OpenDoc Series'

OSWorkflow
中文手册

V2.8

作者：OSWorkflow Team
翻译：陈刚

文档说明

参与人员：

译者	联络
陈刚	cucuchen520(at)yahoo.com.cn

(at) 为 email @ 符号

发布记录

版本	日期	作者	说明
2.8	2007.07.20	陈刚	翻译
2.8	2007.07.20	夏昕	文档格式编排
2.8	2007.08.14	曹晓钢	编辑、校对
2.8	2007.08.19	徐明明	审校

OpenDoc 版权说明

Open Doc Series 目前包括以下几份文档：

- **Spring 开发指南**
- **Hibernate 开发指南**
- **ibatis2 开发指南**
- **Webwork2 开发指南**
- **持续集成实践之 CruiseControl**
- **Using the Rake Build Language 等**

以上文档可从http://www.redsaga.com获取最新更新信息

1 开始教程

1.1 简介

OSWorkflow 与其它大多数工作流非常不同，不论是商用的还是开源的。它的不同之处就在用它极其灵活(extremely flexible)。然而，这也使得我们很难掌握它。举个例子：OSWorkflow 没有好的可视化工具来开发流程，这就意味着我们要手工书写和定义这些 XML 流程描述文件。这需要应用开发者具备一定的勇气，就类似于有勇气写代码或者配置数据库一样。有些人希望寻找一个快速的”即插即用(plug-and-play)”工作流解决方案，但最终发现这样的解决方案没有提供足够的灵活度，从而不能在一个完善的系统中实现所有的需求。

OSWorkflow 提供这样的灵活(OSWorkflow gives you this flexibility)

我们认为 OSWorkflow 是一个”低端”(low level)工作流实现。比如说循环（loops）和条件(conditions)在其它的工作流里面被抽象成是可视化的图标，在 OSWorkflow 里面却是“代码”。这并不是要求你来实现真正的代码，而是用脚本语言去定义这些条件。我们不希望那些不懂技术的用户去修改流程。有些系统提供了 GUI，可以简单地编辑一些流程，但当客户自己运用 GUI 的时候，这些流程最终往往被修改和破坏了。我们相信最好的方式是能让开发者知道这些改变。虽然如此，OSWorkflow 最新的版本也提供 GUI 来辅助开发者编辑流程。

OSWorkflow主要基于有限状态机（*finite state machine*）。每一种状态(state)被描述成为step ID和status。从一种**状态(state)**转移到另一种状态没有动作(action)是不可能发生的。在工作流的生命周期内通常有一个或者多个活动的状态。这些简单的思想表现在OSWorkflow引擎核心包里面，并且通过用一个简单的XML文件来描述业务工作流程。

1.2 必要文件

在 OSWorkflow 发布版本中几乎包含了所有需要的包：

- OSCore 2.0.1及以上版本
- PropertySet 1.2及以上版本
- Jakarta commons-logging
- BeanShell (可选)
- BSF (可选)
- EJB interfaces (不一定需要 EJB 容器)
- XML parser (对于 JDK 1.4 来说是不需要的)

OSWorkflow 的核心 API 可以在 JDK1.3 及以上环境中运行。然而 GUI 设计器则需要 JDK1.4 及以上版本才能工作。

注意，关于SOAP和工作调度(job scheduling)：GLUE是OSWorkflow使用的一种SOAP实现方式。你也可以在OSWorkflow2.8 版中使用XFire。GLUE可以方便地从WebMethods得到。如果你想得到SOAP支持或者**远程**工作调度支持，你必须下载**GLUE专业版**的包(**GLUE Professional** libraries)。XFire 是属于codehaus的一个开源项目。

作为对GLUE的补充，你需要Quartz 来进行任务调度（job scheduling）。如果你不喜欢GLUE或者Quartz，你也可以通过使用OSWorkflow API来非常容易地提供其他实现，如果在你的应用服务器内部运行Quartz，或者任何正确配置了OSWorkflow的地方运行Quartz，你不一定需要Quartz，但是你必须要将JobScheduler的local参数设置成true。

除这些包以外，对于你所选择的不同WorkflowStore，其所依赖的包也不同，要想知道更多相关信息请阅读文档的 **1.4 章节：Persistence的选择**。对于有些工具方法可能需要其它别的包。例如，如果你使用**OSUserGroupCondition**，必须导入OSUser包。

1.3 运行示例

注意自 2.5 版开始，war 示例部署到大多数 web 容器里都不需要外部配置即可运行。现在的例子使用的是内存式持久化 (memory persistence)，所以不需要配置数据源。下面的介绍则是为那些需要配置持久化存储(persistent store)的人而准备的。为了逐步弄清楚工作流示例，推荐你第一次部署例子的时候就按照它本来的样子(memory persistence)进行部署，不要加入持久化。但 ear 示例就要使用 EJB 持久化，所以需要完整的 EJB 应用服务器。

下面的介绍只对你部署基于持久化例子来说才是有效的。如果你第一次运行示例，强烈推荐你不要使用持久化，直接部署示例就好了。

OSWorkflow 附带的是个简单的示例，用于帮助你理解它的原理。这个例子可以在任何一种 servlet 容器上运行。如果要配置持久化数据源，本章下面的指南是由 Ed Yu 和 Egor Kobylkin 所提供的对于几种应用服务器的配置方式。如果你使用的容器没有列出，请选择某个类似的指南，安装工作将是非常相似的。

- Orion
- JRun
- Tomcat 4.0.x
- Tomcat 4.1.x

Orion

这里大体描述一下怎样将 osworkflow_example.war 部署到 Orion 1.5.4 或更高版本上面去。

开始持久化

要获得良好的性能，配置一个池化数据源是非常重要的。
对于 Orion 1.5.4 或者更高版本，增加下面的配置到 config/data-sources.xml 这个文件里面。这个数据源将会是 OSWorkfow 所需要的。

```
<!-- Postgresql Pooled Datasource ->
        <data-source
```

```
                class="com.evermind.sql.DriverManagerDataSource"
                name="NoPoolPostgresql"
                location="jdbc/NoPoolDefaultDS"
                xa-location="jdbc/xa/NoPoolDefaultXADS"
                connection-driver="org.postgresql.Driver"
                username="name"
                password="pass"
                url="jdbc:postgresql://hostname:port/dbname"
                inactivity-timeout="30"
        />
        <data-source
                class="com.evermind.sql.OrionPooledDataSource"
                name="PooledPostgresql"
                location="jdbc/DefaultDS"
                pooled-location="jdbc/DefaultDS"
                xa-location="jdbc/xa/DefaultXADS"
                source-location="jdbc/NoPoolDefaultDS"
                connection-driver="org.postgresql.Driver"
                username="name"
                password="pass"
                url="jdbc:postgresql://hostname:port/dbname"
                inactivity-timeout="30"
        />

        <!- Oracle Pooled Datasource ->
        <data-source
                class="oracle.jdbc.pool.OracleDataSource"
                name="Default"
                location="jdbc/DefaultDS"
                xa-location="jdbc/xa/DefaultXADS"
                ejb-location="jdbc/DefaultDS"
                connection-driver="oracle.jdbc.driver.OracleDriver"
                username="name"
                password="pass"
                url="jdbc:oracle:thin:@hostname:port:SID"
                inactivity-timeout="30"
        />
```

另外，针对你的数据库，还要修改 osworkflow.xml 里面的 sequence。

如果你使用 postgresql 数据库，请执行 tables_postgres.sql。请确认你的 osworkflow.xml 文件中包含如下配置：

```
<!- Postgresql sequence access ->
  <property key="entry.sequence"
                value="SELECT nextVal('seq_os_wfentry')"/>
  <property key="step.sequence"
```

```
                value="SELECT nextVal('seq_os_currentsteps')"/>
```

如果你使用 oracle 数据库，请执行 tables_oracle.sql。请确认你的 osworkflow.xml 文件中包含如下配置：

```
<!- Oracle sequence access ->
        <property key="entry.sequence"
                value="SELECT seq_os_wfentry.nextval FROM dual"/>
        <property key="step.sequence"
                value="SELECT seq_os_currentsteps.nextval FROM dual"/>
```

WAR 部署

在 orion 服务器根目录下创建一个文件夹(如：C:\orion\oswf)，将 osworkflow_example.war 放到它的下面。

修改 config/application.xml，添加下面的配置：

```
<web-module id="oswf" path="../oswf/osworkflow_example.war" />
```

修改 config/default-web-site.xml，添加下面的配置：

```
    <web-app application="default" load-on-startup="true"
                name="oswf" root="/oswf" />
```

最后到http://localhost/oswf/去观看示例。

你也可以部署 ear 示例程序，它使用 EJB persistence 来代替例子程序中使用的内存持久化。

JRun

这里大体描述一下怎样将 osworkflow_example.war 部署到 JRun4 上面去。

开始持久化

要获得良好的性能，配置一个池化数据源是非常重要的。

对于 JRun4，当使用 JMC 定义 JDBC 资源的时候是非常容易配置数据库连接池的。启动 admin server 和 default server，然后使用 JMC，在 default server(端口：8100)中定义 JDBC 数据源。当添加数据源时要把“pool connection”复选框勾选上。添加后记得检查一次数据源。

```
JNDI Name                   jdbc/DefaultDS
Driver Class Name org.postgresql.Driver
URL                         jdbc:postgresql://hostname:port/dbName
Description                 OSWorkflow example Postgresql database.
Pool Connections  X
Pool Statements             X
Native Results              X
User Name                   user
Password          pass
Verify Password   pass      pass

JNDI Name                   jdbc/DefaultDS
Driver Class Name oracle.jdbc.pool.OracleDataSource
URL                         jdbc:oracle:thin:@hostname:port:SID
Description                 OSWorkflow example Oracle database.
Pool Connections  X
Pool Statements             X
```

```
Native Results              X
User Name                   user
Password          pass
Verify Password   pass      pass
```

另外，针对你的数据库，还要修改 osworkflow.xml 里面的 sequence。
如果你使用 postgresql 数据库，请执行 tables_postgres.sql。确定你的 osworkflow.xml 文件包含有下面的配置：

```
<!- Postgresql sequence access ->
  <property key="entry.sequence"
                  value="SELECT nextVal('seq_os_wfentry')"/>
  <property key="step.sequence"
                  value="SELECT nextVal('seq_os_currentsteps')"/>
```

如果使用的是 oracle 数据库，请执行 tables_oracle.sql。请确定你的 osworkflow.xml 文件包含有如下的配置：

```
<!- Oracle sequence access ->
        <property key="entry.sequence"
                  value="SELECT seq_os_wfentry.nextval FROM dual"/>
        <property key="step.sequence"
                  value="SELECT seq_os_currentsteps.nextval FROM dual"/>
```

WAR 部署

要部署 osworkflow_example.war，当配置好上面的一切以后，将它放入 servers/default 文件夹。重启默认服务器(端口：8100)。
最后，到http://localhost:8100/osworkflow_example 观看示例。

Tomcat4.0.x

这里有一个将 osworkflow_example.war 部署到 Tomcat 4.0.X.上面去的快速方法。
开始持久化

要获得良好的性能，配置一个池化数据源是非常重要的。
不幸的是，没有好的文档说明怎样在 Tomcat4.0 里面装入 Tyrex，再考虑到 Tyrex 将会在 Tomcat4.1 及以后版本过期，所以没有好的方法在 Tomcat4.0 里面通过连接池提高性能。聊胜于无，倒有文档写出了怎样在 Tomcat4.0 里面配置 oracle 连接池，链接是
http://www.apachelabs.org/tomcat-user/20020OS:3-7.mbox/threads.html
试一试让我知道是否你成功了。我也尝试着使用 Tomcat 4.0.4 的 LE 版本 (jakarta-tomcat-4.0.4-LE-jdk14.*)，没有取得成功(NPE)，反而是完整版运行的很好。
请先阅读下面 WAR 部署部分，然后回到这里。
非常重要的是：要复制 osworkflow_example.war 到 TOMCAT_HOME/webapps 里面，然后启动并关闭服务器，然后再修改 TOMCAT_HOME/conf/server.xml，添加以下内容：

```
<!-- OSWorkflow Example Context ->
        <Context path="/osworkflow_example"
                 docBase="osworkflow_example" debug="0"
                 reloadable="true" crossContext="true">
          <Resource name="jdbc/DefaultDS" auth="Container"
            type="javax.sql.DataSource"/>
          <ResourceParams name="jdbc/DefaultDS">
```

```
            <parameter><name>user</name><value>user</value></parameter>

            <parameter><name>password</name><value>pass</value></parameter>
             <parameter>
               <name>driverClassName</name>
               <!- Oracle ->
              <value>oracle.jdbc.driver.OracleDriver</value>

               <!- Postgresql
               <value>org.postgresql.Driver</value>
               ->
             </parameter>
             <parameter>
               <name>driverName</name>
               <!- Oracle ->
               <value>jdbc:oracle:thin:@hostname:port:SID</value>

               <!- Postgresql
               <value>jdbc:postgresql://hostname:port/dbName</value>
               ->
             </parameter>
          </ResourceParams>
        </Context>
      </Host>
    </Engine>
  </Service>
```

确定你的 JDBC 驱动在 TOMCAT_HOME/common/lib 里面。对于 oracle，你可能还需要把 classes12.zip 重命名为 classes12.jar 才可以。

除此之外，还要针对你的数据库，修改 osworkflow.xml 里面的 sequence。

如果你使用的是 postgresql 数据库，请执行 tables_postgres.sql。确认你的 osworkflow.xml 文件包含如下配置：

```
<!- Postgresql sequence access ->
  <property key="entry.sequence"
                  value="SELECT nextVal('seq_os_wfentry')"/>
  <property key="step.sequence"
                  value="SELECT nextVal('seq_os_currentsteps')"/>
```

如果使用的是 oracle 数据库，请执行 tables_oracle.sql。确认你的 osworkflow.xml 文件包含如下配置：

```
<!- Oracle sequence access ->
       <property key="entry.sequence"
                  value="SELECT seq_os_wfentry.nextval FROM dual"/>
       <property key="step.sequence"
                  value="SELECT seq_os_currentsteps.nextval FROM dual"/>
```

WAR 部署

要部署 osworkflow_example.war，请将它放入 TOMCAT_HOME/webapps 中，开启并关闭服务器。然后按上面开始持久化进行配置，接着照着上述步骤修改 server.xml。
最后，到http://localhost:8080/osworkflow_example观看示例。

Tomcat4.1.x

<Egor Kobylkin egor.kobylkin@o2.com>
如下是如何将 osworkflow 部署到 Tomcat 4.1.x. 上的概述。(4.1.27 测试通过)

要部署 osworkflow_example.war，请将它放到 TOMCAT_HOME/webapps 下面，开启并关闭服务(如果你已经做了，可以忽略此步骤)。
注意若只需要查看例子，没有必要创建一个*persistent store*，就算不关闭Tomcat，例子也会在没有*persistent store*的情况下运行的很好。

创建持久化

将 jboss-j2ee.jar 放入 TOMCAT_HOME/common/endorsed 中（可从 JBoss.org 获得）以开启 EJB lookup 功能。
接着你必须要修改 TOMCAT_HOME/conf/server.xml，添加如下内容：

```
<!-- OSWorkflow JNDI JDBC Data Source Example. egor.kobylkin@o2.com ->

<Context path="/osworkflow_example" docBase="osworkflow-2.8.0-example"
         debug= "99" reloadable="true" crossContext="true"

verbosity="DEBUG">
<!- debug level is set to paranoid, to know what is happening,
     turn it off once you do not need it ->

<Logger className="org.apache.catalina.logger.FileLogger"
        prefix="OSWorkflow." suffix=".log" timestamp="true"/>
<!-
    put log4j.jar into:
    TOMCAT_ROOT/webapp/osworkflow-2.8.0-example/WEB-INF/lib
    if you want to use it for logging
->

<Resource name="jdbc/DefaultDS" auth="Container"
                 type="javax.sql.DataSource"/>
<!- name="jdbc/DefaultDS" is used in other components of the
     Example App, do not change it here! ->

<ResourceParams name="jdbc/DefaultDS">
  <parameter>
    <name>factory</name>
    <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
```

```
  </parameter>
  <parameter>
    <name>driverClassName</name>
    <value>oracle.jdbc.driver.OracleDriver</value>
  </parameter>
  <parameter>
    <name>url</name>
    <value>jdbc:oracle:thin:@yourserver.com:port:SID</value>
  </parameter>
  <parameter>
    <name>username</name>
    <value>your_database_user_name_here</value>
  </parameter>
  <parameter>
    <name>password</name>
    <value>your_password_here</value>
  </parameter>
  <parameter>
    <name>maxActive</name>
    <value>20</value>
  </parameter>
  <parameter>
    <name>maxIdle</name>
    <value>10</value>
  </parameter>
  <parameter>
    <name>maxWait</name>
    <value>-1</value>
  </parameter>
</ResourceParams>
</Context>

<!- OSWorkflow JNDI Data Source ->
```

以下链接是一些介绍如何配置 JNDI / JDBC 数据源的文档

- http://jakarta.apache.org/tomcat/tomcat-4.1-doc/jndi-datasource-examples-howto.html
- http://jakarta.apache.org/tomcat/tomcat-4.1-doc/jndi-resources-howto.html

确定你的 JDBC 驱动在 TOMCAT_HOME/common/lib 里面。对于 oracle，你可能还需要将 classes12.zip 重命名为 classes12.jar 才可以。
除此之外，还需要针对你的数据库修改 osworkflow.xml 里面的 sequence。
如果你使用 postgresql 数据库，请执行 tables_postgres.sql。确定你的 osworkflow.xml 文件

包含如下配置：

```
<!- Postgresql sequence access ->
  <property key="entry.sequence"
                  value="SELECT nextVal('seq_os_wfentry')"/>
  <property key="step.sequence"
                  value="SELECT nextVal('seq_os_currentsteps')"/>
```

如果使用的是 oracle 数据库，请执行 tables_oracle.sql（留心一下文件中未加进去的符号）。确定你的 osworkflow.xml 文件包含如下配置：

```
<!- Oracle sequence access ->
        <property key="entry.sequence"
                  value="SELECT seq_os_wfentry.nextval FROM dual"/>
        <property key="step.sequence"
                  value="SELECT seq_os_currentsteps.nextval FROM dual"/>
```

最后，重启服务器，然后到http://localhost:your_port/osworkflow-2.8.0-example(或者是你部署的任意web 位置)来观看示例。

1.4 持久化(Persistence)的选择

OSWorkflow 提供插件式的持久化机制,这使得我们在工作流内容的存储方面有了许多选择。OSWorkflow 内置提供了以下实现方式：**MemoryStore (默认), SerializableStore, JDBCStore, OfbizStore, 和 EJBStore 。** 如果以上方式都不能满足你的需求，你可以自己扩展 com.opensymphony.workflow.spi.WorkflowStore 接口。可以查阅 javadoc 获取更多的信息。

同时也请注意，每一种存储实现方式都需要设置必要的或者可选的 property。如果你要使用它们，推荐你首先阅读 javadoc ，以便能正确地进行配置。一个 JDBCStore 的配置的例子如下：

(来自 osworkflow.xml)

```
<persistence class="com.opensymphony.workflow.spi.jdbc.JDBCWorkflowStore">
        <!-- For jdbc persistence, all are required. -->
        <property key="datasource" value="jdbc/DefaultDS"/>
        <property key="entry.sequence"
                    value="SELECT nextVal('seq_os_wfentry')"/>
        <property key="entry.table" value="OS_WFENTRY"/>
        <property key="entry.id" value="ID"/>
        <property key="entry.name" value="NAME"/>
        <property key="entry.state" value="STATE"/>
        <property key="step.sequence"
                    value="SELECT nextVal('seq_os_currentsteps')"/>
        <property key="history.table" value="OS_HISTORYSTEP"/>
        <property key="current.table" value="OS_CURRENTSTEP"/>
        <property key="historyPrev.table" value="OS_HISTORYSTEP_PREV"/>
        <property key="currentPrev.table" value="OS_CURRENTSTEP_PREV"/>
```

```
        <property key="step.id" value="ID"/>
        <property key="step.entryId" value="ENTRY_ID"/>
        <property key="step.stepId" value="STEP_ID"/>
        <property key="step.actionId" value="ACTION_ID"/>
        <property key="step.owner" value="OWNER"/>
        <property key="step.caller" value="CALLER"/>
        <property key="step.startDate" value="START_DATE"/>
        <property key="step.finishDate" value="FINISH_DATE"/>
        <property key="step.dueDate" value="DUE_DATE"/>
        <property key="step.status" value="STATUS"/>
        <property key="step.previousId" value="PREVIOUS_ID"/>
</persistence>
```

如果你使用的 web 容器是 Tomcat，你必须在**$TOMCAT_HOME/conf/server.xml** 里面配置数据源，数据源默认的名字是 **jdbc/DefaultDS**。

还应该在 *WEB-INF/classes* 里面增加一个名为 *propertyset.xml* 的文件来让 propertyset 使用 jdbc，注意数据源的名称一定要和 Tomcat 里面的数据源的名称一致。

```
<propertysets>
    <propertyset name="jdbc"
    class="com.opensymphony.module.propertyset.database.JDBCPropertySet">
        <arg name="datasource" value="jdbc/DefaultDS"/>
        <arg name="table.name" value="OS_PROPERTYENTRY"/>
        <arg name="col.globalKey" value="GLOBAL_KEY"/>
        <arg name="col.itemKey" value="ITEM_KEY"/>
        <arg name="col.itemType" value="ITEM_TYPE"/>
        <arg name="col.string" value="STRING_VALUE"/>
        <arg name="col.date" value="DATE_VALUE"/>
        <arg name="col.data" value="DATA_VALUE"/>
        <arg name="col.float" value="FLOAT_VALUE"/>
        <arg name="col.number" value="NUMBER_VALUE"/>
    </propertyset>
     </propertysets>
```

在 src/etc/deployment/jdbc 的下面有一些创建 OSWorkflow 数据库的脚本文件。

如果你使用的是 HypersonicSQL，你可以遵循以下步骤进行操作

1. 假定你的 hsql 数据库的名称为 **oswf**，并且已经创建于**/db** 目录。
2. 使用 hsql.sql 创建表结构，你可以使用

   java -cp hsqldb.jar org.hsqldb.util.DatabaseManager

   启动工具来执行脚本。
3. 增加 context 配置到**$TOMCAT_HOME/conf/server.xml**

```
<Context path="/osworkflow"
         docBase="/jakarta-tomcat-4.1.27/webapps/osworkflow-2.8.0-example">
          <Resource name="jdbc/oswf" type="javax.sql.DataSource"/>
```

```
        <ResourceParams name="jdbc/DefaultDS">
          <parameter><name>username</name><value>sa</value></parameter>
          <parameter><name>password</name><value></value></parameter>
          <parameter><name>driverClassName</name>
            <value>org.hsqldb.jdbcDriver</value></parameter>
          <parameter><name>url</name>
            <value>jdbc:hsqldb:/db/oswf</value></parameter>
        </ResourceParams>
</Context>
```

4. 增加 **WEB-INF/classes/propertyset.xml**，配置如上。
5. 改变 *WEB-INF/classes/osworkflow.xml* 里面 persistent 的设置，下面的例子适用于所有不支持 sequences 的数据库(如 HSQL)。

```
<persistence class="com.opensymphony.workflow.spi.jdbc.JDBCWorkflowStore">
        <!- For jdbc persistence, all are required. -->
        <property key="datasource" value="jdbc/DefaultDS"/>
        <property key="entry.sequence"
                    value="select count(*) + 1 from os_wfentry"/>
        <property key="entry.table" value="OS_WFENTRY"/>
        <property key="entry.id" value="ID"/>
        <property key="entry.name" value="NAME"/>
        <property key="entry.state" value="STATE"/>
        <property key="step.sequence" value="select sum(c1) from
(select 1 tb, count(*) c1 from os_currentstep
union select 2 tb, count(*) c1 from os_historystep)"/>
        <property key="history.table" value="OS_HISTORYSTEP"/>
        <property key="current.table" value="OS_CURRENTSTEP"/>
        <property key="historyPrev.table" value="OS_HISTORYSTEP_PREV"/>
        <property key="currentPrev.table" value="OS_CURRENTSTEP_PREV"/>
        <property key="step.id" value="ID"/>
        <property key="step.entryId" value="ENTRY_ID"/>
        <property key="step.stepId" value="STEP_ID"/>
        <property key="step.actionId" value="ACTION_ID"/>
        <property key="step.owner" value="OWNER"/>
        <property key="step.caller" value="CALLER"/>
        <property key="step.startDate" value="START_DATE"/>
        <property key="step.finishDate" value="FINISH_DATE"/>
        <property key="step.dueDate" value="DUE_DATE"/>
        <property key="step.status" value="STATUS"/>
        <property key="step.previousId" value="PREVIOUS_ID"/>
</persistence>
```

对于 step.sequence 和 entry.sequence 的详细查询定义，不同的数据库有不同方法。
举个例子，在 MSSQL 数据库中，正确的应该是(假定不使用数据库 sequence)

```
<property key="step.sequence" value="select sum(c1) + 1 from (select 1 as
tb, count(*) as c1 from os_currentstep union select 2 as tb, count(*) as c1
```

```
from os_historystep) as TabelaFinal" />
```

对于 MYSQL，OSWorkflow 提供一个特殊的存储方法，它将使用单独的表来存储 ID 的值（参考 mysql.sql）。

除此之外，相对于标准部署来说，还有两个改变。首先就是调用 ID 的 sequences，它定义在 osworkflow.xml 的 persistence 里面，必须要增加以下四组 property。

```
<property key="step.sequence.increment"
    value="INSERT INTO OS_STEPIDS (ID) values (null)"/>
  <property key="step.sequence.retrieve"
    value="SELECT max(ID) FROM OS_STEPIDS"/>
  <property key="entry.sequence.increment"
    value="INSERT INTO OS_ENTRYIDS (ID) values (null)"/>
  <property key="entry.sequence.retrieve"
     value="SELECT max(ID) FROM OS_ENTRYIDS"/>
```

其次就是 persistence 实现类不同，应为 **com.opensymphony.workflow.spi.jdbc.MySQLWorkflowStore**

1.5 载入流程定义文件

在配置上，OSWorkflow 尽可能地保持着灵活。在 classpath 中只有一个文件是必须的：osworkflow.xml，它定义了 persistence 实现的方法(如 JDBC, EJB, Ofbiz)，同样也定义了用于载入流程定义信息的 workflow 实现工厂。

默认的工厂是 **com.opensymphony.workflow.loader.XMLWorkflowFactory**。它载入了在 classpath 中的一个文件，而该文件又包含一系列指向不限个数的流程定义文件的链接，它们全都是 XML 形式（参阅附录 A）。

当处于测试阶段，如果配置文件改变了，流程定义文件的重新加载(reload)是非常有用的。针对这种情况，你可以为你的工厂定义一个可选的 property 名字叫做 **reload**(true|false)，默认值是 false。如果你想使用自定义的方法实现流程描述的定义，请你自行继承 com.opensymphony.workflow.loader.AbstractWorkflowFactory。

例如说，**com.opensymphony.workflow.loader.JDBCWorkflowFactory** 是另一种可选的工厂，它允许你把流程定义存储在数据库里面，而非使用 XML 文件。

JDBCWorkflowFactory 将会有以下几个特性，但是还没有被实现。

1. 转换支持。
2. 直接存储 XML。
3. 删除功能。

它需要一些配置才可以运行：

osworkflow.xml:

```
<osworkflow>
  <persistence class="com.opensymphony.workflow.spi.jdbc.JDBCWorkflowStore">
    <arg name="foo" value="bar"/>
    ...
  </persistence>
  <factory class="com.opensymphony.workflow.loader.JDBCWorkflowFactory">
```

```
  </factory>
</osworkflow>
```

JDBC Database:

为 HSQLDB 数据库所提供的创建表结构的脚本如下：

```
CREATE CACHED TABLE OS_WORKFLOWDEFS (WF_DEFINITION BINARY,WF_NAME VARCHAR(256) NOT NULL PRIMARY KEY)
```

OSWorkflow 还包含这样一个 workflow 工厂，这个工厂将“正规化(normalized)”形式的流程定义存储在关系数据库中。这个工厂使用 spring 和 hibernate。参考 SpringHibernateWorkflowFactory。

最常见的配置如下

osworkflow.xml:

```
<osworkflow>
  <persistence class="com.opensymphony.workflow.spi.jdbc.JDBCWorkflowStore">
    <arg name="foo" value="bar"/>
    ...
  </persistence>
  <factory class="com.opensymphony.workflow.loader.XMLWorkflowFactory">
    <property key="resource" value="workflows.xml" />
  </factory>
</osworkflow>
```

workflows.xml:

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

2 其它模块整合

2.1 OSCore

OSWorkflow 需要 PropertySet 和 OSCore。因为，OSWorkflow 重度使用了 OSCore 中的很多有用的功能，因此必须使用 OSCore2.2.0 或以上版本。OSUser 不是严格必须的，但是，所有内置的用户/组方法(function)和条件(condition)都使用了它，因此，除非你准备使用你自己版本的条件(condition)。

2.2 PropertySet

OSWorkflow 的核心功能之一就是动态保存变量。这允许函数从 workflow 声明周期的一开始就运行，还能保存一些数据到 OSWorkflow 中。因此，后继的 action

执行的时候，可以取出这些数据，在其他函数中使用。这是非常强力的功能，使用得当的话可以做出高度定制的，具有长生命周期的工作流行为，就算服务器重启也可以持续。

这都归功于 PropertySet 模块。在 propertyset(通常以 **ps** 的名字作为变量出现)能动态保存的变量的类型，取决于 **WorkflowStore** 所选择的 PropertySet 实现，WorkflowStore 是 osworkflow.xml 中所配置的。例如，你选择了 *JDBCWorkflowStore*,你就必须确保在 propertyset.xml 中正确配置了 jdbc propertyset。如何配置 propertyset 后端存储（例如，JDBCPropertySet 的 sql 语句）在 propertyset 下载中可以找到。

2.3 Spring framework

总览

OSWorkflow 可以很容易地同轻量级的容器整合到一起，如 spring,xwork 或者 NanoContainer。使用这些框架的优势是，osworkflow 可以作为一个”钩子(hook)”渗透到容器中，并且很容易地得到依赖解析，组件装配，还有生命周期管理的支持。
OSWorkflow 已经可以和 spring framework 集成了，发行版本中包含了集成所需的代码，以下是所需组件：

1) **SpringHibernateWorkflowStore**，让工作流程实例(如果需要的话)分享当前事务。
2) **SpringTypeResolver**，允许 OSWorkflow 从 Spring ApplicationContext 中获得业务逻辑组件(conditions, functions 等等)。
3) **SpringConfiguration**，这是一个 Workflow Configuration 接口的实现类，它包含指向 store 和 factory 的引用，这样可以在 spring 中注射或者连接。
4) **SpringWorkflowFactory**，这是一个 XMLWorkflowFactory 封装包，它可以允许从容器中注入 configuration，从而不再从其它的 XML 配置文件中读取它们。

这四个类确保 OSWorkflow 紧密地整合到 Spring Framework 中去。

使用实例

下面通过创建一个 osworkflow-spring.xml 配置文件逐步演示整合过程。
这个文件被作为一个子 ApplicationContext 文件，被另一个文件所引用，而父文件已经定义好了 hibernate sessionFactory 配置。
首先必须为 workflow 实例配置一个 store,还必须配置一个读取有限状态机的 factory，如下：

```
<bean id="workflowStore"
   class="com.opensymphony.workflow.spi.hibernate.SpringHibernateWorkflowStore
  ">
        <property name="sessionFactory">
          <ref bean="sessionFactory"/>
        </property>
        <!--
        Optional PropertySet delegate, in case you want
    to use another PropertySet store that is not HibernateStore
        <property name="propertySetDelegate">
```

```
        <ref local="propertySetDelegate"/>
      </property>
       -->
  </bean>

  <bean id="workflowFactory"
        class="com.opensymphony.workflow.loader.SpringWorkflowFactory"
        init-method="init">
    <property name="resource"><value>workflows.xml</value></property>
    <property name="reload"><value>true</value></property>
  </bean>
```

定义这两个 bean 之后，我们可以定义 workflow Configuration 对象。

```
<bean id="osworkflowConfiguration"
        class="com.opensymphony.workflow.config.SpringConfiguration">
    <property name="store"><ref local="workflowStore"/></property>
     <property name="factory"><ref local="workflowFactory"/></property>
  </bean>
```

接着，如果需要，可以配置 SpringTypeResolver，允许 OSWorkflow 使用 spring 管理的 bean 来作为业务逻辑。

```
<bean id="workflowTypeResolver"
        class="com.opensymphony.workflow.util.SpringTypeResolver">
    <!--
    Here you can inject custom resolver for business logic
     <property name="conditions">
      <map>
         <entry key="beanshell">
        <value>mypackage.MyBeanShellCustomCondition</value></entry>
       </map>
     </property>
     -->
  </bean>
```

如果不需要拦截器，这个文件就算定义完成了，OSWorkflow 可以和 Spring 一起被使用了。

```
<bean id="workflow"
        class="com.opensymphony.workflow.basic.BasicWorkflow"
        singleton="false">
    <property name="configuration">
      <ref local="osworkflowConfiguration"/>
    </property>
        <property name="resolver">
      <ref local="workflowTypeResolver"/>
```

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

另一方面，如果需要事务控制或者 AOP 的支持，我们可以为流程实例做些锦上添花的工作，基于以上目的，可在此例的流程方法里面加一个 interceptor，所以最后 bean 的定义将被以下取代：

```
<bean id="transactionInterceptor"
class="org.springframework.transaction.interceptor.TransactionInterceptor">
 <property name="transactionManager">
<ref local="transactionManager"/></property>
 <property name="transactionAttributes">
  <props>
   <prop key="*">PROPAGATION_REQUIRED</prop>
  </props>
 </property>
</bean>

<bean id="workflow" class="org.springframework.aop.framework.ProxyFactoryBean">
 <property name="singleton">
  <value>false</value>
 </property>
 <property name="proxyInterfaces">
  <value>com.opensymphony.workflow.Workflow</value>
 </property>
 <property name="interceptorNames">
  <list>
   <value>transactionInterceptor</value>
   <value>workflowTarget</value>
  </list>
 </property>
</bean>

<bean id="workflowTarget" class="com.opensymphony.workflow.basic.BasicWorkflow"
singleton="false">
 <constructor-arg><value>test</value></constructor-arg>
 <property name="configuration">
<ref local="osworkflowConfiguration"/></property>
</bean>
```

对于这种定义，实例中的每一个方法都可以被一个 surround aspect 所修饰。

最后，也可能通过 SpringTypeResolver 来注射业务功能，定义如下：

```
<bean id="myFunction" class="mypackage.MyFunction" singleton="false" />
```

那么我们可以在 OSWorkflow 的 XML 中这样定义一个 function：

```
<function type="spring">
    <arg name="bean.name">myFunction</arg>
 </function>
```

3 理解 OSWorkflow

这一章详细地分析了 OSWorkflow 的核心思想。主要包括 functions,conditions,actions,steps, 以上这些不同的实体之间的相互作用，以及它们各自的例子。

3.1 工作流程描述

OSWorkflow 的核心是工作流描述文件。这个描述文件是一个 XML 文件(实际上并不一定，但是这是直接就内置支持的。) （——译者注：可能由别的如 JDBCWorkflowFactory 代替，但是 XML 是一个通用的格式，强烈建议使用)。
一个工作流描述文件描述了针对一个特定 workflow 的所有的 steps,states,transitions,functions.

- 一个工作流(workflow)由多个步骤(step)来表示流程(flow)。
- 每个步骤有一个或者多个动作，一个动作可以被设置成为是否自动执行，或者通过与用户的交互来由程序选择执行。
- 每一个动作至少有一个无条件结果(unconditional result)和零到多个条件结果(conditional result)
- 如果定义了多个 conditional result，第一个符合所有条件的 result 将会被执行，如果没有定义 conditonal results 或者没有符合的 conditions，就会执行 unconditional result。
- 执行完当前步骤，它可能会停留在当前步骤，跳转到另一个新的步骤，或者是一个 split，或者是一个 join 中去。以上所有的情况下，workflow 的的状态可以随之改变（例如状态是 Underway,Quened,Finished）。
- 如果一个动作执行的 result(结果)是一个分支(split)，result 中的 split 属性会指向 splits 元素中定义的一个 split 元素。
- 一个分支可以有一个或者多个无条件结果，但不会有条件结果。这些结果指向来自于 split 里面的步骤。
- 注册器（register）是一个全局变量，它在工作流运行时被解析，可以被每个 function 和 condition 使用
- propertyset 是全局范围的持久数据集合（——译者注：如果用数据库存储，它为 os_propertyentry 表。）
- transientVars 是一个保存临时数据的 Map 对象，它应用于所有的 functions 和 conditons。这个 transientVars 里面包括所有的 registers(全局变量)，用户输入，以及当前工作流的上下文和状态。它仅在一次工作流调用的生命周期中存在。

3.2 工作流程思想

对比一下其它比较熟悉的工作流引擎，OSWorkflow 是独一无二的。为了全面领会 OSWorkflow，完全掌握 OSWorkflow 的特性，就必须理解组成 OSWorkflow 的核心概念。

步骤,状态和动作 (Steps,Status and Actions)

任何一个工作流实例都可以有一个或者多个当前步骤(current steps)。每一个当前步骤都有一个状态值。当前步骤的状态值(status values)构成流程实例的工作流状态（workflow status）。**实际的状态值是完全由应用开发者或者项目经理决定的**。例如状态值有可能是"Underway"或"Queued".(译者注:也可能是别的，甚至于你也可以用中文的"处理中"，"排队中" ，"已完成"来表示)。

工作流要运作，*流转（transition）*将一定会在代表工作流实例的有限状态机中发生。一旦一个步骤完成，它将不再是当前步骤，通常一个新的当前步骤将会因此而立即产生，从而保证工作流的延续。完成步骤以后的最终状态值是由 old-status 这个属性所设置的，它会在流转到其它步骤以前发生。当工作流中的新的流转发生时，Old-status 必须已经得到定义。*它的值你爱放什么都可以，不过大多数情况下叫"Finished"就成了。*

*流转(transition)*本身就是*动作(action)*的一个结果。一个步骤可能有多个与它相关联的动作。具体要执行哪个动作，有可能由用户选择，也有可能是外部的事件，或者是由一个 trigger 自动地完成。取决于所进行的动作，一次流转也随之发生。动作通常受限于特定的 group,user 或者当前工作流状态。每一个动作必须有一个 *unconditional result（默认）*和零到多个条件结果。

总结：一个工作流包含多个步骤。每一个步骤都有一个当前状态(例如, Queued, Underway, or Finished)。每一个步骤中都有一个或者多个动作可以被执行。每一个动作都可以设置执行条件(condition)，也可以设置执行函数(pre-function or post-function)。动作产生结果(result)，导致工作流的状态和当前步骤发生改变。

结果,合并和分支(Results,Joins and Splits)

3.2.1 无条件结果(Unconditional Result)

对于每一个动作，都需要存在一个无条件结果，叫做 unconditional-result。这个结果只不过是一些指令，告诉 OSWorkflow 下一步要做什么。这一结果让组成工作流的状态机从一个状态流转到下一个状态。

3.2.2 条件结果(Conditional Results)

Conditional Result 是 Unconditional Result 的一个扩展。除此之外它多了一个或多个 **condition** 子元素。第一个判定为 true 的 conditional（使用 **AND** 或 **OR** 连接各个 conditon），会指明发生流转的步骤，这个切换步骤的发生是由于某个用户执行了某个动作所导致的。后文会更多的讲述条件。

3.2.3 可能发生的三种不同的结果(conditional or unconditional)

- ✓ 单一步骤/状态。
- ✓ split 成两个或多个步骤/状态。
- ✓ join,将这个和其他的步骤/状态组合成一个单一的步骤/状态。

根据你期望的行为的不同种类，你所写的 XML 工作流描述文件会有所不同。你可以阅读附件 A 中的 DTD 文件（它也有文档）。
http://www.opensymphony.com/osworkflow/workflow_2_8.dtd，获取更多的信息。
注意：目前，一个 split 或 join 无法立刻再生成 split 或 join。

3.2.3.1单一步骤/状态(step/status)的结果可以这样描述

```
<unconditional-result old-status="Finished" step="2"
                      status="Underway" owner="${someOwner}"/>
```

如果状态不是 Queued 的话，那么第三个必要条件就是新步骤的所有者（owner）。除了可以指明下一个状态的信息之外，result 也可以指定 validators 和 post-functions，这将在下面讨论。
如果这个步骤并不需要流转到其它的步骤，可以设置 setp 的值为-1。将上面的例子改写成如下：

```
<unconditional-result old-status="Finished" step="-1"
                      status="Underway" owner="${someOwner}"/>
```

3.2.3.2将一个状态分支(split)成多个状态可以这样描述

```
<unconditional-result split="1"/>
...
<splits>
  <split id="1">
    <unconditional-result old-status="Finished" step="2"
                          status="Underway" owner="${someOwner}"/>
    <unconditional-result old-status="Finished" step="2"
                          status="Underway" owner="${someOtherOwner}"/>
  </split>
</splits>
```

3.2.3.3合并(join)是最复杂的情况，一个典型的连接看起来如下

```
<!-- for step id 6 ->
<unconditional-result join="1"/>
...
<!- for step id 8 ->
<unconditional-result join="1"/>
...
<joins>
  <join id="1">
    <conditions type="AND">
      <condition type="beanshell">
```

```
        <arg name="script">
          "Finished".equals(jn.getStep(6).getStatus()
          && "Finished".equals(jn.getStep(8).getStatus())
        </arg>
      </condition>
    </conditions>
  </join>
  <unconditional-result old-status="Finished" status="Underway"
                                owner="test" step="2"/>
  </join>
</joins>
```

上面这段代码看起来有些模糊，最需要关心的应该是使用了 **jn** 的**条件元素(condition element)**。这个特殊的变量 **jn** 可以用来建立表达式，用以决定 join 何时发生。本质上，可以很容易理解出这段 xml 的意思：**当 step6 和 step8 都完成时，流转到这步 join。**

外部方法(External Functions)

OSWorkflow 为定义和执行的外部业务逻辑和服务提供了一个标准的方法，这就是 functions。function 通常代表游离于工作流实例自身之外的功能，可以用来使用 workflow 的信息来更新外部实体或者系统，或是当工作流状态发生改变时通知外部的系统。
有两种类型的 functions: 步骤前和步骤后 pre-function and post-function。
pre-function 是在工作流执行某一特定的流转之前进行的。一个例子就是预置调用者（caller）的名字到变量中去，以便在 result 中使用这个 caller。另外一个例子就是记录某个动作的最近调用者。这二者都作为标准工具方法提供，而且在实用中是相当有效的。

（译者注：举两个例子如下
例子一：

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

例子二：

```
<pre-functions>
          <function type="class">
<arg name="class.name">com.opensymphony.workflow.util.MostRecentOwner</arg>
                                <arg name="stepId">1</arg>
          </function>
```

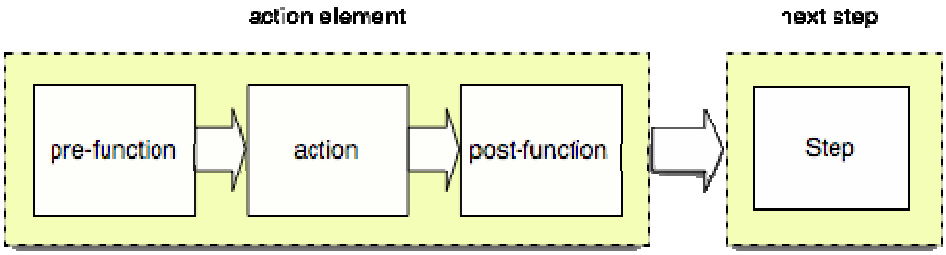
```
</pre-functions>
  <results>
   <unconditional-result old-status="Finished" status="Underway" step="1"
                                                owner="${mostRecentOwner}"/>
  </results>
)
```

post-function 和 pre-function 的适用性差不多，只不过它发生在状态改变以后，最典型的例子就是当执行完某个动作以后发邮件给相关的人员，例如当一份位于“research”步骤的文件被执行了“markReadyForReview”动作，审阅组人员就会收到 email。

使用 pre-function 和 post-function 有很多技巧。假若一个用户点了两次完成按钮，发送两个执行命令，而这个动作的 pre function 将会花很长的时间去完成，这就会造成这个长方法被调用多次，因为传递并未真正开始，OSWorkflow 会认为第二次调用的动作仍然是合法的。因此必须要把这个方法放到后面去。通常 pre-function 是比较简单而快捷的，而 post-function 才是真正的大餐。

Functions 可以被定义在两个不同的地方，steps 中和 actions 中。

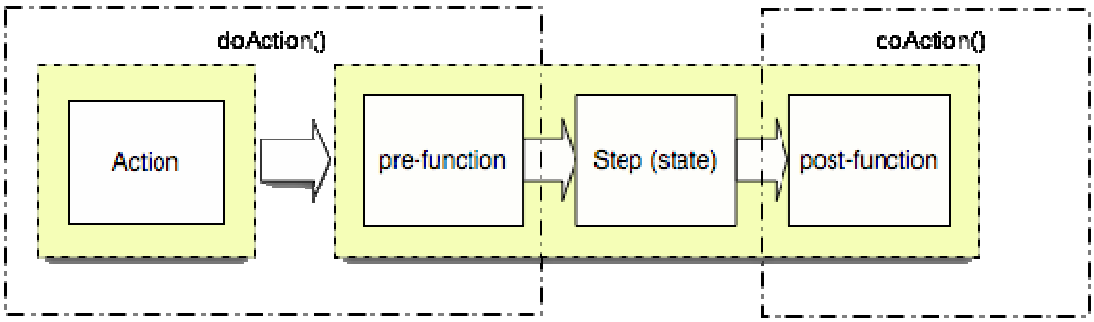
通常，一个 pre-function 或者一个 post-function 被定义在一个 action 中。总的来说，伴随这流转，一个 function 被用于“做事”，如果要通知给第三方，发送邮件或者简单地设定变量为以后所用。下面的图解用以说明 action 级别 functions。



如果是在一个 step 中定义的 pre-function 和 post-function，用法会有所不同，pre-function 将会在工作流流转到这个 step 之前执行。注意这个 pre-function 将会在任何目的是此步骤的流转发生之前执行，甚至是这个步骤本身(举个例子，如果状态从 Queued 转换到 Finished 但并未改变步骤也会执行 pre-function)。

同样的，step post-functions 也将会在工作流传递出这个步骤之前调用，甚至当它自己改变状态但步骤不发生改变也是如此。

下图的图解说明了调用顺序。注意 action 虚线内做了一些抽象，它自己也可以有 per-function 和 post-function。



您可以在后文的**函数(Functions)**章节获得更多信息。

触发器函数(Trigger Functions)

Trigger Functions 与其他函数一样，不同点在于他们不是仅与一个动作相联系的。他们是通过一个唯一的 ID 来标识。之后，通过 Quartz 任务调度（或者其他任务调度）来执行。这些函数通常是运行在系统级用户的上下文中，而非是工作流中的常规用户。Trigger functions 是由外部来源通过 OSWorkflow 的 API 来调用的，例如来自 Quartz 这样的任务调度器。

验证器(Validators)

validator 是用来校验一个动作的输入的有效性的。如果输入符合条件，那么将执行这个动作。如果输入不符合条件，那么将抛出 InvalidInputException 异常到调用者——通常是 JSP 或者 servlet。

Validators 遵循的规则很多和 Functions 相同，你可以在后面的 Validators 章节寻找更多信息。

注册器(Registers)

register 是一个辅助方法，这个方法可以返回一个对象，这个对象可以用在方法中去访问其他的普通对象，特别是 workflow 中涉及到的实体。被注册的对象可以是任何类型，典型的注册对象的例子是：Document, Metadata, Issue, 和 Task。这些确实方便的工具，除了使开发者开发简单一些之外，并没有给 OSWorkflow 增加任何额外的好处。下面是一个 registers 的例子：

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

                        <function type="beanshell">
                            <arg name="script">
                                transientVars.get("log").info("Initiate
Work");
                            </arg>
                        </function>
</pre-functions>
```

条件(Conditions)

Conditions 就象 validators，registers 和 functions 一样，可以用不同的语言和技术来

实现。Conditions 可以用 AND 或 OR 逻辑运算符来组织。任何其他类型的复杂逻辑必须由工作流开发者来实现。Conditions 通常是与 conditional results 联系的，只有条件成立，result 才会执行。
Conditions 与函数很相似，唯一不同的是 Conditions 返回的是 boolean 值，而不是 void。你可以在后文 Conditions 章节寻找更多信息。

变量解析(Variable Interpolation)

在所有的 functions、conditions、validators 和 registers 中，可能要提供一系列的参数(args)。这些参数将会被转化为参数 Map，这将在后面讨论。同样，在 workflow 描述符中的 status，old-status 和 owner 元素也将被动态的解析成变量。一个变量是这样被识别的：${foo}。当 OSWorkflow 识别出这种格式的时候，它首先到 transientVars 中去找关键字为 foo 的对象，如果没有找到，那么就到 propertySet 中去找，如果也没找到，那么变量 foo 将被转换为一个空字串。
一个非常重要的,值得大家注意的问题是参数(args)的类型，如果变量是惟一的参数的话，这个参数将不会是 String 类型，而是变量自己的类型，如果这个参数是字符串加上变量混合型的话，整个参数在任何情况下都将自动转化为 String 类型。这就意味着以下两个参数有极大的不同，foo 是 Date 型，而 bar 则是 String 型。

```
<arg name="foo">${someDate}</arg>
<arg name="bar"> ${someDate} </arg> <!—注意多余的空格-->
```

许可和约束(Permissions and Restrictions)

基于工作流实例的状态，Permissions 可被赋予到用户和/或群组。这些 Permission 和工作流引擎的功能无关，但是它们会对使用 OSWorkflow 的应用相当有好处。举一个例子，一个文档管理系统可能需要一个"file-write-permission"，只让某个组编辑文档(Document Edit)。你的程序便可以使用 API 去判断是否文件应该被修改。如果在工作流中有好多状态都适用“file-write-permission”，这就很有用，你不再需要去检查特定的步骤或者条件，检查是依照某个特定许可进行的。
（译者注：以下是例子：

```
<external-permissions>
                    <permission name="permA">
                        <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>
                    </permission>
</external-permissions>）
```

Permission 和 action 都使用了*约束（restriction）*的概念。**约束是定义在 restrict-to 元素内的一或多个条件。**

自动动作(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。如果要给工作流加入自动化，这就很有用。要达到这个目的，你需要在那些 action 中加入 auto 属性，并把它的值设置为 "true"。流程将考察这个动作的条件和约束，如果条件符合，并且工作流**可以**执行这个动作，那么将自动执行。 Auto action 是由当前的调用者执行的，所以将对引发自动动作的动作的调用者执行权限检查。

和抽象实体的集成(Integrating with Abstract Entities)

因为 OSWorkflow 并不是一个即开即用的解决方案，在你的项目可以正确地运行 OSWorkflow 之前，你需要做一些开发工作。建议给你的核心实体，例如"Document" 或 "Order"，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。

工作流程实例状态(自 2.6 版开始有效)

有时候，为整个 workflow 实例(os_wfentry)指定一个状态是很有帮助的，它独立于流程的执行步骤。OSWorkflow 提供一些元状态（meta-states），实例可以处于其中一种元状态。它们是 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，而且将永远保持着被终止时刻的状态。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。
（译者注：在数据库中
CREATED = 0;ACTIVATED = 1;SUSPENDED = 2;KILLED = 3;COMPLETED = 4;UNKNOWN = -1;）

3.3 通用动作和全局动作(Common and Global Actions)

当我们在定义工作流描述的时候，使用通用动作(common action)和全局动作(global action)这两个方便的特性可以帮助我们减少重复代码。
基于最简单的想法。这两种类型的动作都可以定义在描述文件的头部，就在 initial-actions 之后。
如果工作中存在许多需要一个相同动作的点，通用动作将会很有用。例如"发送邮件"动作。这个动作可能包括一些 pre-functions,或许还有一个验证器，甚至还包含一行一行脚本。总体上大约需要 7~8 行 xml 代码才能搞定。因为这个动作在很多地方都需要，就得在任何需要

的地方都复制一次。
如果将它定义成通用动作就可以避免这种复制。这个动作将作为通用动作被定义一次，然后任何需要使用这一动作的步骤都可以很简单地像这样来引用它：

```
<common-action id="100" />
```

而全局动作则稍微有点不同。它们被定义在相同的地方(initial-actions 之后的 global-actions 元素里面)，它并非像上面的通用动作一样在某个特殊的步骤里被显式的进行引用，它的引用是隐式的，也就是说：一旦它被定义，它将会被所有的 step 引用到。例如，可以有个“terminate workflow”的全局动作，在任何时候都可以用来终结一个流程(例如通过设置工作流的状态为 KILLED)，或许还有些额外动作，例如记录日志，再发送一封邮件（这些被作为 global action 的 function 定义）。
以上两种类型的动作都需要具有唯一的 ID，不能和别的动作 ID 相冲突。
以上的两种类型的动作都不能指明将传递工作流到哪一具体的步骤，在这种情况下通常需要设定下一步 step 为 0，以表明不需要流转。以下是例子。

```
<unconditional-result old-status="Finished" step="0"
                      status="Underway" owner="${someOwner}"/>
```

（译者注：这里官方可能描述错误，step 设置为-1 流程不变，如果设置为 0 会报错；另外可以指明 step 为某一具体的步骤 ID。）

3.4 方法(Functions)

OSWorkflow 中的方法是基于工作流的应用中，你可以真正用来享受的“正餐”部分。它们可以在有限状态机从一种状态流转到另一种状态之前或者之后被执行。OSWorkflow 支持下列 function 形式：

3.4.1 基于 Java 的方法

- 用 ClassLoader 载入 Java 类
- 通过 JNDI 重新获得
- 远程 EJBs
- 本地 EJBs

Java-based Functions（基于 Java 的方法）
基于 Java 的方法必须要实现 **com.opensymphony.workflow.FunctionProvider** 接口。这个接口只有一个方法——execute，这个方法有三个参数：

- **transientVars Map**

transientVars Map： 这个 Map 就是在客户端代码调用 **Workflow.doAction()**时传进来的。当动作完成，这个参数将会对方法非常有用，因为它可以基于用户的不同的输入使方法表现出不同的行为。这个参数也包含了许多特殊变量，这些变量对于访问工作流的各个不同的层面是有帮助的。它也包含了所有的在 Registers 中配置的变量（参考 3.2 工作流程思想）同时包含了下面两种特殊的变量：

entry(com.opensymphony.workflow.spi.WorkflowEntry)

和 **context**(com.opensymphony.workflow.WorkflowContext)。

（译者注：还有别的变量如： store,configuration,descriptor，另外还有用户输入的 Input Map 等等。）

- **The args Map**

args Map 是一个包含<function/>标签中所有<arg/>标签的 Map。这些参数都是 String 类型的并且已经经过了变量替换。这意味着<arg name="foo">this is ${someVar}</arg>将会由配置中的"foo"转换成为"this is [contents of someVar]"。

- **propertySet**

propertySet 包含所有的在 workflow 实例中持久化的变量。

（译者注：在数据库中为 os_propertyentry 表。）

基于 Java 的方法适用于以下**类型**：

class

对于一个类方法，类加载器必须知道方法所属的类的名字。这可以通过 class.name 参数来完成，举例：

```
<function type="class">
        <arg name="class.name">com.acme.FooFunction</arg>
        <arg name="message">The message is ${message}</arg>
</function>
```

jndi

JNDI 方法就象类方法一样，除了它们必须存在于 JNDI 树中。代替 class.name 参数，jndi.location 参数是必要的，举例：

```
<function type="jndi">
     <arg name="jndi.location">java:/FooFunction</arg>
     <arg name="message">The message is ${message}</arg>
</function>
```

remote-ejb

只需要一些处理，远程 EJB 就可以作为 OSWorkflow 中的方法使用。EJB 的远程接口必须继承 com.opensymphony.workflow.FunctionProviderRemote，同样的，ejb.location 也是必要的参数：

```
<function type="remote-ejb">
     <arg name="ejb.location">java:/comp/env/FooEJB</arg>
     <arg name="message">The message is ${message}</arg>
</function>
```

local-ejb

本地 EJBs 非常象远程 EJBs，除了 EJB 本地接口必须继承 com.opensymphony.workflow.FunctionProvider，类似于其它基于 Java 的方法。例子：

```
<function type="local-ejb">
        <arg name="ejb.location">java:/comp/env/FooEJB</arg>
        <arg name="message">The message is ${message}</arg>
</function>
```

3.4.2 BeanShell 类型的方法

OSWorkflow支持BeanShell作为一个脚本语言(scripting language)。你可以到BeanShell站点上获得更多关于BeanShell的信息。要在你自己的流程定义文件中编写脚本，BeanShell的文档就足够了。在开始之前，你必须知道一些准则，那就是：

BeanShell 方法的**类型**必须指定为 **beanshell**，有一个必要参数：**script**。这个参数的值实际上就是要执行的 script，举例：

```
<function type="beanshell">
     <arg name="script">
             System.out.println("Hello, World!");
     </arg>
</function>
```

在表达式范围内中始终存在三个变量，entry、context 和 store。entry 变量是一个实现了 com.opensymphony.workflow.spi.WorkflowEntry 接口的对象，它代表 workflow 实例。context 变量是 com.opensymphony.workflow.WorkflowContext 类型的对象，它允许 BeanShell 方法回滚事务或决定调用者的名字。store 变量是 com.opensymphony.workflow.WorkflowStore 类型的对象，它允许方法访问 workflow 底层的持久化存储区。

和 **3.4.1 基于 Java 的方法**一样，三个变量 **transientVars**, **args** 和 **propertySet** 可用，它们被自动地在 BeanShell 范围内赋值，举例：

```
<function type="beanshell">
     <arg name="script">
             propertySet.setString("world", "Earth");
     </arg>
</function>
<function type="beanshell">
     <arg name="script">
             System.out.println("Hello,
"+propertySet.getString("world"));
     </arg>
</function>
```

这两个脚本的输出将会是"Hello,Earth"。这是因为任何存储在 **propertySet** 中的变量都会被持久化，以便在后面的该 workflow 中的方法中使用。

3.4.3 BSF 类型的方法(perlscript, vbscript, javascript)

作为对 3.4.1 基于 Java 的方法和 3.4.2 BeanShell 类型的方法的补充，OSWorkflow 还支持一种第三方的方法类型：Bean Scripting Framework 方法。BSF 是 IBM AlphaWorks 小组的项目，可以让通用语言，比如 VBScript, Perlscript, Python, 和 JavaScript 等，在通用的环境下运行。这意味着，在 OSWorkflow 中你可以使用 BSF 支持的任何一种语言来编写你的方法，格式如下：

```
<function type="bsf">
        <arg name="source">foo.pl</arg>
        <arg name="row">0</arg>
        <arg name="col">0</arg>
        <arg name="script">
                print $bsf->lookupBean("propertySet").getString("foo");
        </arg>
</function>
```

上面的代码将获得 **propertySet**，然后打印出 key 为 foo 的值。在 BeanShell 方法中的缺省范围的变量，都可以在你的 BSF script 代码中使用。关于如何在你所使用的语言中引用这些 bean，请阅读 BSF 用户手册。

3.4.4 工具方法

OSWorkflow 本身附带一些很实用的工具方法，这些方法都实现了 com.opensymphony.workflow.FunctionProvider 接口。要获取更详细的信息，请阅读这些工具方法的 javadoc 文档。下面是每个工具方法的简单描述，它们都位于 com.opensymphony.workflow.util 包中。

Caller

用当前动作的执行者名字设置临时变量 caller。

WebWorkExecutor

执行一个 WebWork 方法并且方法结束时恢复旧的 ActionContext。

EJBInvoker

调用一个 EJB 的 session bean 方法。请查看 javadoc 获取更多关于参数和 EJB 限制的信息。

JMSMessage

发送一个 TextMessage 给一个 JMStopic 或 queue。

MostRecentOwner

用最近指定的步骤的所有者的名字来设置临时变量 mostRecentOwner。如果所有者未找到时，可以选择将将变量设置为 nothing，或者返回一个内部错误。

ScheduleJob

可以调度一个 Trigger 方法在某时执行。同时支持 cron 表达式和简单的 repeat/delay counts。

UnschduleJob

删除一个 ScheduleJob 和所有与之相关联的 triggers 方法。在 workflow 的状态发生变化而且你不再希望 ScheduleJob 再执行的时候是很有用的。

SendEmail

给一个或多个用户发送邮件。

3.5 验证器(Validations)

与 functions 类似，OSWorkflow 有三种不同形式的 validators：**java-based**，**beanshell** 和 **bsf**。基于 Java 的 validators 必须实现 **com.opensymphony.workflow.Validator** 接口（如果是 **remote-ejb**，则需实现 **com.opensymphony.workflow.ValidatorRemote** 接口）。基于 Java 的验证器通过抛出 InvalidInputException 异常表明一个输入是不合法的，并且停止正在发生的工作流 action。
但在 beanshell 和 bsf 中，有一点小小不同，因为异常是在脚本中抛出的，所以不能抵达到 jre。为了能够得到它， beanshell 和 bsf 中的任意返回值将被用作错误信息。逻辑如下：

- 如果返回值是一个 InvalidInputException 对象，这个对象立刻抛出到 client。
- 如果返回值是一个 Map，这个 Map 将被用做 InvalidInputException 中的一个 error/errormessage 对。
- 如果返回值是一个 String[]，偶数字被做为 key。奇数做为 value 来构造一个 Map 然后被用在以上情况中。
- 其他情况，值将被转换成 string 并且作为一个通用的错误信息来添加。

3.6 注册器(Registers)

OSWorkflow 中的 Register 是一个运行时的变量，它能够动态地注册到 workflow 的定义文件中。Registers 在很多场合都是很有用的。例如：你可能想提供访问实体的便捷途径，这个实体贯穿于流程(区别于真正的工作流)和流程描述文件。这时，你就可以定义一个 Register 来封装这个实体。如果这个实体是一个本地的 session EJB 的话，你就要使用 **com.opensymphony.workflow.util.ejb.local.LocalEJBRegister** 注册类使这个实体生效。例如，在后面的 post-function 中，你就可以访问这个实体，并且可以通过 beanshell script 来调用这

个实体中的方法。
Registers 类似于 3.5 验证器和 3.4 方法，也有三种实现方式：**基于 Java**, **BeanShell** 和 **BSF**。

基于 Java 的注册器

Java-based registers 必须实现 **com.opensymphony.workflow.Register** 接口(或者如果是**远程 ejb** 的话，你要实现 **com.opensymphony.workflow.RegisterRemote** 接口)。

BeanShell 类型和 BSF 类型的注册器

通过 script 返回的值或对象将是你注册的对象。

注册器接口注解(Register interface note)
验证器和方法都必须三个参数（**transientVars**, **args** 和 **propertySet**）一起工作，注册器仅仅需要 args Map 参数（连同 **context** 和 **entry** 一起可以在 **propertySet** 里面被找到）。这是因为注册器的调用根本不管用户的输入，与此同时它封装了变量 map，所以在注册器里拥有那些信息变得没有意义。

一个例子：

下面的例子将说明 register 的功能和用途。在这里 register 被用于一个简单的日志 register，这个 register 暴露一个"log"变量，这个变量可以在工作流生命周期内被访问到。日志记录器可以做很多有用的工作，比如说将工作流的实例 id 加入到日志记录中。
我们在 workflow 的描述符文件的顶层定义 register。

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

也可以从代码中可以看到，我们创建了一个名为 log 的 LogRegister，还定义了一个值为 true 的参数 addInstanceId。

我们现在可以在 workflow 描述符文件中的任何地方使用这个变量。例如：

```
<function type="beanshell" name="bsh.function">
  <arg name="script">transientVars.get("log").info("function called");</arg>
</function>
```

上面的方法将输出"function called"，前面准备的流程实例 ID 也一起输出。
虽然这个例子十分简易，但它的确表现了注册器的威力，并且指明注册器如何被用于访问特定的实体或在流程生命周期内的数据。

3.7 条件(Conditions)

除了一个微小的例外，OSWorkflow 中的条件和方法很相像：**在 BSF 和 Beanshell 的 script 范围中，有一个特别的对象叫做"jn"。**这个变量是 com.opensymphony.workflow.JoinNodes 类的一个实例，被用于 join-conditions 中。除此之外，conditions 与 functions 的不同在于，conditions 必须返回一个 **true 或 false** 的值。这个值可以是一个"true"的字串，也可以是一个"true"的 Boolean 对象，甚至是一个含有 toString()的方法，其返回值为字符串类型的"true"的各种形式(TRUE,True,true 等等)。
每个 condition 必须作为一个 **conditions** 的子标签被定义。这个元素有一个属性叫做 **type**，它的值是 **AND** 或者 **OR**。当你使用"AND"类型，所有的 condition 元素的值必须都是"true"，整个 conditions 才能是 true。否则，整个 conditions 将返回"false"。如果你使用"OR"类型，那么只要有一个 condition 标签的值为"true"，整个 conditions 就为 true，而只有当所有的 condition 标签的值必须都是"false"，整个 conditions 才能是 false。如果你想要更复杂的逻辑判断，那么你就要自己考虑使用 **Condition** 或 **ConditionRemote** 接口、BeanShell 或者是 BSF 来实现。
注意：如果在 conditions 标签中只含有一个 condition 的话，类型可以省略。
从 OSWorkflow2.7 开始，可以通过在<conditions>元素下面定义简单定义附加的<conditons>子元素构造 conditions，它允许你表达比在同一级别里一组简单的由 AND 或 OR 构成的 conditions 更复杂的逻辑运算，
下面是 OSWorkflow 自带的一些标准的 conditions：

- **OSUserGroupCondition -** 使用 OSUser 来判断调用者是否在参数"group"中。
- **StatusCondition -** 判断当前步骤的状态是否与参数"status"相同。
- **AllowOwnerOnlyCondition -** 如果调用者是指定的步骤的所有者的话，那么只返回 true，如果没有指明步骤的话，就返回当前步骤。
- **DenyOwnerCondition** 与 AllowOwnerOnlyCondition 功能相反。

更多的信息，请阅读 JavaDoc。

3.8 SOAP 支持

OSWorkflow 通过使用 SOAP 支持远程调用(remote invocation)。可以用来自于 WebMethods 的 Glue SOAP 实现，也可以用开源的 XFire soap library。

使用 XFire

OSWorkflow 附带的示例程序通过 SOAP 默认暴露了工作流对象。你应该以它作为起点。打开 SOAP 支持极为简单。第一步确保在你的 **WEB-INF/lib** 目录里面有所有必需的 xfire jar 文件。这些文件包含在 **lib/optional/xfire** 目录里面。
下一步就是在你的 web 应用程序里面添加 SOAP servlet。添加以下的内容到 web.xml 文件里面。

```
<servlet>
    <servlet-name>SOAPWorkflow</servlet-name>

<servlet-class>com.opensymphony.workflow.soap.SOAPWorkflowServlet</servlet-clas
s>
```

```
  </servlet>

  <servlet-mapping>
    <servlet-name>SOAPWorkflow</servlet-name>
    <url-pattern>/soap/*</url-pattern>
  </servlet-mapping>
```

一旦你的应用被部署了，你可以通过 **http://<server>/soap/Workflow?wsdl** 访问 WSDL。
要调用这一服务，任何 SOAP 客户端应该都可以。XFire 自己本身有客户端支持，这样允许你使用和服务端相同的 classes。别的客户端库如 Axis,GLUE 或者.net 应该也可以直接工作。

使用 GLUE

OSWorkflow发行包里面**没有**GLUE,你必须自行从WebMethods下载。GLUE在大多数应用中是免费的。下载GLUE时，你可以看到授权许可(the license agreement)。**SOAP和Job Scheduling在 2.1 或者更高版本才能得到支持，而且你必须在你的classpath里面包含 GLUE-STD.jar。**
和 XFire 一样，第一步必须在你的 web 应用程序中加入 GLUE servlet，这在 GLUE 文档里面有详细的说明。使用 Quartz 任务调度器，任务调度**必须打开** SOAP 支持才能运作。
这里有一些示例代码，使用 Glue 与 OSWorkflow 进行对话。

```
import electric.util.Context;
import electric.registry.Registry;
import electric.registry.RegistryException;

...

Context context = new Context();
context.setProperty( "authUser", username );
context.setProperty( "authPassword", password );
Workflow wf = (Workflow) Registry.bind(
  "http://localhost/osworkflow/glue/oswf.wsdl", Workflow.class, context);
```

从这点开始，你就可以用通常的方式来使用 Workflow 接口了。

4 GUI 设计器

OSWorkflow 有一个 GUI 设计器，使用它能够创建流程描述。这个设计器当前支持大部分的工作流功能，包括创建步骤，分支和合并。虽然如此，它的质量尚未达到产品级别，所以建议有规律地备份你的工作！

4.1 设计器的安装

OSWorkflow 的 GUI 设计器是一个基于 Swing 的 Java 应用程序。

仅有的先决条件是需要安装 Java1.4。程序可以通过以下几种不同方式启动。

- Webstart
- 从源代码：在代码树中，运行 ant 中的名为 **client-jar** 的 target 来 build 客户端，然后在命令行中运行 **java -jar dist/designer.jar** 或者双击 **designer.jar** 启动它。
- 从二进制发布包里：下载最新的 OSWorkflow 版本，双击 designer.jar 文件。

4.2 快速启动指南

在编辑和创建流程以前，必须要创建一个工作区间(workspace)。一个工作区间主要是一组可被一起编辑的流程。可以把它想像成为一个由许多流程组成的'流程项目'。
首次启动设计器时，你将会被提示是载入一个现有的工作区间还是创建一个新的。选择创建一个新的工作区间。
在你的眼前会出现一个文件对话框，创建一个新的目录，在这个目录里面为你的工作区间文件输入一个名称。工作区间内所有的文件通常被创建在工作区间文件的旁边，这就是为什么最好让它有自己的目录。
你有一个工作区间载入了，可以加入或者导入流程。让我们导入一个流程，看看完整的流程看起来是怎样的。

导入一个流程

从**文件(file)**菜单中选择**导入工作流(import)**，这时有一个对话框提示你指定流程的位置和导入的类型。你可以导入一个本地的流程配置，也可以根据它的 URL 指定一个远程的。在 OSWorkflow 的测试用例里面使用了大量的流程描述，这些描述都可从远程导入，通过以下地址得到它们：
https://osworkflow.dev.java.net/source/browse/osworkflow/src/test/samples/
选择其中任意一个例子，复制它的地址到 URL 文本框里面，设计器将会导入它，当它导入完成以后它便存在于你的工作区中，你可以打开浏览了。

布局

一旦你成功地导入了一个流程，设计器会对指定的流程进行检查，以确定是否存在一个布局。流程布局决定所有的标签和步骤在图表上放置的位置。如果没有布局存在，将尝试给出一个比较合理的布局。你可以在菜单里面选择**布局(Layout)->布局图形(Layout graph)**强迫设计器自动布局。
如果你改变了布局(举个例子：拖动步骤和标签)，确保你保存了工作区间。这会保存你的布局，这样你以后就可以重用这个布局了。

创建一个流程

除了可以导入一个已经存在的流程，你也可以通过在菜单里面选择**文件（File)->新建（New）->新建工作流（New Workflow）**来创建一个新的流程。你将会被提示定义流程名称。

编辑一个流程

一旦创建了一个流程，一个新的标签将会出现，在它里面存在一个初始步骤（initial step）。你可以从工具栏里面拖动步骤图标，分支图标，合并图标来创建这些流程子项。为了创建步骤与步骤之间的动作，你可以将鼠标移到每一个步骤中间的红点上面直至鼠标的形状发生改

变，然后拖动它到指定的步骤里面。设计器不支持将箭头拖动到初始步骤里面。
为了编辑某一步骤或结果里面的信息，选定它，左边的详细信息菜单里面有 pre-function 和 post-function 选项卡供你编辑，同样还有条件选项卡等供你编辑。你也可以在这里修改步骤名称并显示它，同样的，你可以通过在图标上直接双击的方式去修改名称。
当你完成了对流程的编辑，请保存你的工作区间。注意如果你创建了一个无效的流程，设计器将不允许你保存，而且会弹出一个对话框告诉你这个流程为什么无效。

4.3 工作区间(Workspace)

设计器中的工作区间是所有工作流程的集合。工作区间可以很方便地将流程分组，这样可以使你在同一时间内操作很多流程。它有点儿类似于一个工程。所有文件都和工作区间一起被创建在主配置文件（译者注：*.wsf）所在的文件夹中，这就是在创建新的工作区间时，我们强烈推荐使用空文件夹的原因。
从实现的角度看，工作区间是一种 WorkflowFactory 类型，它知道怎样载入和保存流程，并且它提供存储流程布局数据的功能。

4.4 调色板(Palette)

设计器是相当灵活的，它允许在部署的时候，由部署者规定流程编辑者可以使用的函数和条件。
调色板在设计器的术语中，是个包含有一定数量条件和函数的 XML 文件，当用户创建新流程的时候可以选择它们。调色板指定了特定函数或条件所需的参数，同时也定义了这些参数是否可被更改。
注意调色版是完全支持国际化的，所有的字符串都是 resource bundle 的 key，真正的文本是在 **palette.properties** 里面指定的。
会检查一些 magic keys 。对于每个函数名称或条件名称(举例如 **check.status**)，.long 这个关键字会被用作它的描述。(在我们的例子里，这个描述的 key 将会是 **check.status.long**)
对于参数，命名规则是<element name>.<arg name>。因而，对于 check.status 条件的状态（status）参数，在 properties 里面的 key 是 **check.status.status**。
对于参数，如果一个特定的 key 没有被找到，会使用一个合适的备用值(fallback value) (例如 XML 文件中列出的条件或参数名)。
目前，设计器只支持一个全局的调色板。默认的支持绝大多数内置的函数和条件的调色板放置在 META-INF（译者注：designer.jar 的 META-INF）文件夹里面。鼓励设计者开发包含符合自己用途的函数和条件的调色板。

5 使用 API

这个章节包含了如何使用 OSWorkflow API 的一些代码例子。

5.1 接口的选择

OSWorkflow 提供了 **com.opensymphony.workflow.Workflow** 接口的多个实现类，你可以在你的程序中使用它们。

BasicWorkflow

BasicWorkflow 不支持事务处理，但是可以通过外覆 BasicWorkflow 的方式来支持事务，这要依赖于你的持久化的实现。BasicWorkflow 通过下面的方式创建：

```
Workflow wf = new BasicWorkflow(username);
```

username 是当前请求的用户名。

EJBWorkflow

EJBWorkflow 使用 EJB 容器来管理事务。这是在 **ejb-jar.xml** 文件中配置的。它是这样建立的：

```
Workflow wf = new EJBWorkflow();
```

这里不需要指定用户名（但在 BasicWorkflow 和 OfbizWorkflow 里面必须指定），因为一旦用户被授权，它会从 EJB 容器中自动将用户名载入。

OfbizWorkflow

OfbizWorkflow 与 BasicWorkflow 非常相似，唯一不同的地方在于：需要事务处理的方法使用 ofbiz 的 TransactionUtil 调用来包装。

5.2 创建一个新的工作流

这是份非常简要的指南，教你如何使用 OSWorkflow 的 API 来创建一个新的工作流实例。首先应该使用 1.5 载入流程定义文件创建定义工作流的文件（在 XML 里面）。然后，你的程序必须知道初始化步骤的值 **initalStep**，以便进行流程实例的初始化。在你初始化一个工作流之前，你必须**创建**它，这样的话，你就可以获得一个流程 ID 从而在 AIP 里面得到工作流的引用。下面是例程代码：

```
Workflow wf = new BasicWorkflow(username);
HashMap inputs = new HashMap();
inputs.put("docTitle", request.getParameter("title"));
wf.initialize("workflowName", 1, inputs);
```

注意：一般情况下，你应该选择 Workflow 接口别的实现类，而不应该是 BasicWorkflow。举个例子，EJBWorkflow 或者 OfbizWorkflow。如果你想使用一个没有实现工作流上下文的 Workflow 存储方式（例如 JDBC 或者 Hibernate），你可以使用 BasicWorkflow。我们欢迎大家贡献各种不同的存储方式的实现！

5.3 执行动作

在 OSWorkflow 中，执行一个动作非常简单：

```
Workflow wf = new BasicWorkflow(username);
HashMap inputs = new HashMap();
inputs.put("docTitle", request.getParameter("title"));
long id = Long.parseLong(request.getParameter("workflowId"));
wf.doAction(id, 1, inputs);
```

5.4 查询

在 OSWorkflow 2.6 中，引入了新的 ExpressionQuery API。

注意：不是所有的 workflow 的存储都支持查询。目前，Hibernate，JDBC 和内存存储都支持查询。然而，hibernate 存储不支持混合类型的查询（例如：既有历史步骤又有当前步骤的查询）。要执行查询，就要建立一个 WorkflowExpressionQuery 对象，然后调用 Workflow 对象的 Query 方法。
下面是一些查询的例子：

```
//Get all workflow entry ID's for which the owner is 'testuser'
new WorkflowExpressionQuery(
  new FieldExpression(FieldExpression.OWNER, //Check the OWNER field
  FieldExpression.CURRENT_STEPS,  //Look in the current steps context
  FieldExpression.EQUALS, //check equality
  "testuser")); //the equality value is 'testuser'

//Get all workflow entry ID's that have the name 'myworkflow'
new WorkflowExpressionQuery(
  new FieldExpression(FieldExpression.NAME, //Check the NAME field
  FieldExpression.ENTRY,  //Look in the entries context
  FieldExpression.EQUALS,  //Check equality
  'myworkflow')) //equality value is 'myworkflow'
```

下面是一个嵌套查询的例子：

```
// Get all finished workflow entries where the current owner is 'testuser'
Expression queryLeft = new FieldExpression(
  FieldExpression.OWNER,
  FieldExpression.CURRENT_STEPS,
  FieldExpression.EQUALS, 'testuser');
Expression queryRight = new FieldExpression(
  FieldExpression.STATUS,
  FieldExpression.CURRENT_STEPS,
  FieldExpression.EQUALS,
```

```
    "Finished",
    true);
  WorkflowExpressionQuery query = new WorkflowExpressionQuery(
    new NestedExpression(new Expression[] {queryLeft, queryRight},
    NestedExpression.AND));
```

最后，这里有一个混合查询，注意 hibernate 类型的存储方式是不支持的。

```
//Get all workflow entries that were finished in the past
//or are currently marked finished
Expression queryLeft = new FieldExpression(
  FieldExpression.FINISH_DATE,
  FieldExpression.HISTORY_STEPS,
  FieldExpression.LT, new Date());
Expression queryRight = new FieldExpression(
  FieldExpression.STATUS,
  FieldExpression.CURRENT_STEPS,
  FieldExpression.EQUALS, "Finished");
WorkflowExpressionQuery query = new WorkflowExpressionQuery(
  new NestedExpression(new Expression[] {queryLeft, queryRight},
  NestedExpression.OR));
```

5.5 对比隐式和显式 Configuration

在 OSWorkflow 2.7 版以前，状态的维持在很多场合都是使用静态字段(static fields)来实现的。这种方式固然方便，但是也有一些缺点和制约。最主要的一点就是不能让 OSWorkflow 通过多种不同配置文件产生多种不同实例在同一个项目上面运行。举个简单的例子，你在通过 MemoryStore 运行工作流的时候不能调用 EJB Store。

OSWorkflow2.7 通过引入一个 Configuration 接口解决了这个局限。这个 Configuration 接口的默认实现是 DefaultConfiguration，这是为了向以后的版本兼容（译者注：要显式调用 DefaultConfiguration）。同样为了向以前的版本兼容，**如果不显式调用 Configuration 接口将默认使用 static instance(if no explicit call is made using a Configuration)**。实际上，决定是使用静态实例还是指定配置取决于 AbstractWorkflow 中的 *setConfiguration* 方法。如果这个方法被调用了，那么预置实例(per-instance)的模式将会被使用。如果没有被调用，以往的单例模式将会被使用(singleton static model)。

一旦你使用新的模式（译者注：预置实例模式）创建工作流，AbstractWorkflow 将不再是无状态的了，如果你不使用静态方法（不鼓励你使用它!），你需要将 AbstractWorkflow 的实例保存在某个地方，这样你可以循环使用它，而不用在每一个调用的地方都要重新创建。

听起来好像挺复杂，其实实践起来非常简单，下面举例说明：

以往方法:

```
Workflow workflow = new BasicWorkflow("blah");
long workflowId = workflow.initialize("someflow", 1, new HashMap());
workflow.doAction(workflowId, 2, new HashMap());
...
```

```
//in some other class, called later on
Workflow workflow = new BasicWorkflow("blah");
workflow.doAction(workflowId, 3, new HashMap());
```

推荐方法:

```
Workflow workflow = new BasicWorkflow("blah");
Configuration config = new DefaultConfiguration();
workflow.setConfiguration(config);
long workflowId = workflow.initialize("someflow", 1, new HashMap());
workflow.doAction(workflowId, 2, new HashMap());
//keep track of Workflow object somewhere!
...
//in some other class, called later on
//look up Workflow instance that was held onto earlier
Workflow workflow = ...; //note, do NOT create a new one!
workflow.doAction(workflowId, 3, new HashMap());
```

6 附录

6.1 DTD 文档(V2.8)

<!--
workflow 元素是根元素。一个工作流包含一组 meta 属性、全局注册器以及触发器函数。它还定义了一组初始动作以激活整个工作流。全局动作在任何步骤中都可以被执行，并且在任何步骤中都是有效的。通用动作在步骤中可以被重复使用。它们在此被定义，一个步骤可以指向一个通用动作并激活它。需要注意的是所有动作必须是全局唯一的。
一个工作流包含一组步骤(step)和确定状态应流向哪个分支(split)的分支定义以及确定合并(join)条件和步骤的合并定义。
-->
<!ELEMENT workflow (meta*, registers?, trigger-functions?, global-conditions?, initial-actions, global-actions?, common-actions?, steps, splits?, joins?)>

<!--
一个动作可以执行。动作的 id 在整个工作流定义中必须唯一，而不仅仅是在一个步骤中唯一。
属性：
id:流程描述文件里面具有唯一性。注意，id 必须全局唯一，而不是在某一步骤中唯一。
name:动作的名称。
view:动作视图名。

auto:可以为 true 或者 false。若为 true，动作在符合以下条件时自动执行：
-工作流进入动作所在的步骤。
-满足动作的所有条件和约束。
注意，一个步骤内每一次流转只能有一个自动执行的动作。如果有多个满足以上条件的动作，则仅仅第一个满足条件（以在 XML 文件中的排列顺序）的动作执行。
finish:这个动作是否被终结。一个终结的动作可以导致所有的当前步骤完成并移至历史流程，工作流状态也会被设置成为 COMPLETED。
适用范围：actions。
-->
<!ELEMENT action (meta*, restrict-to? , validators?, pre-functions?, results, post-functions?)>
<!ATTLIST action
id CDATA #REQUIRED
name CDATA #REQUIRED
view CDATA #IMPLIED
auto (TRUE | FALSE | true | false) #IMPLIED
finish (TRUE | FALSE | true | false) #IMPLIED
>

<!--
通用动作指的是在“common-actions”元素中定义的动作。
适用范围：actions。
-->
<!ELEMENT common-action (#PCDATA)>
<!ATTLIST common-action
id CDATA #REQUIRED
>

<!--
在整个步骤中必须有零个或者多个通用动作集合和零个或者多个动作集合。注意：要么定义通用动作集合，要么定义动作集合（译者注：也可以混合定义）。不允许定义一个空的元素集合。
适用范围：step。
-->
<!ELEMENT actions (common-action*, action*)>

<!--
函数中的参数。用户传入参数值。例如，若变量 foo 为"bar"，则"test ${foo}"返回"test bar"。
适用范围：function
-->
<!ELEMENT arg (#PCDATA)>
<!ATTLIST arg

```
    name CDATA #REQUIRED
>


<!--
包含一个或多个条件元素，并确定用 AND 还是 OR 运算符来组合这些条件。
适用范围：join, restrict-to, result
-->
<!ELEMENT conditions (conditions | condition)*>
<!ATTLIST conditions
    type (AND | OR) #IMPLIED
>


<!--
一个条件。
适用范围：conditions
-->
<!ELEMENT condition (arg*)>
<!ATTLIST condition
    type CDATA #REQUIRED
  id CDATA #IMPLIED
  negate CDATA #IMPLIED
  name CDATA #IMPLIED
>


<!--
权限组是在工作流外面定义的，但是却与工作流实体紧密相连。
适用范围：step
-->
<!ELEMENT external-permissions (permission+)>


<!--
函数是 OSWorkflow 自动执行的。查看文档可以知道所选择的函数类型需要的参数(如果存在参数的话)。
适用范围：pre-functions, post-functions
-->
<!ELEMENT function (arg*)>
<!ATTLIST function
    type CDATA #REQUIRED
  id CDATA #IMPLIED
  name CDATA #IMPLIED
```

```
>


<!--
一组全局动作，可以在任何步骤中执行。
适用范围：workflow
-->
<!ELEMENT global-actions (action+)>


<!--
一组初始动作在工作流实例被赋予任何状态之前执行。
适用范围：workflow
-->
<!ELEMENT initial-actions (action+)>


<!--
一组通用动作可以包含在任何步骤中，只需简单地声明一个带 id 属性和不含任何内容的动作标签。这样在多个步骤中可以包含同一个动作。
适用范围：workflow
-->
<!ELEMENT common-actions (action+)>


<!--
两个或多个当前状态一起将在这里被转换成一个新的单一的状态。无条件结果必须是一个真正的 step/status 的集合，而不是其它的 join 或 split。当所有条件运算值为 true 的情况下才会产生 join。
适用范围：joins
-->
<!ELEMENT join (conditions, unconditional-result)>
<!ATTLIST join
      id CDATA #REQUIRED
>


<!--
一组合并元素。
适用范围：workflow
-->
<!ELEMENT joins (join*)>
```

```
<!--
工作流、动作、步骤的一个静态 meta 属性。
适用范围：workflow, step, action
-->
<!ELEMENT meta (#PCDATA)>
<!ATTLIST meta
      name CDATA #REQUIRED
>


<!--
一个外部权限。
适用范围：external-permissions
-->
<!ELEMENT permission (restrict-to)>
<!ATTLIST permission
      name CDATA #REQUIRED
   id CDATA #IMPLIED
>


<!--
当工作流从一种状态转换成另一种新的状态之后所调用的函数集合。
适用范围：action, result, unconditional-result
-->
<!ELEMENT post-functions (function+)>


<!--
当工作流从一种状态转换成另一种新的状态之前所调用的函数集合。
适用范围：action, result, unconditional-result
-->
<!ELEMENT pre-functions (function+)>


<!--
一个将变量 put 到函数和验证器范围内进行使用的类。
适用范围：registers
-->
<!ELEMENT register (arg*)>
<!ATTLIST register
      type CDATA #REQUIRED
      variable-name CDATA #REQUIRED
   id CDATA #IMPLIED
```

```
>


<!--
一组注册器。
适用范围：workflow
-->
<!ELEMENT registers (register+)>


<!--
动作和权限的约束。可以基于群组和当前工作流状态设置，也可以基于调用者是否为拥有者本人设置。
适用范围：action, permission
-->
<!ELEMENT restrict-to (conditions?)>


<!--
一个动作的条件结果。只有在 beanshell 执行条件的结果为 true 时，才能产生 result。
适用范围：results
-->
<!ELEMENT result (conditions, validators?, pre-functions?, post-functions?)>
<!ATTLIST result
      old-status CDATA #REQUIRED
      status CDATA #IMPLIED
      step CDATA #IMPLIED
      owner CDATA #IMPLIED
      split CDATA #IMPLIED
   join CDATA #IMPLIED
   due-date CDATA #IMPLIED
   id CDATA #IMPLIED
   display-name CDATA #IMPLIED
>


<!--
一组可选的条件结果和单一的无条件结果。
适用范围：action
-->
<!ELEMENT results (result*, unconditional-result)>


<!--
```

```
一种当前状态可以在这里被转换成另外一种或者多种状态。这些新状态的结果必须是一个真正的 step/status 的集合，而不是另外的 split 或 join。
适用范围：splits
-->
<!ELEMENT split (unconditional-result+)>
<!ATTLIST split
    id CDATA #REQUIRED
>


<!--
一组分支元素。
适用范围：workflow
-->
<!ELEMENT splits (split+)>


<!--
工作流中的一个步骤(id 在所有步骤中必须唯一)。
适用范围：steps
-->
<!ELEMENT step (meta*, pre-functions?, external-permissions?, actions?, post-functions?)>
<!ATTLIST step
    id CDATA #REQUIRED
    name CDATA #REQUIRED
>


<!--
包含在工作流定义中的一组步骤。
适用范围：workflow
-->
<!ELEMENT steps (step+)>


<!--
没有任何条件的结果。如果所有条件结果都失败，这个结果则为缺省结果。
-->
<!ELEMENT unconditional-result (validators?, pre-functions?, post-functions?)>
<!ATTLIST unconditional-result
    old-status CDATA #REQUIRED
    status CDATA #IMPLIED
    step CDATA #IMPLIED
    owner CDATA #IMPLIED
```

```
      split CDATA #IMPLIED
      join CDATA #IMPLIED
   due-date CDATA #IMPLIED
   id CDATA #IMPLIED
   display-name CDATA #IMPLIED
>


<!--
有唯一 id 的函数，由一个外部作业调度程序调用。
适用范围：trigger-functions
-->
<!ELEMENT trigger-function (function)>
<!ATTLIST trigger-function
      id CDATA #REQUIRED
>


<!--
一组触发器函数。
适用范围：workflow
-->
<!ELEMENT trigger-functions (trigger-function+)>


<!--
一个对用户输入进行校验的类或脚本。查看文档可以知道所选择的函数类型需要的参数(如果存在参数的话)。
适用范围：validators
-->
<!ELEMENT validator (arg*)>
<!ATTLIST validator
      type CDATA #REQUIRED
   name CDATA #IMPLIED
   id CDATA #IMPLIED
>


<!--
一组验证器。
适用范围：action, intial-step, result, unconditional-result
（译者注：根本没有 intial-step，这里很有可能是作者笔误。）
-->
<!ELEMENT validators (validator+)>
```

<!--
一组被用于检查每个动作的全局条件集合。
适用范围：workflow
-->
<!ELEMENT global-conditions (conditions?)>

6.2 从 2.7 版升级

Descriptor 的改变

所有* Descriptor 类将不再有构造器，这是因为它在 2.8 版中是由 DescriptorFactory 创建的。这种改变将使 OSWorkflow 更容易地为第三方提供个性化的描述而不用修改源代码。

Register API 的改变

Register API 改变了，增加了 PropertySet 参数。
译者注：
2.7 版中：

```
public interface Register {
    /**
     * Returns the object to bind to the variable map for this workflow instance.
     *
     * @param context The current workflow context
     * @param entry The workflow entry. Note that this might be null, for example in a pre function
     * before the workflow has been initialised
     * @param args Map of arguments as set in the workflow descriptor

     * @return the object to bind to the variable map for this workflow instance
     */
    public Object registerVariable(WorkflowContext context, WorkflowEntry entry, Map args) throws WorkflowException;
}
```

2.8 版中(红色部分为新增的 PropertySet 参数)：

```
public interface Register {
    /**
     * Returns the object to bind to the variable map for this workflow instance.
     *
     * @param context The current workflow context
     * @param entry The workflow entry. Note that this might be null, for example in a pre function
```

```
     * before the workflow has been initialised
     * @param args Map of arguments as set in the workflow descriptor

     * @param ps
     * @return the object to bind to the variable map for this workflow instance
     */
    public Object registerVariable(WorkflowContext context, WorkflowEntry entry, Map args,
PropertySet ps) throws WorkflowException;
}
```

译者写在最后：这篇译文的出炉，不光是我一个人的功劳，它同时得到了满江红翻译、审校同仁们的大力支持，尤其是曹晓钢先生工作非常之忙碌，但他却挤出时间花了不少心血来校对此译文，给我指出了不少疏漏和错误之处，其严谨的治学作风和专业的翻译水准给我留下了非常深刻的印象。另外我的爱人乔文也给予了我极大的精神支持。我衷心感谢上述所有人员及帮助过我的所有朋友！