用任何輸入；只會傳回問候語字串。
- `say` 採用兩個輸入字串：問候語和名稱。
- 界面具有新的版本編號，在您變更已註冊的界面時會需要。

HelloWorldWorkflowParallel 會實作 `GreeterActivitiesImpl` 中的活動，如下所示：

```
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` 和 `getGreeting` 現在只會傳回一半的問候語字串。`say` 會串連兩個部分來產生完整片語，並將之列印至主控台。

HelloWorldWorkflowParallel 工作流程工作者

HelloWorldWorkflowParallel 工作流程界面在 `GreeterWorkflow` 中實作，如下所示：

```
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();
}
```

類別與 HelloWorldWorkflow 版本相同，差別在於版本編號已為符合活動工作者而變更。

工作流程在 `GreeterWorkflowImpl` 中實作，如下所示：

```
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);
   }
}
```

概述而言，此實作與 HelloWorldWorkflow 十分類似；這三個活動用戶端方法會循序執行。不過，活動不同。

- HelloWorldWorkflow 將 `name` 傳遞給 `getGreeting`。因為 `name` 是 `Promise<T>` 物件，所以 `getGreeting` 延遲執行活動，直到 `getName` 完成，因此這兩個活動會循序執行。
- HelloWorldWorkflowParallel 不會傳遞任何輸入 `getName` 或 `getGreeting`。任一種方法都不會延遲執行，而且會立即平行執行相關聯的活動方法。

`say` 活動會採用 `greeting` 和 `name` 做為輸入參數。因為它們是 `Promise<T>` 物件，所以 `say` 會延遲執行，直到兩個活動完成，然後建構和列印問候語。

請注意，HelloWorldWorkflowParallel 不會使用任何特殊模型程式碼來定義工作流程拓撲。它透過使用標準 Java 流程控制並利用 `Promise<T>` 物件的屬性，隱含地執行此操作。 AWS Flow Framework 對於 Java 應用程式，只需搭配傳統 Java 控制流程建構使用`Promise<T>`物件，即可實作甚至複雜的拓撲。

HelloWorldWorkflowParallel 工作流程和活動主機與啟動者

HelloWorldWorkflowParallel 將 `GreeterWorker` 實作為工作流程和活動實作的主機類別。它與 HelloWorldWorkflow 實作相同，差別在於 `taskListToPoll` 名稱設為 "HelloWorldParallelList"。

`HelloWorldWorkflowParallel` 會在 `GreeterMain` 中實作工作流程啟動者，而且與 HelloWorldWorkflow 實作相同。

若要執行工作流程，請執行 `GreeterWorker` 和 `GreeterMain`，如同 `HelloWorldWorkflow`。

了解適用於 Java AWS Flow Framework 的

AWS Flow Framework 適用於 Java 的 可與 Amazon SWF 搭配使用，讓您輕鬆建立可擴展且容錯的應用程式，以執行長時間執行、遠端或兩者兼具的非同步任務。「Hello World！」 中的範例什麼是適用於 Java AWS Flow Framework 的 ？介紹了如何使用 AWS Flow Framework 實作基本工作流程應用程式的基本概念。本節提供 AWS Flow Framework 應用程式運作方式的概念資訊。第一個區段摘要說明 AWS Flow Framework 應用程式的基本結構，其餘區段則提供 AWS Flow Framework 應用程式運作方式的進一步詳細資訊。

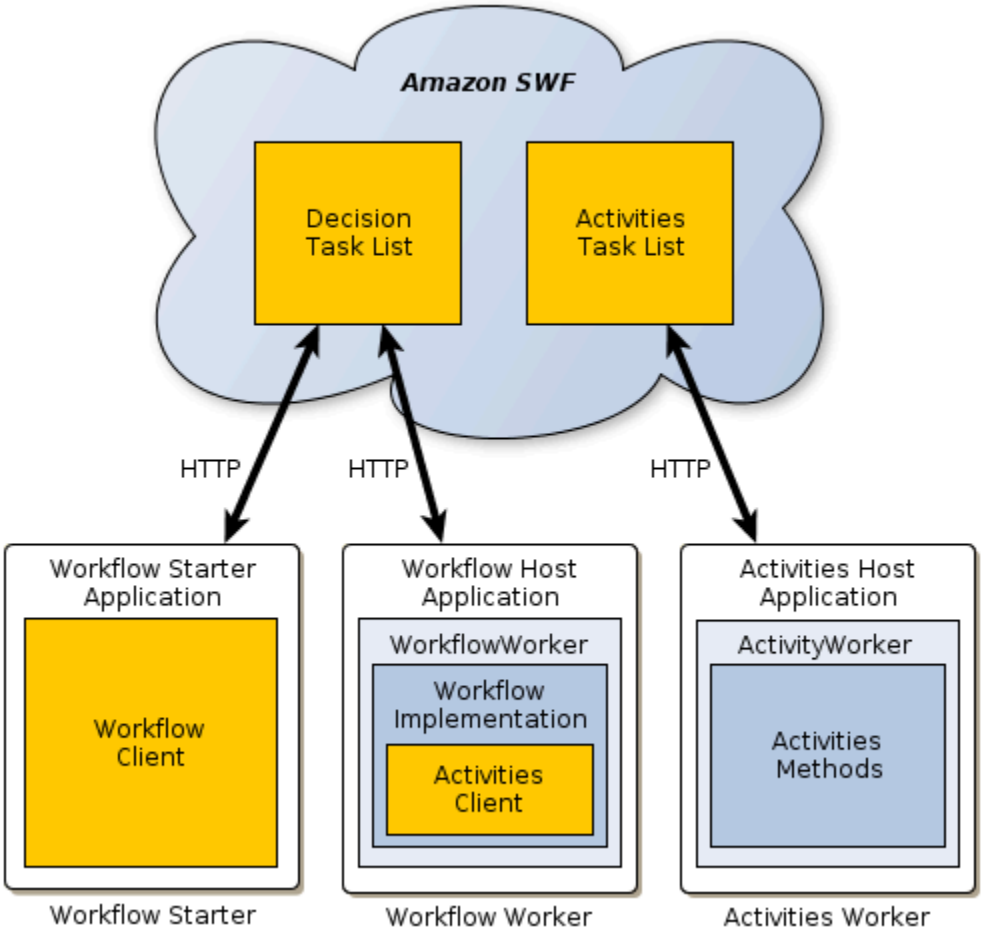
主題

- AWS Flow Framework 基本概念：應用程式結構
- AWS Flow Framework 基本概念：可靠的執行
- AWS Flow Framework 基本概念：分散式執行
- AWS Flow Framework 基本概念：任務清單和任務執行
- AWS Flow Framework 基本概念：可擴展的應用程式
- AWS Flow Framework 基本概念：活動和工作流程之間的資料交換
- AWS Flow Framework 基本概念：應用程式與工作流程執行之間的資料交換
- Amazon SWF 逾時類型

AWS Flow Framework 基本概念：應用程式結構

概念上， AWS Flow Framework 應用程式包含三個基本元件：工作流程啟動者、工作流程工作者和活動工作者。一或多個主機應用程式負責向 Amazon SWF 註冊工作者 （工作流程和活動）、啟動工作者，以及處理清理。工作者處理執行工作流程的機制，並可能在數部主機上實作。

此圖表代表基本 AWS Flow Framework 應用程式：



> **Note**
>
> 在三個不同的應用程式中實作這些元件在概念上很方便，但您可以建立應用程式以各種方式實作此功能。例如，您可以使用單一主機應用程式處理活動和工作流程工作者，或使用不同的活動和工作流程主機。您也可以使用多個活動工作者，每個工作者各在不同的主機上處理不同組的活動等。

這三個 AWS Flow Framework 元件會透過將 HTTP 請求傳送至 Amazon SWF 來間接互動，而 Amazon SWF 會管理請求。Amazon SWF 會執行下列動作：

- 維護一或多份決策任務清單，決定工作流程工作者要執行的下一個步驟。
- 維護一或多份活動任務清單，決定活動工作者要執行哪些任務。
- 維護工作流程執行的詳細逐步歷史記錄。

使用 時 AWS Flow Framework，您的應用程式程式碼不需要直接處理圖中所示的許多詳細資訊，例如傳送 HTTP 請求至 Amazon SWF。您只需呼叫 AWS Flow Framework 方法，架構就會處理幕後的詳細資訊。

活動工作者的角色

活動工作者執行工作流程必須完成的各種任務。包括：

- 活動實作，包含一套針對工作流程執行特定任務的活動方法。
- ActivityWorker 物件，使用 HTTP 長輪詢請求來輪詢 Amazon SWF 以執行活動任務。需要任務時，Amazon SWF 會透過傳送執行任務所需的資訊來回應請求。ActivityWorker 物件接著會呼叫適當的活動方法，並將結果傳回給 Amazon SWF。

工作流程工作者的角色

工作流程工作者協調執行各種活動、管理資料流程，以及處理失敗的活動。包括：

- 工作流程實作，包含活動協調邏輯、處理失敗的活動等。
- 活動用戶端，功能為活動工作者的代理，可讓工作流程工作者排程要非同步執行的活動。
- WorkflowWorker 物件，使用 HTTP 長輪詢請求來輪詢 Amazon SWF 以進行決策任務。如果工作流程任務清單中有任務，Amazon SWF 會傳回執行任務所需的資訊，以回應請求。架構接著會執行工作流程來執行任務，並將結果傳回給 Amazon SWF。

工作流程啟動者的角色

工作流程啟動者啟動工作流程執行個體 (也稱為「工作流程執行」)，可在執行期間與執行個體互動，以將額外資料傳遞給工作流程工作者或取得目前的工作流程狀態。

工作流程啟動者使用工作流程用戶端啟動工作流程執行，在執行期間視需要與工作流程互動，並處理清理。工作流程啟動者可以是本機執行的應用程式、Web 應用程式、 AWS CLI ，甚至是 AWS Management Console。

Amazon SWF 如何與您的應用程式互動

Amazon SWF 會調解工作流程元件之間的互動，並維護詳細的工作流程歷史記錄。Amazon SWF 不會啟動與元件的通訊；它會等待元件的 HTTP 請求，並視需要管理請求。例如：

- 如果請求來自工作者，輪詢可用的任務，Amazon SWF 會在任務可用時直接回應工作者。如需輪詢如何運作的詳細資訊，請參閱《Amazon Simple Workflow Service 開發人員指南》中的輪詢任務。
- 如果請求是來自活動工作者的通知，指出任務已完成，Amazon SWF 會記錄執行歷史記錄中的資訊，並將任務新增至決策任務清單，以通知工作流程工作者任務已完成，讓其繼續進行下一個步驟。
- 如果請求來自工作流程工作者來執行活動，Amazon SWF 會記錄執行歷史記錄中的資訊，並將任務新增至活動任務清單中，以指示活動工作者執行適當的活動方法。

此方法可讓工作者在任何具有網際網路連線的系統上執行，包括 Amazon EC2 執行個體、公司資料中心、用戶端電腦等。它們甚至不必在同一個作業系統上執行。因為 HTTP 請求來自工作者，所以不需要對外部開放連接埠，工作者可在防火牆後面執行。

如需詳細資訊

如需 Amazon SWF 運作方式的更詳細討論，請參閱 Amazon Simple Workflow Service 開發人員指南。

AWS Flow Framework 基本概念：可靠的執行

非同步分散式應用程式必須處理傳統應用程式不會發生的可靠性問題，包括：

- 如何在非同步分散式元件間「提供可靠的通訊」，例如在遠端系統長期執行的元件。
- 如果元件失敗或中斷連線，如何「確保結果不遺失」，特別是長期執行的應用程式。
- 如何「處理失敗的分散式元件」。

應用程式可以依賴 AWS Flow Framework 和 Amazon SWF 來管理這些問題。我們將探索 Amazon SWF 如何提供機制，以確保您的工作流程以可預測的方式可靠地運作，即使它們長時間執行，並且依賴以運算和人類互動執行的非同步任務。

提供可靠的通訊

AWS Flow Framework 使用 Amazon SWF 將任務分派給分散式活動工作者，並將結果傳回給工作流程工作者，藉此在工作流程工作者與其活動工作者之間提供可靠的通訊。Amazon SWF 使用以下方法，以確保工作者與其活動之間的可靠通訊：

- Amazon SWF 持久存放排程的活動和工作流程任務，並保證最多會執行一次。
- Amazon SWF 保證活動任務會順利完成並傳回有效的結果，或者會通知工作流程工作者任務失敗。

- Amazon SWF 會永久存放每個已完成活動的結果，或者，對於失敗的活動，它會存放相關的錯誤資訊。

AWS Flow Framework 然後， 會使用來自 Amazon SWF 的活動結果來決定如何繼續工作流程的執行。

確保結果不遺失

維護工作流程歷史記錄

在一 PB 資料上執行資料採礦操作的活動可能需要「數小時」才能完成，而指揮人力工作者執行複雜任務的活動可能需要「數天」甚至「數週」才能完成！

為了容納這類案例， AWS Flow Framework 工作流程和活動可能需要很長的時間才能完成：工作流程執行最多一年。可靠地執行長期執行的程序需要可持續長期存放工作流程執行歷史記錄的機制。

會根據維護每個工作流程執行個體執行歷史記錄的 Amazon SWF AWS Flow Framework 來處理此問題。工作流程的歷史記錄提供完整且權威的工作流程進度記錄，包括所有已排程和完成的工作流程和活動任務，及已完成或失敗之活動傳回的資訊。

AWS Flow Framework 應用程式通常不需要直接與工作流程歷史記錄互動，但可以在必要時存取。基於大部分的目的，應用程式讓框架在背後與工作流程歷史記錄互動即可。如需工作流程歷史記錄的完整討論，請參閱《Amazon Simple Workflow Service 開發人員指南》中的工作流程歷史記錄。

無狀態的執行

執行歷史記錄允許工作流程工作者為「無狀態」。如果您有多個工作流程或活動工作者的執行個體，則任何工作者皆可執行任何任務。工作者會收到從 Amazon SWF 執行任務所需的所有狀態資訊。

此方法可讓工作流程更可靠。例如，如果活動工作者失敗，您不必重新啟動工作流程。無論何時發生錯誤，只要重新啟動工作者，它就會開始輪詢任務清單並處理清單上的任何任務。您可以使用兩個或以上的工作流程和活動工作者 (可能在不同的系統上)，讓整體的工作流程可容錯。然後，如果其中一個工作者失敗，其他工作者會繼續處理排程的任務，而不中斷工作流程進度。

處理失敗的分散式元件

活動通常因暫時的原因 (如短暫的中斷連線) 而失敗，所以處理失敗活動的常用策略是重試活動。應用程式不需實作複雜的訊息傳遞策略，可依賴 AWS Flow Framework來處理重試程序。它可在工作流程中提供數種機制來重試失敗的活動，以及提供內建的例外狀況處理機制來搭配非同步分散式任務執行運作。

AWS Flow Framework 基本概念：分散式執行

工作流程執行個體基本上是執行的虛擬執行緒，可以跨越在多個遠端電腦上執行的活動和協調邏輯。Amazon SWF 和 AWS Flow Framework 函數作為作業系統，可透過下列方式管理虛擬 CPU 上的工作流程執行個體：

- 維護各執行個體的執行狀態。
- 在執行個體間切換。
- 在關閉執行個體的當下繼續執行執行個體。

重新執行工作流程

因為活動可以長期執行，所以不會只封鎖工作流程到完成。反之， 會使用重播機制來 AWS Flow Framework 管理工作流程執行，這倚賴 Amazon SWF 維護的工作流程歷史記錄在 片段中執行工作流程。

每個部分都會以「每項活動只執行一次」的方式重新執行工作流程邏輯，確保活動和同步方法不執行，直到它們的 Promise 物件就緒。

工作流程啟動者在啟動工作流程執行時起始第一個重新執行的部分。框架會呼叫工作流程進入點方法以及：

1. 執行所有不依賴活動完成的工作流程任務，包括呼叫所有活動用戶端方法。
2. 為 Amazon SWF 提供要排程執行的活動任務清單。在第一個部分，此清單只包含不依賴 Promise 且可立即執行的活動。
3. 通知 Amazon SWF 片段已完成。

Amazon SWF 會將活動任務存放在工作流程歷史記錄中，並將它們放在活動任務清單上以排程執行。活動工作者輪詢任務清單並執行任務。

當活動工作者完成任務時，它會將結果傳回至 Amazon SWF，該結果會記錄在工作流程執行歷史記錄中，並將工作流程工作者的新工作流程任務放在工作流程任務清單上來排程。工作流程工作者輪詢任務清單，在它接到任務時執行下一段的重新執行部分，如下所示：

1. 框架會再次執行工作流程進入點方法以及：
   - 執行所有不依賴活動完成的工作流程任務，包括呼叫所有活動用戶端方法。但是框架會檢查執行歷史記錄，卻不排程重複的活動任務。

- 檢查歷史記錄查看哪些活動任務已完成，執行依賴這些活動的任何非同步工作流程方法。

2. 當可執行的所有工作流程任務都已完成時，架構會向 Amazon SWF 回報：
   - 它為 Amazon SWF 提供自上次事件以來，輸入`Promise<T>`物件已就緒且可排程執行的任何活動清單。
   - 如果事件沒有產生其他活動任務，但仍有未完成的活動，架構會通知 Amazon SWF 該事件已完成。然後，它會等待另一項活動完成，初始化下一個重新執行的部分。
   - 如果事件沒有產生其他活動任務，且所有活動都已完成，則架構會通知 Amazon SWF 工作流程執行已完成。

如需重新執行行為的範例，請參閱「AWS Flow Framework for Java 重播行為」。

重新執行和非同步的工作流程方法

非同步工作流程方法的使用通常很像活動，因為方法會延遲執行直到所有輸入 `Promise<T>` 物件皆就緒。但是，重新執行機制處理非同步方法和處理活動不一樣。

- 重新執行不保證非同步方法只會執行一次。它會延遲執行非同步方法，直到其輸入 Promise 物件皆就緒，但它接著會針對所有後續部分執行該方法。
- 當非同步方法完成時，不會開始新的部分。

「AWS Flow Framework for Java 重播行為」中有提供重新執行非同步工作流程的範例。

重新執行與工作流程實作

大多數情況下，您不需要考慮重新執行機制的細節。它基本上是發生在幕後。不過，重新執行有兩項重要的實作針對您的工作流程實作。

- 不要使用工作流程方法來執行長期執行的任務，因為重新執行會多次重複該任務。即使是非同步的工作流程方法一般都會執行一次以上。請改用活動處理長期執行的任務，重新執行只會執行一次活動。
- 您的工作流程邏輯必須是完全確定的，每一個部分都必須採用相同的控制流程路徑。例如，控制流程路徑不應該相依於目前的時間。如需重新執行和確定性要求的詳細說明，請參閱「不確定性」。

AWS Flow Framework 基本概念：任務清單和任務執行

Amazon SWF 會透過將工作流程和活動任務發佈到具名清單來管理工作流程和活動任務。Amazon SWF 至少維護兩個任務清單，一個用於工作流程工作者，另一個用於活動工作者。

> **Note**
>
> 您可以視需要指定任意數量的任務清單，將不同的工作者指派給各清單。任務清單數目沒有任何限制。當您建立工作者物件時，您通常會在工作者主機應用程式中指定一份工作者任務清單。

下列來自 `HelloWorldWorkflow` 主機應用程式的摘錄會建立新的活動工作者，並將它指派到 `HelloWorldList` 活動任務清單。

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

根據預設，Amazon SWF 會在`HelloWorldList`清單中排程工作者的任務。接著工作者向此清單輪詢任務。您可為任務清單指派任何名稱。您甚至可以讓工作流程和活動清單使用同一個名稱。在內部，Amazon SWF 會將工作流程和活動任務清單名稱放在不同的命名空間中，因此兩個清單會有所不同。

如果您未指定任務清單， 會在工作者向 Amazon SWF 註冊類型時 AWS Flow Framework 指定預設清單。如需詳細資訊，請參閱工作流程和活動類型註冊。

有時候，讓特定的工作者或工作者群組執行某些任務是很有幫助的。例如，影像處理工作流程可能會使用一個活動來下載影像，然後使用另一個活動來處理影像。在同一個系統中執行這兩項任務比較有效率，並可避免在網路上傳輸大型檔案的費用。

若要支援此類案例，您可以使用包含 `schedulingOptions` 參數的多載，在您呼叫活動用戶端方法時明確指定任務清單。您可以透過傳遞適當設定`ActivitySchedulingOptions`物件的方法來指定任務清單。

例如，假設 HelloWorldWorkflow 應用程式的 say 活動由 getName 和 getGreeting 以外的活動工作者主持。下列範例示範如何確保 say 能和 getName 及 getGreeting 使用同一份任務清單，即使它們本來獲派不同的清單。

```
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);
  }
}
```

同步的 runSay 方法從其用戶端物件取得 getGreeting 任務清單。然後建立並設定 ActivitySchedulingOptions 物件，以確保 say 和 getGreeting 輪詢同一份任務清單。

> **Note**
>
> 當您將 schedulingOptions 參數傳遞到活動用戶端方法時，它只會覆寫用於該活動執行的原始任務清單。如果您再次呼叫活動用戶端方法，但未指定任務清單，Amazon SWF 會將任務指派給原始清單，活動工作者會輪詢該清單。

AWS Flow Framework 基本概念：可擴展的應用程式

Amazon SWF 有兩個主要功能，可讓您輕鬆擴展工作流程應用程式來處理目前的負載：

- 完整的工作流程執行歷史記錄，可讓您實作無狀態應用程式。
- 鬆散耦合到任務執行的任務排程，可輕鬆地擴展應用程式以符合目前需求。

Amazon SWF 透過將任務發佈到動態配置的任務清單來排程任務，而不是直接與工作流程和活動工作者通訊。相反地，工作者會使用 HTTP 請求來輪詢其個別任務清單。這種方法鬆散地將任務排程與任務執行耦合，並允許工作者在任何合適的系統上執行，包括 Amazon EC2 執行個體、公司資料中心、用戶端電腦等。由於 HTTP 請求源自於工作者，因此不需要外部可見的連接埠，這可讓工作者甚至在防火牆之後執行。

工作者用來輪詢任務的長時間輪詢機制能夠確保工作者不會過度負載。即使已排定的任務數量飆升，工作者會依自己的速度提取任務。不過，因為工作者是無狀態的，所以您可以啟動其他工作者執行個體來動態擴展應用程式，以符合增加的負載。即使它們是在不同的系統上執行，每個執行個體還是會提取相同的任務清單，而第一個可用的工作者執行個體會執行每個任務，不論工作者所在位置或其啟動時間為何。而負載減少時，您即可據此減少工作者數目。

AWS Flow Framework 基本概念：活動和工作流程之間的資料交換

當您呼叫非同步活動用戶端方法時，會立即傳回 Promise (也稱為 Future) 物件，代表活動方法的傳回值。一開始，Promise 處於未就緒狀態，並且未定義傳回值。在活動方法完成其任務並傳回之後，框架會將傳回值跨網路封送處理至工作流程工作者，以將值指派給 Promise，並讓物件進入就緒狀態。

即使活動方法沒有傳回值，您還是可以使用 Promise 來管理工作流程執行。如果您將傳回的 Promise 傳遞給活動用戶端方法或非同步工作流程方法，將延遲執行，直到物件就緒為止。

如果您將一或多個 Promise 傳遞給活動用戶端方法，框架會將任務置入佇列，但延遲進行排程，直到所有物件都就緒為止。框架接著會擷取每個 Promise 中的資料，並將之跨網際網路封送處理至活動工作者，以將之傳遞給活動方法做為標準類型。

> **Note**
>
> 如果您需要在工作流程與活動工作者之間傳輸大量資料，較建議的方法是在便利的位置存放資料，並且只傳遞擷取資訊。例如，您可以將資料存放在 Amazon S3 儲存貯體中，並傳遞相關聯的 URL。

Promise<T> 類型

`Promise<T>` 類型在某些方面與 Java `Future<T>` 類型類似。兩種類型都代表非同步方法所傳回的值，而且一開始未定義。您可以呼叫物件的 `get` 方法，來存取物件的值。除此之外，兩種類型的運作相當不同。

- Future<T> 是一種同步建構，允許應用程式等待非同步方法完成。如果您呼叫 get，而且物件未就緒，則會阻擋下來，直到物件就緒為止。
- 使用 Promise<T>，同步是由框架進行處理。如果您呼叫 get，而且物件未就緒，則 get 會拋出例外狀況。

Promise<T> 的主要用途是管理兩個活動之間的資料流程。其會在確認輸入資料有效之後，才執行活動。在許多情況下，工作流程工作者不需要直接存取 Promise<T> 物件；它們只會將物件從某個活動傳遞給另一個活動，然後讓框架和工作流程工作者處理詳細資訊。若要存取工作流程工作者中的 Promise<T> 物件值，您必須先確定物件已就緒，再呼叫其 get 方法。

- 較建議的方法是將 Promise<T> 物件傳遞給非同步工作流程方法，並在該處處理值。非同步方法會延遲執行，直到其所有輸入 Promise<T> 物件皆就緒，如此保證能讓您安全地存取其值。
- Promise<T> 會在物件就緒時，公開傳回 true 的 isReady 方法。不建議使用 isReady 來輪詢 Promise<T> 物件，但在特定情況下，isReady 十分有用。

AWS Flow Framework 適用於 Java 的 也包含衍生自 Promise<T>且具有類似行為的 Settable<T>類型。差別在於架構通常會設定 Promise<T> 物件的值，而工作流程工作者負責設定的值Settable<T>。

有一些情況是工作流程工作者需要建立 Promise<T> 物件，並設定其值。例如，傳回 Promise<T> 物件的非同步方法需要建立傳回值。

- 若要建立代表類型值的物件，請呼叫靜態 Promise.asPromise 方法，以建立適當類型的 Promise<T> 物件、設定其值，並讓其進入就緒狀態。
- 若要建立 Promise<Void> 物件，請呼叫靜態 Promise.Void 方法。

> **Note**
>
> Promise<T> 可以代表任何有效的類型。不過，如果必須跨網際網路封送處理資料，則類型必須與資料轉換器相容。如需詳細資訊，請參閱下節。

資料轉換器和封送處理

會使用資料轉換器跨網際網路將資料 AWS Flow Framework 合併。框架預設會使用根據 Jackson JSON 處理器的資料轉換器。不過，此轉換器有一些限制。例如無法封送處理未使用字串做為索引鍵

的對應。如果預設轉換器不適用您的應用程式，則您可以實作自訂資料轉換器。如需詳細資訊，請參閱DataConverters。

AWS Flow Framework 基本概念：應用程式與工作流程執行之間的資料交換

工作流程進入點方法可以有一或多個參數，允許工作流程啟動者將初始資料傳遞給工作流程。它也可以用來在執行期間將其他資料提供給工作流程。例如，如果客戶變更其送貨地址，則您可以通知訂單處理工作流程，以進行適當的變更。

Amazon SWF 允許工作流程實作訊號方法，這可讓工作流程入門等應用程式隨時將資料傳遞至工作流程。訊號方法可以有便利名稱和參數。您可以將之指定為訊號方法，做法為將之包含在您的工作流程界面定義中，並將 `@Signal` 註釋套用至方法宣告。

下列範例顯示可宣告訊號方法 `changeOrder` 的訂單處理工作流程界面，允許工作流程啟動者在工作流程啟動之後變更原始訂單。

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

框架的註釋處理器會建立名稱與訊號方法相同的工作流程用戶端方法，而且工作流程啟動者會呼叫用戶端方法以將資料傳遞給工作流程。如需範例，請參閱AWS Flow Framework 配方

Amazon SWF 逾時類型

為了確保工作流程執行正確執行，您可以使用 Amazon SWF 設定不同類型的逾時。有些逾時會指定工作流程總共可以執行多久。有些逾時則指定活動任務要多久才能指派給工作者，以及從排程開始要多久才可以完成。Amazon SWF API 中的所有逾時都會以秒為單位指定。Amazon SWF 也支援字串`NONE`做為逾時值，表示沒有逾時。

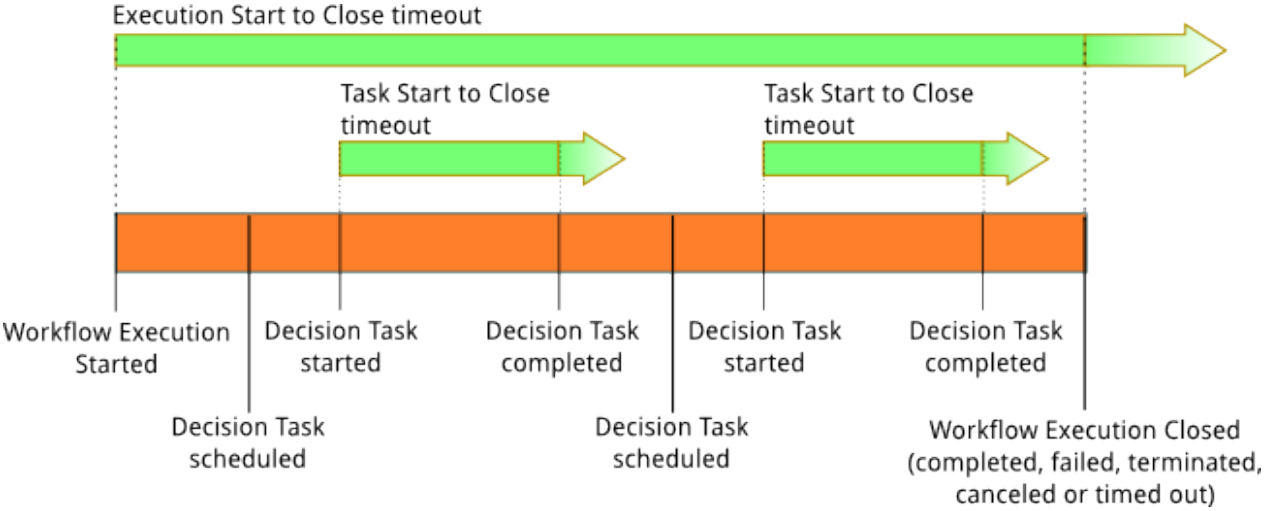
對於與決策任務和活動任務相關的逾時，Amazon SWF 會將事件新增至工作流程執行歷史記錄。事件的屬性提供有關發生哪種類型的逾時，以及哪些決策任務或活動任務受到影響的資訊。Amazon

SWF 也會排程決策任務。當決策者接到新的決策任務時，會在歷史記錄中看到逾時事件，並呼叫 RespondDecisionTaskCompleted 動作以採取適當的動作。

任務從排程起到結束的這段時間，視為開啟。因此，任務在工作者處理它時會回報為開啟。當工作者回報任務為完成、取消或失敗時，任務即結束。由於逾時，Amazon SWF 也可能會關閉任務。

工作流程和決策任務的逾時

下圖示範工作流程和決策逾時與工作流程生命週期的關係：

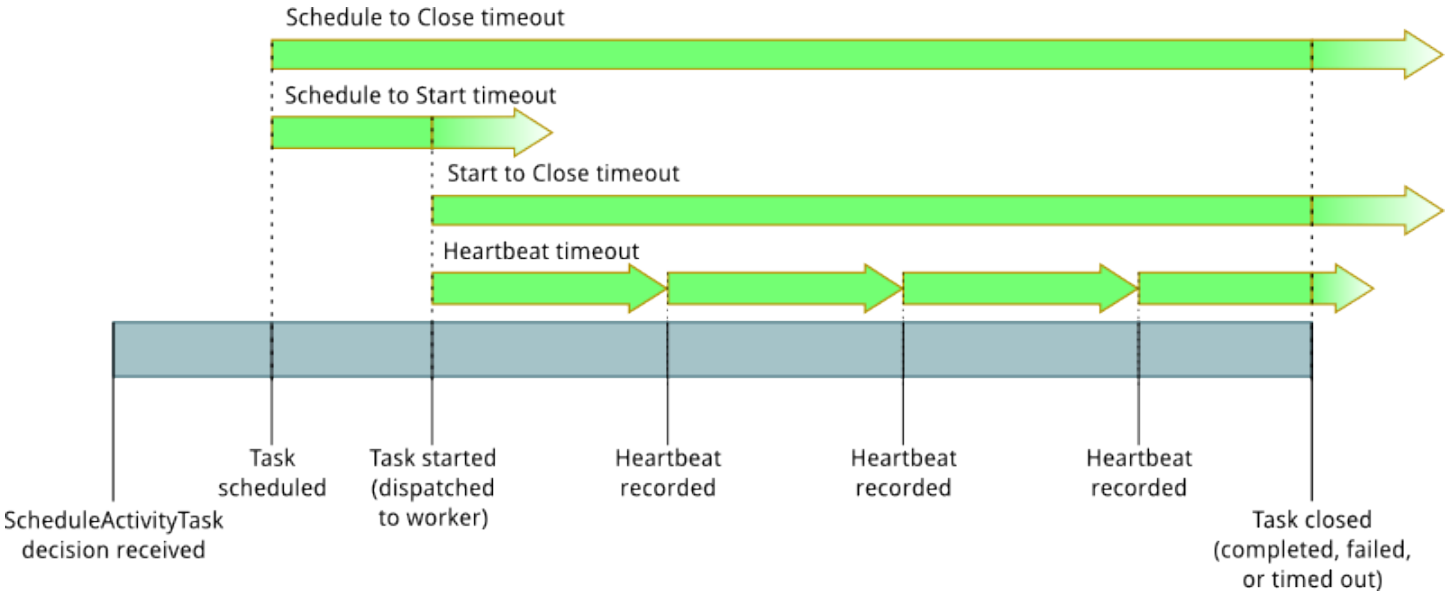


有兩種逾時類型與工作流程和決策任務相關：

- 工作流程開始關閉 (**`timeoutType: START_TO_CLOSE`**) – 此逾時會指定工作流程執行完成所需的時間上限。它在工作流程註冊期間設為預設值，但可在工作流程開始後用不同的值覆寫。如果超過此逾時，Amazon SWF 會關閉工作流程執行，並將類型為 WorkflowExecutionTimedOut 的事件新增至工作流程執行歷史記錄。除 `timeoutType` 之外，事件屬性還會指定對此工作流程執行生效的 `childPolicy`。子政策指定如果父工作流程執行逾時或終止，子工作流程執行的處理方式。例如，如果 `childPolicy` 設為 TERMINATE，則子工作流程執行會予以終止。一旦工作流程執行逾時，除可見度呼叫外，您無法對它採取任何動作。
- 決策任務開始關閉 (**`timeoutType: START_TO_CLOSE`**) – 此逾時會指定對應決策者完成決策任務所需的最長時間。它是在工作流程型註冊期間設定。如果超過此逾時，任務會在工作流程執行歷史記錄中標示為逾時，而 Amazon SWF 會將類型為 DecisionTaskTimedOut 的事件新增至工作流程歷史記錄。這些事件屬性會包含當此決策任務排程 (`scheduledEventId`) 及啟動時 (`startedEventId`) 所對應的事件 ID。除了新增事件之外，Amazon SWF 還會排程新的決策任務，以提醒決策者此決策任務已逾時。此逾時發生後，使用 `RespondDecisionTaskCompleted` 完成逾時決策任務的嘗試會失敗。

活動任務的逾時

下圖示範逾時與活動任務生命週期的關係：



有四種逾時類型與活動任務相關：

- 活動任務開始關閉 (**`timeoutType: START_TO_CLOSE`**) – 此逾時會指定活動工作者在收到任務後處理任務所需的時間上限。嘗試使用 RespondActivityTaskCanceled、RespondActivityTaskCompleted 和 RespondActivityTaskFailed 結束逾時的活動任務都將失敗。
- 活動任務活動訊號 (**`timeoutType: HEARTBEAT`**) – 此逾時會指定任務在提供動作進度之前可執行的時間上限`RecordActivityTaskHeartbeat`。
- 活動任務排程開始 (**`timeoutType: SCHEDULE_TO_START`**) – 此逾時會指定如果沒有工作者可執行任務，Amazon SWF 會等待多久才逾時活動任務。一旦逾時，過期的任務就會不指派給其他工作者。
- 活動任務排程關閉 (**`timeoutType: SCHEDULE_TO_CLOSE`**) – 此逾時會指定任務從排程到完成所需的時間。就最佳實務而言，此值不應大於任務從排程到開始逾時和任務從開始到結束逾時的總和。

> **Note**
>
> 每種逾時類型都有預設值，一般設為 NONE (無限)。但是任何活動執行的時間上限都限制在一年。

您是在活動類型註冊期間設定這些預設值，但在排程活動任務時還可使用新值加以覆寫。當其中一個逾時發生時，Amazon SWF 會將 ActivityTaskTimedOut 類型的事件新增至工作流程歷史記錄。此事件的 `timeoutType` 值屬性會指定這些逾時中的哪一個會發生。這些逾時每一個的 `timeoutType` 值都會出現在括號中。這些事件屬性也會包含當活動任務排程 (`scheduledEventId`) 及啟動時 (`startedEventId`) 所對應的事件 ID。除了新增事件之外，Amazon SWF 還會排程新的決策任務，以提醒決策者發生逾時。

了解 AWS Flow Framework 適用於 Java 的 中的任務

主題

- 任務
- 執行順序
- 工作流程執行
- 不確定性

任務

AWS Flow Framework 適用於 Java 的 用來管理非同步程式碼執行的基礎基本程式碼是 Task類別。Task 類型的物件代表必須同步執行的工作。當您呼叫同步方法時，框架會建立 Task 以在該方法中執行程式碼，並將它置入清單供稍後執行。同樣地，當您呼叫 Activity 時，也會為它建立 Task。方法呼叫會在這之後傳回，通常傳回 Promise<T> 做為呼叫的未來結果。

Task 類別為公有，可直接使用。例如，我們可以重新撰寫 Hello World 範例，讓其使用 Task，而不使用同步方法。

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

當傳遞到 Task 建構函數的所有 Promise 都準備就緒時，框架會呼叫 doExecute() 方法。如需 Task類別的詳細資訊，請參閱 適用於 Java 的 AWS SDK 文件。

框架也包含稱為 Functor 的類別，它代表也是 Promise<T> 的 Task。Functor 物件會在 Task 完成時準備就緒。在下列範例中，會建立 Functor 以取得歡迎訊息：

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

執行順序

只有當傳遞到對應之同步方法或活動的所有 `Promise<T>` 具類型參數都準備就緒時，任務才符合執行資格。隨時可供執行的 `Task` 在邏輯上已移至就緒佇列。換言之，它已排程準備執行。工作者類別會透過叫用您在非同步方法內文中撰寫的程式碼，或在活動方法的情況下排程 Amazon Simple Workflow Service (AWS) 中的活動任務來執行任務。

在任務執行並產生結果時，它們會讓其他任務準備就緒，讓程式持續執行不中斷。框架執行任務的方式對了解您同步程式碼的執行順序很重要。在您的程式中循序出現的程式碼，實際上不一定按照此順序執行。

```
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!");
}
```

上列程式碼列印如下：

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

```
Hello, Bob
```

這可能不是您期望的內容，但仔細思考同步方法的任務是如何執行的，就很容易說明：

1. 呼叫 getUserName 建立 Task。我們稱它為 Task1。由於 getUserName不會使用任何參數，因此 Task1 會立即放入就緒佇列中。
2. 接著，呼叫 printHelloName 建立需要等候 getUserName 結果的 Task。我們稱它為 Task2。由於必要值尚未就緒， Task2 會放入等待清單中。
3. 然後建立 printHelloWorld 任務，並新增至就緒佇列。我們稱它為 Task3。
4. 然後，陳述println式會列印「Hello， Amazon！」 主控台。
5. 此時，Task1 和 Task3 會在就緒佇列中，而 Task2 則在等候清單中。
6. 工作者執行 Task1，其結果讓 Task2 準備就緒。Task2 會新增至 Task3 後面的就緒佇列。
7. Task3 和 Task2 接著依此順序執行。

活動依相同的模式執行。當您在活動用戶端上呼叫方法時，它會建立Task在執行時排程 Amazon SWF 中活動的 。

框架依賴注入邏輯的程式碼產生及動態代理等功能，將方法呼叫轉換成您程式的活動呼叫及同步任務。

工作流程執行

工作流程實作的執行也由工作者類別管理。當您在工作流程用戶端上呼叫方法時，它會呼叫 Amazon SWF 來建立工作流程執行個體。Amazon SWF 中的任務不應與架構中的任務混淆。Amazon SWF 中的任務是活動任務或決策任務。執行活動任務很簡單。活動工作者類別會從 Amazon SWF 接收活動任務、在您的實作中叫用適當的活動方法，並將結果傳回給 Amazon SWF。

決策任務的執行較為複雜。工作流程工作者會從 Amazon SWF 接收決策任務。決策任務實際上是一項請求，詢問工作流程邏輯接下來要做什麼。第一個決策任務是針對工作流程執行個體透過工作流程用戶端啟動時而產生。一旦收到此決策任務，框架即會開始在以 @Execute 標註的工作流程方法中執行程式碼。此方法會執行排程活動的協調性邏輯。當工作流程執行個體的狀態變更時，例如，當活動完成時，進一步排程決策任務。此時，工作流程邏輯可以根據活動的結果決定執行動作，例如，它可能決定排程另一項活動。

框架將決策任務順暢轉譯為工作流程邏輯，對開發人員隱藏所有這些細節。就開發人員的觀點而言，此程式碼看起來就像是一般的程式。在封面下，框架會使用 Amazon SWF 維護的歷史記錄將其映射到 Amazon SWF 的呼叫和決策任務。當決策任務到達時，框架會重新執行外掛在目前已完成活動結果中的程式。解鎖等候這些結果的同步方法和活動並繼續執行程式。

執行的範例影像處理工作流程和對應的歷史記錄會顯示在下表中。

執行縮圖工作流程

執行工作流程程式	Amazon SWF 維護的歷史記錄
初始執行	
1. 分派迴圈 2. getImageUrls 3. downloadImage 4. createThumbnail (在等候佇列中的任務) 5. uploadImage (在等候佇列中的任務) 6. <再重複迴圈>	1. 啟動的工作流程執行個體，id="1" 2. 已排程 downloadImage
重新播放	
1. 分派迴圈 2. getImageUrls 3. 「downloadImage 影像」路徑="foo" 4. createThumbnail 5. uploadImage (在等候佇列中的任務) 6. <再重複迴圈>	1. 啟動的工作流程執行個體，id="1" 2. 已排程 downloadImage 3. 已完成 downloadImage，傳回="foo" 4. 已排程 createThumbnail
重新播放	
1. 分派迴圈 2. getImageUrls 3. 「downloadImage 影像」路徑="foo" 4. createThumbnail 縮圖路徑="bar" 5. uploadImage 6. <再重複迴圈>	1. 啟動的工作流程執行個體，id="1" 2. 已排程 downloadImage 3. 已完成 downloadImage，傳回="foo" 4. 已排程 createThumbnail 5. 已完成 createThumbnail，傳回="bar" 6. 已排程 uploadImage
重新播放	
1. 分派迴圈 2. getImageUrls	1. 啟動的工作流程執行個體，id="1" 2. 已排程 downloadImage

執行工作流程程式	Amazon SWF 維護的歷史記錄
3. 「downloadImage 影像」路徑="foo" 4. createThumbnail 縮圖路徑="bar" 5. uploadImage 6. <再重複迴圈>	3. 已完成 downloadImage，傳回="foo" 4. 已排程 createThumbnail 5. 已完成 createThumbnail，傳回="bar" 6. 已排程 uploadImage 7. 已完成 uploadImage ...

呼叫 `processImage` 時，架構會在 Amazon SWF 中建立新的工作流程執行個體。這是正在啟動之工作流程執行個體的永久記錄。程式會執行直到呼叫`downloadImage`活動為止，這會要求 Amazon SWF 排程活動。工作流程會進一步執行並建立後續活動的任務，但在`downloadImage`活動完成之前，無法執行它們；因此，此重播階段會結束。Amazon SWF 會為執行`downloadImage`活動分派任務，一旦完成，就會在歷史記錄中記錄結果。工作流程現在已準備好繼續進行，並由 Amazon SWF 產生決策任務。框架收到決策任務，並重新執行外掛在歷史記錄所記錄之下載影像結果中的工作流程。這會解除封鎖 的任務`createThumbnail`，並透過在 Amazon SWF 中排程`createThumbnail`活動任務來更進一步地繼續執行程式。`uploadImage` 會重複同樣的程序。程式以此方式繼續執行，直到工作流程處理完所有影像且無任何等待中的任務為止。由於不會在本機存放任何執行狀態，因此每個決策任務都可能在不同機器上執行。這可讓您輕鬆撰寫可容錯且可擴展的程式。

不確定性

由於架構依賴於重播，因此協同運作程式碼（活動實作除外的所有工作流程程式碼）具有決定性。例如，您程式中的控制流程不應該相依於亂數或目前的時間。由於這些物件會在叫用之間變更，因此重播可能不會遵循協同運作邏輯的相同路徑。這會導致未預期的結果或錯誤。框架提供 `WorkflowClock` 讓您以確定方式來取得目前的時間。如需詳細資訊，請參閱「執行內容」一節。

> **Note**
>
> 工作流程實作物件的不正確 Spring 接線也會導致不確定性。工作流程實作 Bean 及其相依的 Bean 都必須在工作流程範圍內 (`WorkflowScope`)。例如，將工作流程實作 Bean 接線到保留狀態且位於全域內容中的 Bean，將會導致未預期的行為。如需詳細資訊，請參閱 Spring 整合區段內容。

AWS Flow Framework for Java 程式設計指南

本節提供如何使用 AWS Flow Framework 適用於 Java 的 功能來實作工作流程應用程式的詳細資訊。

主題

- 使用 實作工作流程應用程式 AWS Flow Framework
- 工作流程和活動合約
- 工作流程和活動類型註冊
- 活動和工作流程用戶端
- 工作流程實作
- 活動實作
- 實作 AWS Lambda 任務
- 執行使用 AWS Flow Framework 適用於 Java 的 編寫的程式
- 執行內容
- 子工作流程執行
- 持續的工作流程
- 在 Amazon SWF 中設定任務優先順序
- DataConverters
- 將資料傳遞給非同步方法
- 可試性與相依性插入
- 錯誤處理
- 重試失敗的活動
- 協助程式任務
- AWS Flow Framework for Java 重播行為

使用 實作工作流程應用程式 AWS Flow Framework

使用 開發工作流程所涉及的一般步驟 AWS Flow Framework 如下：

1. 定義活動和工作流程合約。分析應用程式的需求，然後判斷必要的活動和工作流程拓撲。「活動」會處理必要的處理任務，而「工作流程拓撲」定義工作流程的基本結構和商業邏輯。

例如，媒體處理應用程式可能需要下載檔案，並處理檔案，然後將處理好的檔案上傳至 Amazon Simple Storage Service (S3) 儲存貯體。這可以分為四個活動任務：

1. 從伺服器下載檔案
2. 處理檔案 (例如，將檔案轉碼為不同的媒體格式)
3. 將檔案上傳至 S3 儲存貯體
4. 刪除本機檔案以執行清理

此工作流程會有進入點方法，並會實作可循序執行活動的簡單線性拓撲，這與 HelloWorldWorkflow 應用程式 十分類似。

2. 實作活動和工作流程界面。工作流程和活動合約藉由 Java「界面」定義，並透過 SWF 將其呼叫慣例設為可預測，然後讓您彈性實作工作流程邏輯和活動任務。您程式中的各個部分可做為彼此資料的取用者，但不需要完全了解其他部分的實作詳細資訊。

   例如，您可以定義 `FileProcessingWorkflow` 界面，並提供不同的工作流程實作進行影片編碼、壓縮、縮圖等。所有這些工作流程都可以有不同的控制流程，而且可以呼叫不同的活動方法；您的工作流程啟動者並不需要知道。透過使用界面，也可以使用稍後可取代為工作中程式碼的模擬實作來簡單地測試工作流程。

3. 產生活動和工作流程用戶端。 AWS Flow Framework 不需要您實作管理非同步執行、傳送 HTTP 請求、匯集資料等的詳細資訊。相反地，工作流程啟動者會對工作流程用戶端呼叫方法來執行工作流程執行個體，而且工作流程實作會對活動用戶端呼叫方法來執行活動。框架會在背景處理這些互動的詳細資訊。

   如果您使用的是 Eclipse，而且已如 一樣設定專案設定 AWS Flow Framework 適用於 Java 的，則 AWS Flow Framework 註釋處理器會使用介面定義來自動產生工作流程和活動用戶端，這些用戶端會公開與對應介面相同的一組方法。

4. 實作活動和工作流程主機應用程式。您的工作流程和活動實作必須嵌入主機應用程式中，以輪詢 Amazon SWF 任務、彙整任何資料，並呼叫適當的實作方法。 AWS Flow Framework 適用於 Java 的 包含 WorkflowWorker 和 ActivityWorker 類別，讓實作主機應用程式變得簡單明瞭。
5. 測試您的 workflow. AWS Flow Framework for Java 提供 JUnit 整合，您可以用來在內嵌和本機測試您的工作流程。
6. 部署工作者。您可以視需要部署工作者，例如，您可以將工作者部署到 Amazon EC2 執行個體或資料中心的電腦。部署並啟動後，工作者會開始輪詢 Amazon SWF 任務，並視需要處理它們。
7. 開始執行。應用程式會使用工作流程用戶端呼叫工作流程的進入點，來啟動工作流程執行個體。您也可以使用 Amazon SWF 主控台啟動工作流程。無論您如何啟動工作流程執行個體，都可以使

用 Amazon SWF 主控台來監控執行中的工作流程執行個體，並檢查工作流程歷史記錄是否有執行中、已完成和失敗的執行個體。

適用於 Java 的 AWS SDK 包含一組 AWS Flow Framework 適用於 Java 範例的 ，您可以依照根目錄中 readme.html 檔案中的指示進行瀏覽和執行。還有一組配方 — 簡單的應用程式 — 顯示如何處理各種特定的程式設計問題，這些問題可從AWS Flow Framework 配方取得。

工作流程和活動合約

Java 界面是用來宣告工作流程和活動的簽章。界面形成工作流程 (或活動) 實作與該工作流程 (或活動) 之用戶端之間的合約。例如，使用標註 `@Workflow` 註釋的界面來定義工作流程類型 `MyWorkflow`：

```
@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();
}
```

合約沒有實作專屬設定。這項實作本質合約的使用允許用戶端與實作脫離，從而能彈性地變更實作詳細資訊而不中斷用戶端。您也可以換成變更用戶端，而不需要變更所使用的工作流程或活動。例如，可能會修改用戶端以 Promise (`Promise<T>`) 非同步呼叫活動，而不需要變更活動實作。同樣地，活動實作可能會變更，以便以非同步方式完成，例如由傳送電子郵件的人員完成，而不需要變更活動的用戶端。

在上面的範例中，工作流程界面 `MyWorkflow` 包含 `startMyWF` 方法來啟動新的執行。此方法會標註 `@Execute` 註釋，而且必須具有傳回類型 `void` 或 `Promise<>`。在指定的工作流程界面中，最多一種方法可以標註此註釋。此方法是工作流程邏輯的進入點，而且框架會在收到決策任務時呼叫此方法，以執行工作流程邏輯。

工作流程界面也會定義可能傳送至工作流程的訊號。工作流程執行在收到具有相符名稱的訊號時，便會呼叫訊號方法。例如，MyWorkflow 界面會宣告已標註 @Signal 註釋的訊號方法 signal1。

訊號方法上需要有 @Signal 註釋。訊號方法的傳回類型必須是 void。工作流程界面中可以定義零或多個訊號方法。您可宣告未有 @Execute 方法及含有一些 @Signal 方法的工作流程界面，藉此產生無法啟動執行但可將訊號傳送至運作中之執行的用戶端。

已標註 @Execute 和 @Signal 註釋的方法可能有 Promise<T> 或其衍生物以外任何類型之任意數目的參數。這可讓您在啟動及執行時將強類型輸入傳遞至工作流程執行。@Execute 方法的傳回類型必須是 void 或 Promise<>。

此外，您也可以在工作流程界面中宣告方法，以報告工作流程執行的最新狀態 (例如，先前範例中的 getState 方法)。此狀態不是工作流程的整個套用狀態。此功能的預定用途是可讓您最多存放 32 KB 的資料，指出執行的最新狀態。例如，在訂單處理工作流程中，您可以存放字串，以指出收到、處理或取消的訂單。每次在決策任務完成之時，框架就會呼叫此方法以取得最新狀態。狀態存放在 Amazon Simple Workflow Service (Amazon SWF) 中，並且可以使用產生的外部用戶端擷取。這可讓您檢查工作流程執行的最新狀態。已標註 @GetState 的方法不得採用任何引數，而且不得具有 void 傳回類型。您可以從此方法傳回任何符合您需求的類型。在上述範例中，用來存放字串狀態和完成百分比數值的方法會傳回 MyWorkflowState 的物件 (請參閱下面的定義)。此方法預期會對工作流程實作物件執行唯讀存取，並且進行同步呼叫，以禁止使用任何非同步操作 (例如呼叫已標註 @Asynchronous 的方法)。工作流程界面中最多一種方法能夠標註 @GetState 註釋。

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

同樣地，會使用已標註 @Activities 註釋的界面來定義一組活動。界面中的每個方法都會對應至 活動，例如：

```
@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);
}
```

此界面可讓您為相關的活動分組。您可以在活動界面內定義任意數目的活動，且您可以定義所需數目的活動界面。與 @Execute 和 @Signal 方法類似，活動方法可以採用 Promise<T> 或其衍生物以外任何類型之任意數目的引數。活動的傳回類型不得是 Promise<T> 或其衍生物。

工作流程和活動類型註冊

Amazon SWF 需要先註冊活動和工作流程類型，才能使用。框架會在您新增至工作者的實作中自動註冊工作流程和活動。此架構會尋找實作工作流程和活動的類型，並將其註冊至 Amazon SWF。框架預設會使用界面定義來推斷工作流程和活動類型的註冊選項。所有工作流程界面必須具有 @WorkflowRegistrationOptions 註釋或 @SkipRegistration 註釋。工作流程工作者會註冊用來設定它的所有工作流程類型，而這些工作流程類型具有 @WorkflowRegistrationOptions 註釋。同樣地，每個活動方法都需要標註 @ActivityRegistrationOptions 註釋或 @SkipRegistration 註釋，或者其中一個註釋必須存在於 @Activities 界面上。活動工作者會註冊用來設定它的所有活動類型，而這些活動類型已套用 @ActivityRegistrationOptions 註釋。當您啟動其中一個工作者時，會自動執行註冊。具有@SkipRegistration註釋的工作流程和活動類型未註冊。@ActivityRegistrationOptions和@SkipRegistration註釋已覆寫語意，而最明確的一個會套用到活動類型。

請注意，Amazon SWF 不允許您在註冊後重新註冊或修改類型。框架將嘗試註冊所有類型，但不會重新註冊已經註冊的類型，且不會報告錯誤。

如果您需要修改已註冊的設定，則必須註冊類型的新版本。啟動新執行時，或呼叫使用已產生之用戶端的活動時，您也可以覆寫已註冊的設定。

註冊需要類型名稱以及其他註冊選項。預設實作會判斷這些項目，如下所示：

工作流程類型名稱和版本

框架會從工作流程界面判斷工作流程類型的名稱。預設的工作流程類型名稱格式為 {*prefix*}{*name*}。{*prefix*} 設定為 @Workflow 界面的名稱，後接 '.'，而 {*name*} 設定為 @Execute 方法的名稱。上述範例中工作流程類型的預設名稱為 MyWorkflow.startMyWF。您可以使用 @Execute 方法的 name 參數來覆寫預設名稱。範例中工作流程類型的預設名稱為 startMyWF。名稱不得為空字串。請注意，當您使用 @Execute 覆寫名稱時，框架不會自動在名稱前面加上前綴。您可以自由使用自己的命名方式。

工作流程版本使用 @Execute 註釋的 version 參數所指定。version 沒有預設值，而且必須明確予以指定；version 是任意形式的字串，您可以自由地使用自己的版本控制方式。

訊號名稱

訊號名稱可以使用 @Signal 註釋的 name 參數予以指定。如果未指定，則會預設為訊號方法的名稱。

活動類型名稱和版本

框架會從活動界面判斷活動類型的名稱。預設的活動類型名稱格式為 {*prefix*}{*name*}。{*prefix*} 設定為 @Activities 界面的名稱，後接 '.'，而 {*name*} 設定為方法名稱。您可以在活動界面的@Activities註釋中覆寫預設的 {*prefix*}。您也可以在活動方法上使用 @Activity 註釋，來指定活動類型名稱。請注意，當您使用 @Activity 覆寫名稱時，框架不會自動在名稱前面加上前綴。您可以自由地使用自己的命名方式。

活動版本使用 @Activities 註釋的 version 參數所指定。此版本用來做為界面中所定義之所有活動的預設值，而且可以使用 @Activity 註釋以根據活動予以覆寫。

預設任務清單

預設任務清單的設定方式是使用 @WorkflowRegistrationOptions 和 @ActivityRegistrationOptions 註釋以及設定 defaultTaskList 參數。預設會設定為 USE_WORKER_TASK_LIST。此特殊值指示框架使用工作者物件上所設定的任務清單，而工作者物件用於註冊活動或工作流程類型。您也可以使用這些註釋，將預設任務清單設定為 NO_DEFAULT_TASK_LIST，以選擇不註冊預設任務清單。這可以用於您需要在執行階段指定任務清單時。如果尚未註冊預設任務清單，則在啟動工作流程時，或在產生之用戶端的個別方法多載上使用 StartWorkflowOptions 和 ActivitySchedulingOptions 參數來呼叫活動方法時，您必須指定任務清單。

其他註冊選項

Amazon SWF API 允許的所有工作流程和活動類型註冊選項都可以透過架構指定。

如需完整的「工作流程」註冊選項清單，請參閱下列項目：

- @工作流程
- @Execute
- @WorkflowRegistrationOptions
- @Signal

如需完整的「活動」註冊選項清單，請參閱下列項目：

- @Activity
- @Activities
- @ActivityRegistrationOptions

如果您想要完全掌控類型註冊，請參閱「工作者可擴充性」。

活動和工作流程用戶端

框架會根據 `@Workflow` 和 `@Activities` 界面產生工作流程和活動用戶端。會產生不同的用戶端界面，其中包含只對用戶端有意義的方法和設定。如果您使用 Eclipse 進行開發，則每次儲存包含適當界面的檔案時，Amazon SWF Eclipse 外掛程式都會執行此操作。產生的程式碼會放在專案中產生的來源目錄裡，而專案位於與界面相同的套件中。

> **Note**
>
> 請注意，Eclipse 所使用的預設目錄名稱是 .apt_generated。Eclipse 不會在 Package Explorer (套件瀏覽器) 中顯示名稱開頭為 '.' 的目錄。如果您想要在 Project Explorer (專案瀏覽器) 中檢視產生的檔案，則請使用不同的目錄名稱。在 Eclipse 中，以滑鼠右鍵按一下 Package Explorer (套件瀏覽器) 中的套件，並選擇 Properties (屬性)、Java Compiler (Java 編譯器)、Annotation processing (註釋處理)，然後修改 Generate source directory (產生來源目錄) 設定。

工作流程用戶端

針對工作流程所產生的成品包含三個用戶端界面及實作它們的類別。產生的用戶端包含：

- 「非同步用戶端」，要在提供非同步方法來啟動工作流程執行並傳送訊號的工作流程實作內使用
- 「外部用戶端」，可用來啟動執行、傳送訊號，並在工作流程實作範圍外部擷取工作流程狀態
- 「自主用戶端」，可用來建立持續工作流程

例如，針對範例 `MyWorkflow` 界面所產生的用戶端界面如下：

```
//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);
```

這些界面的多載方法對應至您所宣告之 `@Workflow` 界面中的每個方法。

外部用戶端會鏡射 `@Workflow` 界面上的方法，而此界面具有採用 `StartWorkflowOptions` 之 `@Execute` 方法的一個額外多載。您可以使用此多載，以在啟動新工作流程執行時傳遞其他選項。這些選項可讓您覆寫預設任務清單、逾時設定，以及建立標籤與工作流程執行的關聯。

另一方面，非同步用戶端的方法允許非同步呼叫 `@Execute` 方法。在工作流程界面中，於 `@Execute` 方法的用戶端界面中產生下列方法多載：

1. 依原狀採用原始引數的多載。如果原始方法已傳回 void，則此多載的傳回類型會是 Promise<Void>；否則會是原始方法中所宣告的 Promise<>。例如：

   原始方法：

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

   產生的方法：

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

   在工作流程的所有引數皆可用且不需要等待時，應該使用此多載。

2. 包含原始引數及 Promise<?> 類型之其他變數引數的多載。如果原始方法已傳回 void，則此多載的傳回類型會是 Promise<Void>；否則會是原始方法中所宣告的 Promise<>。例如：

   原始方法：

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

   產生的方法：

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

   如果工作流程的所有引數皆可用且不需要等待，但您想要等待一些其他 Promise 就緒，則應該使用此多載。變數引數可以用來傳遞 Promise<?> 物件，這類物件未宣告為引數，但您想要在執行呼叫之前等待。

3. 包含原始引數、StartWorkflowOptions 類型之其他引數及 Promise<?> 類型之其他變數引數的多載。如果原始方法已傳回 void，則此多載的傳回類型會是 Promise<Void>；否則會是原始方法中所宣告的 Promise<>。例如：

   原始方法：

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

   產生的方法：

   ```
   Promise<void> startMyWF(
   ```

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

如果工作流程的所有引數皆可用且不需要等待、您想要覆寫用於啟動工作流程執行的預設設定，或想要等待一些其他 Promise 就緒，則應該使用此多載。變數引數可以用來傳遞 Promise<?> 物件，這類物件未宣告為引數，但您想要在執行呼叫之前等待。

4. 原始方法中每個引數皆取代為 Promise<> 包裝函式的多載。如果原始方法已傳回 void，則此多載的傳回類型會是 Promise<Void>；否則會是原始方法中所宣告的 Promise<>。例如：

   原始方法：

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

   產生的方法：

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

   要非同步評估要傳遞給工作流程執行的引數時，應該使用此多載。在傳遞給此方法多載的所有引數都就緒前，不會對其執行呼叫。

   如果部分引數已就緒，則會透過 Promise.asPromise(*value*) 方法將它們轉換成已處於就緒狀態的 Promise。例如：

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

5. 原始方法中每個引數皆取代為 Promise<> 包裝函式的多載。多載也會具有 Promise<?> 類型的其他變數引數。如果原始方法已傳回 void，則此多載的傳回類型會是 Promise<Void>；否則會是原始方法中所宣告的 Promise<>。例如：

   原始方法：

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

產生的方法：

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

如果要非同步評估要傳遞給工作流程執行的引數，而且您也想要等待一些其他 Promise 就緒，則應該使用此多載。在傳遞給此方法多載的所有引數都就緒前，不會對其執行呼叫。

6. 原始方法中每個引數皆取代為 `Promise<?>` 包裝函式的多載。多載也會具有 `StartWorkflowOptions` 類型的其他引數以及 `Promise<?>` 類型的變數引數。如果原始方法已傳回 `void`，則此多載的傳回類型會是 `Promise<Void>`；否則會是原始方法中所宣告的 `Promise<>`。例如：

   原始方法：

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

   產生的方法：

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

   如果將會非同步評估要傳遞給工作流程執行的引數，而且您想要覆寫用以啟動工作流程執行的預設設定，則請使用此多載。在傳遞給此方法多載的所有引數都就緒前，不會對其執行呼叫。

也會產生對應至工作流程界面中每個訊號的方法，例如：

原始方法：

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

產生的方法：

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

非同步用戶端不會包含與原始界面中已標註 @GetState 之方法對應的方法。由於狀態的擷取需要 Web 服務呼叫，因此不適合在工作流程中使用。因此，它只能透過外部用戶端提供。

自主用戶端是要從工作流程內使用，以在目前執行完成時啟動新的執行。此用戶端上的方法類似非同步用戶端上的方法，但傳回 void。此用戶端不會具有與已標註 @Signal 和 @GetState 之方法對應的方法。如需詳細資訊，請參閱「持續的工作流程」。

產生的用戶端分別衍生自基本界面：WorkflowClient 和 WorkflowClientExternal，以提供可用來取消或終止工作流程執行的方法。如需這些界面的詳細資訊，請參閱 適用於 Java 的 AWS SDK 文件。

產生的用戶端可讓您以強類型形式與工作流程執行互動。建立之後，產生之用戶端的執行個體會繫結至特定工作流程執行，且只能用於該執行。此外，框架也會提供非工作流程類型或執行專屬的動態用戶端。產生的用戶端隱含地依賴此用戶端。您也可以直接使用這些用戶端。請參閱「動態用戶端」一節。

框架也會產生用於建立強類型用戶端的 Factory。針對範例 MyWorkflow 界面所產生的用戶端 Factory 如下：

```
//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` 基本界面如下：

```
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);
}
```

您應該使用這些 Factory 來建立用戶端的執行個體。Factory 可讓您設定一般用戶端 (應該使用一般用戶端來提供自訂用戶端實作)、用戶端用來封送處理資料的 `DataConverter`，以及用來啟動「工作流程執行」的選項。如需詳細資訊，請參閱「DataConverters」和「子工作流程執行」小節。`StartWorkflowOptions` 包含您可以用來覆寫註冊時指定預設值的設定，例如逾時。如需 `StartWorkflowOptions`類別的詳細資訊，請參閱 適用於 Java 的 AWS SDK 文件。

外部用戶端可以用來在工作流程範圍外部啟動工作流程執行，非同步用戶端則可用來從工作流程內的程式碼啟動工作流程執行。若要啟動執行，您只需使用產生的用戶端來呼叫與工作流程界面中已標註 `@Execute` 之方法對應的方法。

框架也會產生用戶端界面的實作類別。這些用戶端會建立請求並將其傳送至 Amazon SWF，以執行適當的動作。`@Execute` 方法的用戶端版本會啟動新的工作流程執行，或使用 Amazon SWF APIs 建立子工作流程執行。同樣地， `@Signal`方法的用戶端版本會使用 Amazon SWF APIs來傳送訊號。

> **Note**
>
> 外部工作流程用戶端必須使用 Amazon SWF 用戶端和網域設定。您可以使用將它們做為參數的用戶端原廠建構函數，或傳入已使用 Amazon SWF 用戶端和網域設定的一般用戶端實作。

框架會導覽工作流程界面的類型階層，也會產生父工作流程界面的用戶端界面並從中衍生。

活動用戶端

與工作流程用戶端類似，會針對已標註 @Activities 的每個界面產生用戶端。產生的成品包含用戶端界面和用戶端類別。針對上述範例 @Activities 界面所產生的界面 (MyActivities) 如下：

```
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);
}
```

此界面包含對應至 @Activities 界面中每個活動方法的一組多載方法。這些多載可提供便利性，並可非同步呼叫活動。針對 @Activities 界面中的每個活動方法，在用戶端界面中產生下列方法多載：

1. 依原狀採用原始引數的多載。此多載的傳回類型是 Promise<*T*>，其中 *T* 是原始方法的傳回類型。例如：

   原始方法：

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

   產生的方法：

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

在工作流程的所有引數皆可用且不需要等待時，應該使用此多載。

2. 包含原始引數、ActivitySchedulingOptions 類型之引數及 Promise<?> 類型之其他變數引數的多載。此多載的傳回類型是 Promise<*T*>，其中 *T* 是原始方法的傳回類型。例如：

   原始方法：

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

   產生的方法：

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

   如果工作流程的所有引數皆可用且不需要等待、您想要覆寫預設設定，或想要等待其他 Promise 就緒，則應該使用此多載。變數引數可以用來傳遞其他 Promise<?> 物件，這類其他物件未宣告為引數，但您想要在執行呼叫之前等待。

3. 原始方法中每個引數皆取代為 Promise<> 包裝函式的多載。此多載的傳回類型是 Promise<*T*>，其中 *T* 是原始方法的傳回類型。例如：

   原始方法：

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

   產生的方法：

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

   將會非同步評估要傳遞給活動的引數時，應該使用此多載。在傳遞給此方法多載的所有引數都就緒前，不會對其執行呼叫。

4. 原始方法中每個引數皆取代為 Promise<> 包裝函式的多載。多載也會具有 ActivitySchedulingOptions 類型的其他引數以及 Promise<?> 類型的變數引數。此多載的傳回類型是 Promise<*T*>，其中 *T* 是原始方法的傳回類型。例如：

原始方法：

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

產生的方法：

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

如果將會非同步評估要傳遞給活動的引數、您想要覆寫已註冊之類型的預設設定，或想要等待其他 Promise 就緒，則應該使用此多載。在傳遞給此方法多載的所有引數都就緒前，不會對其執行呼叫。產生的用戶端類別會實作此界面。每個界面方法的實作都會建立並傳送請求給 Amazon SWF，以使用 Amazon SWF APIs 排程適當類型的活動任務。

5. 包含原始引數及 Promise<?> 類型之其他變數引數的多載。此多載的傳回類型是 Promise<*T*>，其中 *T* 是原始方法的傳回類型。例如：

   原始方法：

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

   產生的方法：

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

   如果有工作流程的所有引數可用而且不需要等待，但您想要等待其他 Promise 物件就緒，則應該使用此多載。

6. 原始方法中每個引數皆取代為 Promise 包裝函式且包含 Promise<?> 類型之其他變數引數的多載。此多載的傳回類型是 Promise<*T*>，其中 *T* 是原始方法的傳回類型。例如：

   原始方法：

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

   產生的方法：

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

如果將非同步等待活動的所有引數，而且您也想要等待一些其他 Promise 就緒，則應該使用此多載。所有傳遞的 Promise 物件都就緒時，將會非同步執行此方法多載的呼叫。

產生的活動用戶端也會有對應至所有活動多載所呼叫之每個活動方法 (名為 {*activity method name*}Impl()) 的受保護方法。您可以覆寫此方法來建立模擬用戶端實作。此方法將下列項目採用為引數：Promise<> 包裝函式中原始方法的所有引數、ActivitySchedulingOptions，以及 Promise<?> 類型的變數引數。例如：

原始方法：

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

產生的方法：

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

排程選項

產生的活動用戶端可讓您傳入 ActivitySchedulingOptions 做為引數。ActivitySchedulingOptions 結構包含決定架構在 Amazon SWF 中排程之活動任務組態的設定。這些設定會覆寫指定為註冊選項的預設值。若要動態指定排程選項，請建立 ActivitySchedulingOptions 物件、視需要進行設定，並將之傳遞給活動方法。在下列範例中，我們已指定應該用於活動任務的任務清單。這將會覆寫這次活動呼叫的預設已註冊任務清單。

```
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);
    }
}
```

動態用戶端

除了產生的用戶端之外，框架還提供一般用途的用戶端DynamicWorkflowClient，以及可用來動態啟動工作流程執行、傳送訊號、排程活動等的DynamicActivityClient用戶端。例如，建議您排程在設計階段類型未知的活動。您可以使用 DynamicActivityClient 排程這類活動任務。同樣地，您可以使用 DynamicWorkflowClient 動態排程子工作流程執行。在下列範例中，工作流程會從資料庫尋找活動，並使用動態活動用戶端排程該活動：

```
//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);
}
```

如需詳細資訊，請參閱 適用於 Java 的 AWS SDK 文件。

發出訊號和取消工作流程執行

產生的工作流程用戶端具有對應至可傳送到工作流程之每個訊號的方法。您可以在工作流程內使用它們，以將訊號傳送給其他工作流程執行。這提供用來傳送訊號的類型機制。不過，有時您可能需要動態判斷訊號名稱，例如，在訊息中收到訊號名稱時。您可以使用動態工作流程用戶端，將訊號動態傳送給任何工作流程執行。同樣地，您可以使用用戶端，請求另一個工作流程執行的取消。

在下列範例中，工作流程會從資料庫尋找要傳送訊號過去的執行，並使用動態工作流程用戶端來動態傳送訊號。

```
//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);
}
```

工作流程實作

若要實作工作流程，您可以撰寫可實作所需 @Workflow 界面的類別。例如，可以如下述實作範例工作流程界面 (MyWorkflow)：

```
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);
  }
}
```

此類別中的 @Execute 方法是工作流程邏輯的進入點。由於架構會在處理決策任務時使用重播來重建物件狀態，因此會為每個決策任務建立新的物件。

在 @Workflow 界面的 @Execute 方法中，不允許使用 Promise<*T*> 做為參數。這麼做的原因是進行非同步呼叫純粹是發起人的決策。工作流程實作本身不是取決於呼叫是同步還是非同步。因此，產生的用戶端界面具有採用 Promise<*T*> 參數的多載，以非同步呼叫這些方法。

@Execute 方法的傳回類型只能是 void 或 Promise<*T*>。請注意，對應外部用戶端的傳回類型是 void，而不是 Promise<>。由於外部用戶端並非用於非同步程式碼，因此外部用戶端不會傳回Promise物件。若要取得外部陳述的工作流程執行結果，您可以設計工作流程，透過 活動更新外部資料存放區中的狀態。Amazon SWF 的可見APIs 也可以用於擷取工作流程結果以供診斷之用。不建議您使用可見性 APIs擷取工作流程執行的結果作為一般實務，因為 Amazon SWF 可能會調節這些 API 呼叫。可見度 API 需要您使用 WorkflowExecution 結構來識別工作流程執行。您可以呼叫 getWorkflowExecution 方法，以從產生的工作流程用戶端取得此結構。此方法會傳回 WorkflowExecution 結構，其對應至用戶端所繫結的工作流程執行。如需可見性 API 的詳細資訊，請參閱 Amazon Simple Workflow Service API 參考。 APIs

從工作流程實作呼叫活動時，您應該使用產生的活動用戶端。同樣地，若要傳送訊號，請使用產生的工作流程用戶端。

決策內容

只要框架執行工作流程程式碼，框架就會提供環境內容。此內容提供可在工作流程實作 (例如建立計時器) 中存取的內容專屬功能。如需詳細資訊，請參閱「執行內容」小節。

公開執行狀態

Amazon SWF 可讓您在工作流程歷史記錄中新增自訂狀態。工作流程執行報告的最新狀態會透過對 Amazon SWF 服務和 Amazon SWF 主控台的可見性呼叫傳回給您。例如，在訂單處理工作流程中，您可以報告不同階段的訂單狀態，例如「收到訂單」、「送出訂單」等等。在 AWS Flow Framework 適用於 Java 的 中，這會透過工作流程界面上標註 `@GetState`註釋的方法來完成。決策者完成決策任務的處理時，就會呼叫此方法以從工作流程實作取得最新狀態。除了可見度呼叫之外，也可以使用產生的外部用戶端 (在內部使用可見度 API 呼叫) 來擷取狀態。

下列範例示範如何設定執行內容。

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

```
}
```

隨時可以使用產生的外部用戶端來擷取工作流程執行的最新狀態。

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

在上述範例中，會報告各種階段的執行狀態。工作流程執行個體啟動時，`periodicWorkflow` 會將初始狀態報告為「剛啟動」('Just Started')。每個 `callPeriodicActivity` 呼叫接著都會更新工作流程狀態。呼叫 `activity1` 100 次之後，會傳回方法，並完成工作流程執行個體。

工作流程區域變數

您有時可能需要在工作流程實作中使用靜態變數。例如，您可能想要在工作流程實作中，存放能從多種位置 (類別可能不同) 存取的計數器。不過，您無法依賴工作流程中的靜態變數，原因是靜態變數是在多個執行緒之間共享，而這會發生問題，因為工作者可能會同時處理不同執行緒上的不同決策任務。或者，您可以將這類狀態存放在工作流程實作的欄位中，但您接著需要傳遞實作物件。為了解決此需求，框架會提供 `WorkflowExecutionLocal<?>` 類別。任何需要具有靜態變數 (例如語意) 的狀態都應該使用 `WorkflowExecutionLocal<?>` 保持為執行個體區域變數。您可以宣告和使用這類型的靜態變數。例如，在下列程式碼片段中，`WorkflowExecutionLocal<String>` 會用以存放使用者名稱。

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

活動實作

以提供 `@Activities` 界面的實作來實作活動。 AWS Flow Framework 適用於 Java 的 使用工作者上設定的活動實作執行個體，在執行時間處理活動任務。工作者會自動尋找合適類型的活動實作。

您可以使用屬性和欄位將資源傳遞給活動執行個體，例如資料庫連線。由於活動實作物件可從多個執行緒存取，共用資源必須安全執行緒。

請注意，活動實作不接受 `Promise<>` 類型的參數或傳回此類型的物件。這是因為活動實作不應該取決於呼叫方式 (同步還是非同步)。

以前顯示的活動界面可以實作如下：

```
public class MyActivitiesImpl implements MyActivities {

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

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

執行緒本機內容可供活動實作使用，用以擷取任務物件、使用中的資料轉換器物件等等。目前的內容可透過 `ActivityExecutionContextProvider.getActivityExecutionContext()` 存取。如需詳細資訊，請參閱 文件 適用於 Java 的 AWS SDK `ActivityExecutionContext`和 一節執行內容。

手動完成活動

上例中的 `@ManualActivityCompletion` 註釋是選擇性的註釋。它只能出現在實作活動的方法中，且用以設定當活動方法傳回時不會自動完成的活動。當您想要以非同步方式完成活動時，這可能很有用，例如，在人工動作完成後手動完成。

當您的活動方法傳回時，框架預設視活動為完成。這表示活動工作者向 Amazon SWF 報告活動任務的完成情況，並提供結果（如果有的話）。但是，當活動方法傳回時，有些使用案例您不希望將活動任務標記為完成。當您建立人力任務模型時特別實用。例如，活動方法可能要在活動任務完成前，先向必須完成某些工作的某人傳送電子郵件。在這種情況下，您可以使用 `@ManualActivityCompletion` 註釋來註釋活動方法，通知活動工作者它不應該自動完成活動。若要手動完成活動，您可以使用架構中`ManualActivityCompletionClient`提供的 ，或在 Amazon SWF SDK 中提供的 Amazon SWF Java 用戶端上使用 `RespondActivityTaskCompleted`方法。如需詳細資訊，請參閱 適用於 Java 的 AWS SDK 文件。

為完成活動任務，您需要提供任務字符。Amazon SWF 會使用任務字符來唯一識別任務。您可在您的活動實作中從 `ActivityExecutionContext` 存取此字符。您必須將此字符傳遞到負責完成任務的一方。您可從 `ActivityExecutionContext` 呼叫 `ActivityExecutionContextProvider.getActivityExecutionContext().getTaskToken()` 來擷取此字符。

您可實作 Hello World 範例的 `getName` 活動，傳送電子郵件要求某人提供歡迎訊息：

```
@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";
}
```

您也可以使用 ManualActivityCompletionClient，用以下的程式碼片段提供歡迎語並結束任務。或者，您也可以讓任務失敗：

```
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);
    }
}
```

實作 AWS Lambda 任務

主題

- 關於 AWS Lambda
- 使用 Lambda 任務的優點和限制
- 在適用於 Java AWS Flow Framework 的 工作流程中使用 Lambda 任務
- 檢視 HelloLambda 範例

關於 AWS Lambda

AWS Lambda 是一項全受管運算服務，可執行您的程式碼以回應自訂程式碼所產生的事件，或從 Amazon S3、DynamoDB、Amazon Kinesis、Amazon SNS 和 Amazon Cognito 等各種 AWS 服務所產生的事件。如需有關 Lambda 的詳細資訊，請參閱 AWS Lambda 開發人員指南。

Amazon Simple Workflow Service 提供 Lambda 任務，讓您可以執行 Lambda 函數來取代或搭配傳統 Amazon SWF 活動。

> **Important**
>
> AWS 您的帳戶將針對 Amazon SWF 代表您執行的 Lambda 執行（請求）收取費用。如需 Lambda 定價的詳細資訊，請參閱 https：//https://aws.amazon.com/lambda/pricing/。

使用 Lambda 任務的優點和限制

使用 Lambda 任務取代傳統的 Amazon SWF 活動有許多好處：

- Lambda 任務不需要像 Amazon SWF 活動類型一樣進行註冊或版本控制。
- 您可以使用已在工作流程中定義的任何現有 Lambda 函數。
- Lambda 函數由 Amazon SWF 直接呼叫；您不需要實作工作者程式來執行它們，就像傳統活動一樣。
- Lambda 為您提供指標和日誌，用於追蹤和分析函數執行。

您還需要知道數個 Lambda 任務的相關限制：

- Lambda 任務只能在支援 Lambda AWS 的區域執行。如需 Lambda 目前支援區域的詳細資訊，請參閱 Amazon Web Services 一般參考中的 Lambda 區域和端點。
- Lambda 任務目前僅受基礎 SWF HTTP API 和適用於 Java AWS Flow Framework 的 支援。目前不支援 AWS Flow Framework 適用於 Ruby 的 中的 Lambda 任務。

在適用於 Java AWS Flow Framework 的 工作流程中使用 Lambda 任務

在適用於 Java AWS Flow Framework 的 工作流程中使用 Lambda 任務有三個要求：

- 要執行的 Lambda 函數。您可以使用您定義的任何 Lambda 函數。如需如何建立 Lambda 函數的詳細資訊，請參閱 AWS Lambda 開發人員指南。
- IAM 角色，可讓您從 Amazon SWF 工作流程執行 Lambda 函數。
- 在工作流程中排程 Lambda 任務的程式碼。

設定 IAM 角色

您必須先提供 IAM 角色，以便從 Amazon SWF 存取 Lambda，才能叫用 Lambda 函數 Amazon SWF 。您可擇一方法：

- 選擇預先定義的角色 AWSLambdaRole，以授予您的工作流程許可，以叫用與您的帳戶相關聯的任何 Lambda 函數。
- 定義您自己的政策和相關聯的角色，以授予工作流程呼叫特定 Lambda 函數的許可，這些函數由其 Amazon Resource Name (ARNs) 指定。

限制 IAM 角色的許可

您可以使用 資源信任政策中的 `SourceArn`和 `SourceAccount`內容索引鍵，限制您提供給 Amazon SWF 的 IAM 角色許可。這些金鑰會限制 IAM 政策的用量，因此只會從屬於指定網域 ARN 的 Amazon Simple Workflow Service 執行中使用。如果您同時使用兩個全域條件內容索引鍵，則值中參考`aws:SourceAccount`的值和帳戶在相同政策陳述式中使用時，`aws:SourceArn`必須使用相同的帳戶 ID。

在下列信任政策範例中，我們使用`SourceArn`內容索引鍵來限制 IAM 服務角色僅用於屬於`someDomain`帳戶的 Amazon Simple Workflow Service 執行。 `123456789012`

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

```
        }
      }
    }
  ]
}
```

在下列信任政策範例中，我們使用`SourceAccount`內容索引鍵來限制 IAM 服務角色只能用於 帳戶中的 Amazon Simple Workflow Service 執行。 `123456789012`

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

提供 Amazon SWF 調用任何 Lambda 角色的存取權

您可以使用預先定義的角色 AWSLambdaRole，讓 Amazon SWF 工作流程能夠叫用與您的帳戶相關聯的任何 Lambda 函數。

使用 AWSLambdaRole 讓 Amazon SWF 能夠叫用 Lambda 函數

1. 開啟 Amazon IAM 主控台。
2. 選擇 Roles (角色)，然後選擇 Create New Role (建立新角色)。
3. 為您的角色提供名稱，例如 `swf-lambda`，然後選擇 Next Step (下一步)。
4. 在AWS 服務角色下，選擇 Amazon SWF，然後選擇下一步。
5. 在 Attach Policy (附加政策) 畫面上，從清單選擇 AWSLambdaRole。
6. 當您檢閱好角色後，請選擇 Next Step (下一步)，然後選擇 Create Role (建立角色)。

定義 IAM 角色以提供呼叫特定 Lambda 函數的存取權

如果您想要提供從工作流程叫用特定 Lambda 函數的存取權，則需要定義自己的 IAM 政策。

建立 IAM 政策以提供特定 Lambda 函數的存取權

1. 開啟 Amazon IAM 主控台。
2. 選擇 Policies (政策)，然後選擇 Create Policy (建立政策)。
3. 選擇複製 AWS 受管政策，然後從清單中選擇 AWSLambdaRole。即會為您產生政策。您可選擇性編輯其名稱及描述，以符合您的需求。
4. 在政策文件的資源欄位中，新增 Lambda 函數的 ARN (Lambda)。例如：

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

> **Note**
>
> 如需如何在 IAM 角色中指定資源的完整說明，請參閱 使用 IAM 中的 IAM 政策概觀。

5. 選擇 Create Policy (建立政策) 來完成您的政策建立程序。

然後，您可以在建立新的 IAM 角色時選取此政策，並使用該角色來授予 Amazon SWF 工作流程的存取。此程序與使用 AWSLambdaRole 政策建立角色非常類似。差別在於您要在建立角色時選擇您自己的政策。

使用 Lambda 政策建立 Amazon SWF 角色

1. 開啟 Amazon IAM 主控台。

2. 選擇 Roles (角色)，然後選擇 Create New Role (建立新角色)。
3. 為您的角色提供名稱，例如 `swf-lambda-function`，然後選擇 Next Step (下一步)。
4. 在AWS 服務角色下，選擇 Amazon SWF，然後選擇下一步。
5. 在連接政策畫面上，從清單中選擇您的 Lambda 函數特定政策。
6. 當您檢閱好角色後，請選擇 Next Step (下一步)，然後選擇 Create Role (建立角色)。

排程執行 Lambda 任務

定義可讓您叫用 Lambda 函數的 IAM 角色後，您可以將它們排程為工作流程的一部分來執行。

> **Note**
>
> HelloLambda 範例會在 適用於 Java 的 AWS SDK中完整示範此程序。

排程執行 Lambda 任務

1. 在您的工作流程實作中，於 `DecisionContext` 執行個體上呼叫 `getLambdaFunctionClient()` 以取得 `LambdaFunctionClient` 執行個體。

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

2. 使用 上的 `scheduleLambdaFunction()`方法排程任務`LambdaFunctionClient`，並傳遞您建立的 Lambda 函數名稱和 Lambda 任務的任何輸入資料。

```
// 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. 在工作流程執行入門中，使用 將 IAM lambda 角色新增至預設工作流程選項`StartWorkflowOptions.withLambdaRole()`，然後在啟動工作流程時傳遞選項。

```
// 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);
```

檢視 HelloLambda 範例

提供實作使用 Lambda 任務之工作流程的範例，會在 中提供 適用於 Java 的 AWS SDK。若要加以檢視及/或執行，請下載來源。

有關如何建置和執行 HelloLambda 範例的完整說明，請參閱隨 AWS Flow Framework 適用於 Java 的提供的 README 檔案。

執行使用 AWS Flow Framework 適用於 Java 的 編寫的程式

主題

- WorkflowWorker
- ActivityWorker
- 工作者執行緒模型
- 工作者可擴充性

此架構提供工作者類別來初始化 AWS Flow Framework 適用於 Java 的 執行時間，並與 Amazon SWF 通訊。為實作工作流程或活動工作者，您必須建立及啟動工作者類別的執行個體。這些工作者類別負責

管理持續的非同步操作、調用解除封鎖的非同步方法，以及與 Amazon SWF 通訊。它們可使用工作流程和活動實作、執行緒數目、要輪詢的任務清單等設定而成。

框架提供兩種工作者類別，一個用於活動，一個用於工作流程。為執行工作流程邏輯，您要使用 `WorkflowWorker` 類別。同樣地，在活動方面則使用 `ActivityWorker` 類別。這些類別會自動輪詢 Amazon SWF 以取得活動任務，並在您的實作中調用適當的方法。

下列範例說明如何執行個體化 `WorkflowWorker` 並開始輪詢任務：

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

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

建立 `ActivityWorker` 之執行個體和開始輪詢任務的基本步驟如下：

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

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

當您想要關閉活動或決策者時，您的應用程式應該關閉正在使用的工作者類別執行個體以及 Amazon SWF Java 用戶端執行個體。這會確保工作者類型使用的所有資源都正確釋出。

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

為啟動執行，請只建立產生之外部用戶端的執行個體，並呼叫 `@Execute` 方法。

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

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

WorkflowWorker

如名稱所示，此工作者類別目的在供工作流程實作使用。它以任務清單和工作流程實作類型設定而成。工作者類型會執行迴圈來輪詢指定任務清單中的決策任務。收到決策任務時，它會建立工作流程實作的執行個體，並呼叫 `@Execute` 方法處理任務。

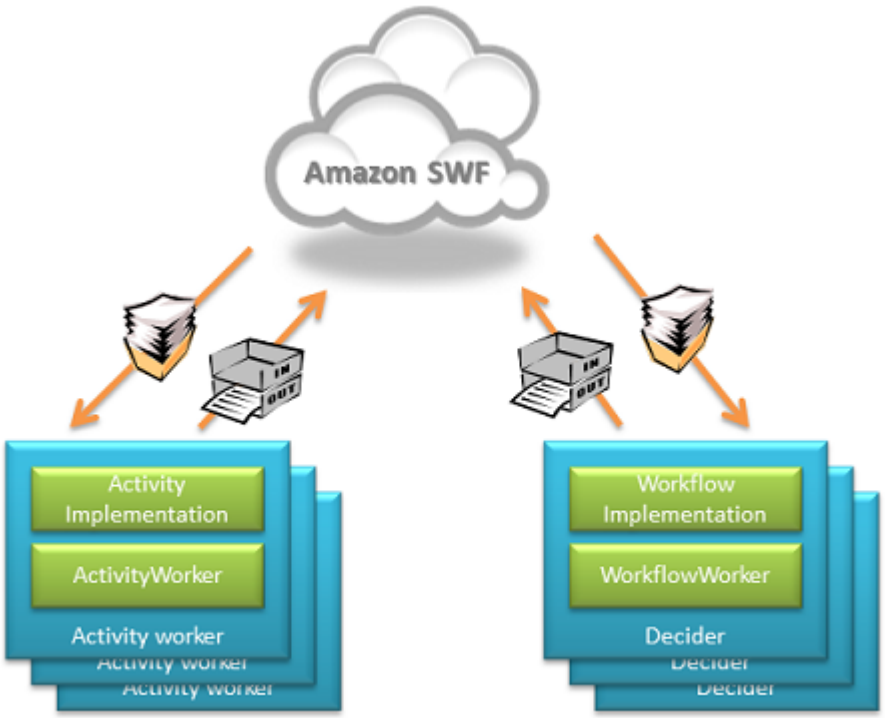
ActivityWorker

若要實作活動工作者，您可使用 `ActivityWorker` 類別以便向任務清單輪詢活動任務。您可使用活動實作物件設定活動工作者。此工作者類型會執行迴圈來輪詢指定任務清單中的活動任務。收到活動任務時，它會尋找您提供的適合實作，並呼叫活動方法處理任務。與呼叫 factory 為每項決策任務建立新執行個體的 `WorkflowWorker` 不同，`ActivityWorker` 只使用您提供的物件。

`ActivityWorker` 類別使用 AWS Flow Framework 適用於 Java 註釋的 來判斷註冊和執行選項。

工作者執行緒模型

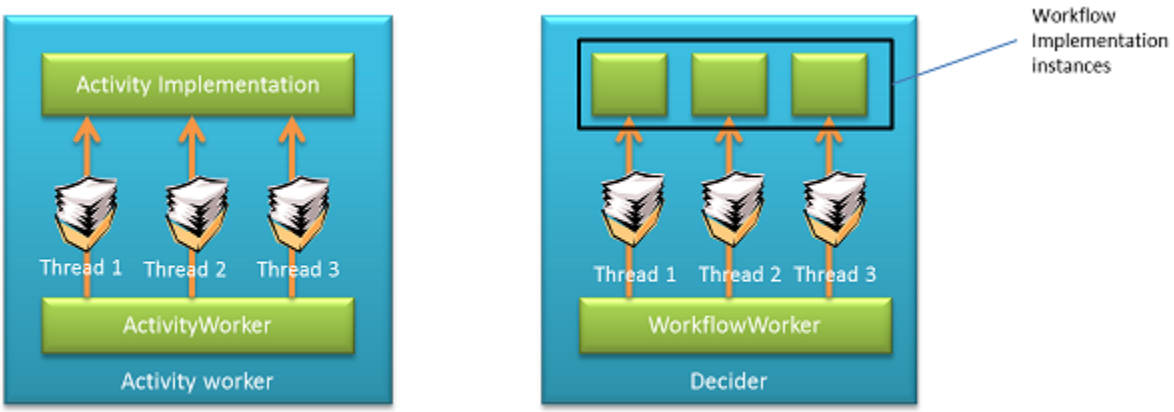
在 AWS Flow Framework 適用於 Java 的 中，活動或決策者是工作者類別的執行個體。您的應用程式負責在應做為工作者的每部電腦和每項程序上設定和執行個體化工作者物件。然後，工作者物件會自動從 Amazon SWF 接收任務，將其分派到您的活動或工作流程實作，並向 Amazon SWF 報告結果。單一工作流程執行個體可能跨多個工作者。當 Amazon SWF 有一或多個擱置中的活動任務時，它會將任務指派給第一個可用的工作者，然後下一個工作者，以此類推。這讓屬於同一個工作流程執行個體的任務可能同時在不同的工作者上處理。



此外，每個工作者都可設定成在多個執行緒上處理任務。這表示即使只有一個工作者，工作流程執行個體的活動任務也可以同時執行。

決策任務的行為類似，但 Amazon SWF 保證對於指定的工作流程執行，一次只能執行一個決策。單一工作流程執行一般需要多項決策任務，因此結果可能也會在多個程序和執行緒上執行。決策者以該類型的工作流程實作設定而成。當決策者收到決策任務時，它會建立工作流程實作的執行個體 (物件)。框架提供可擴充的 factory 模式來建立這些執行個體。預設工作流程 factory 每次都會建立新的物件。您可提供自訂的 factory 來覆寫此行為。

與以工作流程實作類型設定的決策者相反，活動工作者以活動實作的執行個體 (物件) 設定而成。當活動工作者收到活動任務時，它會分派給合適的活動實作物件。



工作流程工作者會維護單一執行緒集區，並在用來輪詢任務 Amazon SWF 的相同執行緒上執行工作流程。由於活動長時間執行（至少與工作流程邏輯相比），活動工作者類別會維護兩個單獨的執行緒集區；一個用於輪詢 Amazon SWF 以進行活動任務，另一個則用於執行活動實作來處理任務。這可讓您分別設定要輪詢任務的執行緒數目，及要執行該任務的執行緒數目。例如，您可使用少數執行緒來輪詢，並用多數執行緒來執行任務。活動工作者類別只會在具有可用輪詢執行緒和處理任務的可用執行緒時輪詢任務的 Amazon SWF。

這種執行緒和執行個體行為表示：

1. 活動實作必須無狀態。您不應使用執行個體變數將應用程式狀態存放在活動物件中。但是，您可以使用欄位存放資源，例如資料庫連線。
2. 活動實作必須是安全執行緒。由於相同的執行個體可以同時處理來自不同執行緒的任務，因此必須同步從活動程式碼存取共用資源。
3. 工作流程實作可以具有狀態，而執行個體變數可用來存放狀態。即使建立了新的工作流程實作執行個體來處理每項決策任務，框架仍會確保正確地重新建立該狀態。但是，工作流程實作必須具有確定性。如需詳細資訊，請參閱「了解 AWS Flow Framework 適用於 Java 的 中的任務」一節。
4. 在使用預設 factory 時，工作流程實作不需安全執行緒。預設實作會確保一次只有一個執行緒使用工作流程實作執行個體。

工作者可擴充性

AWS Flow Framework 適用於 Java 的 也包含幾個低階工作者類別，可為您提供精細的控制和可擴展性。使用它們，您可完全自訂工作流程和活動類型註冊，並設定 factory 來建立實作物件。這些工作者為 `GenericWorkflowWorker` 和 `GenericActivityWorker`。

`GenericWorkflowWorker` 可使用 factory 設定以建立工作流程定義 factory。工作流程定義 factory 負責建立工作流程實作的執行個體，及提供註冊選項這類組態設定。在一般的情況下，您應該直接使用 `WorkflowWorker` 類別。它會自動建立並設定框架中提供的 factory 實作，即 `POJOWorkflowDefinitionFactoryFactory` 和 `POJOWorkflowDefinitionFactory`。factory 要求工作流程實作類別必須具有無引數的建構函數。此建構函數用於在執行時間建立工作流程物件的執行個體。factory 會查看您在工作流程界面和實作上使用的註釋，以建立適合的註冊和執行選項。

您可透過實作 `WorkflowDefinitionFactory`、`WorkflowDefinitionFactoryFactory` 和 `WorkflowDefinition`，提供自己的 factory 實作。工作者類別使用 `WorkflowDefinition` 類別來分派決策任務和訊號。透過實作這些基礎類別，您可完全自訂 factory 和工作流程實作請求的分派。例如，您可使用這些可擴充性點提供自訂的程式設計模型撰寫工作流程，例如，根據您自己的註釋來撰寫或從 WSDL 產生它，而不是框架使用的第一個程式碼方法。為使用您自訂的 factory，您必須使用

GenericWorkflowWorker 類別。如需這些類別的詳細資訊，請參閱 適用於 Java 的 AWS SDK 文件。

同樣地，GenericActivityWorker 可讓您提供自訂活動實作 factory。透過實作 ActivityImplementationFactory 和 ActivityImplementation 類別，您可完全控制活動執行個體化以及自訂註冊和執行選項。如需這些類別的詳細資訊，請參閱 適用於 Java 的 AWS SDK 文件。

執行內容

主題

- 決策內容
- 活動執行內容

此框架提供工作流程和活動實作的環境內容。此內容專門用於處理中的任務，並提供可在實作中使用的一些公用程式。每次工作者處理新任務時，都會建立內容物件。

決策內容

執行決策任務時，框架會透過 DecisionContext 類別提供工作流程實作的內容。DecisionContext 提供工作流程執行之執行 ID，以及時鐘與計時器功能等這類上下文相關資訊。.

存取工作流程實作中的 DecisionContext

您可以使用 DecisionContextProviderImpl 類別存取工作流程實作中的 DecisionContext。或者，您可以使用 Spring 將內容注入工作流程實作的欄位或屬性中，如「可試性與相依性置入」一節中所示。

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

建立時鐘和計時器

DecisionContext 包含類型 WorkflowClock 的屬性，以提供計時器和時鐘功能。由於工作流程邏輯需要確定性，因此您不應在工作流程實作中直接使用系統時鐘。WorkflowClock 上的 currentTimeMills 方法會傳回所處理決策的啟動事件的時間。這確保您在重新執行期間收到相同的時間值，因此得以將工作流程邏輯設為確定性。

WorkflowClock 也具有 createTimer 方法可傳回在指定間隔後就緒的 Promise 物件。您可以使用此值做為其他非同步方法的參數，延遲其在指定期間後執行。因此，您可以有效排程非同步方法或活動，以在稍後執行。

下列清單中的範例示範如何定期呼叫活動。

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

在上述清單中，`callPeriodicActivity` 非同步方法會呼叫 `activity1`，然後使用目前的 `AsyncDecisionContext` 來建立計時器。它會將傳回的 `Promise` 傳遞為遞迴呼叫自己的引數。此遞迴呼叫會等到計時器引發 (在此範例中是 1 小時) 再執行。

活動執行內容

如同 `DecisionContext` 在處理決策任務時提供內容資訊，`ActivityExecutionContext` 會在處理活動任務時提供類似的內容資訊。活動程式碼可以透過 `ActivityExecutionContextProviderImpl` 類別使用此內容。

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

使用 `ActivityExecutionContext`，您可以執行下列作業：

發出長時間執行活動的活動訊號

如果活動長時間執行，則必須定期向 Amazon SWF 報告其進度，讓它知道任務仍在進行。在缺乏這類活動訊號的情形下，如果任務活動訊號已於註冊活動類型或排程活動時設定逾時，則任務可能會逾時。若要傳送活動訊號，您可以對 `ActivityExecutionContext` 使用 `recordActivityHeartbeat` 方法。活動訊號也會提供取消進行中之活動的機制。如需詳細資訊和範例，請參閱「錯誤處理」一節。

取得活動任務的詳細資訊

如果需要，您可以取得執行器取得任務時，Amazon SWF 傳遞之活動任務的所有詳細資訊。這包含任務、任務類型、任務字符等之輸入的相關資訊。如果您想要實作手動完成的活動，例如人為動作，則必須使用 `ActivityExecutionContext`擷取任務字符並將其傳遞至最終完成活動任務的程序。如需詳細資訊，請參閱「手動完成活動」一節。

取得執行器正在使用的 Amazon SWF 用戶端物件

執行器正在使用的 Amazon SWF 用戶端物件可以透過在 上呼叫 `getService`方法來擷取`ActivityExecutionContext`。如果您想要直接呼叫 Amazon SWF 服務，這會很有用。

子工作流程執行

我們在目前提出的範例中，已直接從應用程式啟動工作流程執行。但是，工作流程執行可透過在已產生的用戶端上呼叫工作流程進入點方法，從工作流程內啟動。當工作流程執行從另一個工作流程執行內容中啟動時，它稱之為子工作流程執行。這可以讓您將複雜的工作流程重構成較小的單位，並有可能跨不同工作流程共享它們。例如，您可建立處理付款的工作流程，並從處理訂單的工作流程呼叫它。

在語意上，子工作流程執行和獨立工作流程除下列差異外，二者的運作是一樣的：

1. 當父工作流程因使用者明確動作而終止時，例如，透過呼叫 `TerminateWorkflowExecution` Amazon SWF API，或因逾時而終止時，子工作流程執行的命運將由子政策決定。您可以設定此子政策終止、取消或捨棄 (繼續執行) 子工作流程執行。
2. 子工作流程的輸出 (傳回進入點方法的值) 可由父工作流程執行使用，如同 `Promise<T>` 由同步方法傳回。這與獨立執行不同，其中應用程式必須使用 Amazon SWF APIs 取得輸出。

在下列範例中，`OrderProcessor` 工作流程會建立 `PaymentProcessor` 子工作流程：

```
@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);

}
```

持續的工作流程

在某些使用案例中，您可能會需要一直執行或長期執行的工作流程，例如，監控伺服器群運作狀態的工作流程。

> **Note**
>
> 由於 Amazon SWF 會保留工作流程執行的整個歷史記錄，因此歷史記錄會隨著時間持續成長。當框架執行重新執行時，會從 Amazon SWF 擷取此歷史記錄，如果歷史記錄大小太大，就會變得很昂貴。在這種長期執行或持續的工作流程中，您應該定期關閉目前的執行並開始新的執行，以繼續處理。

這是邏輯性持續工作流程執行。產生的自主用戶端可用於此目的。在您的工作流程實作中，只要在自主用戶端上呼叫 `@Execute` 方法即可。一旦完成目前的執行，框架即會使用相同的工作流程 ID 開始新的執行。

您也可以在可從目前的 `DecisionContext` 擷取的 `GenericWorkflowClient` 上呼叫 `continueAsNewOnCompletion` 方法來繼續執行。例如，以下工作流程實作會設定計時器在一天後觸發，呼叫它自己的進入點開始新的執行。

```
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();
    }
}
```

當工作流程遞迴呼叫自己時，框架會在所有等待中的任務已完成並開始新的工作流程執行時，關閉目前的工作流程。請注意，只要有等待中的任務，就不會關閉目前的工作流程執行。新的執行不會自動繼承原始執行的任何歷史記錄或資料，如果您想要將一些狀態傳遞給新的執行，您必須將它明確傳送為輸入。

在 Amazon SWF 中設定任務優先順序

根據預設，任務清單上的任務根據它們的「到達時間」來交付：先排程的任務一般盡可能先執行。透過設定選用任務優先順序，您可以優先處理某些任務：Amazon SWF 會嘗試在任務清單上交付優先順序較高的任務，然後再處理優先順序較低的任務。

您可以為工作流程和活動都設定任務優先順序。工作流程的任務優先順序並不影響其排程的任何活動任務優先順序，也不影響其啟動的任何子工作流程。活動或工作流程的預設優先順序是在註冊期間設定（由您或 Amazon SWF 設定），除非在排程活動或啟動工作流程執行時覆寫，否則一律會使用已註冊的任務優先順序。

任務優先順序值的範圍介於 "-2147483648" 到 "2147483647"，數字愈大表示優先順序愈高。如果您未設定活動或工作流程的任務優先順序，則會為其指派優先順序零 ("0")。

主題

- 為工作流程設定任務優先順序
- 為活動設定任務優先順序

為工作流程設定任務優先順序

當您註冊或啟動工作流程時，您可為它設定任務優先順序。除非在啟動工作流程執行時被覆寫，否則在註冊工作流程類型時設定的任務優先順序，會用為該類型任何工作流程執行的預設值。

若要以預設任務優先順序註冊工作流程類型，請在宣告工作流程註冊選項時，在 WorkflowRegistrationOptions 中設定 defaultTaskPriority 選項：

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

您也可以在啟動工作流程時，為工作流程設定 taskPriority，來覆寫已註冊的 (預設) 任務優先順序。

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

PriorityWorkflowClientExternalFactory cf
    = new PriorityWorkflowClientExternalFactoryImpl(swfService, domain);

priority_workflow_client = cf.getClient();

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

或者，您可在啟動子工作流程或將工作流程繼續做為新的工作流程使用時，設定任務優先順序。例如，您可以在 ContinueAsNewWorkflowExecutionParameters 或 StartChildWorkflowExecutionParameters 中設定 taskPriority 選項。

為活動設定任務優先順序

您可在註冊或排程活動時，為它設定任務優先順序。除非在排程活動時被覆寫，否則在註冊活動類型時設定的任務優先順序，會用為活動執行的預設值。

若要以預設任務優先順序註冊活動類型，請在宣告活動註冊選項時，在 ActivityRegistrationOptions 中設定 defaultTaskPriority 選項：

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

您也可以在排程活動時，為活動設定 taskPriority，來覆寫已註冊的 (預設) 任務優先順序。

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

ImportantActivitiesClient activityClient = new ImportantActivitiesClientImpl();

activityClient.doSomethingImportant(activityOptions);
```

DataConverters

當您的工作流程實作呼叫遠端活動時，傳遞給它的輸入和活動的執行結果都必須序列化，才可透過線路傳送它們。為達到此目的，框架會使用 DataConverter 類別。這是您可實作以提供您個人序列化程式的抽象類別。架構中提供預設的 Jackson 序列化器型實作 `JsonDataConverter`。如需詳細資訊，請參閱 適用於 Java 的 AWS SDK 文件 。如需 Jackson 如何執行序列化以及可用來影響它的 Jackson 註釋之詳細資訊，請參閱 Jackson JSON Processor 文件。使用的線路格式視為合約的一部分。因此，您可以設定 `@Activities` 和 `@Workflow` 註釋的 `DataConverter` 屬性，以在您的活動和工作流程界面上指定 `DataConverter`。

框架會建立您在 `@Activities` 註釋中指定之 `DataConverter` 類型的物件，以序列化活動的輸入和還原序列化其結果。同樣地，您在 `@Workflow` 註釋中指定之 `DataConverter` 類型的物件，會用來序列化您傳遞至工作流程的參數，並在子工作流程案例中還原序列化結果。除了輸入之外，架構也會將其他資料傳遞至 Amazon SWF，例如例外狀況詳細資訊，工作流程序列化工具也會用於序列化此資料。

如果您不希望框架自動建立 `DataConverter` 的執行個體，您也可以自行提供。產生的用戶端具有包含 `DataConverter` 的建構函數多載。

如果您未指定 `DataConverter` 類型，也未傳遞 `DataConverter` 物件，根據預設會使用 `JsonDataConverter`。

將資料傳遞給非同步方法

主題

- 將集合和對應傳遞給非同步方法
- Settable<T>
- @NoWait
- Promise<Void>
- AndPromise 和 OrPromise

先前各節已解釋如何使用 Promise<*T*>。這裡將討論一些 Promise<*T*> 的進階使用案例。

將集合和對應傳遞給非同步方法

框架支援將陣列、集合和對應以 Promise 類型傳遞給非同步方法。例如，非同步方法可能採用 Promise<ArrayList<String>> 做為引數，如下列清單所示。

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

在語義上，其運作與任何其他 Promise 類型參數一樣，且非同步方法會等到集合變成可用後再執行。如果集合成員是 Promise 物件，則您可以讓框架等待所有成員就緒，如下列程式碼片段所示。這樣會讓非同步方法等待每個集合成員變成可用。

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

請注意，@Wait 註釋必須用於此參數，代表其包含 Promise 物件。

也請注意，活動 printActivity 採用 String 引數，但產生之用戶端中的相符方法採用 Promise<String>。我們會在用戶端上呼叫此方法，而不是直接呼叫活動方法。

Settable<T>

Settable<*T*> 是 Promise<T> 的衍生類型，提供 set 方法讓您手動設定 Promise 的值。例如，下列工作流程透過等待 Settable<?> 來等待收到訊號，此於 signal 方法中設定：

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

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

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

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

Settable<?> 一次也可以鏈結到另一個 Promise。您可以使用 AndPromise 和 OrPromise，將 Promise 分組。您可以在已鏈結的 Settable 上呼叫 unchain() 方法，以將其取消鏈結。鏈結時，Settable<?> 會在其鏈結的 Promise 就緒時自動變成就緒。當您想要在程式的其他部分中使用從 doTry() 範圍內傳回的 Promise 時，鏈結會特別有用。由於 TryCatchFinally 用作巢狀類別，因此您無法Promise<>在父系範圍內宣告 ，並在 中設定它doTry()。原因是 Java 需要在父範圍中宣告變數，並用於要標示為最終的巢狀類別。例如：

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

    new TryFinally() {

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

```
            activity2(resultToChain);

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

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

    return result;
}
```

Settable 一次可以鏈結到一個 Promise。您可以在已鏈結的 Settable 上呼叫 unchain() 方法，以將其取消鏈結。

@NoWait

當您將 Promise 傳遞給非同步方法時，框架預設會等待 Promise 就緒，再執行方法 (集合類型除外)。您可以在非同步方法宣告的參數上使用 @NoWait 註釋，來覆寫此行為。如果您要傳入非同步方法將自行設定的 Settable<T>，這會十分有用。

Promise<Void>

非同步方法中相依性的實作方式是將某個方法所傳回的 Promise 以引數傳遞給另一個方法。不過，您可能會想要從方法傳回 void，但仍想要其他非同步方法於其完成後再執行。在這種情況下，您可以使用 Promise<Void> 做為方法的傳回類型。Promise 類別提供靜態 Void 方法，能讓您用以建立 Promise<Void> 物件。非同步方法完成執行時，此 Promise 會就緒。您可以將此 Promise 傳遞給另一個非同步方法，如同其他 Promise 物件。如果您使用 Settable<Void>，則請在其上以 null 呼叫 set 方法，使之就緒。

AndPromise 和 OrPromise

AndPromise 和 OrPromise 可讓您將多個 Promise<> 物件分組為單一邏輯 Promise。AndPromise 會在用來建構它的所有 Promise 都就緒時就緒。OrPromise 會在用來建構它的 Promise 集合中的任何 Promise 都就緒時就緒。您可以對 AndPromise 和 OrPromise 呼叫 getValues()，以擷取組成 Promise 之值的清單。

可試性與相依性插入

主題

- Spring 整合
- JUnit 整合

框架設計旨在容易控制反轉 (IoC)。活動與工作流程實作以及框架提供的工作者和內容物件，都可以使用如 Spring 的容器予以設定和執行個體化。框架可立即提供與 Spring Framework 的整合。此外，也已提供與 JUnit 的整合，進行工作流程和活動實作的單元測試。

Spring 整合

com.amazonaws.services.simpleworkflow.flow.spring 套裝服務包含的類別，方便您在應用程式中使用 Spring 框架。這些包括自訂的 Scope 和 Spring 感知活動及工作流程工作者：`WorkflowScope`、`SpringWorkflowWorker` 和 `SpringActivityWorker`。這些類別讓您完全透過 Spring 設定您的工作流程和活動實作以及工作者。

WorkflowScope

`WorkflowScope` 是框架提供的自訂 Spring Scope 實作。此範圍讓您在生命週期不超過決策任務生命週期的 Spring 容器中建立物件。每次工作者收到決策任務時，在此範圍內的 Bean 都會執行個體化。您應該為工作流程實作 Bean 及其相依的任何其他 Bean 使用此範圍。Spring 提供的單一和原型範圍不應用於工作流程實作 Bean，因為框架需要為每項決策任務建立新的 Bean。無法執行此作業會造成未預期的行為。

下例示範 Spring 組態的程式碼片段，註冊 `WorkflowScope` 後，用它設定工作流程實作 Bean 和活動用戶端 Bean。

```
<!-- 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>
```

組態程式碼：`<aop:scoped-proxy proxy-target-class="false" />`，用於 `workflowImpl` Bean 的組態，為必要項目，因為 `WorkflowScope` 不支援使用 CGLIB 做為代理。您應該為 `WorkflowScope` 中所有接到不同範圍中其他 Bean 的 Bean 使用此組態。在本例中，`workflowImpl` Bean 需要接到單一範圍中的工作流程工作者 Bean (請參閱以下的完整範例)。

您可以在 Spring Framework 文件中深入了解使用自訂的範圍。

Spring 感知工作者

使用 Spring 時，您應該使用框架提供的 Spring 感知工作者類別：`SpringWorkflowWorker` 和 `SpringActivityWorker`。這些工作者可使用 Spring 插入您的應用程式，如下一個範例所示。Spring 感知工作者預設會實作 Spring 的 `SmartLifecycle` 界面，在 Spring 內容初始化時自動開始輪詢任務。您可以將工作者的 `disableAutoStartup` 屬性設成 `true` 來關閉此功能。

下列範例示範如何設定決策者。本例使用 `MyActivities` 和 `MyWorkflow` 界面 (此處不顯示) 以及對應的實作 `MyActivitiesImpl` 和 `MyWorkflowImpl`。產生的用戶端界面和實作為 `MyWorkflowClient`/`MyWorkflowClientImpl` 和 `MyActivitiesClient`/`MyActivitiesClientImpl` (此處也不顯示)。

活動用戶端會使用 Spring 的自動接線功能插入到工作流程實作中：

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

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

決策者的 Spring 組態如下所示：

```
<?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>
```

由於 `SpringWorkflowWorker` 已在 Spring 中完整設定，並在 Spring 內容初始化時自動開始輪詢，因此決策者的主機程序很簡單：

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

同樣地，活動工作者可設定如下：

```
<?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>
```

活動工作者裝載程序類似決策者：

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

插入決策內容

如果您的工作流程實作依賴這些內容物件，您也可以透過 Spring 輕鬆插入它們。框架自動在 Spring 容器中註冊與內容相關的 Bean。例如，在下列程式碼片段中，已自動接線各種內容物件。不需要內容物件的其他 Spring 組態。

```
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();
   }
}
```

如果您想要透過 Spring XML 組態在工作流程實作中設定內容物件，請在 com.amazonaws.services.simpleworkflow.flow.spring 套裝服務中使用 `WorkflowScopeBeanNames` 類別中宣告的 Bean 名稱。例如：

```
<!-- 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>
```

或者，您可將 `DecisionContextProvider` 插入到工作流程實作 Bean 中，用它建立內容。如果您想提供自訂的提供者和內容實作，這會很有用。

在活動中插入資源

您可以使用控制反轉 (IoC) 容器執行個體化活動實作並予以設定，然後將資源宣告為活動實作類別的屬性，輕鬆插入資料庫連線等資源。這類資源一般範圍限定為單一個。請注意，活動實作在多執行緒上是由活動工作者呼叫。因此，共享資源的存取必須予以同步。

JUnit 整合

框架提供 JUnit 延伸以及內容物件的測試實作 (例如測試時鐘)，您可用來撰寫及執行搭配 JUnit 的單元測試。使用這些延伸，您可以在本機測試您的內嵌工作流程實作。

撰寫簡單的單元測試

為針對您的工作流程撰寫測試，請使用 com.amazonaws.services.simpleworkflow.flow.junit 套裝服務中的 `WorkflowTest` 類別。此類別是架構特定的 JUnit `MethodRule`實作，並在本機執行您的工作流

程程式碼，呼叫內嵌的活動而不是經過 Amazon SWF。這可讓您彈性執行測試，不限次數，也不產生任何費用。

為使用此類別，只要宣告 `WorkflowTest` 類型的欄位並標記以 `@Rule` 註釋即可。執行您的測試之前，請先建立新的 `WorkflowTest` 物件，並將您的活動和工作流程實作新增至此物件。然後，您可以使用產生的工作流程用戶端服務團隊來建立用戶端並開始執行工作流程。框架也提供自訂的 JUnit 執行器 `FlowBlockJUnit4ClassRunner`，您必須將它用於您的工作流程測試中。例如：

```
@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);
    }
```

```
}
```

您也可以為每一個新增至 WorkflowTest 的活動實作另行指定任務清單。例如，如果您的工作流程實作會在主機特定的任務清單中排程活動，您就可以在每部主機的任務清單中註冊活動：

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

請注意，@Test 中的程式碼為非同步。因此，您應該使用非同步工作流程用戶端來啟動執行。為驗證您的測試結果，也會提供 AsyncAssert 協助類別。此類別允許您等候 Promise 就緒再驗證結果。在本例中，我們會先等候工作流程執行的結果就緒，再驗證測試輸出。

如果您要使用 Spring，則可以使用 SpringWorkflowTest 類別，而不是 WorkflowTest 類別。SpringWorkflowTest 提供的屬性，可讓您輕鬆透過 Spring 組態用來設定您的活動和工作流程實作。就像 Spring 感知工作者一樣，您應該使用 WorkflowScope 來設定工作流程實作 Bean。這可確保為每一項決策任務建立新的工作流程實作 Bean。確定設定範圍限定代理 proxy-target-class 設定設為 false 的這些 Bean。如需詳細資訊，請參閱「Spring 整合」一節。在「Spring 整合」一節中示範的 Spring 組態範例可變更為使用 SpringWorkflowTest 來測試工作流程：

```
<?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>
```

模擬活動實作

您可以在測試期間使用真實的活動實作，但若只想要對工作流程邏輯進行單元測試，您應該模擬活動。將活動界面的模擬實作提供給 `WorkflowTest` 類別，即可執行此作業。例如：

```
@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);
    }
}
```

或者，您可以提供活動用戶端的模擬實作，將它插入您的工作流程實作中。

測試內容物件

如果您的工作流程實作取決於架構內容物件，例如 `DecisionContext`，您不需要執行任何特殊動作來測試此類工作流程。透過 `WorkflowTest` 執行測試時，它會自動插入測試內容物件。當您的工作流程實作存取內容物件時，例如使用 `DecisionContextProviderImpl`，它將取得測試實作。您可在您的測試程式碼中 (`@Test` 方法) 操控這些測試內容物件，建立有趣的測試案例。例如，如果您的工作流程建立計時器，您可以在 `WorkflowTest` 類別上呼叫 `clockAdvanceSeconds` 方法建立計時器觸發，向前撥動時鐘的時間。您也可以在 `WorkflowTest` 上使用 `ClockAccelerationCoefficient` 屬性，加速時鐘，建立比平常早的計時器觸發。例如，如果您的工作流程建立一小時的計時器，您可以將 `ClockAccelerationCoefficient` 設成 60，建立一分鐘的計時器觸發。`ClockAccelerationCoefficient` 預設會設定為 1。

如需 com.amazonaws.services.simpleworkflow.flow.test 和 com.amazonaws.services.simpleworkflow.flow.junit 套裝服務的詳細資訊，請參閱 適用於 Java 的 AWS SDK 文件。

錯誤處理

主題

- TryCatchFinally 語意
- 取消
- 巢狀 TryCatchFinally

Java 中的 `try/catch/finally` 建構可簡化錯誤處理，而且隨時可以使用。此建構可讓您建立錯誤處理器與程式碼區塊的關聯。就內部而言，其運作方式為填入呼叫堆疊上錯誤處理器的其他中繼資料。拋出例外狀況時，執行時間會查看相關聯錯誤處理器的呼叫堆疊，並呼叫之；如果找不到適當的錯誤處理器，則會將例外狀況傳播到呼叫鏈。

這適用於同步程式碼，但處理非同步和分散式程式中的錯誤會造成其他挑戰。由於非同步呼叫會立即傳回，因此當非同步程式碼執行時，呼叫者不會在呼叫堆疊上。這表示發起人無法如常處理非同步程式碼中的未處理例外狀況。一般而言，會透過將錯誤狀態傳遞給已傳遞到非同步方法的回呼，來處理源自非同步程式碼的例外狀況。或者，如果使用 `Future<?>`，則會在您嘗試存取它時報告錯誤。這麼做比較不恰當，因為收到例外狀況的程式碼 (使用 `Future<?>` 的回呼或程式碼) 沒有原始呼叫的內容，而且可能無法適當地處理例外狀況。甚至，在分散式非同步系統中，如果並行執行元件，則可能會同時發生多個錯誤。這些錯誤可以是不同的類型和嚴重性，而且需要適當地加以處理。

在非同步呼叫之後清除資源也十分困難。與同步程式碼不同，您無法在呼叫程式碼中使用 try/catch/finally 來清除資源，因為當最後區塊執行時，在嘗試區塊中啟動的工作可能仍在進行中。

框架提供一種機制，讓分散式非同步程式碼中的錯誤處理類似 Java 的 try/catch/finally，而且幾乎像 Java 的 try/catch/finally 一樣簡單。

```
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 類別與其變體 TryFinally 和 TryCatch 的運作方式與 Java 的 try/catch/finally 類似。使用它，即可建立例外狀況處理器與工作流程程式碼區塊的關聯，而工作流程程式碼區塊可能會以非同步與遠端任務執行。doTry() 方法在邏輯上等於 try 區塊。框架會自動執行 doTry() 中的程式碼。Promise 物件清單可以傳遞給 TryCatchFinally 的建構函數。所有傳入建構函數的 Promise 物件都準備就緒時，將會執行 doTry 方法。如果從 doTry() 非同步呼叫的程式碼引發例外狀況，則會取消 doTry() 中的任何待定工作，並呼叫 doCatch() 來處理例外狀況。例如，在上面的清單中，如果 downloadImage 拋出例外狀況，則會取消 createThumbnail 和 uploadImage。最後，完成所有非同步工作 (已完成、失敗或已取消) 時，會呼叫 doFinally()。它可以用於資源清理。您也可以將這些類別巢狀處理，以符合您的需求。

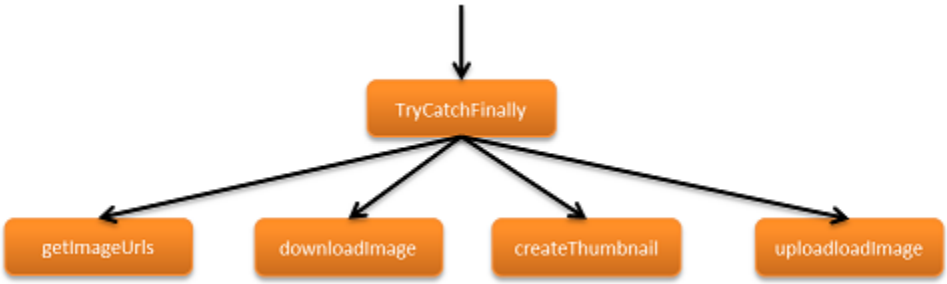
在 doCatch() 中報告例外狀況時，框架會提供包含非同步和遠端呼叫的完整邏輯呼叫堆疊。偵錯時，這可能十分有用，特別是您有呼叫其他非同步方法的非同步方法時。例如，downloadImage 的例外狀況將會產生與下列類似的例外狀況：

```
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 語意

AWS Flow Framework 適用於 Java 的 程式的執行可以視覺化為同時執行分支的樹狀結構。非同步方法、活動和 `TryCatchFinally` 本身的呼叫會在此執行的樹狀目錄中建立新分支。例如，影像處理工作流程可以檢視為下圖中所顯示的樹狀目錄。



某個執行分支中的錯誤將會導致回溯該分支，就像例外狀況導致回溯 Java 程式中的呼叫堆疊一樣。回溯會繼續移至執行分支，直到錯誤已處理或到達樹狀目錄根，在此情況下，會終止工作流程執行。

框架會報告將任務處理為例外狀況時所發生的錯誤。它會建立下列兩者的關聯：`TryCatchFinally` 中所定義的例外狀況處理器 (`doCatch()` 方法) 與對應 `doTry()` 中程式碼所建立的所有任務。如果任務失敗，例如由於逾時或未處理的例外狀況，則會引發適當的例外狀況，並`doCatch()`叫用對應的 來處理。為了達成此目的，框架與 Amazon SWF 一起運作，以傳播遠端錯誤，並在發起人的內容中將其重新作為例外狀況。

取消

同步程式碼中發生例外狀況時，控制會直接跳至 `catch` 區塊，並跳過 `try` 區塊中剩餘的程式碼。例如：

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

在此程式碼中，如果 `b()` 拋出例外狀況，則絕不會呼叫 `c()`。與工作流程比較：

```
new TryCatch() {

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

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

在此情況下，activityA、activityB 和 activityC 呼叫都會順利傳回，並導致建立三個非同步執行的任務。假設稍後 activityB 的任務導致錯誤。此錯誤由 Amazon SWF 記錄在歷史記錄中。若要處理此情況，框架會先嘗試取消源自相同 doTry() 範圍的所有其他任務；在此情況下為 activityA 和 activityC。所有這類任務完成時 (取消、失敗或順利完成)，都會呼叫適當的 doCatch() 方法來處理錯誤。

與永不執行 c() 的同步範例不同，會呼叫 activityC，並排定執行任務；因此，框架會嘗試予以取消，但不保證會確實取消。由於活動可能已經完成、可能忽略取消請求，或可能因錯誤而失敗，因而無法保證取消。不過，框架能夠保證只有在完成從對應 doTry() 啟動的所有任務之後，才會呼叫 doCatch()。也能保證只有在完成從 doTry() 和 doCatch() 啟動的所有任務之後，才會呼叫 doFinally()。例如，如果上述範例中的活動彼此相依，假設 activityB 相依於 activityA和 activityC activityB，則 的取消activityC會立即生效，因為在 activityB 完成之前不會在 Amazon SWF 中排程：

```
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();
```

```
    }
};
```

活動訊號

AWS Flow Framework 適用於 Java 的合作取消機制允許正常取消傳輸中的活動任務。觸發取消時，會自動取消封鎖或等待指派給工作者的任務。不過，如果任務已指派給工作者，則框架會請求取消活動。活動實作必須明確地處理這類取消請求。作法是報告您活動的活動訊號。

報告活動訊號允許活動實作報告進行中之活動任務 (對監控很有幫助) 的進度，且可讓活動檢查取消請求。如果已請求取消，則 `recordActivityHeartbeat` 方法將拋出 `CancellationException`。活動實作可以截獲此例外狀況，並處理取消請求，或者可以抑制例外狀況來忽略請求。若要使用取消請求，活動應該執行所需的清除 (如果有的話)，然後重新拋出 `CancellationException`。從活動實作拋出此例外狀況時，框架會記錄已完成活動任務，但狀態為已取消。

下列範例顯示可下載和處理影像的活動。活動會在處理每個影像之後發出活動訊號，而且，如果請求取消，則會清除和重新拋出例外狀況來確認取消。

```
@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;
        }
    }
}
```

不需要報告活動訊號，但若您的活動是長時間執行，或可能會執行要在錯誤情況下取消的高成本操作時，則建議報告活動訊號。您應該從活動實作定期呼叫 `heartbeatActivityTask`。

如果活動逾時，則會拋出 `ActivityTaskTimedOutException`，而且例外狀況物件上的 `getDetails` 會傳回資料，而這項資料會傳遞給對應活動任務的最後一個成功

heartbeatActivityTask 呼叫。工作流程實作可能會使用這項資訊來判斷活動任務逾時之前的進度。

> **Note**
>
> 由於 Amazon SWF 可能會調節活動訊號請求，因此活動訊號頻率太頻繁不是很好的做法。如需 Amazon SWF 設定的限制，請參閱 Amazon Simple Workflow Service 開發人員指南。

明確地取消任務

除了錯誤情況外，您也可能在其他情況下明確取消任務。例如，如果使用者取消訂單，則可能需要取消使用信用卡處理付款的活動。框架可讓您明確地取消 TryCatchFinally 範圍中所建立的任務。在下列範例中，如果在處理付款時收到訊號，則會取消付款任務。

```
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);
        }
    }
}
```

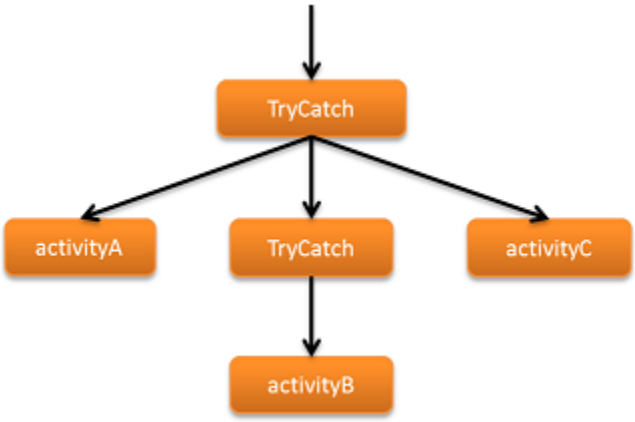
接收已取消之任務的通知

任務完成且處於已取消狀態時，框架會拋出 CancellationException 來通知工作流程邏輯。任務完成且處於已取消狀態時，會在歷史記錄中建立記錄，而且框架會以 CancellationException 呼叫適當的 doCatch()。如上述範例所示，在取消付款處理任務之時，工作流程會收到 CancellationException。

未處理的 CancellationException 會傳播到執行分支，如同其他例外狀況。不過，只有在範圍中沒有其他例外狀況時，doCatch() 方法才會收到 CancellationException；其他例外狀況的優先順序高於取消。

巢狀 TryCatchFinally

您可以將 TryCatchFinally 巢狀處理，以符合您的需求。由於每個 TryCatchFinally 都會在執行樹狀目錄中建立新的分支，因此您可以建立巢狀範圍。父範圍中的例外狀況將會導致嘗試取消透過在其內將 TryCatchFinally 巢狀處理而啟動的所有任務。不過，巢狀 TryCatchFinally 中的例外狀況不會自動傳播至父項。如果您想要將例外狀況從巢狀 TryCatchFinally 傳播至其內含的 TryCatchFinally，則應該在 doCatch() 中重新拋出例外狀況。換言之，只會發出未處理的例外狀況，就像 Java 的 try/catch 一樣。如果您呼叫取消方法來取消巢狀 TryCatchFinally，則會取消巢狀 TryCatchFinally，但不會自動取消內含的 TryCatchFinally。




```
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);
    }
};
```

重試失敗的活動

活動有時會因暫時性原因失敗，例如暫時失去連線。有時活動可能成功，所以處理活動錯誤的適當方法，通常是重試活動，或許要多試幾次。

重試這些活動有各種策略，最好的策略是根據您的工作流程詳細資訊。這些策略分為三大基本分類：

- 重試到成功為止策略只會一直重試活動直到完成。
- 指數重試策略會以指數方式增加重試嘗試之間的時間間隔，直到活動完成或程序達到指定的停止點，例如嘗試次數的上限。
- 自訂重試策略決定是否以及如何在每次嘗試失敗後重試活動。

以下各節會說明如何實作這些策略。範例工作流程工作者全都使用單一活動 `unreliableActivity`，隨機執行下列作業之一：

- 立即完成
- 超過逾時值故意失敗
- 拋出 `IllegalStateException` 故意失敗

重試到成功為止策略

最簡單的重試策略是每次活動失敗就一直重試，直到最後成功。基本模式是：

1. 實作您工作流程進入點方法的 `TryCatch` 或 `TryCatchFinally` 類別。
2. 在 `doTry` 中執行活動
3. 如果活動失敗，框架會呼叫 `doCatch`，再次執行進入點方法。
4. 重複步驟 2 - 3 直到順利完成活動。

以下工作流程會實作重試到成功為止策略。工作流程界面在 `RetryActivityRecipeWorkflow` 中實作，且有一個方法 `runUnreliableActivityTillSuccess`，這是工作流程的進入點。工作流程工作者在 `RetryActivityRecipeWorkflowImpl` 中實作，如下所示：

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

工作流程運作方式如下：

1. `runUnreliableActivityTillSuccess` 會建立 `Settable<Boolean>` 物件，名稱為 `retryActivity`，其用於指出活動是否失敗，以及是否應該重試。`Settable<T>` 是衍生自 `Promise<T>` 並且運作方式相同，但是您手動設定 `Settable<T>` 物件的值。
2. `runUnreliableActivityTillSuccess` 實作匿名的巢狀 `TryCatch` 類別，以處理 `unreliableActivity` 活動拋出的任何例外狀況。如需深入討論如何處理匿名程式碼拋出的例外狀況，請參閱「錯誤處理」。
3. `doTry` 執行 `unreliableActivity` 活動，這樣會傳回 `Promise<Void>` 物件，名為 `activityRanSuccessfully`。
4. `doTry` 呼叫非同步的 `setRetryActivityToFalse` 方法，它有兩個參數：

- activityRanSuccessfully 會採用 unreliableActivity 活動傳回的 Promise<Void> 物件。
- retryActivity 採用 retryActivity 物件。

當 unreliableActivity 完成後，activityRanSuccessfully 就會就緒，且 setRetryActivityToFalse 會將 retryActivity 設為 false。否則，activityRanSuccessfully 絕不會就緒，而 setRetryActivityToFalse 不執行。

5. 如果 unreliableActivity 擲出例外狀況，框架就會呼叫 doCatch 並將例外狀況物件傳遞給它。doCatch 將 retryActivity 設定為 true。
6. runUnreliableActivityTillSuccess 呼叫非同步的 restartRunUnreliableActivityTillSuccess 方法，並將 retryActivity 物件傳遞給它。因為 retryActivity 是 Promise<T> 類型，所以 restartRunUnreliableActivityTillSuccess 延遲執行直到 retryActivity 就緒為止，這會在 TryCatch 完成後發生。
7. 當 retryActivity 就緒時，restartRunUnreliableActivityTillSuccess 會擷取值。
   - 如果該值為 false，表示重試成功。restartRunUnreliableActivityTillSuccess 不執行任何動作，且重試序列會終止。
   - 如果值為 true，表示重試失敗。restartRunUnreliableActivityTillSuccess 會呼叫 runUnreliableActivityTillSuccess 以再次執行活動。
8. 重複步驟 1 - 7 直到 unreliableActivity 完成。

> **Note**
>
> doCatch 不處理例外狀況，只將 retryActivity 物件設成 true，指出活動失敗。重試是由非同步的 restartRunUnreliableActivityTillSuccess 方法處理，這會延遲例外狀況直到 TryCatch 完成。此方法的原因是，如果您以 doCatch 重試活動，您就無法取消它。以 restartRunUnreliableActivityTillSuccess 重試活動可讓您執行可取消的活動。

指數重試策略

使用指數重試策略，框架會在指定期間後 (N 秒) 再次執行失敗的活動。如果該嘗試失敗，框架就會在 2N 秒後、4N 秒後、以此類推，再次執行活動。因為等待時間會變得相當長，您一般會在某個時間點停止重試嘗試，而不是無止境地繼續下去。

框架提供三種方式實作指數重試策略：

- `@ExponentialRetry` 註釋是最簡單的方法，但您必須在編譯階段設定重試組態選項。
- `RetryDecorator` 類別可讓您在執行時間設定重試組態，並視需要予以變更。
- `AsyncRetryingExecutor` 類別可讓您在執行時間設定重試組態，並視需要予以變更。此外，框架會呼叫使用者實作的 `AsyncRunnable.run` 方法，執行每次的重試嘗試。

所有方法都支援下列組態選項，它們的時間值都是以秒計：

- 初始重試等待時間。
- 用來計算重試間隔的退避係數，如下所示：

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

  預設值為 2.0。
- 重試嘗試次數的上限。預設值無限制。
- 重試間隔上限。預設值無限制。
- 過期時間。當程序期間總計超過此值時就會停止重試嘗試。預設值無限制。
- 會觸發重試程序的例外狀況。根據預設，每種例外狀況都會觸發重試程序。
- 不會觸發重試嘗試的例外狀況。根據預設，不排除任何例外狀況。

以下各節說明您可實作指數重試策略的各種方式。

使用 @ExponentialRetry 的指數重試

為活動實作指數重試策略最簡單的方式，是在界面定義中將 `@ExponentialRetry` 註釋套用到活動。如果活動失敗，框架會根據指定的選項值，自動處理重試程序。基本模式是：

1. 將 `@ExponentialRetry` 套用到合適的活動並指定重試組態。
2. 如果註釋的活動失敗，框架會根據註釋引數指定的組態，自動重試活動。

`ExponentialRetryAnnotationWorkflow` 工作流程工作者使用 `@ExponentialRetry` 註釋實作指數重試策略。它使用 `unreliableActivity` 活動，它的界面定義是以 `ExponentialRetryAnnotationActivities` 實作，如下所示：

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

`@ExponentialRetry` 選項指定以下策略：

- 只有當活動拋出 `IllegalStateException` 時才重試。
- 使用 5 秒的初始等待時間。
- 不超過 5 次重試嘗試。

工作流程界面在 `RetryWorkflow` 中實作，且有一個方法 `process`，這是工作流程的進入點。工作流程工作者在 `ExponentialRetryAnnotationWorkflowImpl` 中實作，如下所示：

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

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

工作流程運作方式如下：

1. `process` 執行同步的 `handleUnreliableActivity` 方法。
2. `handleUnreliableActivity` 執行 `unreliableActivity` 活動。

如果活動因拋出 `IllegalStateException` 而失敗，框架會自動執行 `ExponentialRetryAnnotationActivities` 指定的重試策略。

使用 RetryDecorator 類別的指數重試

@ExponentialRetry 簡單好用。不過，組態是靜態的且於編譯階段設定，所以每次活動失敗，框架都會使用相同的重試策略。您可以使用 RetryDecorator 類別，實作更有彈性的指數重試策略，這可讓您在執行時間指定組態，並視需要予以變更。基本模式是：

1. 建立並設定指定重試組態的 ExponentialRetryPolicy 物件。
2. 建立 RetryDecorator 物件，並將步驟 1 中的 ExponentialRetryPolicy 物件傳遞到建構函數。
3. 將活動用戶端的類別名稱傳遞到 RetryDecorator 物件的裝飾方法，將裝飾項目物件套用到活動。
4. 執行活動。

如果活動失敗，框架會根據 ExponentialRetryPolicy 物件的組態，重試活動。您可以修改此物件，視需要變更重試組態。

> **Note**
>
> @ExponentialRetry 註釋和 RetryDecorator 類別互斥。您不能使用 RetryDecorator 動態覆寫 @ExponentialRetry 註釋指定的重試政策。

以下工作流程實作示範如何使用 RetryDecorator 類別來實作指數重試策略。它使用沒有 @ExponentialRetry 註釋的 unreliableActivity 活動。工作流程界面在 RetryWorkflow 中實作，且有一個方法 process，這是工作流程的進入點。工作流程工作者在 DecoratorRetryWorkflowImpl 中實作，如下所示：

```
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();
  }
}
```

工作流程運作方式如下：

1. process 建立並設定 ExponentialRetryPolicy 物件的方法：
   - 將初始重試間隔傳遞到建構函數。
   - 呼叫物件的 withMaximumAttempts 方法來設定嘗試次數上限為 5。ExponentialRetryPolicy 會公開您可以用來指定其他組態選項的其他 with 物件。
2. process 建立名為 retryDecorator 的 RetryDecorator 物件，並將步驟 1 中的 ExponentialRetryPolicy 物件傳遞到建構函數。
3. process 透過呼叫 retryDecorator.decorate 方法並將活動用戶端的類別名稱傳遞給它，將裝飾項目套用到活動。
4. handleUnreliableActivity 執行活動。

如果活動失敗，框架會根據步驟 1 指定的組態，重試活動。

> **Note**
>
> ExponentialRetryPolicy 類別的數個 with 方法有對應的 set 方法，您可隨時呼叫以修改對應的組態選項：setBackoffCoefficient、setMaximumAttempts、setMaximumRetryIntervalSeconds 和 setMaximumRetryExpirationIntervalSeconds。

使用 AsyncRetryingExecutor 類別的指數重試

RetryDecorator 類別設定重試程序比 @ExponentialRetry 更有彈性，但是框架仍會根據 ExponentialRetryPolicy 物件目前的組態自動執行重試嘗試。更彈性的方法是使用 AsyncRetryingExecutor 類別。除在執行時間讓您設定重試程序之外，框架還會呼叫使用者實作的 AsyncRunnable.run 方法來執行每次的重試嘗試，不只是執行活動。

基本模式是：

1. 建立並設定 ExponentialRetryPolicy 物件以指定重試組態。
2. 建立 AsyncRetryingExecutor 物件，並將 ExponentialRetryPolicy 物件和工作流程時鐘執行個體傳遞給它。
3. 實作匿名的巢狀 TryCatch 或 TryCatchFinally 類別。
4. 實作匿名的 AsyncRunnable 類別並覆寫 run 方法，實作自訂的程式碼來執行活動。
5. 覆寫 doTry 來呼叫 AsyncRetryingExecutor 物件的 execute 方法，並將步驟 4 的 AsyncRunnable 類別傳遞給它。AsyncRetryingExecutor 物件呼叫 AsyncRunnable.run 執行活動。
6. 如果活動失敗，AsyncRetryingExecutor 物件會根據步驟 1 指定的重試政策，再次呼叫 AsyncRunnable.run 方法。

以下工作流程示範如何使用 AsyncRetryingExecutor 類別來實作指數重試策略。它會和前文討論的 DecoratorRetryWorkflow 工作流程使用相同的 unreliableActivity 活動。工作流程界面在 RetryWorkflow 中實作，且有一個方法 process，這是工作流程的進入點。工作流程工作者在 AsyncExecutorRetryWorkflowImpl 中實作，如下所示：

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

工作流程運作方式如下：

1. process 呼叫 handleUnreliableActivity 方法，並將組態設定傳遞給它。
2. handleUnreliableActivity 使用步驟 1 的組態設定建立 ExponentialRetryPolicy 物件 retryPolicy。
3. handleUnreliableActivity 建立 AsyncRetryExecutor 物件 executor，並將步驟 2 的 ExponentialRetryPolicy 物件和工作流程時鐘執行個體傳遞到建構函數。
4. handleUnreliableActivity 實作匿名的巢狀 TryCatch 類別，並覆寫 doTry 和 doCatch 方法來執行重試嘗試及處理任何例外狀況。
5. doTry 建立匿名的 AsyncRunnable 類別並覆寫 run 方法，實作自訂的程式碼來執行 unreliableActivity。為簡化起見，run 只執行活動，但您可視情況實作更成熟的方法。
6. doTry 呼叫 executor.execute 並將 AsyncRunnable 物件傳遞給它。execute 呼叫 AsyncRunnable 物件的 run 方法來執行活動。
7. 如果活動失敗，執行器會根據 retryPolicy 物件組態再次呼叫 run。

如需深入討論如何使用 TryCatch 類別處理錯誤，請參閱「AWS Flow Framework 適用於 Java 例外狀況」。

自訂重試策略

重試失敗活動最有彈性的方法是自訂策略，遞迴呼叫執行重試嘗試的非同步方法，非常類似重試到成功為止策略。但您不僅僅可以再次執行活動，還可實作自訂邏輯來決定是否及如何執行每次接續的重試嘗試。基本模式是：

1. 建立 Settable<T> 狀態物件，用以指示活動是否失敗。

2. 實作巢狀的 TryCatch 或 TryCatchFinally 類別。
3. doTry 執行活動。
4. 如果活動失敗，doCatch 會設定狀態物件指出活動失敗。
5. 呼叫非同步的錯誤處理方法，並將狀態物件傳遞給它。此方法會延遲例外狀況直到 TryCatch 或 TryCatchFinally 完成。
6. 錯誤處理方法決定是否重試活動，如果重試，何時重試。

以下工作流程示範如何實作自訂的重試策略。它會和 DecoratorRetryWorkflow 與 AsyncExecutorRetryWorkflow 工作流程使用相同的 unreliableActivity 活動。工作流程界面在 RetryWorkflow 中實作，且有一個方法 process，這是工作流程的進入點。工作流程工作者在 CustomLogicRetryWorkflowImpl 中實作，如下所示：

```
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;
    }
}
```

工作流程運作方式如下：

1. process 呼叫非同步的 callActivityWithRetry 方法。
2. callActivityWithRetry 會建立 Settable<Throwable> 物件，名稱為 failure，其用於指出活動是否已失敗。Settable<T> 是衍生自 Promise<T> 並且運作方式相同，但是您手動設定 Settable<T> 物件的值。
3. callActivityWithRetry 實作匿名的巢狀 TryCatchFinally 類別，以處理 unreliableActivity 拋出的任何例外狀況。如需深入討論如何處理匿名程式碼拋出的例外狀況，請參閱「AWS Flow Framework 適用於 Java 例外狀況」。
4. doTry 執行 unreliableActivity。
5. 如果 unreliableActivity 擲出例外狀況，框架會呼叫 doCatch 並傳遞例外狀況給它。doCatch 會將 failure 設定為例外狀況物件，指出活動失敗並將物件放在就緒狀態。
6. doFinally 檢查 failure 是否就緒，只有當 failure 是由 doCatch 所設定時才為 true。
   - 如果 failure 準備就緒， 不會doFinally執行任何動作。
   - 如果 failure 尚未就緒，活動完成且 doFinally 將錯誤設為 null。
7. callActivityWithRetry 呼叫非同步的 retryOnFailure 方法，並將錯誤傳遞給它。因為錯誤是 Settable<T> 類型，所以 callActivityWithRetry 會延遲執行直到錯誤就緒為止，這會在 TryCatchFinally 完成後發生。
8. retryOnFailure 從錯誤取得值。
   - 如果錯誤設成 null，重試嘗試就會成功。retryOnFailure 不執行任何動作，這會終止重試程序。
   - 如果錯誤設成例外狀況物件且 shouldRetry 傳回 true，retryOnFailure 會呼叫 callActivityWithRetry 重試活動。

     shouldRetry 實作自訂邏輯以決定是否重試失敗的活動。為簡化起見，shouldRetry 一律傳回 true，而 retryOnFailure 會立即執行活動，但您可視需要實作更成熟的邏輯。
9. 步驟 2-8 會重複，直到 unreliableActivity 完成或shouldRetry決定停止程序為止。

> **Note**
>
> doCatch 不處理重試程序，只會設定錯誤指出活動失敗。重試程序是由非同步的 retryOnFailure 方法處理，這會延遲例外狀況直到 TryCatch 完成。此方法的原因是，如果您以 doCatch 重試活動，您就無法取消它。以 retryOnFailure 重試活動可讓您執行可取消的活動。

協助程式任務

AWS Flow Framework 適用於 Java 的 允許將特定任務標記為 daemon。這可讓您建立執行某些背景工作的任務，這些工作應該在所有其他工作完成時取消。例如，當其餘的工作流程完成時，就應該取消運作狀態監控任務。您可以在同步方法或 TryCatchFinally 執行個體上設定 daemon 標記，即可完成此作業。在下列範例中，同步方法 monitorHealth() 標記為 daemon。

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

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

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

在上例中，當 doUsefulWorkActivity 完成時，就會自動取消 monitoringHealth。這將會取消以此同步方法為根目錄的整個執行分支。取消的語意和 TryCatchFinally 中的一樣。同樣地，您也可以將布林值標記傳遞到建構函數來標記 TryCatchFinally 協助程式。

```
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
            }
        };
    }
}
```

在 內啟動的協助程式任務會`TryCatchFinally`範圍限定於其建立於其中的內容，也就是說，它將範圍限定於 `doTry()`、 `doCatch()`或 `doFinally()`方法。例如，在下列範例中，startMonitoring 同步方法標記為協助程式並從 `doTry()` 呼叫。為它建立的任務會在於 `doTry()` 內啟動的其他任務 (本例中為 `doUsefulWorkActivity`) 完成時立刻取消。

```
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 重播行為

本主題使用「什麼是適用於 Java AWS Flow Framework 的 ？」一節中的範例來討論重新執行行為範例。將會討論同步及非同步藍本。

範例 1：同步重新執行

如需同步工作流程中的重新執行運作方式範例，請在個別的 HelloWorldWorkflow 工作流程和活動實作內新增 `println` 呼叫，以對這些實作進行修改，如下所示：

```
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);
   }
}
```

如需程式碼的詳細資訊，請參閱「HelloWorldWorkflow 應用程式」。以下編輯過的輸出版本包含註解，指出每個重新執行部分的開始。

```
//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
```

此範例的重新執行程序運作方式如下：

- 第一個部分排程無相依性的 `getName` 活動任務。
- 第二個部分排程相依於 `getName` 的 `getGreeting` 活動任務。
- 第三個部分排程相依於 `getGreeting` 的 `say` 活動任務。
- 最後一個部分未排程任何額外任務，且找不到未完成的活動，其可終止工作流程執行。

> **Note**
>
> 三種活動用戶端方法在每個部分各呼叫一次。不過，其中只有一個呼叫會導致活動任務，因此一個任務只會執行一次。

範例 2：非同步重新執行

與同步重新執行範例類似，您可以修改 HelloWorldWorkflowAsync 應用程式以查看非同步重新執行運作方式。產生的輸出如下：

```
//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
```

因為只有兩個活動，所以 HelloWorldAsync 使用三個重新執行部分。`getGreeting` 活動已取代為 getGreeting 非同步工作流程方法，後者不會在完成時開始重新執行部分。

第一個部分不會呼叫 `getGreeting`，因為它相依於 name 活動完成。不過，在 getName 完成之後，重新執行會針對每個後續部分呼叫 getGreeting 一次。

另請參閱

- AWS Flow Framework 基本概念：分散式執行

最佳實務

使用這些最佳實務，充分利用 AWS Flow Framework 適用於 Java 的 。

主題

- 變更決策者程式碼：版本控制和功能標記

變更決策者程式碼：版本控制和功能標記

本節顯示如何使用兩種方法避免對決策者進行回溯不相容變更：

- 版本控制提供基本解決方案。
- 含功能標記的版本控制是以版本控制解決方案為基礎：未介紹工作流程的新版本，而且不需要推送新程式碼來更新版本。

在您嘗試這些解決方案之前，請熟讀「範例藍本」小節，其中說明回溯不相容決策者變更的原因和影響。

重播程序和程式碼變更

當 AWS Flow Framework 適用於 Java 的決策者工作者執行決策任務時，必須先重建執行的目前狀態，才能新增步驟。決策者使用稱為「重播」的程序來執行這項作業。

重播程序會從頭重新執行決策者程式碼，同時瀏覽已發生事件的歷史記錄。瀏覽事件歷史記錄可允許框架對訊號或任務完成做出反應，並解鎖程式碼中的 `Promise` 物件。

當架構執行決策者程式碼時，它會透過遞增計數器，將 ID 指派給每個排程任務 （活動、Lambda 函數、計時器、子工作流程或傳出訊號）。框架會將此 ID 傳達給 Amazon SWF，並將 ID 新增至歷史記錄事件，例如 `ActivityTaskCompleted`。

為使重播程序成功，確定決策者程式碼十分重要，以及針對每個工作流程執行中的每個決策，依相同順序排定相同任務。如果您未遵守此需求，則框架可能會無法比對 `ActivityTaskCompleted` 事件中的 ID 與現有 `Promise` 物件。

範例藍本

有一個程式碼變更的類別視為回溯不相容。這些變更包含可修改已排程之任務的數目、類型或順序的更新。請思考下列範例：

您可以撰寫決策者程式碼來排定兩個計時器任務。您可以啟動執行，並執行決策。因此，已排定 ID 為 1 和 2 的兩個計時器任務。

如果您更新決策者程式碼在執行下個決策之前只排定一個計時器，則在下個決策任務期間，框架將無法重播第二個 `TimerFired` 事件，因為 ID 2 不符合程式碼產生的任何計時器任務。

藍本大綱

下列大綱顯示此藍本的步驟。此藍本的最後目標為遷移至系統，且此系統只排定一個計時器，但不會導致遷移前啟動的執行發生錯誤。

1. 初始決策者版本
   a. 撰寫決策者。
   b. 啟動決策者。
   c. 決策者排定兩個計時器。
   d. 決策者啟動五次執行。
   e. 停止決策者。
2. 回溯不相容決策者變更
   a. 修改決策者。
   b. 啟動決策者。
   c. 決策者排定一個計時器。
   d. 決策者啟動五次執行。

下列各節所包含的 Java 程式碼範例顯示如何實作此藍本。「解決方案」小節中的程式碼範例顯示各種方法來修正回溯不相容變更。

> **Note**
> 您可以使用 適用於 Java 的 AWS SDK 的最新版本來執行此程式碼。

常見程式碼

下列 Java 程式碼在此藍本的範例之間不會變更。

`SampleBase.java`

```
package sample;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflow;
import com.amazonaws.services.simpleworkflow.AmazonSimpleWorkflowClientBuilder;
import com.amazonaws.services.simpleworkflow.flow.JsonDataConverter;
import com.amazonaws.services.simpleworkflow.model.DescribeWorkflowExecutionRequest;
import com.amazonaws.services.simpleworkflow.model.DomainAlreadyExistsException;
import com.amazonaws.services.simpleworkflow.model.RegisterDomainRequest;
import com.amazonaws.services.simpleworkflow.model.Run;
import com.amazonaws.services.simpleworkflow.model.StartWorkflowExecutionRequest;
import com.amazonaws.services.simpleworkflow.model.TaskList;
import com.amazonaws.services.simpleworkflow.model.WorkflowExecution;
import com.amazonaws.services.simpleworkflow.model.WorkflowExecutionDetail;
import com.amazonaws.services.simpleworkflow.model.WorkflowType;

public class SampleBase {

    protected String domain = "DeciderChangeSample";
    protected String taskList = "DeciderChangeSample-" + UUID.randomUUID().toString();
    protected AmazonSimpleWorkflow service =
 AmazonSimpleWorkflowClientBuilder.defaultClient();
    {
        try {
            AmazonSimpleWorkflowClientBuilder.defaultClient().registerDomain(new
 RegisterDomainRequest().withName(domain).withDescription("desc").withWorkflowExecutionRetentio
        } catch (DomainAlreadyExistsException e) {
        }
    }

    protected List<WorkflowExecution> workflowExecutions = new ArrayList<>();

    protected void startFiveExecutions(String workflow, String version, Object input) {
        for (int i = 0; i < 5; i++) {
            String id = UUID.randomUUID().toString();
            Run startWorkflowExecution = service.startWorkflowExecution(
                    new
 StartWorkflowExecutionRequest().withDomain(domain).withTaskList(new
 TaskList().withName(taskList)).withInput(new JsonDataConverter().toData(new
```

```
 Object[] { input })).withWorkflowId(id).withWorkflowType(new
 WorkflowType().withName(workflow).withVersion(version)));
            workflowExecutions.add(new
 WorkflowExecution().withWorkflowId(id).withRunId(startWorkflowExecution.getRunId()));
            sleep(1000);
        }
    }

    protected void printExecutionResults() {
        waitForExecutionsToClose();
        System.out.println("\nResults:");
        for (WorkflowExecution wid : workflowExecutions) {
            WorkflowExecutionDetail details = service.describeWorkflowExecution(new
 DescribeWorkflowExecutionRequest().withDomain(domain).withExecution(wid));
            System.out.println(wid.getWorkflowId() + " " +
 details.getExecutionInfo().getCloseStatus());
        }
    }

    protected void waitForExecutionsToClose() {
        loop: while (true) {
            for (WorkflowExecution wid : workflowExecutions) {
                WorkflowExecutionDetail details = service.describeWorkflowExecution(new
 DescribeWorkflowExecutionRequest().withDomain(domain).withExecution(wid));
                if ("OPEN".equals(details.getExecutionInfo().getExecutionStatus())) {
                    sleep(1000);
                    continue loop;
                }
            }
            return;
        }
    }

    protected void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

}
```

Input.java

```
package sample;

public class Input {

    private Boolean skipSecondTimer;

    public Input() {
    }

    public Input(Boolean skipSecondTimer) {
        this.skipSecondTimer = skipSecondTimer;
    }

    public Boolean getSkipSecondTimer() {
        return skipSecondTimer != null && skipSecondTimer;
    }

    public Input setSkipSecondTimer(Boolean skipSecondTimer) {
        this.skipSecondTimer = skipSecondTimer;
        return this;
    }

}
```

撰寫初始決策者程式碼

以下是決策者的初始 Java 程式碼。此程式碼註冊為第 1 版，並且排定兩個五秒計時器任務。

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);
        }

    }
}
```

模擬回溯不相容變更

下列修改過的決策者 Java 程式碼是個不錯的回溯不相容變更範例。程式碼仍然註冊為第 1 版，但只排定一個計時器。

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);
        }

    }
}
```

下列 Java 程式碼可讓您藉由執行修改過的決策者，來模擬進行回溯不相容變更的問題。

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();
    }

}
```

當您執行程式時，那三個失敗的執行就是在決策者初始版本下啟動並在遷移之後繼續的執行。

解決方案

您可以使用下列解決方案來避免回溯不相容變更。如需詳細資訊，請參閱「變更決策者程式碼」和「範例藍本」。

使用版本控制

在此解決方案中，您可以將決策者複製至新類別，並修改決策者，然後在新的工作流程版本下註冊決策者。

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);
        }

    }

}
```

在更新過的 Java 程式碼中，第二個決策者工作者會同時執行兩個版本的工作流程，讓進行中的執行得以單獨繼續，無關乎第 2 版中的變更。

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();
    }

}
```

當您執行程式時，所有執行會順利完成。

使用功能標記

回溯相容性問題的另一種解決方案是分支處理程式碼，以支援根據輸入資料 (而非工作流程版本) 來分支處理相同類別中的兩個實作。

當您採取這種方法時，可以在每次引進敏感變更時將欄位新增至輸入物件 (或修改輸入物件的現有欄位)。針對在遷移之前啟動的執行，輸入物件不會有欄位 (或有不同值)。因此，您不需要增加版本編號。

> **Note**
>
> 如果您新增欄位，則請確定 JSON 還原序列化程序具有回溯相容。引進欄位之前序列化的物件仍然應該在遷移之後成功還原序列化。因為 JSON 只要欄位遺漏就會設定 `null` 值，所以一律會使用已封箱的類型 (`Boolean`，而非 `boolean`)，並處理值為 `null` 的情況。

`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);
            }
        }

    }
}
```

在更新過的 Java 程式碼中，仍然會針對第 1 版註冊兩個工作流程版本的程式碼。不過，在遷移之後，會啟動輸入資料的 `skipSecondTimer` 欄位設為 `true` 的新執行。

`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();
    }

}
```

當您執行程式時，所有執行會順利完成。

適用於 Java AWS Flow Framework 的 疑難排解和偵錯秘訣

主題

- 編譯錯誤
- 未知的資源錯誤
- 在 Promise 上呼叫 get() 時的例外狀況
- 非確定性工作流程
- 版本控制引起的問題
- 對工作流程執行進行故障診斷和偵錯
- 遺失的任務
- 由於 API 參數長度限制導致的驗證失敗

本節說明使用 AWS Flow Framework for Java 開發工作流程時，您可能會遇到的一些常見陷阱。本節也會提供一些提示，協助您診斷和偵錯問題。

編譯錯誤

如果您使用 AspectJ 編譯時間繫繞選項，則可能會發生編譯時間錯誤，在這類錯誤中，編譯器找不到針對工作流程和活動所產生的用戶端類別。這類編譯錯誤的可能原因在於 AspectJ 建置器會在編譯期間忽略已產生的用戶端。您可以移除專案中的 AspectJ 功能並重新予以啟用，以修復此問題。請注意，每次工作流程或活動界面變更時，您都需要執行這項作業。有鑑於此問題，建議您改為使用載入時間繫繞選項。如需詳細資訊，請參閱「設定 AWS Flow Framework 適用於 Java 的」一節。

未知的資源錯誤

當您嘗試對無法使用的資源執行 操作時，Amazon SWF 會傳回未知的資源錯誤。此故障的常見原因如下：

- 您用來設定工作者的網域並不存在。若要修正此問題，請先使用 Amazon SWF 主控台或 Amazon SWF 服務 API 註冊網域。
- 您嘗試建立之類型的工作流程執行或活動任務並未註冊。如果您嘗試在執行工作者之前建立工作流程執行，即可能發生此問題。由於工作者在第一次執行時註冊其類型，因此您必須在嘗試開始執行之前至少執行一次（或使用主控台或服務 API 手動註冊類型）。請注意，在類型註冊之後，即使未執行任何工作者，您還是可以建立執行。

- 工作者嘗試完成已逾時的任務。例如，如果工作者因處理任務的時間太長而逾時，則會在嘗試完成或讓任務失敗時收到 UnknownResource 故障。 AWS Flow Framework 工作者將繼續輪詢 Amazon SWF 並處理其他任務。不過，您應該考慮調整逾時。調整逾時需要您註冊新版本的活動類型。

在 Promise 上呼叫 get() 時的例外狀況

與 Java Future 不同，`Promise` 是一種非封鎖建構，對尚未就緒的 `Promise` 呼叫 `get()` 將會拋出例外狀況，而非封鎖。使用 的正確方法是將其`Promise`傳遞至非同步方法 （或任務），並在非同步方法中存取其值。 AWS Flow Framework for Java 可確保只有在傳遞給它的所有`Promise`引數都就緒時，才會呼叫非同步方法。如果您認為程式碼正確，或者您在執行其中一個 AWS Flow Framework 範例時遇到這種情況，則很可能是因為 AspectJ 未正確設定。如需詳細資訊，請參閱「設定 AWS Flow Framework 適用於 Java 的」一節。

非確定性工作流程

如「不確定性」一節所述，您工作流程的實作必須具有確定性。可能導致非確定性的一些常見錯誤為使用系統時鐘、使用亂數，以及產生 GUID。由於這些建構可能會在不同的時間傳回不同的值，因此每次執行工作流程時，工作流程的控制流程可能會採用不同的路徑 (了解 AWS Flow Framework 適用於 Java 的 中的任務如需詳細資訊，請參閱章節 AWS Flow Framework 基本概念：分散式執行和)。如果框架在執行工作流程時偵測到非確定性，即拋出例外狀況。

版本控制引起的問題

當您實作工作流程或活動的新版本時，例如，當您新增功能時，您應該使用適當的註釋來增加 類型的版本：`@Workflow`、 `@Activites`或 `@Activity`。在部署新版本的工作流程時，您現有的執行版本通常正在執行。因此，您需要確定具有適當工作流程和活動版本的工作者取得任務。達成此目標的方式是為每個版本使用一組不同的任務清單。例如，您可以在任務清單名稱的後方附加版本編號。如此能確保將屬於不同工作流程和活動版本的任務指派給適當的工作者。

對工作流程執行進行故障診斷和偵錯

對工作流程執行進行故障診斷的第一步是使用 Amazon SWF 主控台來查看工作流程歷史記錄。工作流程歷史記錄為一完整與可信賴的記錄，內含變更工作流程執行之執行狀態的所有事件。此歷史記錄由 Amazon SWF 維護，對於診斷問題非常寶貴。Amazon SWF 主控台可讓您搜尋工作流程執行並深入了解個別歷史記錄事件。

AWS Flow Framework 提供了一個`WorkflowReplayer`類別，您可以用來在本機重播工作流程執行並進行偵錯。使用此類別，您可以對已關閉和執行中的工作流程執行進行偵錯。 `WorkflowReplayer` 依賴存放在 Amazon SWF 中的歷史記錄來執行重播。您可以將其指向 Amazon SWF 帳戶中的工作流程執行，或將其提供歷史記錄事件 （例如，您可以從 Amazon SWF 擷取歷史記錄，並將其在本機序列化以供日後使用）。當您使用 `WorkflowReplayer` 重新執行工作流程執行時，不會影響您帳戶中執行的工作流程執行。重新執行完全在用戶端上執行。您可以對工作流程進行偵錯、建立中斷點，並如常使用偵錯工具來逐步執行程式碼。如果您使用的是 Eclipse，請考慮新增步驟篩選條件來篩選 AWS Flow Framework 套件。

例如，下列程式碼片段可以用來重新執行工作流程執行：

```
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 也可讓您取得工作流程執行的非同步執行緒傾印。此執行緒傾印可將所有開啟之非同步任務的呼叫堆疊提供給您。此資訊可以用來判斷執行中的哪些為待定且可能停擺的任務。例如：

```
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);
}
```

遺失的任務

有時您可能會關機工作者並接著啟動一連串的新工作者，卻發現任務都交付過去給舊的工作者。因為系統中的競爭條件分散到數個程序，所以可能發生這種情況。當您以緊密迴圈執行單位測試時，也可能會出現此問題。而在 Eclipse 中停止測試有時也可能會導致這種情況，因為有可能未呼叫關機處理器。

為了確定問題實際上是由舊工作者取得任務所造成，您應該查看工作流程歷史記錄，判斷哪個程序收到了您預期新工作者應收到的任務。例如，歷史記錄中的 `DecisionTaskStarted` 事件包含已收到任務的工作流程工作者的身分。Flow Framework 使用的 ID 格式為：{*processId*}@{*host name*}。例如，以下是 Amazon SWF 主控台中範例執行`DecisionTaskStarted`的事件詳細資訊：

事件時間戳記	Mon Feb 20 11:52:40 GMT-800 2012
Identity	2276@ip-0A6C1DF5
排程事件 ID	33

為了避免此情況，請為每個測試使用不同的任務清單。另外，請考慮新增關閉舊工作者與啟動新工作者之間的延遲。

由於 API 參數長度限制導致的驗證失敗

Amazon SWF 會對 API 參數強制執行長度限制。如果您的工作流程或活動實作超過限制條件，您將會收到`HTTP 400`錯誤。例如，當呼叫 `recordActivityHeartbeat` `ActivityExecutionContext`來傳送執行中活動的活動訊號時，字串長度不得超過 2048 個字元。

另一個常見案例是活動因例外狀況而失敗。框架會呼叫 ，RespondActivityTaskFailed並以序列化例外狀況做為詳細資訊，向 Amazon SWF 報告活動失敗。如果序列化例外狀況的長度大於 32，768 個位

元組，API 呼叫將報告 400 錯誤。若要緩解這種情況，您可以截斷例外狀況訊息或符合長度限制的原因。

AWS Flow Framework for Java 參考

主題

- AWS Flow Framework 適用於 Java 註釋
- AWS Flow Framework 適用於 Java 例外狀況
- AWS Flow Framework 適用於 Java 套件

AWS Flow Framework 適用於 Java 註釋

主題

- @Activities
- @Activity
- @ActivityRegistrationOptions
- @異步
- @Execute
- @ExponentialRetry
- @GetState
- @ManualActivityCompletion
- @Signal
- @SkipRegistration
- @Wait 和 @NoWait
- @工作流程
- @WorkflowRegistrationOptions

@Activities

此註釋可以用於界面，以宣告一組活動類型。標註此註釋之界面中的每個方法皆代表活動類型。界面不能同時有 `@Workflow` 和 `@Activities` 註釋。

在此註釋上可以指定下列參數：

`activityNamePrefix`

指定界面中所宣告之活動類型名稱的前綴。如果設定為空白字串 (預設值)，則會使用後接 '.' 的界面名稱做為前綴。

`version`

指定界面中所宣告之活動類型的預設版本。預設值為 `1.0`。

`dataConverter`

指定 `DataConverter` 的類型，以用於在建立此活動類型的任務和其結果時序列化/還原序列化資料。預設為 `NullDataConverter`，指出應該使用 `JsonDataConverter`。

@Activity

此註釋可以用於已標註 `@Activities` 之界面內的方法。

在此註釋上可以指定下列參數：

`name`

指定活動類型的名稱。預設值是空白字串，指出應該使用預設前綴和活動方法名稱來判斷活動類型的名稱 (格式為 {prefix}{name})。請注意，當您在 `@Activity` 註釋中指定名稱時，框架不會自動在名稱前面加上前綴。您可以自由使用自己的命名方式。

`version`

指定活動類型的版本。這會覆寫內含界面之 `@Activities` 註釋中所指定的預設版本。預設為空字串。

@ActivityRegistrationOptions

指定活動類型的註冊選項。此註釋可以用於已標註 `@Activities` 的界面或其內的方法。如果兩個位置皆指定，則用於方法上註釋會生效。

在此註釋上可以指定下列參數：

`defaultTasklist`

指定要向 Amazon SWF 註冊用於此活動類型的預設任務清單。使用 `ActivitySchedulingOptions` 參數在已產生的用戶端上呼叫活動方法時，可以覆寫此預設值。預設為 `USE_WORKER_TASK_LIST`。此特殊值指出應該使用執行註冊之工作者所使用的任務清單。

`defaultTaskScheduleToStartTimeoutSeconds`

指定此活動類型的向 Amazon SWF 註冊defaultTaskScheduleToStartTimeout。這是此活動類型的任務在指派給工作者之前，所允許等待的最長時間。如需詳細資訊，請參閱 Amazon Simple Workflow Service API 參考。

`defaultTaskHeartbeatTimeoutSeconds`

指定已向 Amazon SWF `defaultTaskHeartbeatTimeout`註冊的此活動類型。活動工作者必須在此持續時間內提供活動訊號；否則，任務將會逾時。預設為 -1，此特殊值指出應該停用這個逾時。如需詳細資訊，請參閱 Amazon Simple Workflow Service API 參考。

`defaultTaskStartToCloseTimeoutSeconds`

指定此活動類型的向 Amazon SWF 註冊的 defaultTaskStartToCloseTimeout。此逾時決定工作者可用來處理這類型之活動任務的最長時間。如需詳細資訊，請參閱 Amazon Simple Workflow Service API 參考。

`defaultTaskScheduleToCloseTimeoutSeconds`

指定已向 Amazon SWF `defaultScheduleToCloseTimeout`註冊的此活動類型。此逾時決定任務可保持開啟狀態的總持續時間。預設為 -1，此特殊值指出應該停用這個逾時。如需詳細資訊，請參閱 Amazon Simple Workflow Service API 參考。

@異步

用於工作流程協調性邏輯中的方法時，指出應該非同步執行方法。將立即傳回方法的呼叫，但會在傳遞給方法的所有 Promise<> 參數皆準備就緒時以非同步實際執行。已標註 @Asynchronous 的方法必須具有傳回類型 Promise<> 或 void。

`daemon`

指出針對非同步方法所建立的任務，是否應該為協助程式任務。預設為 `False`。

@Execute

用於已標註 `@Workflow` 註釋之界面中的方法時，識別工作流程的進入點。

> **Important**
>
> 界面中只有一個方法可以裝飾 `@Execute`。

在此註釋上可以指定下列參數：

`name`

指定工作流程類型的名稱。如果未設定，則名稱預設為 {prefix}{name}，其中 {prefix} 是後接 '.' 的工作流程界面名稱，而 {name} 是工作流程中 `@Execute` 裝飾方法的名稱。

`version`

指定工作流程類型的版本。

@ExponentialRetry

用於活動或非同步方法時，在方法拋出未處理的例外狀況時設定指數重試政策。重試嘗試是在退避期間之後進行，而退避期間是以嘗試次數的次方計算而來。

在此註釋上可以指定下列參數：

`intialRetryIntervalSeconds`

指定要在第一次重試嘗試之前等待的持續時間。此值不應該大於 `maximumRetryIntervalSeconds` 和 `retryExpirationSeconds`。

`maximumRetryIntervalSeconds`

指定重試嘗試之間的持續時間上限。達到之後，會將重試間隔設為此值。預設為 -1，這表示無限的持續時間。

`retryExpirationSeconds`

指定指數重試將在之後停止的持續時間。預設為 -1，這表示不會有過期的狀況。

`backoffCoefficient`

指定用來計算重試間隔的係數。請參閱 指數重試策略。

`maximumAttempts`

指定指數重試將在之後停止的嘗試次數。預設為 -1，這表示重試嘗試次數無限制。

`exceptionsToRetry`

指定應觸發重試的例外狀況類型清單。這些類型的未處理例外狀況將不會進一步傳播，而且將會在計算的重試間隔之後重試方法。此清單預設會包含 `Throwable`。

`excludeExceptions`

指定不應觸發重試的例外狀況類型清單。將允許傳播這類型的未處理例外狀況。此清單預設為空的。

@GetState

用於界面中已標註 `@Workflow` 註釋的方法時，識別使用的方法來擷取最新工作流程執行狀態。在具有 `@Workflow` 註釋的界面中，最多可以有一種方法具有此註釋。具有此註釋的方法不得採用任何參數，而且必須要有 `void` 以外的傳回類型。

@ManualActivityCompletion

此註釋可以用於活動方法，指出在傳回方法時不應該完成活動任務。活動任務不會自動完成，而且需要直接使用 Amazon SWF API 手動完成。這適用於下列使用案例：將活動任務指派給未自動化或需要人為介入才能完成的某個外部系統。

@Signal

用於界面中已標註 `@Workflow` 註釋的方法時，識別界面所宣告之工作流程類型的執行可以收到的訊號。需要使用此註釋來定義訊號方法。

在此註釋上可以指定下列參數：

`name`

指定訊號名稱的名稱部分。如果未設定，則會使用方法名稱。

@SkipRegistration

在註釋註釋的界面上使用 時`@Workflow`， 表示工作流程類型不應向 Amazon SWF 註冊。`@WorkflowRegistrationOptions` 和 `@SkipRegistrationOptions` 註釋中的其中一個必須用於已標註 `@Workflow` 的界面，但兩者不能同時使用。

@Wait 和 @NoWait

這些註釋可用於 類型的參數`Promise<>`，以指示 AWS Flow Framework 適用於 Java 的 是否應等待就緒，再執行方法。根據預設，傳入 `@Asynchronous` 方法的 `Promise<>` 參數必須先準備就緒，

再執行方法。在特定藍本中，需要覆寫此預設行為。傳入 @Asynchronous 方法且標註 @NoWait 的 Promise<> 參數則不會等待。

包含 Promise 的 Collections 參數 (或其子類別) (例如 List<Promise<Int>>) 必須標註 @Wait 註釋。根據預設，框架不會等待集合的成員。

@工作流程

此註釋用於界面，以宣告「工作流程」類型。已裝飾此註釋的界面只應該包含一個已裝飾 @Execute 註釋的方法，來宣告您工作流程的進入點。

> **Note**
>
> 界面不能同時宣告 @Workflow 和 @Activities 註釋；兩者會互斥。

在此註釋上可以指定下列參數：

dataConverter

指定要在將請求傳送至此工作流程類型的工作流程執行以及接收其結果時使用的 DataConverter。

預設為 NullDataConverter，其接著會回復為 JsonDataConverter，以將所有請求和回應資料都處理為「JavaScript 物件標記法 (JSON)」。

範例

```
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

在以 註釋的界面上使用 時`@Workflow`， 會在註冊工作流程類型時提供 Amazon SWF 使用的預設設定。

> **Note**
>
> `@WorkflowRegistrationOptions` 或 `@SkipRegistrationOptions` 必須用於已標註 `@Workflow` 的界面，但您不能同時指定兩者。

在此註釋上可以指定下列參數：

描述

工作流程類型的選用文字描述。

`defaultExecutionStartToCloseTimeoutSeconds`

指定向 Amazon SWF `defaultExecutionStartToCloseTimeout`註冊的工作流程類型。這是此類型的工作流程執行完成所需的總時間。

如需工作流程逾時的詳細資訊，請參閱「Amazon SWF 逾時類型」。

`defaultTaskStartToCloseTimeoutSeconds`

指定向 Amazon SWF `defaultTaskStartToCloseTimeout`註冊的工作流程類型。這指定此類型之工作流程執行的單一決策任務完成所需的時間。

如果您未指定 `defaultTaskStartToCloseTimeout`，則會預設為 30 秒。

如需工作流程逾時的詳細資訊，請參閱「Amazon SWF 逾時類型」。

`defaultTaskList`

用於此工作流程類型執行之決策任務的預設任務清單。啟動工作流程執行時，可以使用 `StartWorkflowOptions` 來覆寫這裡設定的預設值。

如果您未指定 `defaultTaskList`，則會預設為 `USE_WORKER_TASK_LIST`。這指出應該使用執行工作流程註冊之工作者所使用的任務清單。

`defaultChildPolicy`

指定要在終止這類型的執行時用於子工作流程的政策。預設值為 `ABANDON`。可能值如下：

- `ABANDON` – 允許子工作流程執行持續執行
- `TERMINATE` – 終止子工作流程執行
- `REQUEST_CANCEL` – 請求取消子工作流程執行

AWS Flow Framework 適用於 Java 例外狀況

適用於 Java AWS Flow Framework 的 使用下列例外狀況。本節說明例外狀況的概觀。如需詳細資訊，請參閱個別例外狀況 適用於 Java 的 AWS SDK 的文件。

主題

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

ActivityFailureException

框架會在內部使用此例外狀況，以溝通活動失敗。當活動因未處理的例外狀況而失敗時，會包裝`ActivityFailureException`並回報給 Amazon SWF。只有在您使用活動工作者擴充點時，才需要處理此例外狀況。您的應用程式碼永遠不需要處理此例外狀況。

ActivityTaskException

這是活動任務失敗例外狀況的基本類別：ScheduleActivityTaskFailedException、ActivityTaskFailedException、ActivityTasl
它包含失敗任務的任務 ID 和活動類型。您可以在工作流程實作中截獲此例外狀況，以使用通用方法來處理活動失敗。

ActivityTaskFailedException

拋出 ActivityTaskFailedException，以將活動中的未處理例外狀況回報給工作流程實作。可以從此例外狀況的 cause 屬性擷取原始例外狀況。例外狀況也會提供適用於進行偵錯的其他資訊，例如歷史記錄中的唯一活動識別符。

框架透過從活動工作者序列化原始例外狀況，可提供遠端例外狀況。

ActivityTaskTimedOutException

如果 Amazon SWF 逾時活動，則會擲出此例外狀況。無法在所需期間內將活動任務指派給工作者，或工作者在所需時間內無法完成活動任務，則可能會發生此狀況。呼叫活動方法時，您可以使用 @ActivityRegistrationOptions 註釋或使用 ActivitySchedulingOptions 參數在活動上設定這些逾時。

ChildWorkflowException

用來報告子工作流程執行失敗之例外狀況的基本類別。例外狀況包含子工作流程執行的 ID 以及其工作流程類型。您可以截獲此例外狀況，以使用通用方法來處理子工作流程執行失敗。

ChildWorkflowFailedException

拋出 ChildWorkflowFailedException，以將子工作流程中的未處理例外狀況回報給父工作流程實作。可以從此例外狀況的 cause 屬性擷取原始例外狀況。例外狀況也會提供適用於進行偵錯的其他資訊，例如子執行的唯一識別符。

ChildWorkflowTerminatedException

父工作流程執行中會拋出此例外狀況，以報告子工作流程執行的終止。如果您想要處理子工作流程終止 (例如，執行清理或補償)，則您應會截獲此例外狀況。

ChildWorkflowTimedOutException

父工作流程執行中擲出此例外狀況，以報告子工作流程執行已逾時，並由 Amazon SWF 關閉。如果您想要處理子工作流程的強制關閉 (例如，執行清理或補償)，則您應會截獲此例外狀況。

DataConverterException

框架使用 `DataConverter` 元件來封送處理和取消封送處理透過線路傳送的資料。如果 `DataConverter` 無法封送處理和取消封送處理資料，則會拋出此例外狀況。這可能會因各種原因而發生，例如，用來封送處理和取消封送處理資料的 `DataConverter` 元件中有不相符項目。

DecisionException

這是例外狀況的基本類別，表示無法由 Amazon SWF 制定決策。您可以截獲此例外狀況，以使用一般方式來處理這類例外狀況。

ScheduleActivityTaskFailedException

如果 Amazon SWF 無法排程活動任務，則會擲出此例外狀況。這可能是由於各種原因所造成：例如，活動已棄用，或您的帳戶已達到 Amazon SWF 限制。例外狀況中的 `failureCause` 屬性指出無法排定活動的確切原因。

SignalExternalWorkflowException

如果 Amazon SWF 無法處理工作流程執行的請求，以發出另一個工作流程執行的訊號，則會擲出此例外狀況。如果找不到目標工作流程執行，也就是您指定的工作流程執行不存在或處於關閉狀態，就會發生這種情況。

StartChildWorkflowFailedException

如果 Amazon SWF 無法啟動子工作流程執行，則會擲回此例外狀況。這可能是由於各種原因所造成：例如，指定的子工作流程類型已棄用，或您的帳戶已達到 Amazon SWF 限制。例外狀況中的 `failureCause` 屬性指出無法啟動子工作流程執行的確切原因。

StartTimerFailedException

如果 Amazon SWF 無法啟動工作流程執行請求的計時器，則會擲出此例外狀況。如果指定的計時器 ID 已在使用中，或您的帳戶已達到 Amazon SWF 限制，則可能會發生這種情況。例外狀況中的 `failureCause` 屬性指出失敗的確切原因。

TimerException

這是計時器相關例外狀況的基本類別。

WorkflowException

框架會在內部使用此例外狀況，以報告工作流程執行中的失敗。只有在您使用工作流程工作者擴充點時，才需要處理此例外狀況。

AWS Flow Framework 適用於 Java 套件

本節提供 AWS Flow Framework 適用於 Java 的 隨附的套件概觀。如需每個套件的詳細資訊，請參閱適用於 Java 的 AWS SDK API 參考中的 com.amazonaws.services.simpleworkflow.flow。

com.amazonaws.services.simpleworkflow.flow

包含與 Amazon SWF 整合的元件。

com.amazonaws.services.simpleworkflow.flow.annotations

包含 AWS Flow Framework 用於 Java 程式設計模型的註釋。

com.amazonaws.services.simpleworkflow.flow.aspectj

包含 AWS Flow Framework @異步和 等功能所需的 Java 元件@ExponentialRetry。

com.amazonaws.services.simpleworkflow.flow.common

包含通用公用程式，例如框架定義的常數。

com.amazonaws.services.simpleworkflow.flow.core

包含核心功能，例如 `Task` 和 `Promise`。

com.amazonaws.services.simpleworkflow.flow.generic

包含核心元件，例如其他功能建立依據的一般用戶端。

com.amazonaws.services.simpleworkflow.flow.interceptors

包含提供裝飾項目 (包括 `RetryDecorator`) 之框架的實作。

com.amazonaws.services.simpleworkflow.flow.junit

包含提供 Junit 整合的元件。

com.amazonaws.services.simpleworkflow.flow.pojo

包含實作註釋型程式設計模型之活動和工作流程定義的類別。

com.amazonaws.services.simpleworkflow.flow.spring

包含提供 Spring 整合的元件。

com.amazonaws.services.simpleworkflow.flow.test

包含單元測試工作流程實作的協助程式類別，例如 `TestWorkflowClock`。

com.amazonaws.services.simpleworkflow.flow.worker

包含活動和工作流程工作者的實作。

文件歷史記錄

下表說明自適用於 AWS Flow Framework Java 的 開發人員指南上次發行以來文件的重要變更。

- API 版本：2012-01-25
- 最新文件更新時間：2018 年 6 月 25 日

變更	描述	變更日期
更新	在 `backoffCoefficient` 的說明中為 `@ExponentialRetry` 修訂錯誤。請參閱 @ExponentialRetry。	2018 年 6 月 25 日
更新	清除整本指南的程式碼範例。	2017 年 6 月 5 日
更新	簡化並改善本指南的組織和內容。	2017 年 5 月 19 日
更新	簡化並改善「變更決策者程式碼：版本控制和功能標記」小節。	2017 年 4 月 10 日
更新	在「最佳實務」小節新增變更決策者程式碼的新指導方針。	2017 年 3 月 3 日
新功能	除了工作流程中的傳統活動任務之外，您還可以指定 Lambda 任務。如需詳細資訊，請參閱實作 AWS Lambda 任務。	2015 年 7 月 21 日
新功能	Amazon SWF 包含支援在任務清單上設定任務優先順序，嘗試在優先順序較低的任務之前交付優先順序較高的任務。如需詳細資訊，請參閱在 Amazon SWF 中設定任務優先順序。	2014 年 12 月 17 日
更新	進行更新和修正。	2013 年 8 月 1 日
更新	• 進行更新和修正，包括更新 Eclipse 4.3 和 適用於 Java 的 AWS SDK 1.4.7 的設定指示。	2013 年 6 月 28 日

變更	描述	變更日期
	• 新增一組建置入門藍本的新教學	
新功能	AWS Flow Framework 適用於 Java 的 初始版本。	2012 年 2 月 27 日

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。