# OSWorkFlow 入门指南

## 目的

这篇指导资料的目的是介绍 OSWorkflow 的所有概念，指导你如何使用它，并且保证你逐步理解 OSWorkflow 的关键内容。

本指导资料假定你已经部署 OSWorkflow 的范例应用在你的 container 上。范例应用部署是使用基于内存的数据存储，这样你不需要担心如何配置其他持久化的例子。范例应用的目的是为了说明如何应用 OSWorkflow，一旦你精通了 OSWorkflow 的流程定义描述符概念和要素，应该能通过阅读这些流程定义文件而了解实际的流程。

本指导资料目前有 3 部分：

1. 你的第一个工作流
2. 测试你的工作流
3. 更多的流程定义描述符概念

# 1. Your first workflow

## 创建描述符

首先，让我们来定义工作流。你可以使用任何名字来命名工作流。一个工作流对应一个 XML 格式的定义文件。让我们来开始新建一个“myworkflow.xml”的文件，这是样板文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE workflow PUBLIC
  "-//OpenSymphony Group//DTD OSWorkflow 2.7//EN"
  "http://www.opensymphony.com/osworkflow/workflow_2_7.dtd">
<workflow>
  <initial-actions>
    ...
  </initial-actions>
  <steps>
    ...
  </steps>
</workflow>
```

首先是标准的 XML 头部，要注意的是 OSWorkflow 将会通过这些指定的 DTD 来验证 XML 内容的合法性。你可以使用绝大多数的 XML 编辑工具来编辑它，并且可以 highlight 相应的错误。

## 步骤和动作

接下来我们来定义初始化动作和步骤。首先需要理解的 OSWorkflow 重要概念是 steps （步骤） 和 actions （动作）。一个步骤是工作流所处的位置，比如一个简单的工作流过程，它可能从一个步骤流转到另外一个步骤（或者有时候还是停留在一样的步骤）。举例来说，一个文档管理系统的流程，它的步骤名称可能有“First Draft - 草案初稿”， “Edit Stage -编辑阶段”， “At publisher - 出版商”等。

动作指定了可能发生在步骤内的转变，通常会导致步骤的变更。在我们的文件管理系统中，在“草案初稿”这个步骤可能有“start first draft - 开始草案初稿”和“complete first draft - 完成草案初稿”这样 2 个动作。

简单的说，步骤是“在哪里”，动作是“可以去哪里”。

初始化步骤是一种特殊类型的步骤，它用来启动工作流。在一个工作流程开始前，它是没有状态，不处在任何一个步骤，用户必须采取某些动作才能开始这个流程。这些特殊步骤被定义在 <initial-actions>。

在我们的例子里面，假定只有一个简单的初始化步骤：“Start Workflow”，它的定义在里面<initial-actions>：

```
<action id="1" name="Start Workflow">
  <results>
    <unconditional-result old-status="Finished" status="Queued"
step="1"/>
  </results>
</action>
```

这个动作是最简单的类型，只是简单地指明了下一个我们要去的步骤和状态。

## 工作流状态

工作流状态是一个用来描述工作流程中具体步骤状态的字符串。在我们的文档管理系统中，在“草案初稿”这个步骤可能有 2 个不同的状态：“Underway - 进行中”和“Queued - 等候处理中”

我们使用“Queued”指明这个条目已经被排入“First Draft”步骤的队列。比如说某人请求编写某篇文档，但是还没有指定作者，那么这个文档在“First

Draft”步骤的状态就是“Queued”。“Underway”状态被用来指明一个作者已经挑选了一篇文档开始撰写，而且可能正在锁定这篇文档。

## 第一个步骤

让我们来看第一个步骤是怎样被定义在<steps>元素中的。我们有 2 个动作：第一个动作是保持当前步骤不变，只是改变了状态到“Underway”，第二个动作是移动到工作流的下一步骤。我们来添加如下的内容到<steps>元素：

```
<step id="1" name="First Draft">
  <actions>
    <action id="1" name="Start First Draft">
      <results>
        <unconditional-result old-status="Finished"
        status="Underway" step="1"/>
      </results>
    </action>
    <action id="2" name="Finish First Draft">
      <results>
        <unconditional-result old-status="Finished"
        status="Queued" step="2"/>
      </results>
    </action>
  </actions>
</step>
<step id="2" name="finished" />
```

这样我们就定义了 2 个动作，old-status 属性是用来指明当前步骤完成以后的状态是什么，在大多数的应用中，通常用"Finished"表示。

上面定义的这 2 个动作是没有任何限制的。比如，一个用户可以调用 action 2 而不用先调用 action 1。很明显的，我们如果没有开始撰写草稿，是不可能去完成一个草稿的。同样的，上面的定义也允许你开始撰写草稿多次，这也是毫无意义的。我们也没有做任何的处理去限制其他用户完成别人的草稿。这些都应该需要想办法避免。

让我们来一次解决这些问题。首先，我们需要指定只有工作流的状态为“Queued”的时候，一个 caller （调用者）才能开始撰写草稿的动作。这样就可以阻止其他用户多次调用开始撰写草稿的动作。我们需要指定动作的约束，约束是由 Condition（条件）组成。

## 条件

OSWorkflow 有很多有用的内置条件可以使用。在此相关的条件是“StatusCondition - 状态条件”。 条件也可以接受参数，参数的名称通常被定义在 javadocs 里（如果是使用 Java Class 实现的条件的话）。在这个例子里面，状态条件接受一个名为“status”的参数，指明了需要检查的状态条件。我们可以从下面的 xml 定义里面清楚的理解：

```
<action id="1" name="Start First Draft">
  <restrict-to>
    <conditions>
      <condition type="class">
        <arg name="class.name">
          com.opensymphony.workflow.util.StatusCondition</arg>
        <arg name="status">Queued</arg>
      </condition>
    </conditions>
  </restrict-to>
  <results>
    <unconditional-result old-status="Finished" status="Underway"
step="1"/>
  </results>
</action>
```

希望对于条件的理解现在已经清楚了。上面的条件定义保证了动作 1 只能在当前状态为“Queued”的时候才能被调用，也就是说在初始化动作被调用以后。

## 函数

接下来，我们想在一个用户开始撰写草稿以后，设置他为“owner”。为了达到这样的目的，我们需要做 2 件事情：

1) 通过一个函数设置“caller”变量在当前的环境设置里。
2) 根据“caller”变量来设置“owner”属性。

函数是 OSWorkflow 的一个功能强大的特性。函数基本上是一个在工作流程中的工作单位，他不会影响到流程本身。举例来说，你可能有一个“SendEmail”的函数，用来在某些特定的流程流转发生时来发送 email 提醒。

函数也可以用来添加变量到当前的环境设置里。变量是一个指定名称的对象，可以用来在工作流中被以后的函数或者脚本使用。

OSWorkflow 提供了一些内置的常用函数。其中一个称为“Caller”，这个函数会获得当前调用工作流的用户，并把它放入一个名为“caller”的字符型变量中。

因为我们需要追踪是哪个用户开始了编写草稿，所以可以使用这个函数来修改我们的动作定义：

```
<action id="1" name="Start First Draft">
  <pre-functions>
    <function type="class">
      <arg name="class.name">
      com.opensymphony.workflow.util.Caller</arg>
    </function>
  </pre-functions>
  <results>
    <unconditional-result old-status="Finished"status="Underway"
    step="1" owner="${caller}"/>
  </results>
</action>
```

h3 组合起来
把这些概念都组合起来，这样我们就有了动作 1：

```
<action id="1" name="Start First Draft">
  <restrict-to>
    <conditions>
      <condition type="class">
        <arg name="class.name">
          com.opensymphony.workflow.util.StatusCondition
        </arg>
        <arg name="status">Queued</arg>
      </condition>
    </conditions>
  </restrict-to>
  <pre-functions>
    <function type="class">
      <arg name="class.name">
        com.opensymphony.workflow.util.Caller
      </arg>
    </function>
  </pre-functions>
  <results>
    <unconditional-result old-status="Finished"status="Underway"
    step="1"  owner="${caller}"/>
  </results>
</action>
```

我们使用类似想法来设置动作 2：

```
<action id="2" name="Finish First Draft">
  <restrict-to>
    <conditions type="AND">
      <condition type="class">
        <arg name="class.name">
          com.opensymphony.workflow.util.StatusCondition
        </arg>
        <arg name="status">Underway</arg>
      </condition>
      <condition type="class">
        <arg name="class.name">
          com.opensymphony.workflow.util.AllowOwnerOnlyCondition
         </arg>
      </condition>
    </conditions>
  </restrict-to>
  <results>
    <unconditional-result old-status="Finished" status="Queued"
step="2"/>
  </results>
</action>
```

在这里我们指定了一个新的条件：“allow owner only”。这样能够保证只有开始撰写这份草稿的用户才能完成它。而状态条件确保了只有在“Underway”状态下的流程才能调用“finish first draft”动作。

把他们组合在一起，我们就有了第一个流程定义：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE workflow PUBLIC
  "-//OpenSymphony Group//DTD OSWorkflow 2.7//EN"
  "http://www.opensymphony.com/osworkflow/workflow_2_7.dtd">
<workflow>
  <initial-actions>
    <action id="1" name="Start Workflow">
      <results>
        <unconditional-result old-status="Finished"
        status="Queued" step="1"/>
      </results>
    </action>
  </initial-actions>
  <steps>
    <step id="1" name="First Draft">
      <actions>
        <action id="1" name="Start First Draft">
```

```
            <restrict-to>
              <conditions>
                <condition type="class">
                  <arg name="class.name">
                     com.opensymphony.workflow.util.StatusCondition
                  </arg>
                  <arg name="status">Queued</arg>
                </condition>
              </conditions>
            </restrict-to>
            <pre-functions>
              <function type="class">
                <arg name="class.name">
                   com.opensymphony.workflow.util.Caller
                </arg>
              </function>
            </pre-functions>
            <results>
              <unconditional-result old-status="Finished"
status="Underway"
              step="1"  owner="${caller}"/>
            </results>
          </action>
          <action id="2" name="Finish First Draft">
            <restrict-to>
              <conditions type="AND">
                <condition type="class">
                  <arg name="class.name">
                      com.opensymphony.workflow.util.StatusCondition
                  </arg>
                  <arg name="status">Underway</arg>
                </condition>
                <condition type="class">
                  <arg name="class.name">
                    com.opensymphony.workflow.util.
                    AllowOwnerOnlyCondition
                  </arg>
                </condition>
              </conditions>
            </restrict-to>
            <results>
              <unconditional-result old-status="Finished"
              status="Queued" step="2"/>
            </results>
```

```
        </action>
      </actions>
    </step>
    <step id="2" name="finished" />
  </steps>
</workflow>
```

现在这个工作流的定义已经完整了，让我们来测试和检查它的运行。

# 2. Testing your workflow

现在我们已经完成了一个完整的工作流定义，下一步是检验它是否按照我们预想的方式执行。

在一个快速开发环境中，最简单的方法就是写一个测试案例。通过测试案例调用工作流，根据校验结果和捕捉可能发生的错误，来保证流程定义的正确性。

我们假设你已经熟悉 Junit 和了解怎样编写测试案例。如果你对这些知识还不了解的话，可以去 JUnit 的网站查找、阅读相关文档。编写测试案例会成为你的一个非常有用的工具。

在开始载入流程定义、调用动作以前，我们需要配置 OSWorkflow 的数据存储方式和定义文件的位置等。

### 配置 osworkflow.xml

我们需要创建的第一个文件是 *osworkflow.xml*。子：

```
<osworkflow>
  <persistence class="com.opensymphony.workflow.
  spi.memory.MemoryWorkflowStore"/>
  <factory
class="com.opensymphony.workflow.loader.XMLWorkflowFactory">
    <property key="resource" value="workflows.xml" />
  </factory>
</osworkflow>
```

这个例子指明了我们准备使用内存 (MemoryWorkflowStore) 来保存流程数据。这样可以减少设置数据库的相关信息，减少出问题的可能性。用内存持久化对于测试来说是非常方便的。

### Workflow factories

上面的配置文件还指明了我们工作流工厂（XMLWorkflowFactory），工作流工厂的主要功能是管理流程定义文件，包括读取定义文件和修改定义文件的功能。通过'resource'这个属性指明了采用通过从 classpath 中读取流程定义文件的方式，按照这个定义，接下来我们需要在 classpath 中创建一个名为 workflows.xml 的文件。

workflows.xml 的内容：

```
<workflows>
  <workflow name="mytest" type="resource" location="myworkflow.xml"/>
</workflows>
```

我们把 *myworkflow.xml* 和 workflows.xml 放在同一目录，这样它就能够被工作流工厂读取了。

这样就完成了配置，接下来是初始化一个流程并调用它。

## Initialising OSWorkflow

OSWorkflow 的调用模式相当简单：通过一个主要的接口来执行大部分操作。这个接口就是 *Workflow* interface，及其扩展 *AbstractWorkflow* 的实现，例如 EJBWorkflow 和 SOAPWorkflow. 为了简单起见，我们使用最基本的一种：BasicWorkflow。

首先，我们来创建 Workflow。在实际项目中，这个对象应该被放在一个全局的位置上以供重用，因为每次都创建一个新的 Workflow 对象是需要耗费比较昂贵的系统资源。在这里的例子，我们采用 BasicWorkflow，它的构建器由一个当前调用者的用户名构成，当然我们很少看到单用户的工作流应用，可以参考其他的 Workflow 实现有不同的方式去获得当前调用者。

为了简单起见，我们采用 BasicWorkflow 来创建一个单一的用户模式，避免编写其他获取用户方法的麻烦。

这样我们来创建一个'testuser'调用的 workflow:

```
Workflow workflow = new BasicWorkflow("testuser");
```

下一步是提供配置文件，在大多数情况下，只是简单的传递一个 DefaultConfiguration 实例：

```
DefaultConfiguration config = new DefaultConfiguration();
workflow.setConfiguration(config);
```

现在我们已经创建并且配置好了一个 workflow，接下来就是开始调用它了。

## 启动和进行一个工作流程

首先我们需要调用 initialize 方法来启动一个工作流程,这个方法有 3 个参数，workflow name （定义在 workflows.xml 里，通过 workflow factory 处理），action ID （我们要调用的初始化动作的 ID），和初始化变量。 因为在例子里面不需初始化变量，所以我们只是传递一个 null，

```
long workflowId = workflow.initialize("mytest", 1, null);
```

我们现在已经有了一个工作流实例，返回的 workflowId 可以在后续的操作中来代表这个实例。这个参数会在 Workflow interface 的绝大部分方法中用到。

## 检验工作流

现在让我们来检验启动的工作流实例是否按照我们所预期的那样运行。根据流程定义，我们期望的当前步骤是第一步，而且应该可以执行第一个动作（开始编写草稿）。

```
Collection currentSteps = workflow.getCurrentSteps
(workflowId);
//校验只有一个当前步骤
assertEquals("Unexpected number of current steps",
1, currentSteps.size());
//校验这个步骤是 1
Step currentStep = (Step)currentSteps.iterator().next();
assertEquals("Unexpected current step", 1,
currentStep.getStepId());

int[] availableActions =
workflow.getAvailableActions(workflowId);
//校验只有一个可执行的动作
assertEquals("Unexpected number of available actions", 1,
availableActions.length);
//校验这个步骤是 1
assertEquals("Unexpected available action", 1, availableActions[0]);
```

## 执行动作

现在已经校验完了工作流实例正如我们所期望的到了第一个步骤，让我们来开始执行第一个动作：

```
workflow.doAction(workflowId, 1, null);
```

这是简单的调用第一个动作，工作流引擎根据指定的条件，改变状态到‘Underway’，并且保持在当前步骤。

现在我们可以类似地调用第 2 个动作，它所需要的条件已经都满足了。

在调用完第 2 个动作以后，根据流程定义就没有可用的动作了，getAvailableActions 将会返回一个空数组。

Congratulations, 你已经完成了一个工作流定义并且成功地调用了它。下一节我们将会讲解 osworkflow 一些更深入的概念。

# 3. Further descriptor concepts

## 定义条件和函数

你也许已经注意到，到目前为止，我们定义的条件和函数类型都是“class”。这种类型的条件和函数接受一个参数：“class.name”，以此来指明一个实现 FunctionProvider 或 Condition 接口的完整类名。

在osworkflow里面也有一些其他内置的类型,包括beanshell,无状态的session bean，JNDI 树上的函数等。我们在下面的例子里使用 beanshell 类型。

### Property sets

我们可能需要在工作流的任意步骤持久化一些少量数据。在 osworkflow 里，这是通过 OpenSymphony 的 PropertySet library 来实现。一个 PropertySet 基本上是一个可以持久化的类型安全 map,你可以添加任意的数据到 propertyset(一个工作流实例对应一个 propertyset），并在以后的流程中再读取这些数据。除非你特别指定操作，否则 propertyset 中的数据不会被清空或者被删除。任意的函数和条件都可以和 propertyset 交互，以 beanshell script 来说，可以在脚本上下文中用“propertyset”这个名字来获取。下面来看具体写法是怎么样的，让我们增加如下的代码在“Start First Draft”动作的 pre-functions 里面：

```
<function type="beanshell">
  <arg name="script">propertySet.setString("foo", "bar")</arg>
</function>
```

这样我们就添加了一个持久化的属性“foo”，它的值是“bar”。这样在以后的流程中，我们就可以获得这个值了。

### Transient Map 临时变量

另外一个和 propertyset 变量相对的概念是临时变量：“transientVars”。临时变量是一个简单的 map，只是在当前的工作流调用的上下文内有效。它包括当前的工作流实例，工作流定义等对应值的引用。你可以通过 FunctionProvider 的 javadoc 来查看这个 map 有那些可用的 key。

还记得我们在教程的第 2 部分传入的那个 null 吗？如果我们不传入 null 的话，那么这些输入数据将会被添加到临时变量的 map 里。

## inputs 输入

每次调用 workflow 的动作时可以输入一个可选的 map，可以在这个 map 里面包含供函数和条件使用的任何数据，它不会被持久化，只是一个简单的数据传递。

## Validators 校验器

为了让工作流能够校验输入的数据，引入了校验器的概念。一个校验器和函数，条件的实现方式非常类似（比如，它可以是一个 class，脚本，或者 EJB）。在这个教程里面，我们将会定义一个校验器，在“finish first draft”这个步骤，校验用户输入的数据“working.title”不能超过 30 个字符。这个校验器看起来是这样的：

```
package com.mycompany.validators;

public class TitleValidator implements Validator
{
  public void validate(Map transientVars, Map args,
  PropertySet ps)
        throws InvalidInputException, WorkflowException
  {
    String title =
    (String)transientVars.get("working.title");
    if(title == null)
      throw new InvalidInputException("Missing working.title");
    if(title.length() > 30)
      throw new InvalidInputException("Working title too long");
  }
}
```

然后通过在流程定义文件添加 validators 元素，就可以登记这个校验器了：

```
<validators>
  <validator type="class">
    <arg name="class.name">
```

```
        com.mycompany.validators.TitleValidator
      </arg>
    </validator>
</validators>
```

这样，当我们执行动作 2 的时候，这个校验器将会被调用，并且检验我们的输入。这样在测试代码里面，如果加上：

```
Map inputs = new HashMap();
inputs.put("working.title",
  "the quick brown fox jumped over the lazy dog," +
  " thus making this a very long title");
workflow.doAction(workflowId, 2, inputs);
```

我们将会得到一个 InvalidInputException，这个动作将不会被执行。减少输入的 title 字符，将会让这个动作成功执行。

我们已经介绍了输入和校验，下面来看看寄存器。

## Registers 寄存器

寄存器是一个工作流的全局变量。和 propertyset 类似，它可以在工作流实例的任意地方被获取。和 propertyset 不同的是，它不是一个持久化的数据，而是每次调用时都需要重新计算的数据。

它可以被用在什么地方呢？在我们的文档管理系统里面，如果定义了一个“document”的寄存器，那么对于函数、条件、脚本来说就是非常有用的：可以用它来获得正在被编辑的文档。

寄存器地值会被放在临时变量（transientVars map）里，这样能够在任意地方获得它。

定义一个寄存器和函数、条件的一个重要区别是，它并不是依靠特定的调用（不用关心当前的步骤，或者是输入数据，它只是简单地暴露一些数据而已），所以它不用临时变量里的值。

寄存器必须实现 Register 接口，并且被定义在流程定义文件的头部，在初始化动作之前。

举例来说，我们将会使用一个 osworkflow 内置的寄存器:LogRegister。这个寄存器简单的添加一个“log”变量，能够让你使用 Jakarta 的 commons-logging 输出日志信息。它的好处是会在每条信息前添加工作流实例的 ID。

```
<registers>
```

```
  <register type="class" variable-name="log">
    <arg name="class.name">
      com.opensymphony.workflow.util.LogRegister
    </arg>
    <arg name="addInstanceId">true</arg>
    <arg name="Category">workflow</arg>
  </register>
</registers>
```

这样我们定义了一个可用的“log”变量，可以通过其他的 pre－function 的脚本里面使用它：

```
<function type="beanshell">
  <arg name="script">transientVars.get("log").info("executing action
2")
  </arg>
</function>
```

日志输出将会在前面添加工作流实例的 ID

## 结论

这个教程的目的是希望可以阐明一些主要的 osworkflow 概念。你还可以通过 API 和流程定义格式去获取更多的信息。有一些更高级的特性没有在此提到，比如 splits 分支、joins 连接，nested conditions 复合条件、auto stpes 自动步骤等等。你可以通过阅读手册来获得更进一步的理解。

# OSWORKFLOW 分析

收藏：

OSWorkFlow 分析

1. OSWorkFlow 基本概念

在商用和开源世界里，OSWorkflow 都不同于这些已有的工作流系统。最大不同在于 OSWorkflow 有着非常优秀的灵活性。在开始接触 OSWorkflow 时可能较难掌握（有人说不适合工作流新手入门），比如，OSWorkflow 不要求图形化工具来开发工作流，而推荐手工编写 xml 格式的工作流程描述符。它能为应用程序开发者提供集成，也能与现有的代码和数据库进行集成。这一切似乎给正在寻找快速"即插即用"工作流解决方案的人制造了麻烦，但研究发现，那些"即插即用"方案也不能在一个成熟的应用程序中提供足够的灵活性来实现所有需求。

2. OSWorkFlow 主要优势

OSWorkflow 给你绝对的灵活性。OSWorkflow 被认为是一种"低级别"工作流实现。与其他工作流系统能用图标表现"Loops(回路)"和"Conditions(条件)"相比，OSWorkflow 只是手工"编码(Coded)"来实现的。但这并不能说实际的代码是需要完全手工编码的，脚本语言能胜任这种情形。 OSWorkflow 不希望一个非技术用户修改工作流程，虽然一些其他工作流系统提供了简单的 GUI 用于工作流编辑，但像这样改变工作流，通常会破坏这些应用。所以，进行工作流调整的最佳人选是开发人员，他们知道该怎么改变。不过，在最新的版本中，OSWorkflow 也提供了 GUI 设计器来协助工作流的编辑。

OSWorkflow 基于有限状态机概念。每个 state 由 step ID 和 status 联合表现（可简单理解为 step 及其 status 表示有限状态机的 state）。一个 state 到另一 state 的 transition 依赖于 action 的发生，在工作流生命期内有至少一个或多个活动的 state。这些简单概念展现了 OSWorkflow 引擎的核心思想，并允许一个简单 XML 文件解释工作流业务流程。

3. OSWorkFlow 核心概念

3.1. 概念定义

步骤（Step）

一个 Step 描述的是工作流所处的位置。可能从一个 Step Transtion（流转）到另外一个 Step，或者也可以在同一个 Step 内流转（因为 Step 可以通 Status 来细分，形成多个 State）。一个流程里面可以多个 Step。

状态（Status）

工作流 Status 是用来描述工作流程中具体 Step（步骤）状态的字符串。OSWorkflow 的有 Underway（进行中）、Queued （等候处理中）、Finished（完成）三种 Status。一个实际 State（状态）真正是由两部分组成：State = (Step + Status) 。

流转（Transtion）

一个 State 到另一个 State 的转移。

动作（Action）

Action 触发了发生在 Step 内或 Step 间的流转，或者说是基于 State 的流转。一

个 step 里面可以有多个 Action。Action 和 Step 之间的关系是，Step 说明“在哪里”，Action 说明“去哪里”。 一个 Action 典型地由两部分组成：可以执行此 Action （动作）的
Condition（条件），以及执行此动作后的 Result（结果）。

条件（Condition）
类似于逻辑判断，可包含“AND”和“OR”逻辑。比如一个请假流程中的“本部门审批阶段”，该阶段利用“AND”逻辑，判断流程状态是否为等候处理中，以及审批者是否为本部门主管。

结果（Result）
Result 代表执行 Action（动作）后的结果，指向新的 Step 及其 Step Status，也可能进入 Split 或者 Join。Result 分为两种， Contidional-Result （有条件结果），只有条件为真时才使用该结果，和 Unconditional-Result（无条件结果），当条件不满足或没有条件时使用该结果。

分离/连接（Split/Join）
流程的切分和融合。很简单的概念，Split 可以提供多个 Result（结果）；Join 则判断多个 Current Step 的态提供一个 Result（结果）。

### 3.2. 步骤、状态和动作(Step, Status, and Action)

工作流要描述步骤(Step)、步骤的状态(Status)、各个步骤之间的关系以及执行各个步骤的条件和权限，每个步骤中可以含有一个或多个动作(Action)，动作将会使一个步骤的状态发生改变。

对于一个执行的工作流来讲，步骤的切换是不可避免的。一个工作流在某一时刻会有一个或多个当前步骤，每个当前步骤都有一个状态值，当前步骤的状态值组成了工作流实例的状态值。一旦完成了一个步骤，那么这个步骤将不再是当前步骤（而是切换到一个新的步骤），通常一个新的当前步骤将随之建立起来，以保证工作流继续执行。完成了的步骤的最终状态值是用 Old-Status 属性指定的，这个状态值的设定将发生在切换到其他步骤之前。Old-Status 的值可以是任意的，但在一般情况下，我们设置为 Finished。

切换本身是一个动作（Action）的执行结果。每个步骤可以含有多个动作，究竟要载入哪个动作是由最终用户、外部事件或者 Tiggerd 的自动调用决定的。随着动作的完成，一个特定的步骤切换也将发生。动作可以被限制在用户、用户组或当前状态。每一个动作都必须包含一个 Unconditional Result 和 0 个或多个 Conditional Results。

所以，总体来说，一个工作流由多个步骤组成。每个步骤有一个当前状态（例如：Queued, Underway or Finished），一个步骤包含多个动作。每个步骤含有多个可以执行的动作。每个动作都有执行的条件，也有要执行的函数。动作包含有可以改变状态和当前工作流步骤的 results。

### 3.3. 结果、分支和连接(Results, Joins, and Splits)

#### 3.3.1. 无条件结果(Unconditional Result)

对于每一个动作来讲，必须存在一个 Unconditional Result。一个 result 是一系列指令，这些指令将告诉 OSWorkFlow 下一个任务要做什么。这包括使工作流从一个状态切换到另一个状态。

### 3.3.2. 有条件结果(Conditional Result)

Conditional Result 是Unconditional Result的一个扩展。它需要一个或多个Condition子标签。第一个为 true 的Conditional（使用 AND 或 OR 类型），会指明发生切换的步骤，这个切换步骤的发生是由于某个用户执行了某个动作的结果导致的。

### 3.3.3. 三种不同的 Results(conditional or unconditional)

一个新的、单一的步骤和状态的组合。

一个分裂成两个或多个步骤和状态的组合。

将这个和其他的切换组合成一个新的单一的步骤和状态的组合。

每种不同的 result 对应了不同的 xml 描述，你可以阅读 http://www.opensymphony.com/osworkflow/workflow_2_7.dtd，获取更多的信息。

注意：通常，一个 split 或一个 join 不会再导致一个 split 或 join 的发生。

## 3.4. 自动步骤(Auto actions)

有的时候，我们需要一些动作可以基于一些条件自动地执行。为了达到这个目的，你可以在 action 中加入 auto="true"属性。流程将考察这个动作的条件和限制，如果条件符合，那么将执行这个动作。 Auto action 是由当前的调用者执行的，所以将对该动作的调用者执行权限检查。

## 3.5. 整合抽象实例(Integrating with Abstract Entities)

建议在你的核心实体中，例如"Document" 或 "Order"，在内部创建一个新的属性：workflowId。这样，当新的 "Document" 或 "Order"被创建的时候，它能够和一个 workflow 实例关联起来。那么，你的代码可以通过 OSWorkflow API 查找到这个 workflow 实例并且得到这个 workflow 的信息和动作。

## 3.6. 工作流实例状态(Workflow Instance State)

有的时候，为整个 workflow 实例指定一个状态是很有帮助的，它独立于流程的执行步骤。OSWorkflow 提供一些 workflow 实例中可以包含的 "meta-states"。这些 "meta-states" 可以是 CREATED, ACTIVATED, SUSPENDED, KILLED 和 COMPLETED。当一个工作流实例被创建的时候，它将处于 CREATED 状态。然后，只要一个动作被执行，它就会自动的变成 ACTIVATED 状态。如果调用者没有明确地改变实例的状态，工作流将一直保持这个状态直到工作流结束。当工作流不可能再执行任何其他的动作的时候，工作流将自动的变成 COMPLETED 状态。

然而，当工作流处于 ACTIVATED 状态的时候，调用者可以终止或挂起这个工作流（设置工作流的状态为 KILLED 或 SUSPENDED）。一个终止了的工作流将不能再执行任何动作，而且将永远保持着终止状态。一个被挂起了的工作流会被冻结，他也不能执行任何的动作，除非它的状态再变成 ACTIVATED。

# 4. OSWorkFlow 包用途分析及代码片断

## 4.1. com.opensymphony.workflow

该包为整个 OSWorkflow 引擎提供核心接口。例如 com.opensymphony.workflow.Workflow 接口，可以说，实际开发中的大部分工作都是围绕该接口展开的，该接口有 BasicWorkflow、EJBWorkflow、OfbizWorkflow 三个实现类。

## 4.2. com.opensymphony.workflow.basic

该包有两个类，BasicWorkflow 与 BasicWorkflowContext。BasicWorkflow 不支持事

务，尽管依赖持久实现，事务也不能包裹它。BasicWorkflowContext 在实际开发中很少使用。

```
public void setWorkflow(int userId) {
    Workflow workflow = new BasicWorkflow(Integer.toString(userId));
}
```

### 4.3. com.opensymphony.workflow.config

该包有一个接口和两个该接口的实现类。在 OSWorkflow 2.7 以前，状态由多个地方的静态字段维护，这种方式很方便，但是有很多缺陷和约束。最主要的缺点是无法通过不同配置运行多个 OSWorkflow 实例。实现类 DefaultConfiguration 用于一般的配置文件载入。而 SpringConfiguration 则是让 Spring 容器管理配置信息。

```
public void setWorkflow(int userId) {
    Workflow workflow = new BasicWorkflow(Integer.toString(userId));
}
```

### 4.4. com.opensymphony.workflow.ejb

该包有两个接口 WorkflowHome 和 WorkflowRemote。该包的若干类中，最重要的是 EJBWorkflow，该类和 BasicWorkflow 的作用一样，是 OSWorkflow 的核心，并利用 EJB 容器管理事务，也作为工作流 session bean 的包装器。

### 4.5. com.opensymphony.workflow.loader

该包有若干类，用得最多的是 XxxxDescriptor，如果在工作流引擎运行时需要了解指定的动作、步骤的状态、名字，等信息时，这些描述符会起到很大作用。

```
public String findNameByStepId(int stepId,String wfName) {
    WorkflowDescriptor wd = workflow.getWorkflowDescriptor(wfName);
    StepDescriptor stepDes = wd.getStep(stepId);
    return stepDes.getName();
}
```

### 4.6. com.opensymphony.workflow.ofbiz

OfbizWorkflow 和 BasicWorkflow 在很多方面非常相似，除了需要调用 ofbiz 的 TransactionUtil 来包装事务。

### 4.7. com.opensymphony.workflow.query

该包主要为查询而设计，但不是所有的工作流存储都支持查询。通常，Hibernate 和 JDBC 都支持，而内存工作流存储不支持。值得注意的是 Hibernate 存储不支持混合型查询（例如，一个查询同时包含了 history step 上下文和 current step 上下文）。执行一个查询，需要创建 WorkflowExpressionQuery 实例，接着调用 Workflow 对象的 query 方法来得到最终查询结果。

```
public List queryDepAdmin(int userId,int type) {
    int[] arr = getSubPerson(userId,type);

    //构造表达式
    Expression[] expressions = new Expression[1 + arr.length];
```

```
    Expression expStatus = new FieldExpression(FieldExpression.STATUS,
    FieldExpression.CURRENT_STEPS, FieldExpression.EQUALS, "Queued");
    expressions[0] = expStatus;

    for (int i = 0; i < arr.length; i++) {
        Expression expOwner = new FieldExpression(FieldExpression.OWNER,
        FieldExpression.CURRENT_STEPS, FieldExpression.EQUALS,
        Integer.toString(arr[i]));
        expressions[i + 1] = expOwner;
    }

    //查询未完成流编号
    List wfIdList = null;
    try {
        WorkflowExpressionQuery query = new WorkflowExpressionQuery(
        new NestedExpression(expressions, NestedExpression.AND));
        wfIdList = workflow.query(query);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

4.8. com.opensymphony.workflow.soap

OSWorkflow 通过 SOAP 来支持远端调用。这种调用借助 WebMethods 实现。

4.9. com.opensymphony.workflow.spi

该包可以说是 OSWorkflow 与持久层打交道的途径，如当前工作流的实体，其中包括：EJB、Hibernate、JDBC、Memory、Ofbiz、OJB、Prevayler。

```
HibernateWorkflowEntry hwfe = (HibernateWorkflowEntry) getHibernateTempl
ate()
    .find("from HibernateWorkflowEntry where Id="
        + wfIdList.get(i)).get(0);
```

4.10. com.opensymphony.workflow.util

该包是 OSWorkflow 的工具包，包括了对 BeanShell、BSF、EJB Local、EJB Remote、JNDI 的支持。

5. OSWorkFlow 表结构分析

5.1. OS_WFENTRY

工作流主表，存放工作流名称和状态

| 字段名 | 数据类型 | 说明 |
| --- | --- | --- |
| ID | NUMBER | 自动编号 |
| NAME | VARCHAR2(20) | 工作流名称 |
| STATE | NUMBER | 工作流状态 |

5.2. OS_CURRENTSTEP

当前步骤表，存放当前正在进行步骤的数据

| 字段名 | 数据类型 | 说明 |
| --- | --- | --- |
| ID | NUMBER | 自动编号 |
| ENTRY_ID | NUMBER | 工作流编号 |
| STEP_ID | NUMBER | 步骤编号 |
| ACTION_ID | NUMBER | 动作编号 |
| OWNER | VARCHAR2(20) | 步骤的所有者 |
| START_DATE | DATE | 开始时间 |
| FINISH_DATE | DATE | 结束时间 |
| DUE_DATE | DATE | 授权时间 |
| STATUS | VARCHAR2(20) | 状态 |
| CALLER | VARCHAR2(20) | 操作人员的帐号名称 |

### 5.3. OS_CURRENTSTEP_PREV

前步骤表，存放当前步骤和上一个步骤的关联数据

| 字段名 | 数据类型 | 说明 |
| --- | --- | --- |
| ID | NUMBER | 当前步骤编号 |
| PREVIOUS | NUMBER | 前步骤编号 |

### 5.4. OS_HISTORYSTEP

历史步骤表，存放当前正在进行步骤的数据

| 字段名 | 数据类型 | 说明 |
| --- | --- | --- |
| ID | NUMBER | 自动编号 |
| ENTRY_ID | NUMBER | 工作流编号 |
| STEP_ID | NUMBER | 步骤编号 |
| ACTION_ID | NUMBER | 动作编号 |
| OWNER | VARCHAR2(20) | 步骤的所有者 |
| START_DATE | DATE | 开始时间 |
| FINISH_DATE | DATE | 结束时间 |
| DUE_DATE | DATE | 授权时间 |
| STATUS | VARCHAR2(20) | 状态 |
| CALLER | VARCHAR2(20) | 操作人员的帐号名称 |

### 5.5. OS_HISTORYSTEP_PREV

前历史步骤表，存放历史步骤和上一个步骤的关联数据

| 字段名 | 数据类型 | 说明 |
| --- | --- | --- |
| ID | NUMBER | 当前历史步骤编号 |
| PREVIOUS | NUMBER | 前历史步骤编号 |

### 5.6. OS_PROPERTYENTRY

属性表，存放临时变量

| 字段名 | 数据类型 | 说明 |
| --- | --- | --- |
| GLOBAL_KEY | VARCHAR2(255) | 全局关键字 |
| ITEM_KEY | VARCHAR2(255) | 条目关键字 |
| ITEM_TYPE | NUMBER | 条目类型 |
| STRING_VALUE | VARCHAR2(255) | 字符值 |
| DATE_VALUE | DATE | 日期值 |
| DATA_VALUE | BLOB | 数据值 |
| FLOAT_VALUE | FLOAT | 浮点值 |
| NUMBER_VALUE | NUMBER | 数字值 |