OSWorkflow 中文文档(版本 2.0)

Author:陈刚(Chris Chen)

Date:2007-07-16

简介(Introduction)

OSWorkflow 非常不同于大多数其它别的工作流，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，这就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地去书写和定义这些 XML 流程描述文件。这便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。寻找一个快速的”即插即用(plug-and-play)”工作流解决方案成为了一些人现有的问题，但我们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)

OSWorkflow 认为是一个”低端”(low level)工作流实现。比如说循环（loops）和条件(conditions)在其它别的工作流里面被抽象成是可视化的图标，然而在 OSWorkflow 里面却是代码。这并不是就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。我们不期待

那些不懂技术的用户去修改流程。我们发现有些系统提供了 GUI 可以简单地编辑一些流程，但当运用 GUI 的时候这些流程最终被修改和破坏了。我们相信最好的方式是能让这些开发者知道这些改变。我们也曾说过，OSWorkflow 最终的版本将提供 GUI 来辅助开发者编辑流程。

OSWorkflow 基于有限状态机制（*finite state machine*）这种思想。每一种状态(state)被描述成为 step ID 和 status(包括 status 和 old-status)。从一种状态(state)转到另一种状态没有动作(action)是不可能发生的。在一个工作流的生命周期内通常有一个或者多个有效的状态。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 文件来描述一个商业的工作流程。

跑通 OSWorkflow2.8 例子**(Run OSWorkflow 2.8 Example)**

所需 JAR 包**(Required JAR Library)**

OSWorkflow 必要包**(OSWorkflow Required Library)**

OSWorkflow 必要包(%osworkflow 解压包%\)：

osworkflow-2.8.0.jar

OSWorkflow 核心包**(OSWorkflow Core Library)**

OSWorkflow 核心包(%osworkflow 解压包%\lib\ core)：

commons-logging.jar：必要，支持日志。
propertyset-1.4.jar：必要，支持 propertyset 的 aggregate,cached,,memory,jdbc,file,javabeans,map,xml 接口实现，并不支持 hibernate3，如果要支持 hernate3，要自己写代码。这个下面再谈。
oscore-2.2.5.jar:必要，提供了一些工具等。

OSWorkflow 可选包**(OSWorkflow Optional Library)**

OSWorkflow 可选包(%osworkflow 解压包%\lib\ optional)：

bsf.jar:支持 bsf，可选。
bsh-1.2b7.jar:支持 beanshell，可选。
ehcache.jar:支持缓存，可选。
osuser-1.0-dev-2Feb05.jar:支持例子里面的用户和群组管理，在涉及到用户和群组的操作建议加上此包。

Spring2 对 OSWorkflow 的支持(Spring2 support OSWorkflow)

Spring2 对 OSWorkflow 的支持(%spring 解压包%\dist) ：
support
spring.jar(version:2.05)

Hibernate3 对 OSWorkflow 的支持(Hibernate3 support OSWorkflow)

Hibernate3 对 OSWorkflow 的支持(%hibernate 解压包%\lib) ：

antlr.jar
cglib.jar
asm.jar
asm-attrs.jars
commons-collections.jar
commons-logging.jar
hibernate3.jar
jta.jar
dom4j.jar
log4j.jar

WorkflowStore

MemoryWorkflowStore

在例子里面有现成的可以参照，不过我还是要讲一下，最重要的也就是要把 persistence class 设置成为 **com.opensymphony.workflow.spi.memory.MemoryWorkflowStore**
具体在\%webapp%\WEB-INF\classes 下的 osworkflow.xml 为：

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

JDBCWorkflowStore

第一步：web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
```

```
<web-app>
      <display-name>OSWorkflow Example App</display-name>
      <description>OSWorkflow Example App</description>
 <servlet>
            <servlet-name>SOAPWorkflow</servlet-name>
        <servlet-class>com.opensymphony.workflow.soap.SOAPWorkflowServlet</servlet-class>
 </servlet>
  <servlet-mapping>
            <servlet-name>SOAPWorkflow</servlet-name>
            <url-pattern>/soap/*</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
            <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

第二步：配置数据源

如果是 tomcat5.0 及以上版本，在\%tomcathome%\conf\catalina\localhost 建一个 osworkflow.xml，osworkflow.xml 里面的内容如下，请注意**红色**部分为数据源名称：

```
<?xml version="1.0" encoding="UTF-8"?>
<Context crossContext="true" displayName="Welcome to Tomcat"
docBase="E:/workspace/osworkflow/exploded" path="" >
  <Resource auth="Container" name="jdbc/oswf" type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/oswf">
    <parameter>
      <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>
    <parameter>
      <name>url</name>
<value>jdbc:mysql://127.0.0.1:3306/osworkflow?useUnicode=true&amp;characterEncoding=GB
K</value>
    </parameter>
    <parameter>
      <name>password</name>
      <value></value>
    </parameter>
    <parameter>
      <name>maxActive</name>
      <value>10</value>
    </parameter>
    <parameter>
      <name>maxWait</name>
      <value>10000</value>
    </parameter>
```

```
        <parameter>
          <name>driverClassName</name>
          <value>com.mysql.jdbc.Driver</value>
        </parameter>
        <parameter>
          <name>username</name>
          <value>root</value>
        </parameter>
        <parameter>
          <name>maxIdle</name>
          <value>3</value>
        </parameter>
    </ResourceParams>
</Context>
```

如果是 resin，请在 web.xml 里面加入以下代码：

```
<resource-ref>
            <res-ref-name>jdbc/oswf</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <init-param driver-name="com.mysql.jdbc.Driver"/>
            <init-param url="jdbc:mysql://localhost:3306/osworkflow"/>
            <init-param user="root"/>
            <init-param password=""/>
            <init-param max-connections="20"/>
            <init-param max-idle-time="30"/>
        </resource-ref>
```

最终 web.xml 如下，红色部分为新加的数据源：

```
        <?xml version="1.0" encoding="ISO-8859-1"?>
        <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
        <web-app>
            <display-name>OSWorkflow Example App</display-name>
            <description>OSWorkflow Example App</description>
            <servlet>
                <servlet-name>SOAPWorkflow</servlet-name>
<servlet-class>com.opensymphony.workflow.soap.SOAPWorkflowServlet</servlet-class>
            </servlet>
            <servlet-mapping>
                <servlet-name>SOAPWorkflow</servlet-name>
                <url-pattern>/soap/*</url-pattern>
            </servlet-mapping>
            <welcome-file-list>
                <welcome-file>default.jsp</welcome-file>
            </welcome-file-list>
            <resource-ref>
```

```
        <res-ref-name>jdbc/oswf</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <init-param driver-name="com.mysql.jdbc.Driver"/>
        <init-param url="jdbc:mysql://localhost:3306/osworkflow"/>
        <init-param user="root"/>
        <init-param password=""/>
        <init-param max-connections="20"/>
        <init-param max-idle-time="30"/>
    </resource-ref>
</web-app>
```

第三步：配置 osworkflow 必要文件。

确定在\%webapp%\WEB-INF\classes\下面有以下几个文件：

example.xml:例子，不作修改。

workflows.xml:例子，不作修改。

osuser.xml,修改如下，请注意**红色**的数据源部分：

```
<opensymphony-user>
    <provider class="com.opensymphony.user.provider.jdbc.JDBCAccessProvider">
        <property name="user.table">os_user</property>
        <property name="group.table">os_group</property>
        <property name="membership.table">os_membership</property>
        <property name="user.name">username</property>
        <property name="user.password">passwordhash</property>
        <property name="group.name">groupname</property>
        <property name="membership.userName">username</property>
        <property name="membership.groupName">groupname</property>
        <property name="datasource">jdbc/oswf</property>
    </provider>
    <provider class="com.opensymphony.user.provider.jdbc.JDBCCredentialsProvider">
        <property name="user.table">os_user</property>
        <property name="group.table">os_group</property>
        <property name="membership.table">os_membership</property>
        <property name="user.name">username</property>
        <property name="user.password">passwordhash</property>
        <property name="group.name">groupname</property>
        <property name="membership.userName">username</property>
        <property name="membership.groupName">groupname</property>
        <property name="datasource">jdbc/oswf</property>
    </provider>
    <provider class="com.opensymphony.user.provider.jdbc.JDBCProfileProvider">
        <property name="user.table">os_user</property>
        <property name="group.table">os_group</property>
        <property name="membership.table">os_membership</property>
        <property name="user.name">username</property>
        <property name="user.password">passwordhash</property>
```

```
            <property name="group.name">groupname</property>
            <property name="membership.userName">username</property>
            <property name="membership.groupName">groupname</property>
            <property name="datasource">jdbc/oswf</property>
        </provider>
        <authenticator class="com.opensymphony.user.authenticator.SmartAuthenticator"/>
    </opensymphony-user>
```

osworkflow.xml 修改如下：

```
    <osworkflow>
        <persistence class="com.opensymphony.workflow.spi.jdbc.MySQLWorkflowStore">
            <!-- For jdbc persistence, all are required. -->
            <property key="datasource" value="jdbc/oswf"/>
            <!-- McKoi -->
            <property key="entry.sequence" value="SELECT max(ID)+1 FROM
OS_WFENTRY"/>
            <property key="step.sequence" value="SELECT max(ID)+1 FROM
OS_STEPIDS"/>

            <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="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"/>

            <!—mysql 特殊配置-->
            <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"/>
</persistence>
<factory class="com.opensymphony.workflow.loader.XMLWorkflowFactory">
<property key="resource" value="workflows.xml"/>
</factory>
</osworkflow>

对 osworkflow.xml 的一点说明：如果说是 mysql 数据库如上即可，如果是别的数据库，就不需要 mysql 特殊配置四项，并且要把 persistence 的 class 改为 com.opensymphony.workflow.spi.jdbc.JDBCWorkflowStore

propertyset.xml 配置如下,**红色**部分为数据源名称：

<propertysets>
<propertyset name="jdbc" class="com.opensymphony.module.propertyset.database.JDBCPropertySet">
<arg name="datasource" value="**jdbc/oswf**"/>
<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>

第四步：将 osworkflow 解压包\src\etc\deployment\jdbc 里面相应数据库的 sql 导到库中。

SpringHibernateWorkflowStore

第一步：配置 web.xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<display-name>OSWorkflow Example App</display-name>
<description>OSWorkflow Example App</description>
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/classes/applicationContext-hibernate3.xml
</param-value>
</context-param>
<listener>

```
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
        </listener>
        <servlet>
            <servlet-name>SOAPWorkflow</servlet-name>
<servlet-class>com.opensymphony.workflow.soap.SOAPWorkflowServlet</servlet-class>
        </servlet>
        <servlet-mapping>
            <servlet-name>SOAPWorkflow</servlet-name>
            <url-pattern>/soap/*</url-pattern>
        </servlet-mapping>
        <welcome-file-list>
            <welcome-file>default.jsp</welcome-file>
        </welcome-file-list>
    </web-app>
```

第二步：配置数据源，同上。

第三步：配置 OSWorkflow 必要文件。

确定在\%webapp%\WEB-INF\classes\下面有以下几个文件：

example.xml:例子，不作修改。

workflows.xml:例子，不作修改。

osuser.xml,因为 osuser 是比较独立的模块，目前还没有支持 hibernate3，修改可以同上，这就意味着 osuser 还是要用 JDBC 存储方式。

osworkflow.xml 由于是采用的 hibernate3，所以便要去掉这个文件，而增加一个 spring 配置文件 applicationContext-hibernate3.xml，其配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>java:comp/env/jdbc/oswf</value>
        </property>
    </bean>
    <!-- Hibernate SessionFactory -->
    <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource">
            <ref bean="dataSource"/>
        </property>
        <property name="mappingResources">
            <list>

<value>com/opensymphony/workflow/spi/hibernate3/HibernateWorkflowEntry.hbm.xml</value>
<value>com/opensymphony/workflow/spi/hibernate3/HibernateHistoryStep.hbm.xml</value>
```

```
<value>com/opensymphony/workflow/spi/hibernate3/HibernateCurrentStep.hbm.xml</value>
<value>com/opensymphony/workflow/spi/hibernate3/HibernateCurrentStepPrev.hbm.xml</value
>
<value>com/opensymphony/workflow/spi/hibernate3/HibernateHistoryStepPrev.hbm.xml</value
>
<value>com/opensymphony/module/propertyset/hibernate3/PropertySetItem.hbm.xml</value>
                    </list>
                </property>
                <property name="hibernateProperties">
                    <props>
                        <prop
key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                        <prop key="hibernate.show_sql">false</prop>
                    </props>
                </property>
            </bean>
            <!-- Transaction manager for a single Hibernate SessionFactory (alternative to JTA) -->
            <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
                <property name="sessionFactory">
                    <ref bean="sessionFactory"/>
                </property>
            </bean>
            <!-- Transaction template for Managers, from:
    http://blog.exis.com/colin/archives/2004/07/31/concise-transaction-definitions-spring-11/ -->
            <bean id="txProxyTemplate" abstract="true"
              class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
                <property name="transactionManager" ref="transactionManager"/>
                <property name="proxyTargetClass" value="true"/>
                <property name="transactionAttributes">
                    <props>
                        <prop key="do*">PROPAGATION_REQUIRED</prop>
                        <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
                    </props>
                </property>
            </bean>
            <bean id="propertySetDelegate"
class="com.opensymphony.workflow.spi.hibernate3.DefaultHibernatePropertySetDelegate">
                <property name="sessionFactory">
                    <ref bean="sessionFactory"/>
                </property>
            </bean>
            <bean id="workflowStore"
class="com.opensymphony.workflow.spi.hibernate3.SpringHibernateWorkflowStore">
```

```
            <property name="sessionFactory">
                <ref bean="sessionFactory"/>
            </property>
            <property name="propertySetDelegate">
                <ref bean="propertySetDelegate"/>
            </property>
        </bean>
        <bean id="workflowFactory"
class="com.opensymphony.workflow.spi.hibernate.SpringWorkflowFactory" init-method="init">
            <property name="resource">
                <value>workflows.xml</value>
            </property>
            <property name="reload">
                <value>true</value>
            </property>
        </bean>
        <bean id="osworkflowConfiguration"
class="com.opensymphony.workflow.config.SpringConfiguration">
            <property name="store">
                <ref local="workflowStore"/>
            </property>
            <property name="factory">
                <ref local="workflowFactory"/>
            </property>
        </bean>
        <bean id="workflowTypeResolver"
class="com.opensymphony.workflow.util.SpringTypeResolver">
        </bean>
    </beans>
```

由配置文件可以看出和官方或者一般的 hibernate3 配置方式有很大的不同。在数据库中就增加了和 JDBCWorkflowStore 数据库相同的两张表 os_current_step_prev 还有 os_history_step_prev.另外 os_propertyentry 表结构和 JDBCWorkflowStore 中的 os_propertyentry 表结构有所不同，对比如下：

JDBCWorkflowStore. os_propertyentry

```
create table OS_PROPERTYENTRY
(
    GLOBAL_KEY varchar(250) NOT NULL,
    ITEM_KEY varchar(250) NOT NULL,
    ITEM_TYPE tinyint,
    STRING_VALUE varchar(255),
    DATE_VALUE datetime,
    DATA_VALUE blob,
    FLOAT_VALUE float,
```

NUMBER_VALUE numeric,
primary key (GLOBAL_KEY, ITEM_KEY)
)TYPE=InnoDB;

SpringHibernateWorkflowStore. os_propertyentry

```
CREATE TABLE `os_propertyentry` (
  `entity_name` varchar(125) NOT NULL,
  `entity_id` bigint(20) NOT NULL,
  `entity_key` varchar(255) NOT NULL,
  `key_type` int(11) default NULL,
  `boolean_val` int(1) default '0',
  `double_val` double default NULL,
  `string_val` varchar(255) default NULL,
  `long_val` bigint(20) default NULL,
  `int_val` int(11) default NULL,
  `date_val` date default NULL,
  PRIMARY KEY  (`entity_name`,`entity_id`,`entity_key`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

os_entryids 表和 os_stepids 表都不曾用到,故可以删除。

程序也有极大不同，增加了 com\opensymphony\module\propertyset\hibernate3 对 hibernate3 的 propertyset 支持；修改了 com\opensymphony\workflow\spi\hibernate3 中的 java 文件特别是 AbstractHibernateWorkflowStore.java

*.hbm.xml 文件都要加上 default-lazy=”false”，另外必需说明 HibernateHistoryStep.hbm.xml 里面的生成 ID 的 class 要设置成为 assigned，因为它的 ID 其实就是从 HibernateCurrentStep.hbm.xml 里面移过来的，所以不能自增长。

另外请读者注意：如果用的是非 mysql 数据库，你可以在 applicationContext-hibernate3.xml 里面 sessionFactory bean 里面<prop key="hibernate.show_sql">false</prop>的后面加上一个<prop key="hibernate.hbm2ddl.auto">**create-drop**</prop>，这样可以不用创建除用户和群组之外的 *.hbm.xml 表，在启动服务的时候系统会自动为*.hbm.xml 创建所对应的表。不过注意要在创建之后把它注释掉，不然每次启动服务的时候都会重新创建一次，并且上次的数据也会因此而丢失。

propertyset.xml 中增加对 hibernate3 的支持，配置如下：

```
<propertysets>
<propertyset name="hibernate3"
class="com.opensymphony.module.propertyset.hibernate3.HibernatePropertySet"/>
</propertysets>
```

详细代码请登陆我的 blog 下载。下载地址：http://cucuchen520.javaeye.com/

JDBCTemplateWorkflowStore

如果说大家对 spring 中的 JDBCTempate 存储方式有应用的话，不防可以采用这种方式，在我的代码中集成了此功能。注：此方式所用数据库和 jdbc 方式完全相同。

第一步：配置 web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>OSWorkflow Example App</display-name>
    <description>OSWorkflow Example App</description>
    <context-param>
        <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/classes/applicationContext-jdbc.xml
            </param-value>
        </context-param>
    <listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>SOAPWorkflow</servlet-name>
<servlet-class>com.opensymphony.workflow.soap.SOAPWorkflowServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SOAPWorkflow</servlet-name>
        <url-pattern>/soap/*</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

第二步：配置数据源，同上。

第三步：配置 osworkflow 必要文件。

确定在\%webapp%\WEB-INF\classes\下面有以下几个文件：

example.xml:例子，不作修改。

workflows.xml:例子，不作修改。

osuser.xml,因为 osuser 是比较独立的模块，目前还没有支持 spring jdbcTempate，修改可以同上，这就意味着 osuser 还是要用普通 JDBC 存储方式。

osworkflow.xml 由于是采用的 jdbcTempate，所以便要去掉这个文件，而增加一个 spring 配置文件 applicationContext-jdbc.xml，其配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>java:comp/env/jdbc/oswf</value>
```

```
            </property>
        </bean>
        <!--
        <bean id="jdbcTemplateWorkflowStore"
              class="com.opensymphony.workflow.spi.jdbc.JDBCTemplateWorkflowStore">
            <property name="dataSource">
                <ref bean="dataSource"/>
            </property>
            <property name="jdbcTemplateProperties">
                <props>
                    <prop key="history.table">OS_HISTORYSTEP</prop>
                    <prop key="historyPrev.table">OS_HISTORYSTEP_PREV</prop>
                    <prop key="current.table">OS_CURRENTSTEP</prop>
                    <prop key="currentPrev.table">OS_CURRENTSTEP_PREV</prop>
                    <prop key="step.id">ID</prop>
                    <prop key="step.entryId">ENTRY_ID</prop>
                    <prop key="step.stepId">STEP_ID</prop>
                    <prop key="step.actionId">ACTION_ID</prop>
                    <prop key="step.owner">OWNER</prop>
                    <prop key="step.caller">CALLER</prop>
                    <prop key="step.startDate">START_DATE</prop>
                    <prop key="step.finishDate">FINISH_DATE</prop>
                    <prop key="step.dueDate">DUE_DATE</prop>
                    <prop key="step.status">STATUS</prop>
                    <prop key="step.previousId">PREVIOUS_ID</prop>
                    <prop key="step.sequence">SELECT max(ID)+1 FROM
OS_STEPIDS</prop>
                    <prop key="entry.sequence">SELECT max(ID)+1 FROM
OS_WFENTRY</prop>
                    <prop key="entry.table">OS_WFENTRY</prop>
                    <prop key="entry.id">ID</prop>
                    <prop key="entry.name">NAME</prop>
                    <prop key="entry.state">STATE</prop>
                </props>
            </property>
        </bean>
        -->


        <bean id="mysqlTemplateWorkflowStore"

class="com.opensymphony.workflow.spi.jdbc.MySQLTemplateWorkflowStore">
            <property name="dataSource">
                <ref bean="dataSource"/>
```

```
                </property>
                <property name="propertySetDelegate">
                    <ref bean="propertySetDelegate"/>
                </property>
                <property name="jdbcTemplateProperties">
                    <props>
                        <prop key="history.table">OS_HISTORYSTEP</prop>
                        <prop key="historyPrev.table">OS_HISTORYSTEP_PREV</prop>
                        <prop key="current.table">OS_CURRENTSTEP</prop>
                        <prop key="currentPrev.table">OS_CURRENTSTEP_PREV</prop>
                        <prop key="step.id">ID</prop>
                        <prop key="step.entryId">ENTRY_ID</prop>
                        <prop key="step.stepId">STEP_ID</prop>
                        <prop key="step.actionId">ACTION_ID</prop>
                        <prop key="step.owner">OWNER</prop>
                        <prop key="step.caller">CALLER</prop>
                        <prop key="step.startDate">START_DATE</prop>
                        <prop key="step.finishDate">FINISH_DATE</prop>
                        <prop key="step.dueDate">DUE_DATE</prop>
                        <prop key="step.status">STATUS</prop>
                        <prop key="step.previousId">PREVIOUS_ID</prop>
                        <prop key="step.sequence">SELECT max(ID)+1 FROM
OS_STEPIDS</prop>
                        <prop key="entry.sequence">SELECT max(ID)+1 FROM
OS_WFENTRY</prop>
                        <prop key="entry.table">OS_WFENTRY</prop>
                        <prop key="entry.id">ID</prop>
                        <prop key="entry.name">NAME</prop>
                        <prop key="entry.state">STATE</prop>

                        <prop key="step.sequence.increment">INSERT INTO OS_STEPIDS (ID)
values (null)</prop>
                        <prop key="step.sequence.retrieve">SELECT max(ID) FROM
OS_STEPIDS</prop>
                        <prop key="entry.sequence.increment">INSERT INTO OS_ENTRYIDS
(ID) values (null)</prop>
                        <prop key="entry.sequence.retrieve">SELECT max(ID) FROM
OS_ENTRYIDS</prop>
                    </props>
                </property>
            </bean>
            <bean id="propertySetDelegate"
class="com.opensymphony.workflow.spi.jdbc.DefaultJDBCTemplatePropertySetDelegate">
                <property name="dataSource">
```

```
                <ref bean="dataSource"/>
            </property>
        </bean>
        <bean id="workflowFactory"
class="com.opensymphony.workflow.spi.hibernate.SpringWorkflowFactory" init-method="init">
            <property name="resource">
                <value>workflows.xml</value>
            </property>
            <property name="reload">
                <value>true</value>
            </property>
        </bean>
        <bean id="osworkflowConfiguration"
class="com.opensymphony.workflow.config.SpringConfiguration">
            <property name="store">
                <ref local="mysqlTemplateWorkflowStore"/>
            </property>
            <property name="factory">
                <ref local="workflowFactory"/>
            </property>
        </bean>
        <bean id="workflowTypeResolver"
class="com.opensymphony.workflow.util.SpringTypeResolver">
        </bean>
    </beans>
```

propertyset.xml 中增加对 jdbcTempate 的支持，配置如下：

```
    <propertysets>
    <propertyset name="hibernate3"
class="com.opensymphony.module.propertyset.hibernate3.HibernatePropertySet"/>
    </propertysets>
    <propertyset name="jdbcTemplate"
class="com.opensymphony.module.propertyset.database.JDBCTemplatePropertySet">
            <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>
```

详细代码请登陆我的 blog 下载。下载地址：http://cucuchen520.javaeye.com/

HibernateWorkflowStore

这种方式我是不推荐使用的，因为它需要自己控制 Hibernate Session，而且官方的例子我也一直没有配置成功过，如果想用这种方式，必需要花一翻苦功夫改写源代码☹(((

理解 OSWorkflow(Understanding OSWorkflow)

这一章节要很详细地讲述 OSWorkflow 的核心思想。主要包 functions,conditions,actions,steps,变量之间的相互作用还有一些例子。

工作流程描述(Workflow Definition)

OSWorkflow 的心脏是工作流描述文件。这个描述文件是一个 XML 文件(虽然它不是必需的，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所有的 steps,states,transitions,functions.

- 一个工作流由多个步骤(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。
- 每一个动作最少有一个无条件结果(unconditional result)和零到多个有条件结果
- 当有多个 unconditional result 时，第一个符合的 result 会被执行，如果没有符合的 condition 或者没有定义 conditons，就会执行 unconditional result。
- 执行完当前步骤，它可能会跳转到另一个新的步骤，或者是一个 split，或者是一个 join 中去，它的状态也会随之改变，有可能是 Underway,Quened,Finished。
- 如果一个动作执行的结果是一个分支(split)，动作根据配置的 split 之 ID 进入相应的分支。
- 一个分支很可能有一个或者多个无条件结果，而没有有条件结果。这些结果指向 split 里面定义的步骤。
- register 是一个全局变量，它在工作流运行时被解析并存于 transientVars 之中，通常被应用于 functions 和 conditions
- propertyset 是持久数据，也是全局变量，如果用数据库存储，它为 os_propertyentry 表。
- transientVars 是一个传输数据的 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)，当前工作流的 context,entry,store,configuration,descriptor 等等（全局变量），另外还有用户输入的传递变量(局部变量，仅仅作用于当前步骤和下一步骤的 pre-function)。 transientVars 仅仅存在于当前工作流生命周期内。

工作流程思想(Workflow Concepts)

对比一下其它比较熟悉的工作流引擎，OSWorkflow 是独一无二的。为了完全掌握 OSWorkflow 的特性，就必需理解它的核心思想。

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更多个当前步骤(current steps)，当前步骤的状态值(status values)构成流程实例的工作流状态。事实上这个真实的状态值是真正由应用开发者或者项目管理者所决定的。一个状态也有可能是 Underway,Queued,Finished，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。
对于一个工作流进程，"transition"将一定会在工作流实例的内部发生。一旦一个步骤完成以后，它将不会再是当前步骤，通常一个新的当前步骤将会因此而立即产生，从而保证工作流的流转和延续。old-status 是指完成步骤以后的最终状态值，通常是在传递到其它步骤以前执行某一具体动作之后发生。通常完成以后的状态都是"Finished"。
传递本身就是动作的一个结果。一个步骤的诸多动作都是和传递紧密相连的。一个将被执行的动作有可能被一个用户完成，也有可能是外部的事件，或者是由一个 trigger 自动地完成。如果一个动作完成，那么一个传递必定也会随之发生。动作通常受限于特定的 group,user 或者当前工作流状态。每一个动作必需有一个 *unconditional result（默认）*。零到多个有条件结果。
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(for example, Queued, Underway, or Finished)。每一个步骤都有一个或者多个动作可以被执行从而改变这个状态。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)

Unconditional Result
对于每一个动作，需要存在一个无条件结果，叫做 unconditional-result。这个结果告诉 OSWorkflow 下一步要做什么。这个使工作流从一个状态传递到下一个状态。
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 连接各个 conditon），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结果所导致。
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 split 成两个或多个步骤和状态的组合。
✓ 一个 join 将这个和其他的步骤和状态组合成一个新的单一的步骤和状态。
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_8.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。
单一步骤(step)和状态(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传递到下其它的步骤，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}"/>
```

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>
```

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>
```

上面这段代码中最需要关心的就应该是 jn。当 join 实际发生的时候，这个特殊的变量 jn 可以被用来建立表达式。本质上，可以很容易理解出这个段 xml 的意思就是：**当 step6 和 step8 都完成时候在此处进行 join。**

外部函数(External Functions)

OSWorkflow 为外部的商业逻辑和需要定义执行的服务提供了一个标准的方法，这就是 functions。一个 function 通常用于封装外部的功能，它可以改变外部的实体或和工作流相关的信息系统，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:pre-function and post-function

pre-function 是在工作流执行某一特定的传递之前进行的。一个例子就是预置 caller 的名字到变量中去，在 result 中调用这个 caller，另外一个例子就是找到一个动作的 mostRecentCaller。以上两种在工作流的工具包中都有支持，而且是相当有用处的。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>
```

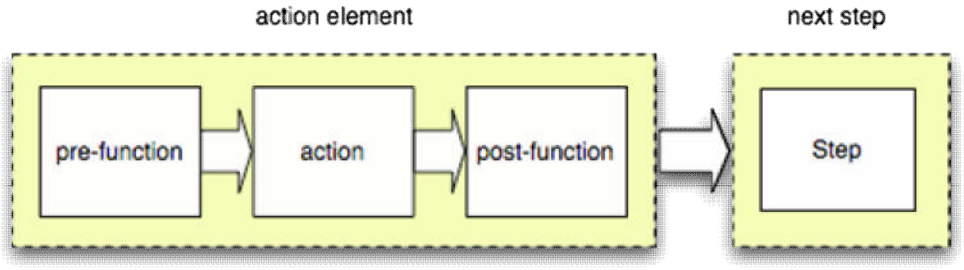
例子二：

```
<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>
```

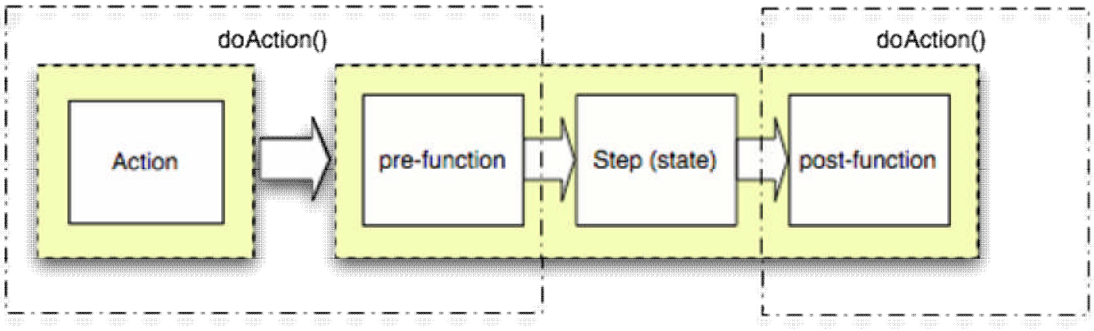
post-function 和 pre-function 类型差不多，只不过它发生在状态改变以后，最典型的例子就是当执行完某个动作以后发邮件给相关的人员。

对于 post-function 和 pre-function 有很多值得注意的地方。一个就是当一个用户点了两次完成按钮发送两个执行命令，这个动作将会花很长的时间去完成，很有可能这个长 function 将会花很长的时间，因为传递并未真正开始，OSWorkflow 会认为第二次调用的动作是合法的。如果一个 function 前置与后置效果一样，最好后置，这样节省开销。通常 pre-function 是要比较简单而快捷的，而 post-function 就是”等菜来”。

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。常见的情况是一个 function 被用于处理某些事情，如果要通知给第三方，发送邮件或者设定变量为以后所用。
下面的图解用以说明 action 和 functions 之间的关系。



如果是在一个 step 中定义的 pre-function 和 post-function，这个用法有所不同，pre-function 将会在工作流传递到这个 step 之前执行。注意这个 pre-function 将会被工作流将要到达的所有步骤调用到，甚至是自己本身的步骤(举个例子，如果状态从 Queued 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。
下图的图解说明了调用顺序。注意 action 虚线内省略了一些内容，并未标明它自己的 per-function 和 post-function。



您可以在以下的**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与一个动作相联系的。他们也是通过一个唯一的 ID 来标识。这些函数通常是运行在系统级用户或非正规用户的环境中。Trigger functions 是通过 OSWorkflow 的 API 来使用。

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。
Validators 和 Functions 有着相同的规则，你可以在以下 Validators 章节寻找更多信息。

注册器(Registers)

A register 是一个辅助函数，这个函数可以返回一个对象，这个对象可以用在函数中去访问其他的普通对象，特别是出现在 workflow 中的实体。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。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。Conditions 通常是与 conditional results 联系的，只有当条件成立，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。你可以在以下 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。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 标签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是参数(args)的类型，如果参数是一个变量的话，这个参数将不会是 String 类型，取而代之的是这个任意此变量的类型，如果这个参数是字符串加上变量混合型的话，传入的变量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)

Permissions 可被应用于基于状态的工作流实例中的用户和/或群组。这些 Permission 和工作流引擎的功能无关，但是如果拥有它们会对扩展 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"使之能够为某个特殊的群组在编辑文档(Document Edit)的时候发挥作用。通常这样的方式便可以使用 API 去判断是否

文件应该被修改。当"file-write-permission"被应用，这在工作流内是非常有用的，它将发挥其巨大的作用，这样的 Permissions 非常简单，它可以替代检查特定的步骤或条件。
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>
```

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。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。

和抽象实体的集成(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。

工作流程实例状态(自 2.6 版开始有效)-Workflow Instance State (Available since OSWorkflow 2.6)

有的时候，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 workflow 实例中可以包含的 state(os_wfentry.state)。这些 state 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。

注：在数据库中

CREATED = 0;ACTIVATED = 1;SUSPENDED = 2;KILLED = 3;COMPLETED = 4;UNKNOWN = -1;

普通动作和全局动作(Common and Global Actions)

当我们在定义工作流描述文件的时候，使用普通动作(common action)和全局动作(global action)这两个方便的特性可以帮助我们成倍减少代码量。
最简单的做法是，这两种类型的动作都可以定义在描述文件的头部，但是要在 initial-actions 之后。
普通动作在某些场合是非常有用的，比如说别的动作都需要一个相同功能的时候。举个例子如发送邮件功能。此时发送邮件就是一个动作，它像 pre-function 或者 validator 一样大约需要 7~8 行 script 代码才能搞定。如果每一个动作都要处理信件的话，这就成倍的增加了配置文件的代码量。还好，现在如果用 common action 只需要在头部定义一次，然后其它别的每一个动作都可以调用到它，调用的代码非常简单，如下：
<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 里面)，它并非像上面的 common action 一样被显式的进行调用，它的调用是隐式的，也就是说：一旦它被定义，它将会被其它别的所有的 action 引用到。举个例子，我们可以用全局动作去终结一个流程，不管在什么场合，我们都可以通过设置工作流的状态为 KILLED 来杀死流程，但我们也可以用它去干些别的事情，例如记录日志，或者和上面一样发送邮件。
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并不需要指明将传递工作流到哪一具体的步骤，在这种情况下通常只需要设定 step 为 0 即可。以下是例子。

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

函数(Functions)

基于 Java 的函数(Java-based Functions)

Java-based Functions（基于 Java 的函数）
基于 Java 的函数必须要实现 com.opensymphony.workflow.FunctionProvider 接口。这个接口只有一个函数——execute，这个函数有三个参数

transientVars Map

transientVars Map：是客户端代码调用 Workflow.doAction()时传进来的。这个参数可以基于用户的不同的输入使函数有不同的行为。
这个参数也包含了一些特别变量，这些变量对于访问 workflow 的信息是很有帮助的。它也包含了所有的在 Registers 中配置的变量和下面两种特别的变量：
entry： (com.opensymphony.workflow.spi.WorkflowEntry)
and
context：(com.opensymphony.workflow.WorkflowContext)。
另外还有别的如： store,configuration,descriptor 还有用户输入的 Input Map 等等

The args Map

args Map 是一个包含在所有的<function/>标签中的<arg/>标签的 Map。这些参数都是 String 类型的。这意味着<arg name="foo">this is ${someVar}</arg> 标签中定义的参数将在 arg Map 中通过"foo"，可以映射到"this is [contents of someVar]"字串。

propertySet

propertySet 包含所有的在 workflow 实例中持久化的变量。在数据库中为 os_propertyentry 表。

Java-based 的函数适用于以下类型：

class

对于一个类的函数，类加载器一定得知道你的函数所属的类的名字。这可以通过 class.name 参数来完成：

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

jndi

JNDI 函数就象类函数一样，唯一的不同是 JNDI 函数的对象一定是已经在 JNDI 树中存在了，这里需要 jndi.location 参数：

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

remote-ejb

远程 EJBs 可以在 OSWorkflow 中作为一个函数使用。EJB 的远程接口一定要扩展 com.opensymphony.workflow.FunctionProviderRemote，这里需要 ejb.location 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 与远程 EJBs 的不同在于，本地 EJBs 要扩展 com.opensymphony.workflow.FunctionProvider 接口，例如：

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

BeanShell 类型的函数(BeanShell Functions)

OSWorkflow 支持 BeanShell 作为一个 scripting 语言。到 http://www.beanshell.org/上，你可以获得更多的 BeanShell 的信息。
BeanShell 函数的类型一定要指定为 beanshell，还需要一个参数，名字叫 script。这个参数的值实际上就是要执行的 script，例如：

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

在表达式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 函数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，它允许函数访问 workflow 的持久化存储区。

和上面 Java-based 函数一样，也要使用三个变量 transientVars, args 和 propertySet，例如：

```
<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 scripts 的输出是"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 中的函数使用。

BSF 类型的函数(BSF Functions)

SWorkflow 还支持一种 Bean Scripting Framework 的函数。BSF 是 IBM AlphaWorks 小组的项目，它允许通用的 Script 语言，如 VBScript, Perlscript, Python, and JavaScript 在通用的环境中运行。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函数：

```
<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 的对象的值。在 BeanShell 函数中的缺省范围的相同的变量，可以在你的 BSF script 代码中获得。关于如何在你所选择的语言中获取变量，请阅读 BSF 用户手册。

工具函数(Utility Functions)

OSWorkflow 本身附带一些很实用的工具函数，这些函数都实现了 com.opensymphony.workflow.FunctionProvider 接口。要获取更详细的信息，请阅读这些工具函数的 javadoc 文档。下面是每个函数的简单描述，你在 com.opensymphony.workflow.util 包中可以找到所有的类。

Caller

用当前动作的执行者名字设置持久化变量 caller。

WebWorkExecutor

执行一个 WebWork 函数并且在动作结束时存储旧的 ActionContext。

EJBInvoker

调用一个 EJB 的 session bean 方法。

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持久化变量 mostRecentOwner。可选特性可以在有所有者时将变量设置为 nothing，或者返回一个内部错误。

ScheduleJob

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

UnschduleJob

删除一个 ScheduleJob 和所有与之联系的 triggers 函数。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。

验证器(Validators)

与 functions 类似，OSWorkflow 有下面几种不同形式的 validators：java-based，beanshell，和 bsf。Java-based 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）。Java-based 这种情况是通过抛出个 InvalidInputException 异常表明一个输入是不合法的，并且停止工作流 action。
在 beanshell 和 bsf 中，有一点小小不同，即使异常可以在脚本中抛出，但是不能抵达到 jre。所以在 beanshell 和 bsf 中用错误信息来完成。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，map 被用做一个 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。
- 其他情况，把值转换成 string 并且作为一个普通的错误信息来添加。

寄存器(Registers)

OSWorkflow 中的 Register 是一个运行时的变量，它能够动态的注册到 workflow 的定义文件中。Registers 在很多场合都是很有用的。例如：你想提供让 workflow 在它的描述符文件中就能访问一个实体的方法。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。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 也有三种实现方式：Java-based, BeanShell 和 BSF。

基于 Java 的寄存器(Java-based)

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 类型的寄存器(BeanShell and BSF registers)

通过 script 返回的值或对象将是你注册的对象。

注：在 registers 的接口中，你只需要一个 args Map 参数就行了，这是因为 registers 的调用根本不管用户的输入。

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，它有一个可访问的变量"log"，这个 log 变量可以在可以在整个工作流的生命期内被访问。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>
```

从代码中可以看到，我们创建了一个名为 log 的 LogRegister，还指定了一个值为 true 的参数 addInstanceId。

我们可以在 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>
```

上面的函数将输出"function called"，同时在输出前附加了 workflow 的实例 id。

条件(Conditions)

在 BSF 和 Beanshell 的 script 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"或"false"的字串，也可以是一个"true"或"false"的布尔值，甚至是一个含有 toString()的函数，其返回值为"true"或"false"的对象。
每个 condition 一定是作为一个 conditions 的子标签被定义的。当你使用"AND"类型，所有的 condition 标签的值必须都是"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。
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。

SOAP 支持(SOAP Support)

OSWorkflow 通过 SOAP 支持 remote invocation。可以用 Glue SOAP 实现 WebMethods 接口或者用开源的 XFire soap library。

使用 XFire(Using XFire)

OSWorkflow2.8 自带的例子默认和 SOAP 绑定在了一起。如果你想使用 SOAP，其配置相当的烦琐。
第一步：保证你的 **WEB-INF/lib** 下面有 xfire jar。这些文件可以在 **lib/optional/xfire** 找到。
下一步：在你的 web.xml 里面要有以下 servlet 配置。

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

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

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

当你的应用部署成功了，你可以用以下网址观看效果：
http://<server>/soap/Workflow?wsdl
当执行服务端的时候，任何 SOAP 客户端也必需工作。XFire 自己本身有客户端支持，这样允许你用和服务端相同的 classes。别的客户端如 Axis,GLUE 或者.net 也必需即开即用(out of the box)。

使用 GLUE(Using GLUE)

OSWorkflow 里面没有 GLUE,必需在 WebMethods 上下载。GLUE 在很多应用中是开放的。如果你下载了 GLUE 你可以看到认证许可(the license agreement)。只有下载 2.1 或者更高版本方支持 SOAP 和 Job Scheduling。你必需在你的 classpath 里面包含 GLUE-STD.jar。
和 XFire 一样，第一步必需在你的 web 应用中加入 GLUE servlet，这个在 GLUE 文档里面有详细的说明。SOAP 必需能够调度工作(scheduled jobs)，使用 Quartz job scheduler。
以下的例子就是 Glue 与 OSWorkflow 会话的代码。

```
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);
```

从上面的描述可以说明，你可以很正常的使用工作流接口。

使用 API(Using the API)

接口的选择(Interface Choices)

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直接使用。

(1) 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，这要以来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。

(2) 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，因为一旦用户被授权，它会从 EJB 容器中自动将用户名载入。

(3) OfbizWorkflow
OfbizWorkflow 与 BasicWorkflow 唯一不同的地方在于，通过 ofbiz 的 TransactionUtil 调用可以支持事务处理。

创建一个新的工作流(Creating a new Workflow)

下面简单介绍了如何使用 OSWorkflow 的 API 来创建一个新的工作流。首先应该创建定义工作流的文件。然后，你的程序必须知道初始化步骤的值，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，你就可以得到这个工作流的引用。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，你应该使用一个 Workflow 类型的引用，而不应该是 BasicWorkflow 的引用。
如果是创建一个 Spring 类型的工作流或者 SpringHibernate 类型的工作流，代码应该修改成如下：

```
ApplicationContext cxt =
WebApplicationContextUtils.getWebApplicationContext(this.getServletConfig().get
ServletContext());
Configuration conf = (SpringConfiguration)
cxt.getBean("osworkflowConfiguration");
Workflow wf = new BasicWorkflow(username);
wf.setConfiguration(conf);
Map inputs = new HashMap();
inputs.put("docTitle", request.getParameter("title"));
wf.initialize("workflowName", 1, inputs);
```

执行动作(Executing Actions)

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);
```

如果是一个 Spring 类型的工作流或者 SpringHibernate 类型的工作流，代码应该修改成如下：

```
ApplicationContext cxt =
WebApplicationContextUtils.getWebApplicationContext(this.getServletConfig().get
ServletContext());
Configuration conf = (SpringConfiguration)
cxt.getBean("osworkflowConfiguration");
Workflow wf = new BasicWorkflow(username);
wf.setConfiguration(conf);
HashMap inputs = new HashMap();
inputs.put("docTitle", request.getParameter("title"));
long id = Long.parseLong(request.getParameter("workflowId"));
wf.doAction(id, 1, inputs);
```

查询(Queries)

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：一个历史和当前步骤的查询信息的查询）。要执行查询，就要建立一个 WorkflowExpressionQuery 对象，然后在 WorkflowExpressionQuery 对象中调用查询方法。

```
//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 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));
```

对比隐式和显式 Configuration (Implicit vs Explicit Configuration)

在 OSWorkflow 2.7 版以前，状态的维持在很多场合都使用的是静态字段(static fields)。这种方式固然方便，但是也有一些小小的缺点。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 通过传入一个 Configuration 接口修复了这个局限性。这个 Configuration 接口的默认实现是 DefaultConfiguration，显式调用 DefaultConfiguration 是为了向以后的版本兼容。如果不显式调用 Configuration 接口将默认实现 static instance(**if no explicit call is made using a Configuration**)。这同样也是为了向以前的版本兼容。换句话说：如果显式的调用 AbstractWorkflow 中的 *setConfiguration* 方法，工作流将使用 per-instance 模式；反之如果不调用此 *setConfiguration* 方法，那么原始的 singleton static 模式将会被使用。

如果你不用 singleton static 模式创建工作流，AbstractWorkflow 将不再是无状态的了；如果你用现在的 per-instance 模式创建了一个工作流实例，你要将这个 Workflow 对象保存在某个地方，那么在以后的调用过程中便可以直接使用你先前保存的那个实例，而不用每一次都要

重新创建了，这就是 per-instance 模式的妙处。
说了这么多感觉好像挺复杂，其实实践起来非常之简单，下面举例说明：
原始方法(Legacy approach):

```
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(Recommend approach):

```
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());
```

附加的(Additional)

Input Map

workflow 的 initialize 和 doAction 方法还有 wf.getAvailableActions(id, map);都有一个 HashMap 的参数，是用来传递工作流中所需要的数据或者对象的,但是只能在当前步骤或者下一步的 pre-function 有效。可应用在 function,condition,validator 等当中。

Workflow 接口里面的主要方法

getAvailableActions(id, null)—得到当前工作流实例所有有效动作.
第一个参数:工作流实例 ID
第二个参数:要传递的对象

getCurrentSteps(id) —根据工作流实例 ID 得到所有当前步骤.
getEntryState(id) —根据工作流实例 ID 得到所有当前实例状态.
getHistorySteps(id) —根据工作流实例 ID 得到所有历史步骤.
getPropertySet(id) —根据工作流实例 ID 得到 PropertySet 对象.
getSecurityPermissions(id) —根据工作流实例 ID 得到当前状态权限列表.
getWorkflowDescriptor(id) —根据工作流实例 ID 得到工作流 XML 描述文件对象.
getWorkflowName(id) —根据工作流实例 ID 得到当前实例名称.
getWorkflowNames()—得到所有的工作流程名称.

WorkflowDescriptor 对象里面的主要方法

getAction(id) —根据动作 ID 得到当前动作描述信息.
getCommonActions()—得到通用动作列表.
getGlobalActions()—得到全局动作列表.
getGlobalConditions()—得到全局条件列表.
getInitialAction(id)—根据初始动作 ID 得到初始动作描述信息.
getInitialActions()—得到所有初始动作描述信息.
getJoin(id)—根据 Join ID 得到相应 Join 描述信息.
getJoins()—得到所有 Join 描述信息.
getName()—得到工作流程描述文件名称.
getRegisters()—得到所有的寄存器.
getSplit(id) —根据 Split ID 得到相应 Split 描述信息.
getSplits()—得到所有 Split 描述信息.
getSep(id) —根据 Step ID 得到相应 Step 描述信息.
getSeps() —得到所有 Step 描述信息.
getTriggerFunction(id)—根据 TriggerFunction ID 得到相应 TriggerFunction 描述信息.
getTriggerFunctions()—得到所有 TriggerFunction 描述信息.

OSWorkflow 包的描述**(OSWorkflow Packages Description)**

说明:所有包均以 com.opensymphony.workflow 开头

	定义了 Workflow,Condition,FunctionProvider,Register,Validator 等接口,还定义了一些常用的 Exception 和很重要的 JoinNodes 对象等
basic	只有两个对象:BasicWorkflow,未实现事务回滚;BasicWorkflowContext:上下文对象的 Basic 实现方式
config	里面主要是读取 XML 配置文件的一些常用类
ejb	Workflow 的 EJB 实现方式,主要对象是 EJBWorkflow 和 EJBWorkflowContext
loader	在工作流初始化时必需要加载的一些类.主要有 WorkflowFactory 接口及 XMLWorkflowFactory 实现类,还有 WorkflowDescriptor 等描述 XML 配置文件的实体对象等
ofbiz	Workflow 的 ofbiz 实现方式,只有两个对象分别是 OfbizWorkflow 和 OfbizWorkflowContext
query	OSWorkflow 查询分析器,完成各种单一或者嵌套或者组合查询.最主要的对象应该是 WorkflowExpressionQuery
soap	Workflow 的 soap 实现方式
spi	与后台打交道的关于 workflow 存储的一些类或者对象有 WorkflowEntry 接口,此接口返回一个工作流实例对对象;WorkflowStore 接口非常重要,里面定义的全部都是工作流的存储及查询方法,JDBCWorkflowStore 实现 SimpleWorkflowEntry,hibernate 或者 spring+hibernate 实现 HibernateWorkflowEntry;Step 接口被 SimpleStep 或者 HibernateStep 调用,SimpleStep 是 JDBCWorkflowStore 实现方式，HibernateStep 是 HibernateWorkflowStore 或者 SpringHibernateWorkflow 实现方式
spi.ejb	EJB 形式的存储实现方式
spi.hibernate	Hibernate2 或者 spring+hibernate2 形式的存储实现方式
spi.hibernate3	Hibernate3 或者 spring+hibernate3 形式的存储实现方式,目前这种是主流方式，代码我有作重大修改，详细敬请下载我的源代码；另外请大家注意：hibernate 不支持复合查询
spi.jdbc	Jdbc 形式的存储实现方式,非常常见但又非常经典的方式！另外我扩展了一个 springTemplate 实现方式，详细请大家看文档的 JDBCTemplateWorkflowStore 部分及相应源代码
spi.memory	Memory 形式的存储实现方式,例子里面默认就是用的这种方式.个人认为只可作示例用，不推荐在项目中使用
spi.ofbiz	Ofbiz 形式的存储实现方式，非常少见的方式，略
spi.ojb	org.apache.ojb 形式的存储实现方式
spi.prevayler	org.prevayler 形式的存储实现方式
timer	定时器，用来处理工作流中的定时任务

util	主要是扩展 OSWorkflow 用以实现对 WebWork. jndi, jmail，springframework, Log4j，OSUser 等等的支持
util.beanshell	Beanshell 形式的 condition, function, registor 和 validator 实现方式
util.bsf	bsf 形式的 condition, function, registor 和 validator 实现方式
util.ejb	EJB 的本地与远程两种形式的 condition, function, registor 和 validator 实现方式
util.jndi	jndi 形式的 condition, function, registor 和 validator 实现方式

OSWorkflo 数据库的描述(OSWorkflow Database Description)

JDBCWorkflowStore 和 JDBCTemplateWorkflowStore 这两种方式所用的表结构完全相同。现将所要用到的表结构开列如下并对每一张表每一字段进行详细说明。在此我不对 hibernate 表结构进行说明，因为有一点点的不同，请自行到我网站上下载源代码进行对比。
注：以下表结构全部以 mysql5.0 为例。

os_currentstep

```
CREATE TABLE `os_currentstep` (
  `ID` bigint(20) NOT NULL,
  `ENTRY_ID` bigint(20) default NULL,
  `STEP_ID` int(11) default NULL,
  `ACTION_ID` int(11) default NULL,
  `OWNER` varchar(35) default NULL,
  `START_DATE` datetime default NULL,
  `FINISH_DATE` datetime default NULL,
  `DUE_DATE` datetime default NULL,
  `STATUS` varchar(40) default NULL,
  `CALLER` varchar(35) default NULL,
```

```
  PRIMARY KEY  (`ID`),
  KEY `ENTRY_ID` (`ENTRY_ID`),
  KEY `OWNER` (`OWNER`),
  KEY `CALLER` (`CALLER`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

ALTER TABLE `os_currentstep`
  ADD FOREIGN KEY (`ENTRY_ID`) REFERENCES `os_wfentry` (`ID`),
  ADD FOREIGN KEY (`OWNER`) REFERENCES `os_user` (`USERNAME`),
  ADD FOREIGN KEY (`CALLER`) REFERENCES `os_user` (`USERNAME`);
```

ID：当前步骤的 ID 号。
ENTRY_ID：工作流程实例 ID，与 os_wfentry 的 ID 相关联。
STEP_ID：当前步骤 ID。
ACTION_ID：当前动作 ID。
OWNER：当前状态下流程所有者，与 os_user 用户 ID 关联。
START_DATE：当前流程开始时间。
FINISH_DATE：当前流程结束时间。
DUE_DATE：好像并没有起什么作用☹(((
STATUS：流程当前状态。
CALLER：当前流程调用者，与 os_user 用户 ID 关联。

os_currentstep_prev

```
CREATE TABLE `os_currentstep_prev` (
  `ID` bigint(20) NOT NULL,
  `PREVIOUS_ID` bigint(20) NOT NULL,
  PRIMARY KEY  (`ID`,`PREVIOUS_ID`),
  KEY `ID` (`ID`),
  KEY `PREVIOUS_ID` (`PREVIOUS_ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

ALTER TABLE `os_currentstep_prev`
  ADD FOREIGN KEY (`ID`) REFERENCES `os_currentstep` (`ID`),
  ADD FOREIGN KEY (`PREVIOUS_ID`) REFERENCES `os_historystep` (`ID`);
```

ID：当前步骤 ID，与 os_currentstep 的 ID 作关联。
PREVIOUS_ID：当前步骤之前步骤 ID，与 os_historystep 的 ID 作关联。

os_historystep

```
CREATE TABLE `os_historystep` (
```

```
  `ID` bigint(20) NOT NULL,
  `ENTRY_ID` bigint(20) default NULL,
  `STEP_ID` int(11) default NULL,
  `ACTION_ID` int(11) default NULL,
  `OWNER` varchar(35) default NULL,
  `START_DATE` datetime default NULL,
  `FINISH_DATE` datetime default NULL,
  `DUE_DATE` datetime default NULL,
  `STATUS` varchar(40) default NULL,
  `CALLER` varchar(35) default NULL,
  PRIMARY KEY  (`ID`),
  KEY `ENTRY_ID` (`ENTRY_ID`),
  KEY `OWNER` (`OWNER`),
  KEY `CALLER` (`CALLER`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

ALTER TABLE `os_historystep`
  ADD FOREIGN KEY (`ENTRY_ID`) REFERENCES `os_wfentry` (`ID`),
  ADD FOREIGN KEY (`OWNER`) REFERENCES `os_user` (`USERNAME`),
  ADD FOREIGN KEY (`CALLER`) REFERENCES `os_user` (`USERNAME`);
```

ID：历史步骤的 ID 号。
ENTRY_ID：工作流程实例 ID，与 os_wfentry 的 ID 相关联。
STEP_ID：历史步骤 ID。
ACTION_ID：历史动作 ID。
OWNER：历史状态下流程所有者，与 os_user 用户 ID 关联。
START_DATE：历史流程开始时间。
FINISH_DATE：历史流程结束时间。
DUE_DATE：好像并没有起什么作用☹(((
STATUS：流程历史状态。
CALLER：历史流程调用者，与 os_user 用户 ID 关联。

os_historystep_prev

```
CREATE TABLE `os_historystep_prev` (
  `ID` bigint(20) NOT NULL,
  `PREVIOUS_ID` bigint(20) NOT NULL,
  PRIMARY KEY  (`ID`,`PREVIOUS_ID`),
  KEY `ID` (`ID`),
  KEY `PREVIOUS_ID` (`PREVIOUS_ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

ALTER TABLE `os_historystep_prev`
  ADD FOREIGN KEY (`ID`) REFERENCES `os_historystep` (`ID`),
```

ADD FOREIGN KEY (`PREVIOUS_ID`) REFERENCES `os_historystep` (`ID`);
ID：历史步骤 ID，与 os_historystep 的 ID 作关联。
PREVIOUS_ID：历史步骤之前步骤 ID，与 os_historystep 的 ID 作关联。

os_wfentry

```
CREATE TABLE `os_wfentry` (
  `ID` bigint(20) NOT NULL,
  `NAME` varchar(60) default NULL,
  `STATE` int(11) default NULL,
  PRIMARY KEY  (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

ID：流程的 ID 号。
NAME：流程名称。
STATE：流程状态。有五种：CREATED = 0;ACTIVATED = 1;SUSPENDED = 2;KILLED = 3;COMPLETED = 4;UNKNOWN = -1。

os_entryids

```
CREATE TABLE `os_entryids` (
  `ID` bigint(20) NOT NULL auto_increment,
  PRIMARY KEY  (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

说明：hibernate 无此表。
ID：由数据库自动生成的 workflow_entry_id。供 os_wfentry 表用。

os_stepids

```
CREATE TABLE `os_stepids` (
  `ID` bigint(20) NOT NULL auto_increment,
  PRIMARY KEY  (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

说明：hibernate 无此表。
ID：由数据库自动生成的 step_id。供 os__currentstep 表或 os_historytstep 表使用。

os_propertyentry

```
CREATE TABLE `os_propertyentry` (
  `GLOBAL_KEY` varchar(250) NOT NULL,
  `ITEM_KEY` varchar(250) NOT NULL,
  `ITEM_TYPE` tinyint(4) default NULL,
  `STRING_VALUE` varchar(255) default NULL,
```

```
  `DATE_VALUE` datetime default NULL,
  `DATA_VALUE` blob,
  `FLOAT_VALUE` float default NULL,
  `NUMBER_VALUE` bigint(10) default NULL,
  PRIMARY KEY  (`GLOBAL_KEY`,`ITEM_KEY`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

说明：hibernate 的 os_ propertyentry 表与此表有点差别。

GLOBAL_KEY：全局变量名，如果为 JDBCWorkflowStore 即为 osff_加上流程实例 ID；如果为 JDBCTemplateWorkflowStore 即为 osff_temp_加上流程实例 ID；

ITEM_KEY：局部变量名，通常是指真正要在程序中调用的”key”。

ITEM_TYPE：以数字标明其 property 的类型。如 5 是 String 型。

DATE_VALUE：如果是 Date 类型的数据，存入这个字段里。

DATA_VALUE：如果是 Data 类型的数据，存入这个字段里。

FLOAT_VALUE：如果是 Float 类型的数据，存入这个字段里。

NUMBER_VALUE：如果是 Number 类型的数据，存入这个字段里。

os_user

```
CREATE TABLE `os_user` (
  `USERNAME` varchar(100) NOT NULL,
  `PASSWORDHASH` mediumtext,
  PRIMARY KEY  (`USERNAME`),
  KEY `USERNAME` (`USERNAME`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

USERNAME：用户名，主键。

PASSWORDHASH：用户密码。

os_group

```
CREATE TABLE `os_group` (
  `GROUPNAME` varchar(20) NOT NULL,
  PRIMARY KEY  (`GROUPNAME`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

GROUPNAME：群组名，主键

os_membership

```
CREATE TABLE `os_membership` (
  `USERNAME` varchar(20) NOT NULL,
  `GROUPNAME` varchar(20) NOT NULL,
  PRIMARY KEY  (`USERNAME`,`GROUPNAME`),
  KEY `USERNAME` (`USERNAME`),
```

```
  KEY `GROUPNAME` (`GROUPNAME`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

USERNAME：用户名，主键。
GROUPNAME：群组名，主键

```
ALTER TABLE `os_membership`
  ADD FOREIGN KEY (`USERNAME`) REFERENCES `os_user` (`USERNAME`),
  ADD FOREIGN KEY (`GROUPNAME`) REFERENCES `os_group` (`GROUPNAME`);
```

以上三张表对真正的 OSWorkflow 没有多大的关系，这是标准而经典的权限模式，用户和群组是多对多的关系，我不想做深入讨论。尤其要注意的是：如果不用这三张表，osuser.xml 可以不用配置，但在 XML 流程定义文件里就不能再引用 **com.opensymphony.workflow.util.OSUserGroupCondition** 了，切记切记！
有人要问，如果不用那用户和权限怎么办？我想这恰巧就是 OSWorkflow 的灵活之处，建自己的用户和权限表就 OK 了，想怎么建都成啊。呵呵。。。
Hibernate 的数据库我就不多说了，大同小异，不过和以上的表还是有点差别的。自己琢磨吧☺)))
一点补充：在我的网站 http://cucuchen520.javaeye.com/
上面的 osworkflow2.8.rar 里面有全套的代码及数据库 script 文件和配置文件。详情请您自行下载。

OSWorkflow 核心代码剖析

initialize 方法

初始化方法主要完成了以下几个功能

- 得到 WorkflowStore 实现类，利用里面的 createEntry 方法创建一个 WorkflowEntry 对象。
- 执行 populateTransientMap 方法，将 context(WorkflowContext),entry(WorkflowEntry),store(WorkflowStore), configuration(Configuration), descriptor(WorkflowDescriptor)装进 transientVars；将当前要执行的 actionId 和 currentSteps 装进 transientVars；将所有 XML 中配置的 register 装进 transientVars。
- 根据 restrict-to 里面配置的条件来判断是否能初始化工作流，如果不能则回滚并抛出 InvalidRoleException 异常。
- 执行 transitionWorkflow 方法传递工作流，这个 transitionWorkflow 方法是 Workflow 中重中之重的方法，以下会进行详细讲解。
- 返回当前工作流程实例 ID。

transitionWorkflow 方法

transitionWorkflow 方法是工作流最最核心的方法，它主要是完成以下功能

- 利用 getCurrentStep 方法取得当前步骤：如果只有一个有效当前步骤，直接返回；如果

有多个有效当前步骤，返回符合条件的第一个。

- 调用动作验证器来验证 transientVars 里面的变量。
- 执行当前步骤中的所有 post-function。
- 执行当前动作中的所有 pre-function。
- 检查当前动作中的所有有条件结果，如果有符合条件的，验证 transientVars 里面的变量并执行有条件结果里面的 pre-function；如果动作里面没有一个有条件结果，执行无条件结果，验证 transientVars 里面的变量并执行无条件结果里面的 pre-function。
- 如果程序进入到一个 split 中：1）验证 transientVars 里面的变量；2）执行 split 中的 pre-functions；3）如果动作中的 finish 不等于 true，执行 split 中所有的 result；4)结束当前步骤并将其移至历史步骤中去，创建新的步骤并执行新步骤中的 pre-function；5)执行 split 中的 post-function。
- 如果程序进入到一个 join 中：1) 结束当前步骤并将其移至历史步骤中去；2)将刚结束的步骤和在 join 中的当前步骤还有历史步骤加到 joinSteps 集合中并产生 JoinNodes 对象，将此对象 put 到 transientVars 里；3）检查 join 条件；4）执行 join 有条件结果中的 validator；5)执行 join 有条件结果中的 pre-function；6)如果当前步骤不在历史步骤里面，把它移到历史步骤里面去；7) 如果刚刚结束的当前动作中的 finish 不等于 true，创建新的步骤并执行新步骤中的 pre-function；8）执行 join 中的 post-function。
- 如果程序进入到另一个 step 中：结束当前步骤并将其移至历史步骤中去，创建新的步骤并执行新步骤中的 pre-function。
- 如果动作里面有符合条件的有条件结果，执行有条件结果里面的 post-function；如果动作里面没有一个有条件结果，则执行无条件结果里面的 post-function。
- 执行动作里面的 post-function。
- 如果动作一开始是一个初始状态，将设置 ACTIVATED 标识；如果动作在 XML 里面有完成状态的标识，将设置 COMPLETED 标识。
- 执行有效的自动动作(auto action)。
- 最后返回流程是否完成的布尔值：如果流程实例已经完结，返回 true；否则返回 false。

doAction 方法

- 判断工作流程实例的状态，如果状态不为 ACTIVATED(1)，直接返回。
- 利用 findCurrentSteps 方法得到当前所有步骤列表。
- 执行 populateTransientMap 方法，将 context(WorkflowContext),entry(WorkflowEntry),store(WorkflowStore), configuration(Configuration), descriptor(WorkflowDescriptor)装进 transientVars；将当前要执行的 actionId 和 currentSteps 装进 transientVars；将所有 XML 中配置的 register 装进 transientVars。
- 检查全局动作(Global Action)和当前步骤里面所有动作的有效性。如果有无效动作，直接抛出 InvalidActionException 异常。
- 执行 transitionWorkflow 方法传递工作流，如果捕获到 WorkflowException，抛出异常并回滚。
- 如果没有动作中没有显式地标明 finish 的状态为 true，那么这时要执行 checkImplicitFinish 方法，查找当前步骤中是否还有有效动作，如果没有一个有效动作，则直接调用 completeEntry 方法结束流程并将流程的状态设置成为 COMPLETED(4)。

如何绑定现有系统

有很多网友问我如何绑定 OSWorkflow 到现有的系统(常见的如 OA)中去，其实这是一件非常容易的事情，其实在以上工作流程思想章节：和抽象实体的集成中有所提及，在此我更详细的解释如下：

在初始化一个新的工作流时，必需要在你的 Service 层执行以下方法：

```
public long doInitialize(String un)throws Exception{
    Workflow wf = new BasicWorkflow(un);
    Long wf_id = -1;
try {
     wf_id = wf.initialize("example", 100, null);
}
catch (Exception e) {
     throw e;
}
return wf_id;
}
```

这时候要加入自己的业务逻辑代码，看起来如下：

```
this.getJdbcTemplate().update("insert into t_doc (wf_id,title,content) values (?,?,?)",
new Object[]{wf_id,"my title","my content"});
```

t_doc 有四个字段：id(自动增长)；wf_id（非常重要，绑定工作流 ID）；title(文档标题), 要从创建工作流的前台 jsp 中传过来；content（文档内容）也要从创建工作流的前台 create_doc.jsp 中传过来。

create_doc.jsp 中的 form 表单提供三个参数：用户名；文档标题和文档内容；另加一个提交按钮提到后台处理☺)))

后台 Service 合并起来的代码应该如下：

```
public long doInitialize(String un,String title,String content)throws Exception{
    Workflow wf = new BasicWorkflow(un);
    Long wf_id = -1;
try {
     wf_id = wf.initialize("example", 100, null);
     this.getJdbcTemplate().update("insert into t_doc (wf_id,title,content) values (?,?,?)",
     new Object[]{wf_id,title,content});
}
catch (Exception e) {
     throw e;
}
return wf_id;
}
```

后台中的 Action 代码看起来如下：

```
String un = request.getParameter("un");
String title = request.getParameter("title");
```

String content = request.getParameter(“content”);
Long wf_id = workflowSerivce. doInitialize(un,title,content);
//根据 wf_id 来返回刚刚存入数据库中的 title 和 content 等信息。
DocumentVO vo = workflowSerivce.getDocumentVO(wf_id);
request.setAttribute(“un”,un);
request.setAttribute(“wf_id”,wf_id);
request.setAttribute(“title”,vo.getTitle());
request.setAttribute(“content”,vo.getContent());

这时 Struts 中的 Action 要前转到显示文档的 view_doc.jsp 中去，并把工作流 ID(wf_id)和用户 ID(un),title,content 统统带过去。

在 view_doc.jsp 中，列出所有权限和有效动作，并可以列出绑定文档的相关信息。

在进行文档传输的时候，绑定的方法和上面类似，如下：

view_doc.jsp 中的 form 表单提供五个参数：用户名，流程 ID，动作 ID，文档标题和文档内容。

后台 Service 合并起来的代码应该如下：

```
public void doAction(String un,long wf_id,int action_id,String title,String content)throws Exception{
    Workflow wf = new BasicWorkflow(un);
  try {
      wf.doAction(wf_id, action_id, null);
      this.getJdbcTemplate().update(“update t_doc set title=?,content=? where wf_id=?”,
      new Object[]{title,content,wf_id});
  }
  catch (Exception e) {
      throw e;
  }
  }
```

后台中的 Action 代码看起来如下：

```
String un = request.getParameter(“un”);
Long wf_id =Long.valueOf( request.getParameter(“wf_id”));
Int action_id =Integer.valueOf(request.getParameter(“action_id”));
String title = request.getParameter(“title”);
String content = request.getParameter(“content”);
workflowSerivce. doAction(un,wf_id,action_id,title,content);
//根据 wf_id 来返回刚刚存入数据库中的 title 和 content 等信息。
DocumentVO vo = workflowSerivce.getDocumentVO(wf_id);
request.setAttribute(“un”,un);
request.setAttribute(“wf_id”,wf_id);
request.setAttribute(“title”,vo.getTitle());
request.setAttribute(“content”,vo.getContent());
```

这时 Struts 中的 Action 要前转到显示文档的 view_doc.jsp 中去，也要把工作流 ID(wf_id)

和用户 ID(un),title,content 这些重要信息统统带过去。

从 2.7 版升级(Migrating from version 2.7)

Descriptor 的改变(Descriptor changes)

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 changes)

Register API 改变了，增加了 PropertySet 参数，如下：

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;
}
```

后记(Afterword)

V1.1 和第一版相比，主要是更正了不太恰当的语法；参照官方的文档重新整理了一下目录结构；另外增加的章节有

- ✓ Common and Global actions。
- ✓ Implicit vs Explicit Configuration
- ✓ Migrating from version 2.7

V1.2 和 V1.1 相比，主要是依照网友的要求更改了标题为中文加英文对照模式，的确这样效果更佳☺；更正了 os_currentstep 表和 os_historystep 表中 OWNER 和 STATUS 字段错误的描述；在 SpringHibernateWorkflowStore 里面加了用 create-drop 自动创建非 mysql 数据库的描述。

V2.0 和 V1.2 相比，主要是增加了 OSWorkflow 核心源代码剖析和如何绑定现有系统两大章节，至此，此文档才真正算是比较完善的 OSWorkflow 中文文档了☺

申明：本文档版权归陈刚(Chris Chen)所有，由于本人花了大量心血来写这份文档，如有转载，请注明出处。

在本人写此文档时很大部分参考了 OSWorkflow 的官方文档，同时也查阅了相关牛人们写的文章，在此由衷地感谢他们！本人发布此文档的目的是在于我个人认为 OSWorkflow 是一个十分优秀且灵活的工作流引擎，重在推广！使大家都来了解它、理解它，从而能更好的应用它！由于本人的水平有限，如有疏漏，恳请大家批评指正！本人联系方式如下：

MSN:cucuchen520@hotmail.com

Email:cucuchen520@yahoo.com.cn