

# JBPM开发指南

# 目 录

一、概述.....	4
二、第一个流程.....	4
2.1、开始前的准备.....	4
2.2 、Hello World 例子.....	5
三、学习JPDL.....	9
3.1 、简介.....	9
3.2 、流程版本 (Version) .....	10
3.3 、流程定义.....	11
3.3.1 process-definition(流程定义) .....	11
3.3.2 node(自动节点) .....	11
3.3.3 start-state(开始状态) .....	12
3.3.4 end-state(结束节点) .....	12
3.3.5 state(状态) .....	13
3.3.6 task-node (任务节点) .....	13
3.3.7 fork(分支) .....	15
3.3.8 join(联合) .....	16
3.3.9 decision(决策) .....	17
3.3.10 transition(转换) .....	18
3.3.11 event(事件) .....	19
3.3.12 action(动作) .....	19
3.3.13 script(脚本) .....	20
3.3.14 expression (表达式) .....	21
3.3.15 variable(变量) .....	22
3.3.16 handler(句柄) .....	22
3.3.17 timer(定时器) .....	23
3.3.18 create-timer(创建定时器) .....	24
3.3.19 cancel-timer(取消定时器) .....	25
3.3.20 task(任务) .....	25
3.3.21 swimlane(泳道) .....	26
3.3.22 assignment(委派) .....	27
3.3.23 controller(控制器) .....	28
3.3.24 process-state 子流程.....	28
3.3.25 sub-process 子流程.....	29
3.3.26 condition 条件.....	29
3.3.27 exception-handler 异常处理.....	30
小结.....	32
四、流程中任务的分配.....	35
4.1 assignment-handler方式的任务分配.....	36
4.2 swimlane方式的任务分配.....	37

五、JBPM持久化.....	39
5.1 特殊数据库支持.....	39
5.2 JBPM数据库的安装.....	39
5.2 JBPM流程发布.....	42
5.2.1 搭建JBPM的WEB应用.....	43
5.2.2 发布第一个流程.....	45
六、日历(Scheduler).....	50
6.1 Scheduler在C/S程序上的应用.....	51
6.2 Scheduler 在Web上的应用.....	53
6.3 Scheduler时间的分类.....	55
七、异步执行.....	58
八、JBPM流程建模与应用.....	58
7.1 JBPM的建模工具.....	58
7.1.1 建模工具的安装.....	59
7.2 公司报销流程示例.....	61
7.2.1 流程建模.....	61
7.2.2 流程数据库搭建.....	71
7.2.3 构建业务表.....	74
7.2.4 报销流程的发布.....	77
7.2.5 应用程序搭建.....	79
九、写在最后.....	91

## 一、概述

JBPM 是一个扩展性很强的 workflow 系统, 百分百用 JAVA 语言开发, 持久层采用 Hibernate 实现, 理论上说, 只要 Hibernate 支持的数据库 JBPM 都支持。同时它还能被部署在任何一款 JAVA 应用服务器上

## 二、第一个流程

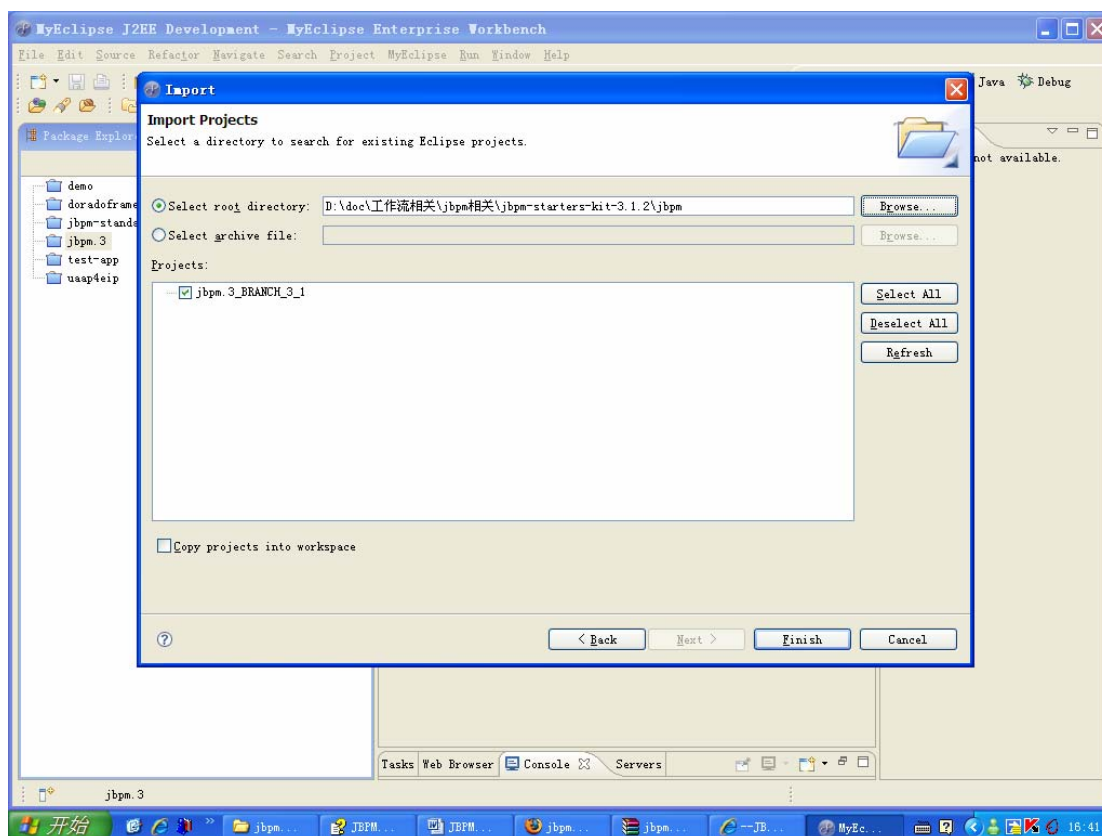
### 2.1、开始前的准备

JBPM 的工程文件, 大家可以到如下网站上去下载:

<http://www.jboss.com/products/jbpm> 目前的最新版本是 3.2.1, 本文就以此版本为例。

在这里请大家下载 `jbpm-starters-kit-3.1.2` 这样一个版本。在这个版本里包括一个 JBPM 流程设计器的 Eclipse 插件, 和一个用 JBOSS 作为服务器的示例流程等相关文件。解压 `jbpm-starters-kit-3.1.2.rar` 到某个特定目录, 这里我们首先用到的是包里的 JBPM 目录下的文件。

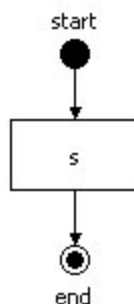
JBPM 目录里面是 JBPM 的 Eclipse 的工程文件, 我们可以用 Eclipse 导入该工程。从 Eclipse 的 File 菜单里选择 `import`——> `Existing Projects into Workspace`——> `next...` 根据向导找到前面提到的 JBPM 目录就可以把该工程导入到 Eclipse 当中。如下图:



好了，接下来，我们就在这个工程的基础之上开始我们的第一个流程。

## 2.2 、Hello World 例子

我们的第一个流程示例源自 JBPM 的 reference。流程图如下：



JBPM 的流程定义采用 XML 的方式（实际绝大多数的流程引擎的流程定义都采用的是这种方式），作为测试 XML 定义我们既可以写在代码当中，也可以以一个独立的 XML 文件的形式存在，接下来的例子我们将分别为大家介绍一下这两种情况。我们首先来看看把 XML 流程定义写在代码中的方式。

新建一个 Junit 的测试用例，测试代码如下：

```
package org.jbpm.tutorial.helloworld;

import junit.framework.TestCase;

import org.jbpm.graph.def.ProcessDefinition;

import org.jbpm.graph.exe.ProcessInstance;

import org.jbpm.graph.exe.Token;
```

```
public class HelloWorldTest extends TestCase {
```

```
    public void testHelloWorldProcess() {
```

```
        /*
```

这个段测试方法演示了一个流程的在代码中以字符串形式定义和这个流程定义的具体执行。

这个流程定义包含三个节点：一个未命名的开始状态（start-state），  
一个名字为's'的状态（state）和一个名字为'end'的结束状态（end-state）。

下一行的功能是把一段xml文本解析为一个ProcessDefinition，

一个ProcessDefinition是一个java对象的形式对流程的正式的描述。

```
        */
```

```
        ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
            "<process-definition>" +
            "    <start-state>" +
            "        <transition to='s' />" +
            "    </start-state>" +
            "    <state name='s'>" +
            "        <transition to='end' />" +
            "    </state>" +
            "    <end-state name='end' />" +
            "</process-definition>"
```

```
);
```

```
/*
```

下边的一行根据流程定义构造了一个具体的执行实例。构造以后，执行的流程就有了一个被定位在开始状态（**start-state**）上的主要的执行路径

```
*/
```

```
ProcessInstance processInstance =  
    new ProcessInstance(processDefinition);
```

```
/*
```

构造以后，执行的流程就有了一个主要的执行路径（**root token**）

```
*/
```

```
Token token = processInstance.getRootToken();
```

```
/*
```

当然，构造以后，流程定义的主要的执行路径被定位在开始状态（**start-state**）

```
*/
```

```
assertSame(processDefinition.getStartState(), token.getNode());
```

```
/*
```

开始流程执行，通过默认转换（**transition**）离开开始状态（**start-state**）

```
*/
```

```
token.signal();
```

```
/*
```

直到运行的流程进入一个等待状态，**signal** 方法将一直被阻塞，运行的流程将要进入第一个等待状态：状态 ‘**s**’。因此现在主要的执行路径，定位到了状态 ‘**s**’ 上。

```
*/
```

```
assertSame(processDefinition.getNode("s"), token.getNode());
```

```
/*
```

执行**signal**，流程将继续执行，将通过默认转换（**transition**）离开状态 ‘**s**’

```
*/
```

```
token.signal();
```

```

/*
流程实例已经到达了结束状态。
*/

assertSame(processDefinition.getNode("end"), token.getNode());
}

```

运行测试，我们看到流程和我们预想的结果完全符合。

在这里我们的流程定义是写在一个代码中，XML 的定义方式是以通过拼字符串的方式完成的，这种方式给我们带来的结果是不直观，同时流程定义起来也很不方便。除了这种定义方式之外我们可以把刚才那段写在代码里的流程定义信息搬到我们的 XML 文件里，同样可以达到相同的效果。接下来我们就来看一下这种做法。

流程定义文件：helloWorld.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<process-definition xmlns="urn:jbpn.org:jpd1-3.1" name="Helloworld">
    <start-state>
        <transition to='s' />
    </start-state>
    <state name='s'>
        <transition to='end' />
    </state>
    <end-state name='end' />
</process-definition>

```

测试代码：HelloWorldTest.java

```

package org.jbpm.tutorial.helloworld;

import junit.framework.TestCase;
import org.jbpm.graph.def.ProcessDefinition;
import org.jbpm.graph.exe.ProcessInstance;
import org.jbpm.graph.exe.Token;

```



```
public class HelloWorldTest extends TestCase {  
    public void testHelloWorldProcess() {  
        ProcessDefinition processDefinition =  
        ProcessDefinition.parseXmlResource("helloWorld.xml");  
        /*  
        从这里可以看出，与上面那段代码唯一不同之处就是没有字符串形式的流程定义信息了，取而代之的是对  
        流程定义的XML进行解析  
        */  
        ProcessInstance processInstance =  
            new ProcessInstance(processDefinition);  
        Token token = processInstance.getRootToken();  
        assertEquals(processDefinition.getStartState(), token.getNode());  
        token.signal();  
        assertEquals(processDefinition.getNode("s"), token.getNode());  
        token.signal();  
        assertEquals(processDefinition.getNode("end"), token.getNode());  
    }  
}
```

运行测试，得到的结果同上例完全相同。

到这里为止，我们已经做了一个非常简单的流程示例，对 JBPM 的流程定义及使用方法也有了初步的概念，在下面的内容中我们将着重来讨论 JBPM 的流程定义方法，及各个节点的主要含义及使用方法。

## 三、学习 JPDL

### 3.1 、简介

JPDL (JBPM Process Definition Language) 是 JBPM 流程定义语言。JPDL 详细定义了这个状态图的每个部分，如：开始、结束状态，状态之间的转换等。这种语言的定义对于用户来说比较容易理解，也比较容易对其进行扩展。

一个 JBPM 的流程定义 XML 文件中包含一个< process-definition>元素，而一个< process-definition>元素又包含零个或一个< description>元素，零个或多个的< swimlane>元素，一个< start-state>元素，零个或多个的< state>元素或< decision>元素或< fork>元素或< join>元素，以及零个或多个的< action>元素，零个或多个< task-node>和< node>元素，一个< end-state>元素等等。此外，< process definition>元素有一个标示符，以“name”属性来表示，这个属性必须存在，用来表示该流程的名称。

## 3.2 、流程版本 (Version)

我们的流程 XML 文件定义完成之后，接下来的工作就是要将其发布到对应的数据库中，当我们每次将我们的流程定义部署到数据库时，部署时流程的名称就是前面提高的<process definition>里定义的 name” 属性的值。

JBPM 的版本机制允许在数据库中多个同名流程定义共存，流程实例以当时的最新版本来启动，并且在它的整个生命周期中将保持以相同的流程定义执行。当一个新的版本被部署，新的流程实例以新版本启动，而老的流程实例则以老的流程定义继续执行。

在部署的时候，jbpm 安排一个版本 (version) 号码 (数字) 给流程定义。为了实现安排 version 号码,如果它是第一个版本(version),JBPM 采取 1+或者 1。从 ProcessDefinition

```
pd=JbpmContext.getGraphSession()
```

```
.findLatestProcessDefinition("processName")
```

中可以通过一个给定的 processName 查找最近的流程定义,这里的 processName 就是前面我们在定义流程的时候在<process definition>里定义的 name” 属性的值,这个属性就是用来表示该流程的名称。如我们的下列代码就是要列出 JBPM 数据库里的有所有最后一次发布的流程定义的版本:

```
JbpmContext context=JbpmContext.getCurrentJbpmContext();
List ls=context.getGraphSession().findLatestProcessDefinitions();

/*
这里返回的 List 是 ProcessDefinition 的集合
*/
```

### 3.3 、流程定义

#### 3.3.1 process-definition(流程定义)

流程定义的根节点，是所有节点的父节点

名称	类型	数量	描述
name	属性	可选的	流程的名称。
swimlane	元素	[0..*]	流程中使用的泳道。泳道表示流程角色，它们被用于任务分配。
start-state	元素	[0..1]	流程起始状态。注意，没有起始状态的流程是合法的，但是不能被执行。
end-state state node task-node process-state super-state fork join decision	元素	[0..*]	流程定义的节点。注意，没有节点的流程是合法的，但是不能被执行。
event	元素	[0..*]	作为一个容器服务于动作的流程事件。
action script create-timer cancel-timer	元素	[0..*]	全局定义的的动作，可以在事件和转换中引用。注意，为了被引用，这些动作必须指定名称。
task	元素	[0..*]	全局定义的任务，可以在动作中使用。
exception-handler	元素	[0..*]	一个异常处理器列表，用于这个流程定义中的委托类所抛出的所有异常。

#### 3.3.2 node(自动节点)

这种节点和 State 相反，也称自动节点。当业务程序实例执行到这个节点，不会停止执行。而是会继续往下执行。如果该节点存在多个离开转向。那么，就会执行其中的第一个离开转向，在 Node 状态中，不需要外部参与者的参与，业务流程的这个部分是自动的、即时

完成的。

名称	类型	数量	描述
action script create-timer cancel-timer	事件	1	用于表示这个节点行为的定制动作。
普通节点元素			请参考普通节点元素。

### 3.3.3 start-state(开始状态)

start-state 是我们整个流程的开始节点，所有的流程实例从这里开始。

名称	类型	数量	描述
Name	属性	可选的	节点的名称。
Task	元素	[0..1]	起始一个流程实例的任务，或者用来捕获流程发起者
Event	元素	[0..*]	支持的事件类型：{node-leave}。
transition	元素	[0..*]	离开转换，每个离开节点的转换必须有一个不同的名称。
exception-handler	元素	[0..*]	一个异常处理器列表，用于这个流程节点中的委托类所抛出的所有异常。

### 3.3.4 end-state(结束节点)

对于每一个流程定义都会有一个结束节点，与开始节点对应

名称	类型	数量	描述
Name	属性	必需的	结束状态的名称。
event	元素	[0..*]	支持的事件类型：{node-enter}。
exception-handler	元素	[0..*]	一个异常处理器列表，用于这个流程节点中的委托类所抛出的所有异常。

### 3.3.5 state(状态)

State 节点也叫手工节点，进入到这种节点，整个流程的执行就会中断。直到系统外参与者发起继续执行的命令，即调用 `signal` 或 `end` 方法，业务程序实例的执行才能够继续下去。

名称	类型	数量	描述
name	属性	必需的	节点的名称。
async	属性	{true false}, 默认是 false	如果设置为 true，这个节点将会异步执行。请参考”异步执行”章节。
transition	元素	[0..*]	离开转换。每个离开节点的转换必须有一个不同的名称，最多只允许所有离开转换中的一个没有名称。第一个转换被指定为默认转换，当离开节点而没有指定转换时，默认转换发生。
event	元素	[0..*]	支持的事件类型： {node-enter node-leave}。
exception-handler	元素	[0..*]	一个异常处理器列表，用于这个流程节点中的委托类所抛出的所有异常。
timer	元素	[0..*]	指定一个定时器，用来监视节点中的一个执行所持续的时间。

### 3.3.6 task-node（任务节点）

其性质和 node 节点一样，在没有 task 的时候，也都是自动执行，不等待。task-node 被归类为一个等待节点，是指在 task-node 中的 task 列表中的 task 没有全部执行完之前，它会一直等待。Task 可以在 task-node 节点下定义，也可以挂在 process-definition 节点下。最普遍的方式是在 task-node 节点下定义一个或多个任务。默认情况下，流程在 task-node 节点会处于等待状态，直到所有的任务被执行完毕。Task 的执行是按顺序执行的，任务都完成后，token 仍然不会指向后面的节点；需要自己手动调用

`processInstance.signal()` 才会驱动流程到下面的节点。

名称	类型	数量	描述
signal	属性	可选的	{unsynchronized never first first-wait last last-wait}，默认是 last。signal 指定了任务的完成对流程执行继续的影响。
create-tasks	属性	可选的	{yes no true false}，默认是 true。当需要在运行时通过计算来决定哪个任务将被创建时，可以设置为 false，如果这样的话，在 node-enter 事件上加一个动作，在动作中创建任务，并且把 create-tasks 设置为 false。
end-tasks	属性	可选的	{yes no true false}，默认是 false。如果设置 end-tasks 为 true，在离开节点时，所有打开的任务将被结束。
task	元素	[0..*]	当执行到达本节点时所应被创建的任务。
普通节点元素			请参考普通节点元素。

为了帮助读者理解 task-node 节点的 signal 属性，这里举例如下：

对于这样的流程定义：

```

<task-node name='a'>
    <task name='laundry' />
    <task name='dishes' />
    <task name='change nappy' />
    <transition to='b' />
</task-node>

```

a) 这里没有定义 signal 属性的值，这就表明当节点中的三个任务都完成后，流程才进入后面的节点

b) 当<task-node name='a' signal='unsynchronized'>表明 token 不会在本节点停留，而是直

接到后面的节点

- c) 当<task-node name='a' signal='never'>表明三个任务都完成后, token 仍然不会指向后面的节点; 需要自己手动调用 `processInstance.signal()` 才会驱动流程到下面的节点
- d) 当<task-node name='a' signal='first'>表明只要有一个任务完成后, token 就指向后面的节点
- e) 当<task-node name='a' signal='first-wait'>表明当第一个任务实例完成时继续执行; 当在 a 节点入口处没有任务创建时, token 在 a 任务节点处等待, 直到任务被创建或完成。
- f) 当<task-node name='a' signal='last'>时, 这是默认值, 和不设置 signal 属性的情况相同。
- g) 当<task-node name='a' signal='last-wait'>时, 当最后一个任务实例完成时候继续执行下去。 当 a 这个任务节点没有任务被建立时, 任务节点等待直到任务被建立。

### 3.3.7 fork(分支)

一个 fork 把一个执行路线分割成多个执行路线。默认分支的行为是为每个离开分支转换建立一个子令牌, 在令牌要到达的分支之间建立一个父母-子女关系

名称	类型	数量	描述
name	属性	必需的	节点的名称。
async	属性	{true false}, 默认是 false	如果设置为 true, 这个节点将会异步执行。请参考”异步执行”章节。
transition	元素	[0..*]	离开转换。每个离开节点的转换必须有一个不同的名称, 最多只允许所有离开转换中的一个没有名称。第一个转换被指定为默认转换, 当

			离开节点而没有指定转换时，默认转换发生。
event	元素	[0..*]	支持的事件类型：{node-enter node-leave}。
exception-handler	元素	[0..*]	一个异常处理器列表，用于这个流程节点中的委托类所抛出的所有异常。
timer	元素	[0..*]	指定一个定时器，用来监视节点中的一个执行所持续的时间。

### 3.3.8 join(联合)

默认联合(join)假设所有来自同一个父母的子令牌联合，当在上使用 fork(分支)这个情形就出现了并且所有令牌分支建立，并且到达同一个联合(join)。当全部令牌都进入联合的时候联合就结束了，然后联合将检查父母-子女，当所有兄弟令牌到达联合(join)，父母令牌将传播(唯一的)离开转换，当还有兄弟令牌活动时，联合的行为将作为等待状态。

名称	类型	数量	描述
name	属性	必需的	节点的名称。
async	属性	{true false} }, 默认是 false	如果设置为 true, 这个节点将会异步执行。
transition	元素	[0..*]	离开转换。每个离开节点的转换必须有一个不同的名称，最多只允许所有离开转换中的一个没有名称。第一个转换被指定为默认转换，当离开节点而没有指定转换时，默认转换发生。
event	元素	[0..*]	支持的事件类型： {node-enter node-leave}。
exception-handler	元素	[0..*]	一个异常处理器列表，用于这个流程节点中的委托类所抛出的所有异常。
timer	元素	[0..*]	指定一个定时器，用来监视节点中的一个执行所持续的时间。

对于 Join 节点，我们知道默认是要等到所有分支都到了流程才能往下继续走，要改变



这一情况，我们可以通过给该节点加 Action 的方法改变该 Join 节点的 Discriminator，就可以使只要有一个分支到达流程就可以继续执行的效果了，如下面的 Action：

```
package workflow.test.action;

//这里通过设置Discriminator可以实现只要有一个分支到达流程就可以继续了，
//它的默认值是false

import org.jbpm.graph.def.ActionHandler;

import org.jbpm.graph.exe.ExecutionContext;

import org.jbpm.graph.node.Join;

public class JoinAction implements ActionHandler{

    public void execute(ExecutionContext arg0) throws Exception {

        Join join=(Join)arg0.getNode();

        join.setDiscriminator(true);

    }

}
```

### 3.3.9 decision(决策)

一个 decision 用以决定在多个执行路径中哪个才可以被执行。如果你是一个程序员，把它可以理解成 switch case 结构即可，一个 decision 能够具有许多离开的 transition。

名称	类型	数量	描述
handler	元素	要么指定 “handler” 元素，或者在转换上指定条件。	一个 org.jbpm.jpdl.Def.DecisionHandler 的实现名称。
transition	元素	[0..*]	离开转换。决策的离开转换可以被扩展为拥有一个条件，决策会查找条件计算为true的第一个转换，没有条件的转换被认为计算为true（为了建模“otherwise”分支）。请参考 condition元素。
普通节点元素			请参考普通节点元素。

Handler 所指定的 DecisionHandler 的实现类里的 decide 方法返回一个字符串，表示要执行哪个 transition，如下例：

```
package workflow.qingjia.shenpi.common;

import org.jbpm.graph.exe.ExecutionContext;
import org.jbpm.graph.node.DecisionHandler;

public class ForkDecision implements DecisionHandler{

    public String decide(ExecutionContext arg0) throws Exception {

        String dayCount=(String)arg0.getVariable("dayCount");

        String go="to boss approve";

        if(Integer.parseInt(dayCount)>10){

            go="to join";

        }

        return go;

    }

}
```

### 3.3.10 transition(转换)

转换用来指定节点之间的连接。transition 元素放在 node 里面，那么这个 transition 就会从这个节点出离开。

名称	类型	数量	描述
name	属性	可选的	转换的名称。注意，每个节点的离开转换必须有一个不同的名称。
to	属性	必需的	目标节点的分级名称，表示将要达到的那个节点名称。
action script  create-timer  cancel-timer	元素	[0..*]	发生转换时将要执行的动作。注意，转换的动作无需放入事件（因为只有一个事件）。

exception-handler	元素	[0..*]	一个异常处理器列表,用于这个流程节点中的委托类所抛出的所有异常。
-------------------	----	--------	----------------------------------

### 3.3.11 event(事件)

JBPM 定义了一系列与 workflow 节点元素相关联的事件,例如,流程实例运行过程中,可以触发节点进入 (node-enter)、节点离开 (node-leave)、流程启动 (process-start)、流程结束 (process-end)、任务创建 (task-create)、任务分派 (task-assign)、任务启动 (task-start) 等事件。

在流程定义时,JBPM 的事件均与 action 绑定。事件的触发将导致相应 actions 的执行。

名称	类型	数量	描述
type	属性	必需的	表示相对于事件要放置的元素事件类型。
action script create-timer cancel-timer	元素	[0..*]	在这个事件上将要执行的动作列表。

### 3.3.12 action(动作)

一个 action 是一段 java 代码。在流程执行期间在一些事件之上定义,这样会在相关事件触发时自动在 workflow 引擎上执行。

名称	类型	数量	描述
name	属性	必需的	动作的名称。当动作被指定名称后,它们可以在流程定义中被查出,这对于运行时动作以及仅一次声明动作是有用的。
class	属性	或者用 ref-name, 或	实现 org.jbpm.graph.def.ActionHandler 接口的类的全名。

		者用 expression。	
ref-name	属性	或者用 class。	所引用动作的名称。如果指定一个引用动作，则本动作不需要再做处理。
expression	属性	或者指定一个 class，或者 ref-name。	一个解决一个方法的 jPDL 表达式。
accept-propagated-events	属性	可选的	{yes no true false}，默认是yes true。如果设置为false，则动作仅在本动作元素的触发事件上被执行。更多信息，请参考“第 9.5.4 事件传播”。
config-type	属性	可选的	{field bean constructor configuration-property}。指定动作对象将被怎样创建以及本元素的内容怎样象配置信息那样被动作对象所使用。
async	属性	{true false}	默认 false，这意味着动作将在当前执行的线程中被执行。如果设置为 true，一个消息将被发送到命令执行器，并且执行器组件将在一个独立的事务中同步执行动作。请参考”异步执行”章节。
	{内容}	可选的	action的内容可以被作为你定制动作实现的配置信息，这是考虑到可重用的委托类的创建。有关委托配置的更多信息，请参考“第 16.2.3 节委托配置”。

### 3.3.13 script(脚本)

Script里是动作执行的beanshell脚本。更多有关beanshell的的信息请参考Beanshell的网站：<http://www.beanshell.org>。如下所示

```
<process-definition>
  <event type="node-enter">
```

```

<script>

    System.out.println("this script is entering node "+node);

</script>

</event>

...

</process-definition>

```

表格 16.16

名称	类型	数量	描述
name	属性	可选的	脚本动作的名称。当动作被指定名称后，它们可以在流程定义中被查出，这对于运行时动作以及仅一次声明动作是有用的。
Accept -propagated -events	属性	可选的 [0..*]	{yes no true false}，默认是 yes true。如果设置为 false，则动作仅在本动作元素的触发事件上被执行。
expression	元素	[0..1]	beanshell脚本。如果你没有指定variable元素，可以写表达式作为脚本元素的内容（忽略expression元素标签）。
variable	元素	[0..*]	脚本所需变量。如果没有指定变量，则当前令牌的所有变量将被装载到脚本，当你想要限制装载到脚本中的变量数量时使用 variable。

### 3.3.14 expression（表达式）

Expression 里可书写 Beanshell 脚本

名称	类型	数量	描述
	{内容}		一个 beanshell 脚本。

### 3.3.15 variable(变量)

一个变量是一种 key-value 对。它与过程实例（一次过程执行）相关联。Key 是 `java.lang.string`, value 是任何 java 类型的任何 pojo。所以任何是 java 类型, 即使不给 jbpmm 知道也能被应用到变量中。JBPM 的流程变量在尽量模仿 `java.util.map` 的语义。这一点可以通过 JBPM 的 API 来了解。也就是说一个变量只能当它被插入时被赋值, 任何 java 类型都可以作为变量中的 value。

名称	类型	数量	描述
name	属性	必需的	流程变量的名称。
access	属性	可选的	默认是 read,write, 用逗号分割的一个访问列表。 迄今为止, 使用的访问仅为 read, write 和 required。
mapped-name	属性	可选的	默认是变量的名称。用来指定变量名称被映射的名称, mapped-name 的含义依赖于这个元素所被使用的上下文。对于一个脚本, 将是一个脚本变量名称; 对于一个任务控制器, 将是任务表单参数的标签; 对于一个 process-state, 将是在子流程中使用的变量名称。

### 3.3.16 handler(句柄)

Handler 是在定义一个 decision 时需要为其定义一个 DecisionHandler 时采用。

名称	类型	数量	描述
expression	属性	或者用 class	一个 jPDL 表达式, 返回结果被用 toString() 方法转换为字符串, 结果字符串应该与某个离开转换匹配。
class	属性	或者用 ref-name	实现了 org.jbpm.graph.node.DecisionHandler 接口的类的全名。

Config -type	属性	可选的	{field bean constructor configuration-property} }。指定动作对象将被怎样创建以及本元素的内容怎样 象配置信息那样被动作对象所使用。
	{内 容}	可选的	Action 里的内容可以用来帮助结合我们的业务来处理 我们的流程, 同时我们可以在 Action 里加上业务处理 逻辑, 以更好的利用流程.

### 3.3.17 timer(定时器)

定时器 timer 可以被用于 decision fork join node process-state state super-state task-node, 可以设置开始时间 due-date 和频率 repeat, 定时器动作可以是所支持的任何动作元素, 如 action 或 script。

timer 还有一个很重要的属性 cancel-event, 这个是 timer 和 task 结合时使用的, 任务定时器的 cancel-event 可以被定制。默认情况下, 当任务被结束时 (=完成) 任务上的定时器将被取消, 这是通过在定时器上使用 cancel-event 属性, 流程开发者可以定制诸如 task-assign 或 task-start。cancel-event 支持多个事件, 通过在属性中指定一个用逗号分割的列表, 可以组合 cancel-event 的类型。

名称	类型	数量	描述
name	属性	可选的	定时器的名称。如果没有指定名称, 则采用外部的节点名称。注意, 每个定时器应该有一个唯一的名称。
due-date	属性	必需的	所指定的定时器创建到定时器执行之间的期限 (可以用业务时间来表示)。
repeat	属性	可选的	{duration yes true} 当一个定时器在预期时间执行后, “repeat” 可选项指定了在离开节点之前重复的执行定时器之间的期限。如果指定为 true 或 false, 则与 due-date 相同的期限被使用。

transition	属性	可选的	当定时器执行、定时器事件触发后以及执行动作时所使用的转换名称。
cancel-event	属性	可选的	这个属性只用在任务的定时器中，它指定了定时器将被取消的事件。默认是 task-end 事件，但是也可以被设置为如 task-assign 或 task-start。 cancel-event 的类型也可以通过指定一个用逗号分割的列表被组合。
action script  create-timer  cancel-timer	元素	[0..*]	当定时器被触发时所应被执行的动作。

### 3.3.18 create-timer(创建定时器)

Create-timer 是定时器的创建

名称	类型	数量	描述
name	属性	可选的	定时器的名称。这个名称可被用于用一个 cancel-timer 动作取消定时器。
duedate	属性	必需的	所指定的定时器创建到定时器执行之间的期限(可以用业务时间来表示)。请参考“第 14.1 节期限”中的语法。
repeat	属性	可选的	{duration 'yes' 'true'} 当一个定时器在预期时间执行后，“repeat”可选项指定了在离开节点之前重复的执行定时器之间的期限。如果指定为true或yes，则与duedate相同的期限被使用。请参考“第 14.1 节期限”的语法。
transition	属性	可选的	当定时器执行、定时器事件触发后以及执行动作时时（如果要）所获取的转换名称。



### 3.3.19 cancel-timer(取消定时器)

Cancel-timer 是定时器的取消

名称	类型	数量	描述
name	属性	可选的	要被取消的定时器的名称。

### 3.3.20 task(任务)

Task 是流程定义里的一部分，它决定了 task instance 的创建和分配

名称	类型	数量	描述
name	属性	可选的	任务的名称。命名的任可以被引用并且可以通过 TaskMgmtDefinition 被查出。
blocking	属性	可选的	{yes no true false} 如果 blocking 设置为 true，当任务没有结束时节点不能被离开（必须要通过 taskInstance.end() 方法离开节点）；如果设置为 false（默认），允许用户通过 signal 继续执行和离开节点。默认设置为 false，因为通常是由用户接口来强制阻塞。
signalling	属性	可选的	{yes no true false}，默认是 true。如果设置 signalling 为 false，则任务没有触发令牌继续的能力。
duedate	属性	可选的	延迟时间（任务执行的延迟时间）。请见业务日历中的解释。
swimlane	属性	可选的	引用一个swimlane，如果在任务上指定了一个swimlane，则assignment将被忽略。
priority	属性	可选的	{highest, high, normal, low, lowest} 之一。作为选择，可以为 priority 指定任何整数，供参考： (highest=1, lowest=5)。
assignment	元	可选的	描写一个委托，该委托将在任务被创建时把任务分配给一

	素		个参与者。
event	元素	[0..*]	支持的事件类型：  {task-create task-start task-assign task-end}。为了任务分配，我们特别的为 TaskInstance 添加了一个非持久化的属性 previousActorId。
exception-handler	元素	[0..*]	一个异常处理器列表，用于这个流程节点中的委托类所抛出的所有异常。
timer	元素	[0..*]	指定一个监视本任务执行期限的一个定时器。对于任务定时器特殊的是可以指定 cancel-event，cancel-event 默认是 task-end，但是它可以被自定义如 task-assign 或 task-start。
controller	元素	[0..1]	指定流程变量怎样被转换为任务表单参数。任务表单参数有用户界面使用，用力向用户表现一个任务表单。

### 3.3.21 swimlane(泳道)

实际应用中，一个人是一个流程中多个 Task 的参与者(actor)的情况是很常见的。在 jbpm 中通过创建一个 swimlane 并且把 swimlane 赋给一个 task 的方式来设置当前 task 的参与者(actor)。一个业务流程中的 swimlane 可以被看做为一个参与者的参与者对象的名称，当然它不一定是固定的某个人，它可以是一个用户组，一个特定用户的角色等。首次执行到达一个 Task，赋给该 Task 的一个 swimlane 就会算出参与者(actor)。

名称	类型	数量	描述
name	属性	必需的	泳道的名称。泳道可以被引用并且可以通过 TaskMgmtDefinition 被查出。
assignment	元素	[1..1]	指定泳道的分配。这个分配在本泳道中的第一个任务实例被创建时完成。

### 3.3.22 assignment(委派)

当流程执行到某个 Task 的时候，引时流程引擎要调用相应的 swimlane 或 assignment 将当前的 task 分配（委派）给某个参与者，外部参与者可以是一个人也可以是某个系统等。

名称	类型	数量	描述
expression	属性	可选的	由于历史原因，这个属性的表达式不是jPDL表达式，而是对JBPM身份组件的一个分配表达式。
actor-id	属性	可选的	一个actorId，可以与pooled-actors协同使用。actor-id被作为一个表达式，因此你可以引用一个固定的actorId，如actor-id=" bobthebuilder"；或者你可以引用一个可以返回一个字符串的属性或方法，如actor-id=" myVar.actorId"，这将调用任务实例变量“myVar”上的getActorId方法。
Pooled-actors	属性	可选的	一个逗号分割的actorId列表，可以与actor-id协同使用。一个固定的参与者池可以指定如下： pooled-actors=" chicagobulls,pointersisters"。 pooled-actors被作为一个表达式，因此你可以引用一个返回String[]、Collection、或一个逗号分割的池中的参与者的属性或方法。
class	属性	可选的	一个实现 org. jbpm. taskmgmt. def. AssignmentHandler 接口的类的全名称。
config-type	属性	可选的	{field bean constructor configuration-property}。指定分配处理器对象（assignment-handler-object）对象将被怎样创建以及本元素的内容怎样象配置信息那样被分配处理器对象所使用。
	{内容}	可选的	assignment 元素的内容可以被作为分配处理器（AssignmentHandler）实现的配置信息，这是考虑到可重用的委托类的创建。

### 3.3.23 controller(控制器)

在任务执行时，可能需要读、写流程变量；在任务完成并提交时，可能需要写流程变量。为此，JBPM 提供了“任务变量”的概念。在某些情况下，任务变量和流程变量并非简单的一一对应关系，例如，三个流程变量代表三个月的销售额，任务变量只需要它们的平均值。为实现任务与流程实例之间的信息交流，JBPM 设置了任务控制器机制。该机制也采用递进模式：首先，JBPM 提供基本（默认）的任务控制器；如果不敷使用，二次开发人员可以使用自定义的任务控制器。JBPM 的任务控制器机制在流程变量和任务变量之间架起了一座桥梁。

名称	类型	数量	描述
class	属性	可选的	一个实现 org.jbpm.taskmgmt.def.TaskControllerHandler 接口的类的全名称。
Config-type	属性	可选的	{field bean constructor configuration-property}。指定分配处理器对象（assignment-handler-object）对象将被怎样创建以及本元素的内容怎样象配置信息那样被分配处理器对象所使用。
	{内容}		controller 元素的内容要么是指定的任务控制处理器的配置信息（如果指定了 class 属性），要么必须是一个 variable 元素列表（如果没有指定任务控制器）。
variable	元素	[0..*]	如果没有通过 class 属性指定任务控制处理器，则 controller 元素的内容必须是变量列表。

### 3.3.24 process-state 子流程

process-state 是 JBPM 提供的用来处理子流程的节点，一个 process-state 只能对应一个子流程，究竟指到哪个子流程可以在 process-state 的 action 里指定，当 token 执行到指定的子流程时，子流程就已经启动，不用像启动主流程一样手工启动子流程。其它部分的处理就和普通的流程没有区别了。

名称	类型	数量	描述
name	属性	必需的	名称。
Sub-process	元素	只能定义一个	子流程
variable	变量	[0...*]	Variable 是用来指定如何把数据从父流程 copy 到子流程

### 3.3.25 sub-process 子流程

名称	类型	数量	描述
name	属性	必需的	子流程的名称
version	属性	可选	子流程的版本。如果没有指定该属性，默认将会采用该子流程的最后一个版本

### 3.3.26 condition 条件

名称	类型	数量	描述
	{内容}或属性表达式	必需的	condition 元素的内容是一个计算结果为布尔值的 jPDL 表达式。决策采用第一个表达式处理结果为 true 的转换（按在 processdefinition.xml 中的顺序），如果没有条件处理结果为 true，则采用默认离开转换（也就是第一个）。

### 3.3.27 exception-handler 异常处理

Jbpm 的异常处理机制仅仅集中于 java 异常，流程定义本身的执行不会导致什么异常，只有在执行委托类时才会导致异常。

在流程定义（process-definitions）添加的 exception-handler 对整个流程起作用、节点（nodes）上添加异常只对当前的节点起作用（同时如果在 process-definitions 里也设置了 exception-handler 那么将不会再执行 process-definitions 里的 exception-handler），和转换（transitions）添加 exception-handler 只对当前的 transitions 起作用（同时如果在 process-definitions 里也设置了 exception-handler 那么将不会再执行 process-definitions 里的 exception-handler），可以指定一个异常处理（exception-handlers）清单，每个异常处理（exception-handler）有一个动作列表，当在委托类中发生异常时，会在流程元素的父层次搜索一个适当的异常处理（exception-handler），当它被搜索到，则异常处理（exception-handler）的动作将被执行。

注意，Jbpm 的异常处理机制与 java 异常处理不完全相似。在 java 中，一个捕获的异常可以影响控制流，而在 Jbpm 中，流程不会被 Jbpm 异常处理机制所改变。异常要么被捕获，要么不捕获，没有被捕获的异常被抛向客户端（例如客户端调用 token.signal()），而被捕获的异常则是通过 Jbpm 的 exception-handler，对于被捕获的异常，图执行仍会继续，就像没有异常发生一样。

在处理异常的动作中，可以使用 Token.setNode(Node node)把令牌放入图中的任何节点。

名称	类型	数量	描述
exception-class	属性	可选的	指定与本异常处理器所匹配的 java throwable 类，如果这个没有指定这个属性，则它匹配所有异常（java.lang.Throwable）。
action	元素	[1..*]	当异常被异常处理器捕获时将要执行的动作列表。

定义示例：

```
<?xml version="1.0" encoding="UTF-8"?>

<process-definition

  xmlns="urn:jbpn.org:jpd1-3.1"  name="test">

<!--这里的exception-handler对整个processInstance起作用-->

  <exception-handler>

    <action class="gj.action.ProcessException"/>

  </exception-handler>

  <start-state name="start">

    <transition name="to state" to="statel"></transition>

  </start-state>

  <state name="statel">

    <event type="node-leave">

      <action name="enter node action"

class="gj.action.EnterStateNodeAction"></action>

    </event>

    <transition name="to end" to="endl">

      <action name="action1"

class="gj.action.ProcessStateAction"></action>

    </transition>

  </state>

  <end-state name="endl"></end-state>

</process-definition>
```

ProcessException 类的代码如下:

```
package gj.action;
```

```
import org.jbpm.graph.def.ActionHandler;

import org.jbpm.graph.exe.ExecutionContext;

public class ProcessException implements ActionHandler{

    public void execute(ExecutionContext executionContext) throws
Exception {
        String errorMsg=executionContext.getException().getMessage();

        System.out.println("异常类型
"+executionContext.getException().toString()+" 异常消息: "+errorMsg);
    }
}
```

这个类就可以用来处理整个 ProcessInstance 中发生的异常。其它在各种类型 node 里和在 transition 里定义的 exception-handler 的处理方式类似，只不过其作用范围仅限制为当前的 node 或 transition。

## 小结

看到这里，我们已经对 JPD 的流程定义语言有了较深的理解，接下来我们可以自己动手写一些流程定义的文件，以此加深对 JPD 的理解。我们来看一下下面的流程定义文件内容：

```
<?xml version="1.0" encoding="UTF-8"?>

<process-definition xmlns="" name="test">

<!--定义一个开始结点，名为 start，指向 fork1-->

    <start-state name="start">
```



```
<transition name="" to="fork1"></transition>
```

```
</start-state>
```

《!—在 fork1 分支结点上，我们定义了两个走向（transition）tr1 和 tr2，其中 tr1 指向 top-state 节点，tr2 指向一个 process-state，当 token 到达 for1 时会自动形成两个 child token，沿着 tr1 和 tr2 的指向继续向执行-->

```
<fork name="fork1">
```

```
<transition name="tr1" to="top-state"></transition>
```

```
<transition name="tr2" to="test-sub-process"></transition>
```

```
</fork>
```

《!—top-state 是一个 state 类型的节点，也就是前面提到的手工节点-->

```
<state name="top-state">
```

```
<transition name="to test-task" to="test-task"></transition>
```

```
</state>
```

《!—test-task 是一个 task-node 类型的节点，它可以有多个 task 节点，生成我们的 taskInstance，这里我们定义了三个节点-->

```
<task-node name="test-task" signal="first">
```

```
<task name="task-a"></task>
```

```
<task name="task-b"></task>
```

```

    <task name="task-c"></task>

    <transition name="to buttom-node" to="buttom-node"></transition>

</task-node>

<!-- buttom-node 是一个 node 节点，可以自动执行，不用人工干预处理-->

<node name="buttom-node">

    <transition name="" to="join1"></transition>

</node>

<!--在 test-sub-process 里我们加上了一个子流程处理，子流程定义我们写在了里面定义的
enter-sub-process-action 里-->

<process-state name="test-sub-process">

    <event type="node-enter">

        <action name="enter-sub-process-action"
class="gaojie.process.action.EnterSubProcessAction"></action>

    </event>

    <transition name="" to="join1"></transition>

</process-state>

<!--join1 是一个 join 节点，与 fork 节点对应，从 fork 分支出发的所有 child token 汇
集到 join 后继续向下执行-->

<join name="join1">

```

```
<transition name="" to="end"></transition>

</join>

<!--最后是结束结点，流程执行到此就宣告结束了一-->

<end-state name="end"></end-state>

</process-definition>
```

## 四、流程中任务的分配

JBPM 中任何一个 task 都必须指定一个任务的接收者，这个接收者可以是一个用户，也可以是一个用户组。如果指定给一个用户那么可以用这个用户的 ID 得到当前的 task。如果是一个用户组那么这个组的任何一个用户都可以看到这个 task，当这个组中的任何一个用户处理该任务后那么这个 task 对这个组中的其它用户就不再可见。

JBPM 中任务的分配方式有两种：一种是为 task 指定一个 assignment-handler，既一个实现了 AssignmentHandler 接口的类；另外一种是为 task 指定一个 swimlane(泳道)，swimlane 可以在流程中定义好，一个流程中可以定义若干个 swimlane，在定义一个 swimlane 时同样也是指定了个实现了 AssignmentHandler 接口的类，当我们的 task 指定了一个 swimlane 后，其效果同我们指定一个 assignment-handler 效果是一样的，只不过可以简化我们任务的分配工作。一个典型的实现了 AssignmentHandler 接口的类如下：

```
package test.assignment;

import org.jbpm.graph.exe.ExecutionContext;
import org.jbpm.taskmgmt.def.AssignmentHandler;
import org.jbpm.taskmgmt.exe.Assignable;

public class ManagerAssignment implements AssignmentHandler{

    public void assign(Assignable arg0, ExecutionContext arg1) throws
Exception {
        String[] s=new String[5];
```

```

    for (int i = 0; i < 5; i++) {
        s[i]="manager"+i;
    }
    //arg0.setActorId("manager0");//将任务分配给单个用户
    arg0.setPooledActors(s);//将任务分配给一个用户组
}
}

```

#### 4.1 assignment-handler 方式的任务分配

在 JBPM 中可以在 start-node 中添加一个 task，在一个 task-node 中添加若干个 task。每一个 task 我们都必须为其指定一个 assignment-handler 或一个 swimlane，二者只能选其一。

```

...
<start-state name="start">
    <task name="start task">
        <controller>
            <variable name="dayCount" access="read,write,required"
mapped-name="请假天数"></variable>
        </controller>
        <assignment
class="test.assignment.IssuePersonAssignment"></assignment>
    </task>
    <transition name="to manager approve" to="manager approve
task"></transition>
    </start-state>
...

```

上面的代码中 start 节点中我们添加了一个 start task 的 task，同时为其指定了一个 assignment，所对应的 class 为 test.assignment.IssuePersonAssignment，该类的代码如下：

```

package test.assignment;

import org.jbpm.graph.exe.ExecutionContext;
import org.jbpm.taskmgmt.def.AssignmentHandler;
import org.jbpm.taskmgmt.exe.Assignable;

import test.common.Constants;

public class IssuePersonAssignment implements AssignmentHandler {

```

```

public void assign(Assignable arg0, ExecutionContext arg1) throws Exception {
    String issuePerson = arg1.getVariable(Constants.ISSUE_PERSON)
        .toString();
    arg0.setActorId(issuePerson);
}
}

```

在该类中，我们从整个流程中的取出一个名为 Constants.ISSUE\_PERSON 的流程变量，并将其赋给当前的 task，这里采用的是 Assignable 的 setActorId 的方法。这样当用户登录时就可以用

```

JbpmContext
context=JbpmConfiguration.getInstance().createJbpmContext();
List
ls=context.getTaskList(session.getAttribute("username").toString());

```

方法来取出对应的任务列表。

#### 4.2 swimlane 方式的任务分配

该种方式的任务分配实际上是对 assignment-handler 方式任务分配的简化。首先用户需要在流程中定义好若干个 swimlane，接下来只需要在 task 中指定一个 swimlane 就可以完成任务的分配工作。

```

...
<swimlane name="manager">
    <assignment
class="test.assignment.ManagerAssignment"></assignment>
    </swimlane>
...
    <task-node name="manager approve task">
        <task name="manager approve" swimlane="manager">
            <controller>
                <variable name="dayCount" access="read" mapped-name="员工请假
天数"></variable>
                <variable name="managerApprove" access="read,write,required"
mapped-name="经理意见"></variable>
            </controller>
        </task>
        <transition name="to user try" to="user try"></transition>
        <transition name="to fork" to="fork1"></transition>
    </task-node>

```

```
</task-node>
```

```
...
```

上面的代码中我们指定了一个叫 `manager` 的 `swimlane`，然后我们在一个名为 `manager approve` 的 `task` 里将该 `task` 指定给该 `swimlane`，这样就完成了任务的分配。从这里我们可以看到与 `assignment-handler` 方式分配最大不同之处是该种分配任务的方式简单、明了。我们只需要预先定义好若干个 `swimlane` 之后就可以在 `task` 里重复使用了。  
`test.assignment.ManagerAssignment` 类的代码如下：

```
package test.assignment;

import org.jbpm.graph.exe.ExecutionContext;
import org.jbpm.taskmgmt.def.AssignmentHandler;
import org.jbpm.taskmgmt.exe.Assignable;

public class ManagerAssignment implements AssignmentHandler{

    public void assign(Assignable arg0, ExecutionContext arg1) throws
Exception {
        String[] s=new String[5];
        for (int i = 0; i < 5; i++) {
            s[i]="manager"+i;
        }
        arg0.setPooledActors(s); //将任务分配给一个用户组
    }
}
```

在这个类当中我们把任务分配给一个用户组（一个由用户 `ID` 组成的数组），这样该组中的每个用户登录后都可以采用以下方法看到任务列表：

```
JbpmContext
context=JbpmConfiguration.getInstance().createJbpmContext();
List
list=context.getTaskMgmtSession().findPooledTaskInstances(session.get
Attribute("username").toString());
```

用这种方法，用户可以看到所有尚未处理的分配到该用户所在用户组中的 `task` 列表。一旦该组中有一个用户处理了该任务，那么这个任务对于其它用户就不再可见了。

## 五、JBPM 持久化

JBPM 采用 Hibernate 持久化到数据库，和其它一些 workflow 引擎一样，它的流程定义信息也要持久化到数据库中。由于 JBPM 采用 Hibernate 来和数据库打交道，理论上来说只要 Hibernate 支持的数据库 JBPM 都支持。在下面的篇幅里我们将介绍一下如何将 JBPM 的流程定义信息发布到数据库中，以及如何在 Tomcat 中进行流程的调用与处理。

### 5.1 特殊数据库支持

1) 在 DB2 上运行 JBPM，我们首先需要运行类似下面的命令，以更改 DB2 的配置，然后才能开始我们 JBPM 的建表工作。

```
create bufferpool jbpn immediate size 1000 pagesize 32K
create tablespace jbpn pagesize 32K managed by database using (file '/db2inst1/db2inst1/NODE0000/JBPM' 10M)
AUTORESIZE YES bufferpool jbpn
create temporary tablespace jbpntemp pagesize 32K managed by database using (file
'/db2inst1/db2inst1/NODE0000/JBPM_TMP' 10M) AUTORESIZE YES bufferpool jbpn
```

### 5.2 JBPM 数据库的安装

我们以 MSSQL2000 为例和大家一起来看一下如何完成 JBPM 数据库的安装(其它数据库大家可以照葫芦画瓢完成)。

- a) 在 MSSQL 里新建一数据库，名为 jbpntest(这里是我所用的数据库名，大家可以根据喜好，自己随意命名)。
- b) 找到我们开始部分下载的 jbpn-starters-kit-3.1.2 文件夹，找到其中的 jbpn 文件夹。
- c) 将 MSSQL 的 JDBC 驱动 copy 到 jbpn 文件夹的 lib 目录下(我这里采用的是 sourceforge 的 jtds 驱动)
- d) 在 jbpn 文件夹里在其中的 src/resources 下新建一文件夹名为 mssql，将 src/resources/hssqldb 文件夹下的 create.db.hibernate.properties 和 identity.db.xml 文件 copy 到 mssql 目录下。
- e) 修改 create.db.hibernate.properties 文件(这个文件里配置的是目标数据库的

连接属性信息), 修改内容如下:

```
# these properties are used by the build script to create
# a hypersonic database in the build/db directory that contains
# the jbpmm tables and a process deployed in there

hibernate.dialect=org.hibernate.dialect.SQLServerDialect
hibernate.connection.driver_class=net.sourceforge.jtds.jdbc.Driver
hibernate.connection.url=jdbc:jtds:sqlserver://localhost:1433/jbpmtest
hibernate.connection.username=sa
hibernate.connection.password=gj
hibernate.show_sql=true
```

- f) 在 jbpmm 文件夹下, 找到 src\config.files\hibernate.cfg.xml. (此文件主要是系统运行时数据库连接属性配置), 需要修改的内容如下:

```
<!-- jdbc connection properties -->
<property name="hibernate.dialect">
org.hibernate.dialect.SQLServerDialect</property>
<property name="hibernate.connection.driver_class">
net.sourceforge.jtds.jdbc.Driver</property>
<property name="hibernate.connection.url">
jdbc:jtds:sqlserver://localhost:1433/jbpmtest</property>
<property name="hibernate.connection.username">sa</property>
<property name="hibernate.connection.password">gj</property>
```

- g) 在 Jbpmm 根目录下, 打开 build.deploy.xml 文件, 找到其中的 create.db 并做如下



修改，create.db 节点修改好的内容如下：

```
<!-- ===== -->

<!-- === SERVER === -->

<!-- ===== -->

<target name="create.db" depends="declare.jpdm.tasks, db.clean,
db.start" description="creates a hypersonic database with the jbpdm tables
and loads the processes in there">

    <jbpdm schema actions="create"

        cfg="${basedir}/src/config.files/hibernate.cfg.xml"

        properties="${basedir}/src/resources/mssql/create.db.hibernate.properties"/>

    <loadidentities

        file="${basedir}/src/resources/mssql/identity.db.xml"

        cfg="${basedir}/src/config.files/hibernate.cfg.xml"

        properties="${basedir}/src/resources/mssql/create.db.hibernate.properties"/>

    <ant antfile="build.xml" target="build.processes" inheritall="false"
/>

    <deployprocess cfg="${basedir}/src/config.files/hibernate.cfg.xml"

        properties="${basedir}/src/resources/mssql/create.db.hibernate.properties">

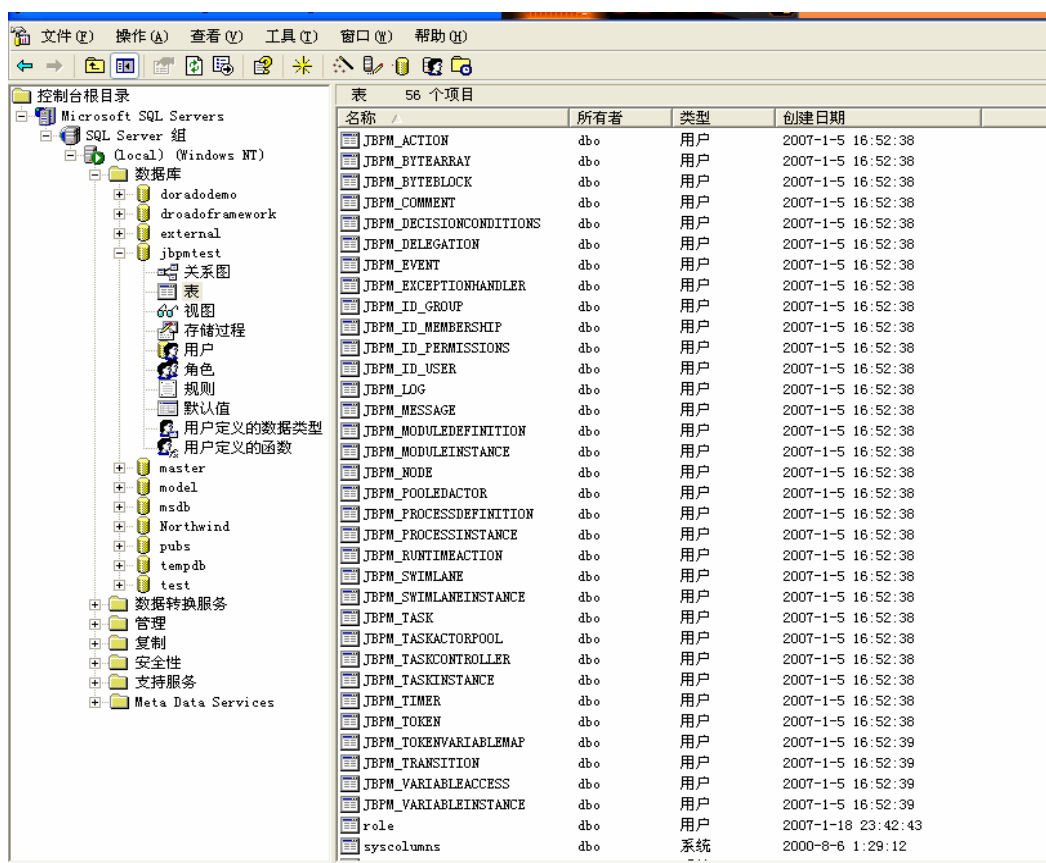
        <fileset dir="build" includes="*.process" />

    </deployprocess>

    <antcall target="db.stop" />

</target>
```

- h) 配置好 ant 工具，将 ant/bin 目录加上系统的 path 环境变量。
- i) 在命令行模式下进入 jbpm 目录，输入 `ant create.db -buildfile build.deploy.xml`，构建 JBPM 所需要的 table。这里大家可以观察屏幕输出，如果没有意外那就表明建表成功了！
- j) 打开 MSSQL 的企业管理器，我们可以看到一系列以“JBPM\_”开头的 table 了，如下图所示：



名称	所有者	类型	创建日期
JBPM_ACTION	dbo	用户	2007-1-5 16:52:38
JBPM_BYTEARRAY	dbo	用户	2007-1-5 16:52:38
JBPM_BYTEBLOCK	dbo	用户	2007-1-5 16:52:38
JBPM_COMMENT	dbo	用户	2007-1-5 16:52:38
JBPM_DECISIONCONDITIONS	dbo	用户	2007-1-5 16:52:38
JBPM_DELEGATION	dbo	用户	2007-1-5 16:52:38
JBPM_EVENT	dbo	用户	2007-1-5 16:52:38
JBPM_EXCEPTIONHANDLER	dbo	用户	2007-1-5 16:52:38
JBPM_ID_GROUP	dbo	用户	2007-1-5 16:52:38
JBPM_ID_MEMBERSHIP	dbo	用户	2007-1-5 16:52:38
JBPM_ID_PERMISSIONS	dbo	用户	2007-1-5 16:52:38
JBPM_ID_USER	dbo	用户	2007-1-5 16:52:38
JBPM_LOG	dbo	用户	2007-1-5 16:52:38
JBPM_MESSAGE	dbo	用户	2007-1-5 16:52:38
JBPM_MODULEDEFINITION	dbo	用户	2007-1-5 16:52:38
JBPM_MODULEINSTANCE	dbo	用户	2007-1-5 16:52:38
JBPM_NODE	dbo	用户	2007-1-5 16:52:38
JBPM_POOLEDACTOR	dbo	用户	2007-1-5 16:52:38
JBPM_PROCESSDEFINITION	dbo	用户	2007-1-5 16:52:38
JBPM_PROCESSINSTANCE	dbo	用户	2007-1-5 16:52:38
JBPM_RUNTIMEACTION	dbo	用户	2007-1-5 16:52:38
JBPM_SWIMLANE	dbo	用户	2007-1-5 16:52:38
JBPM_SWIMLANEINSTANCE	dbo	用户	2007-1-5 16:52:38
JBPM_TASK	dbo	用户	2007-1-5 16:52:38
JBPM_TASKACTORPOOL	dbo	用户	2007-1-5 16:52:38
JBPM_TASKCONTROLLER	dbo	用户	2007-1-5 16:52:38
JBPM_TASKINSTANCE	dbo	用户	2007-1-5 16:52:38
JBPM_TIMER	dbo	用户	2007-1-5 16:52:38
JBPM_TOKEN	dbo	用户	2007-1-5 16:52:38
JBPM_TOKENVARIABLEMAP	dbo	用户	2007-1-5 16:52:39
JBPM_TRANSITION	dbo	用户	2007-1-5 16:52:39
JBPM_VARIABLEACCESS	dbo	用户	2007-1-5 16:52:39
JBPM_VARIABLEINSTANCE	dbo	用户	2007-1-5 16:52:39
role	dbo	用户	2007-1-18 23:42:43
syscolumns	dbo	系统	2000-8-6 1:29:12

- k) 至此，JBPM 的建表工作完成。

## 5.2 JBPM 流程发布

上一章节，我们已完成了对 JBPM 的 table 的建立，接下来我们将继续探讨如何将我们的流程定义文件发布到 JBPM 数据库当中。

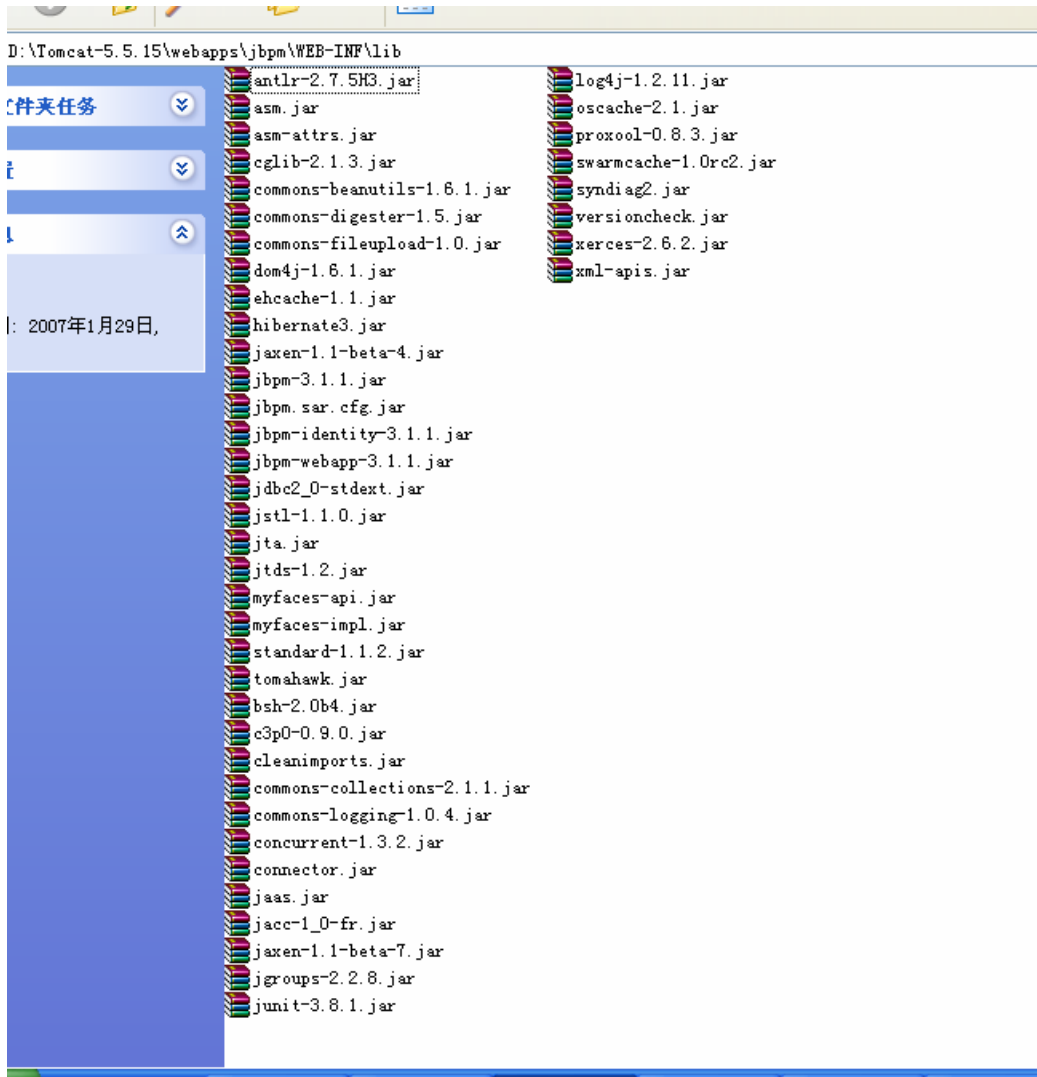
我们这里采用 WEB 应用的形式将流程定义文件发布到数据库当中。我们采用 Tomcat

5.5.15 做为我们的 web server 进行。

### 5.2.1 搭建 JBPM 的 WEB 应用

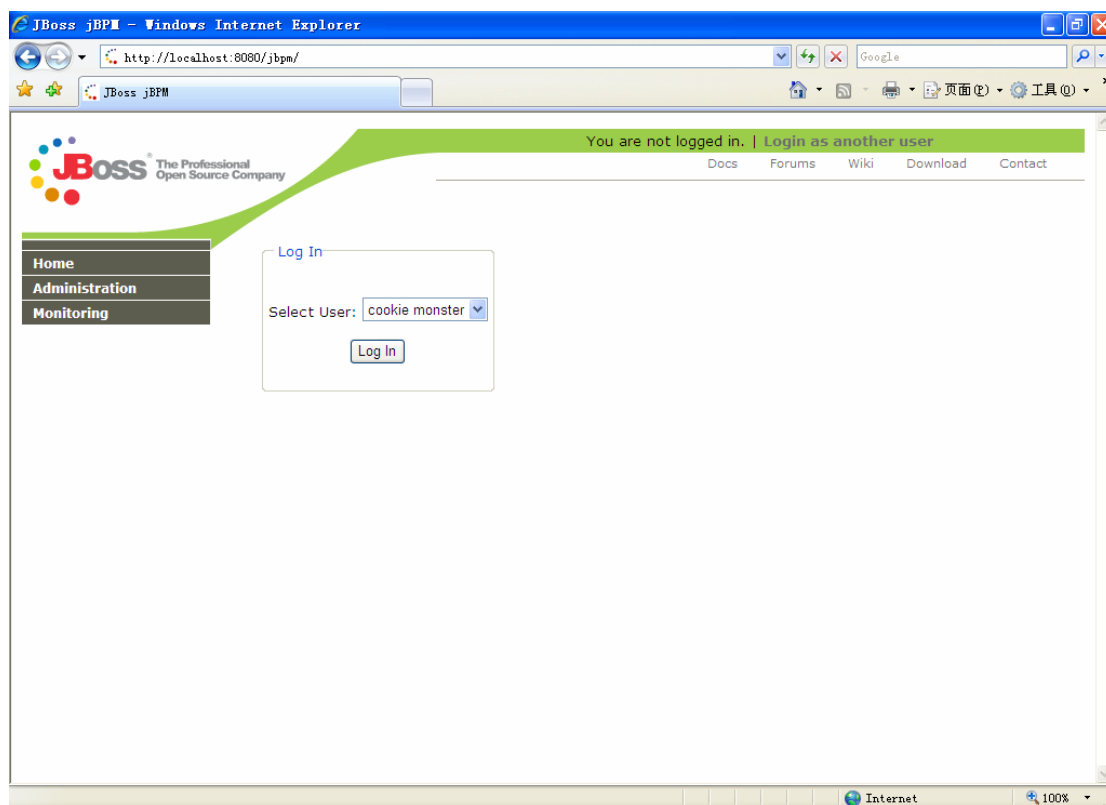
在 JBPM 的发布包中，已经为我们准备了一个 JSF 架构的 WEB 应用，我们可以在这个应用基础之上搭建我们的发布工具。

- a) 在命令行模式下，进入 jbpn 目录，运行 ant 命令编译打包 JBPM 工程。打包完成后进行 jbpn/build 目录，找到 jbpn.war.dir 文件夹，将其 copy 到 tomcat 5.5.15 的 webapps 目录下（由于我们前面在做 JBPM 建表工作的时候已经配置好 JBPM 的数据库连接信息，所以 jbpn.war.dir 的 web 应用里我们就不要再操心数据库的连接问题了）。
- b) 将 webapps 下的 jbpn.war.dir 改名为 jbpn（目的是为了简单）
- c) 打开 jbpn 目录，我们知道 Hibernate 运行时需要一些第三方 jar 包支持，但我们的 jbpn 目录里只有 hibernate 自己的 jar 包，如果这样运行 tomcat5.5.15 我们可以很明显地看到 jbpn 应用发布失败，如何解决这个问题呢，方法很简单。我们下载一个 hibernate3 的工程文件，将其中必须的第三方 jar 文件 copy 到我们的 jbpn 工程里就 OK 了，下图是我 copy 完成后的 jbpn/WEB-INF/lib 目录下的 jar 包列表，供参考：



## 运行 tomcat5.5.15

打开 IE, 在地址栏里输入: `http://localhost:8080/jbpm`, 这时如果我们操作正确的话, 应该可以看到如下信息了:



这个 JBPM 的示例里已经有了一个流程示例，我们看到的便是这个应用的主窗口，大家可以用一下，体验一下 JBPM 工作流带来的乐趣。

## 5.2.2 发布第一个流程

JBPM 里的那个流程示例这里不介绍了，有兴趣的可以自己研究一下。接下来我们来写一个 servlet 来发布我们的第一个流程。

启动 Eclipse，编写发布的 JSP 页面 `deploy.jsp`，并将其 copy 到我们的 `jbpm` 应用的根目录下。

```
<%@ page language="java" contentType="text/html; charset=GB18030"
    pageEncoding="GB18030"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=GB18030">
```

```

<title>Insert title here</title>

</head>

<body>

<form action="deplyWorkflow.servlet" method="post"
enctype="multipart/form-data" name="form1">

    <table width="100%" border="0">

        <tr>

            <td colspan="2">&nbsp;</td>

        </tr>

        <tr>

            <td>请选择要发布的流程定义文件: </td>

            <td><input name="workflowfile" type="file" id="workflowfile"></td>

        </tr>

        <tr>

            <td>&nbsp;</td>

            <td>&nbsp;</td>

        </tr>

        <tr>

            <td colspan="2" align="center"><input type="submit" name="Submit"
value=" 提交  "></td>

        </tr>

    </table>

</form>

</body>

</html>

```

编写我们的流程发布 Servlet，代码如下：

```
package servlet;
```

```
import java.io.IOException;

import java.io.InputStream;

import java.io.PrintWriter;


import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;


import org.apache.commons.fileupload.FileItem;

import org.jbpm.JbpmConfiguration;

import org.jbpm.JbpmContext;

import org.jbpm.graph.def.ProcessDefinition;


import common.UploadFile;


public class DeployWorkflowServlet extends HttpServlet{


    protected void doPost(HttpServletRequest arg0, HttpServletResponse
arg1) throws ServletException, IOException {

<!-- UploadFile 是我这边为了方便对上传文件处理而写的一个 apache 的
common-file 的简单封装-->

        UploadFile upload=new UploadFile(arg0);

        FileItem fileItem=upload.getFile("workflowfile");

        InputStream fin=fileItem.getInputStream();

<!--通过 JbpmConfiguration 创建一个 JbpmContext 实例-->

        JbpmConfiguration config=JbpmConfiguration.getInstance();

        JbpmContext context=config.createJbpmContext();
```

```
        ProcessDefinition
pd=ProcessDefinition.parseXmlInputStream(fin);

        context.deployProcessDefinition(pd); //发布上传过来的流程

        context.close();

        PrintWriter write=arg1.getWriter();

        arg1.setContentType("text/html");

        write.write("WorkFlow deploy success....."); //显示发布成功
信息

        write.close();

    }
}
```

编译将 DeployWorkflowServlet.java, 并将编译好的 class 其放入 jbpn 应用的 classes 目录下。配置到 web.xml 中, 配置代码如下:

```
<servlet>

    <servlet-name>deplyWorkflow</servlet-name>

    <servlet-class>servlet.DeployWorkflowServlet</servlet-class>

</servlet>

<servlet-mapping>

    <servlet-name>deplyWorkflow</servlet-name>

    <url-pattern>/deplyWorkflow.servlet</url-pattern>

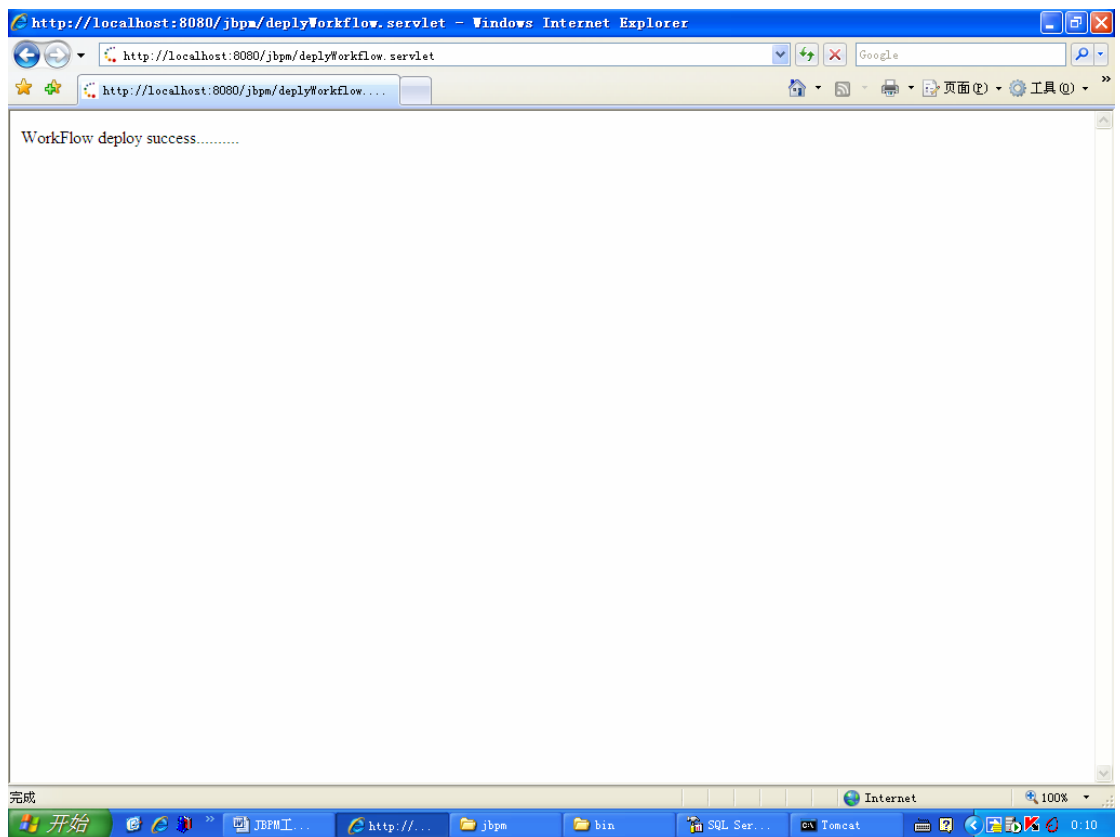
</servlet-mapping>
```

启动Tomcat5.5.15, 访问<http://localhost:8080/jbpm/deploy.jsp>, 我们将看到如下页面



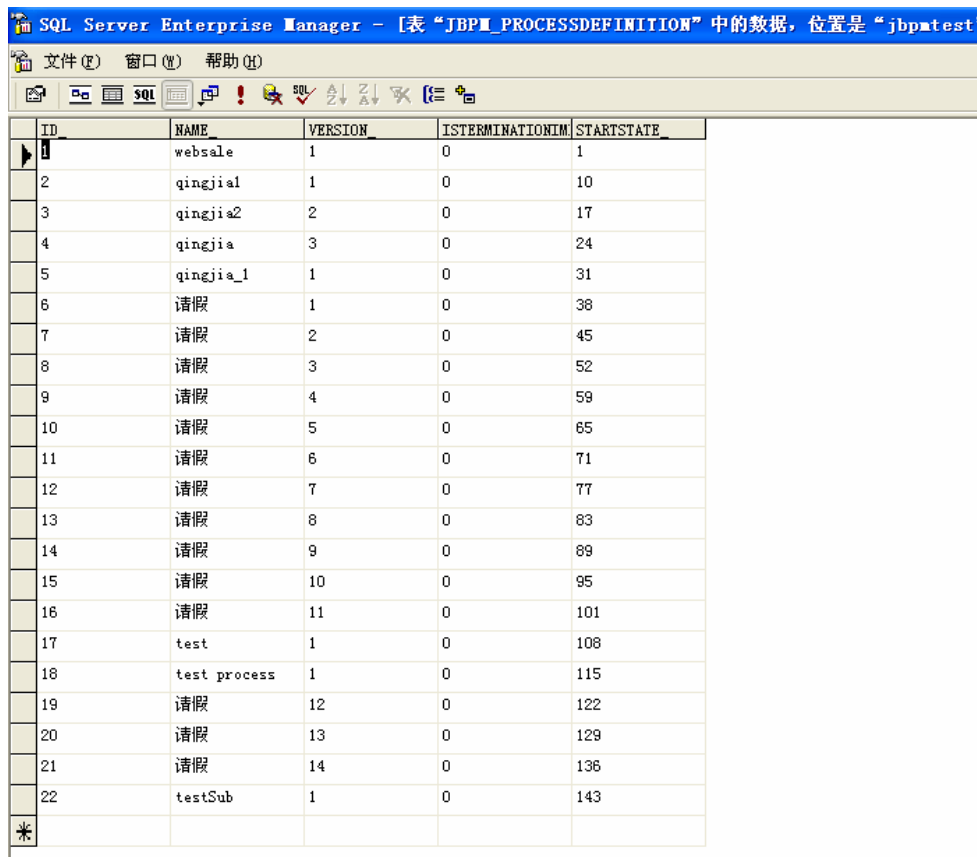


选择我们前面做过的流程定义文件，点击“提交”按钮



显示发布成功信息，打开我们的 MSSQL 的企业管理器，打开表

“JBPM\_PROCESSDEFINITION”，最后一条就是我们刚才发布的 testsub 流程，如下图



ID	NAME	VERSION	ISTERMINATIONIM	STARTSTATE
1	websale	1	0	1
2	qingjia1	1	0	10
3	qingjia2	2	0	17
4	qingjia	3	0	24
5	qingjia_1	1	0	31
6	请假	1	0	38
7	请假	2	0	45
8	请假	3	0	52
9	请假	4	0	59
10	请假	5	0	65
11	请假	6	0	71
12	请假	7	0	77
13	请假	8	0	83
14	请假	9	0	89
15	请假	10	0	95
16	请假	11	0	101
17	test	1	0	108
18	test process	1	0	115
19	请假	12	0	122
20	请假	13	0	129
21	请假	14	0	136
22	testSub	1	0	143

OK，我们的第一个流程发布成功。

## 六、日历(Scheduler)

JBPM 的 Scheduler 可以实现在 JBPM 流程中定时触发某一动作。在流程中 JBPM 提供了 timer 节点供我们使用，通过这个节点我们可以实现节点动作的定时触发。定时器 timer 可以被用于 decision、fork、join、node、process-state、state、super-state、task-node，可以设置开始时间 duedate 和频率 repeat，定时器动作可以是所支持的任何动作元素，如 action、script、create-timer、cancel-timer，来执行我们设置的商务动作。我们可以把 Scheduler 理解成一个后台线程在不停的监听着 timer(jbpm\_timer 表)，如果有需要触发的 timer 生成了，就按照 timer 的属性定时或者循环触发它。

jbpm 提供了 2 种调用 scheduler 的方法：

一种是用在 web 应用的，采用 `org.jbpm.scheduler.impl.SchedulerServlet`，具体的使用方法在 JBPM 提供的 javadoc 里有很好的示例，我们只需在 `web.xml` 中加载它就行了；

另一种是针对性的 C/S 程序，jbpm 提供了一个很好的示例

`org.jbpm.scheduler.impl.SchedulerMain`，我们可以参照它编写我们自己的 Scheduler。

## 6.1 Scheduler 在 C/S 程序上的应用

下面我就编写一个 cs 程序来实现 Scheduler，并调用一个最简单的 timer。这个 timer 从第 5 秒开始每隔 3 秒执行 script 中的内容。

```
<process-definition xmlns="" name="test">
  <start-state name="start">
    <transition name="" to="a">transition</transition>
  </start-state>
  <state name="a">
    <timer name="reminder" due-date="5 seconds"
      repeat="3 seconds">
      <script>System.out.println("node enter");</script>
    </timer>
    <transition name="" to="end">transition</transition>
  </state>
  <end-state name="end">end-state</end-state>
</process-definition>
```

```
package test;
import org.jbpm.*;
import org.jbpm.graph.def.ProcessDefinition;
import org.jbpm.graph.exe.*;
import org.jbpm.scheduler.impl.Scheduler;

public class Test {
  static JbpmConfiguration jbpmConfiguration =
    JbpmConfiguration.getInstance();
  static ProcessDefinition processDefinition = null;
  static ProcessInstance processInstance = null;
  static Scheduler scheduler = null;
  public static void initScheduler() { //设置Scheduler的属性
    scheduler = new Scheduler();
    int interval = 5000;
  }
}
```

```

        scheduler.setInterval(interval);
        int historyMaxSize = 0;
        scheduler.setHistoryMaxSize(historyMaxSize);
        scheduler.start();
    }

    public static void destroy() { //这个例子没用到
        scheduler.stop();
    }

    static class MySchedulerThread extends Thread { //实际业务处理线程
        public void run() {
            JbpmContext jbpmContext =
jbpmConfiguration.createJbpmContext();
            try {
                long processInstanceId = 1;
                processInstance =
jbpmContext.loadProcessInstance(processInstanceId);
                Token token = processInstance.getRootToken();
                System.out.println(token.getNode());
                //一定要运行到有timer生成, 触发
                token.signal();
                System.out.println(token.getNode());
                jbpmContext.save(processInstance);
                //如果这里程序到这里退出的话可以看到jbpm_timer表里有一条数据
                Thread.sleep(30*1000); //为模拟效果, 此线程停止30秒
                //节点跳过, timer结束, jbpm_timer表该数据清空
                token.signal();
                System.out.println(token.getNode());
                jbpmContext.save(processInstance);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                jbpmContext.close();
            }
        }
    }

    public static void main(String[] args) {
        initScheduler ();
        MySchedulerThread mst=new MySchedulerThread();
        mst.start();
    }
}

```

运行结果:

```

StartState(start)
State(a)
node enter
node enter
node enter
node enter
node enter
node enter
EndState(end)

```

从上面的例子的运行结果当中我们可以看到，当流程开始后进入名为“a”的 state 节点，进入名为“a”的 state 节点五秒后开始启动名为 reminder 的 timer，该 timer 每隔 3 秒运行一次，直到 token 离开当前节点。

## 6.2 Scheduler 在 Web 上的应用

Scheduler 在 Web 上的应用相对来说比较简单，我们只需把 org.jbpm.scheduler.impl.SchedulerServlet 配置到我们的 web.xml 中，然后在我们的流程中配置好 timer 就可以完成我们的流程的调度。

```

.....

<servlet>
  <servlet-name>SchedulerServlet</servlet-name>
  <servlet-class>
    org.jbpm.scheduler.impl.SchedulerServlet
  </servlet-class>
  <init-param>
    <param-name>interval</param-name>
    <param-value>5000</param-value>
  </init-param>
  <init-param>
    <param-name>historyMaxSize</param-name>
    <param-value>50</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>SchedulerServlet</servlet-name>
  <url-pattern>/jbpmScheduler</url-pattern>
</servlet-mapping>

```

.....

流程配置示例:

```
<?xml version="1.0" encoding="UTF-8"?>

<process-definition
  xmlns="" name="test">
  <start-state name="start">
    <transition name="" to="statel"></transition>
  </start-state>
  <state name="statel">
    <timer name="reminder" dueDate="5 seconds" repeat="6 seconds">
      <script>
        System.out.println("reminder start ***"+new Date());
      </script>
    </timer>
    <timer name="endReminder" dueDate="30 seconds" transition="tr2">
      <script>
        System.out.println("endReminder start ***"+new Date());
      </script>
      <cancel-timer name="reminder"/>
    </timer>
    <transition name="tr1" to="end1"></transition>
    <transition name="tr2" to="node1"></transition>
  </state>
  <node name="node1">
    <event type="node-enter">
      <script>
        System.out.println("enter node1 ***"+new Date());
      </script>
    </event>
    <transition name="" to="end1"></transition>
  </node>
  <end-state name="end1"></end-state>

</process-definition>
```

启动流程代码:

```
JbpmConfiguration config=JbpmConfiguration.getInstance();
JbpmContext context=config.getCurrentJbpmContext();
ProcessDefinition
pd=context.getGraphSession().findLatestProcessDefinition("test");
ProcessInstance pi=pd.createProcessInstance();
```

```

        pi.getContextInstance().setVariable("reminderTestDueDate",
arg1.getParameter("reminderTestDueDate"));
        pi.getContextInstance().setVariable("taskuser",
arg1.getParameter("taskuser"));

        pi.signal();

```

在上面的示例流程中，一旦我们在代码中启动流程，流程就开始进入 state1 节点，进入该节点后 5 秒开始 reminder 这个 timer，reminder 会每隔 6 秒钟触发一次，在进入 state1 后的 30 秒后触发 endReminder 这个 timer，该 timer 只会触发一次，触发的时候会启动其内部的 script 和 cancel-timer，cancel-timer 可以用来取消其 name 属性所指定的 timer。我们这里指定的是前面的 reminder，所以当 endReminder 启动后会结束掉前面的 reminder 这个 timer。同时当 endReminder 执行完成后，流程会按 endReminder 的 transition 属性所指定的 transition 离开当前的 state 节点进入 tr2 所指定的路线进入 node1 节点，同时执行 node1 节点的 script 动作，运行结果如下：

```

reminder start ***Sun Jun 24 14:15:16 CST 2007
reminder start ***Sun Jun 24 14:15:22 CST 2007
reminder start ***Sun Jun 24 14:15:28 CST 2007
reminder start ***Sun Jun 24 14:15:34 CST 2007
reminder start ***Sun Jun 24 14:15:40 CST 2007
endReminder start ***Sun Jun 24 14:15:45 CST 2007
enter node1 ***Sun Jun 24 14:15:45 CST 2007

```

## 6.3 Scheduler 时间的分类

关于 dueDate 的格式，可以分为两种。一种是我们上面所用到的可以把它叫做绝对时间，比如 5 seconds, 5 days, 5 months... (5 秒, 5 天, 5 个月...), 可以使用的有 second, seconds, minute, minutes, hour, hours, day, days, week, weeks, month, months, year, years; 第二种叫业务时间格式，就是在绝对时间里加上 business，也就是说加了 business 就叫业务时间。具体的业务时间怎么去界定呢？我们可以打开 jbp\*.jar 里的文件 org/jbpm/calendar/jbpm.business.calendar.properties 指定了什么是业务时间

```
hour.format=HH:mm
```

```
#weekday ::= [<daypart> [& <daypart>]*]

#daypart ::= <start-hour>-<to-hour>

#start-hour and to-hour must be in the hour.format

#dayparts have to be ordered

weekday.monday=    9:00-12:00 & 12:30-17:00

weekday.tuesday=   9:00-12:00 & 12:30-17:00

weekday.wednesday= 9:00-12:00 & 12:30-17:00

weekday.thursday=  9:00-12:00 & 12:30-17:00

weekday.friday=    9:00-12:00 & 12:30-17:00

weekday.saturday=

weekday.sunday=

day.format=dd/MM/yyyy

# holiday syntax: <holiday>

# holiday period syntax: <start-day>-<end-day>

# below are the belgian official holidays

holiday.1= 01/01/2005 # nieuwjaar

holiday.2= 27/3/2005  # pasen

holiday.3= 28/3/2005  # paasmaandag

holiday.4= 1/5/2005   # feest van de arbeid

holiday.5= 5/5/2005   # hemelvaart

holiday.6= 15/5/2005  # pinksteren

holiday.7= 16/5/2005  # pinkstermaandag
```



```

holiday.8= 21/7/2005 # my birthday

holiday.9= 15/8/2005 # moederkesdag

holiday.10= 1/11/2005 # allerheiligen

holiday.11= 11/11/2005 # wapenstilstand

holiday.12= 25/12/2005 # kerstmis

business.day.expressed.in.hours= 8

business.week.expressed.in.hours= 40

business.month.expressed.in.business.days= 21

business.year.expressed.in.business.days= 220

```

上面定义通俗的理解就是它归定了从星期一到星期五的 9:00-12:00 & 12:30-17:00 这段时间为上班时间也就是业务时间，星期六和星期日没有定义也就是放假的时间，再往下就是定义了 12 个节假日，最后是一些工作时长统计如一天 8 小时，一个星期 40 小时等。

下面我们定义一个定时器：

```

<state name='catch crooks'>
  <timer name='reminder' dueDate='3 business day'
    repeat='2 business day'
    transition='time-out-transition' >
  <action class='the-remainder-action-class-name' />
  </timer>
</state>

```

这里的定时器（timer）的名字是 reminder，它的 dueDate 定义的是 3 business day, repeat 定义的是 2 business day。也就是说从定时器启动开始在 3 个业务日的时间后每隔两个业务日执行一次 action 的方法，直到 timer 结束。现在就可以来区别业务时间和绝对时间了，假设我是在星期五的早上 10 点启动了这个 timer，那它第一次执行 action 的方法是在什么时候呢？因为这里的定义是有加上 business 的所以要结束工作日历的定义来算了。工作日历中定义了星期六和星期天是不上班的（没有定义），所以在计算时间时就跳过，要到星期三到早上 10 点（这时只是执行完 dueDate 的时间也就是第一次触发 action

的时间), 到星期五早上 10 点第二次触发 action, 到下个星期二早上第二次触发 action (因为星期六和星期日没定义不是业务时间), 以此类推。如果在 timer 定义时没有加上 business 的话, 就以绝对时间进行计算, 也就是在星期一的 10 点第一次触发 action, 到星期三 10 点第二次触发 action, 以此类推。

## 七、异步执行

从前面的 JPDL 流程定义语言里我们知道了很多节点都有 async 属性, 当该属性设置为 true 时表示是异步执行, 否则表示同步执行, 其默认值为 false。通常在同步状态下, 在 JBPM 流程中节点总是在令牌(Token)进入之后被执行, 因此, 节点在客户端线程中被执行。同步执行要把所有的业务代码执行完成, 和流程离开当前节点后进入下面的节点后代码全部执行完成后才能完成整个一个操作过程, 所以如果流程中的逻辑比较复杂的话, 给我们的感觉就是时间较久, 如果是异步则恰恰相反。

在 jBPM 中, 异步执行通过使用一个异步通知系统来实现。当流程执行到达需要异步执行的节点时, jBPM 将挂起执行, 产生一个命令消息并发送该命令消息到命令执行器, 命令执行器是一个单独的组件, 在收到的消息之上它将在流程挂起的地方恢复流程执行。事务因此也将由一个被分裂为两个独立的事务, 每个事务对应于一部分。这样对于一些在流程中在节点里或 Action 里需要花费较长时间执行的逻辑代码可以把节点或 Action 的 async 的属性打开, 设置成异步执行。

## 八、JBPM 流程建模与应用

### 7.1 JBPM 的建模工具

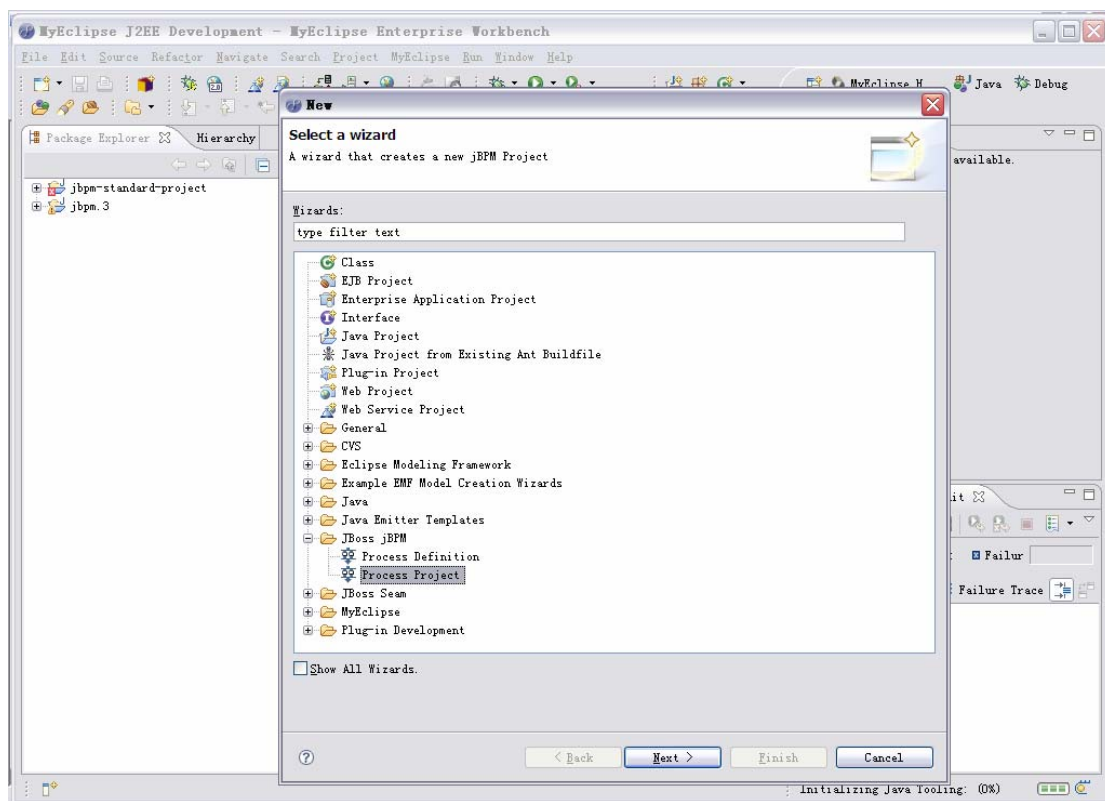
我们前面所做的几个流程, 全部是我们通过手写 XML 的方式实现的, 个人觉得, 这种方式对于我们熟练 JPDL 语法, 熟悉 JBPM 的建模方式是很有帮助的。如果我们的对 JPDL 已熟悉了要做一些复杂流程建模时手写的工作效率是很难尽如人意的。这个时候我们就迫切需

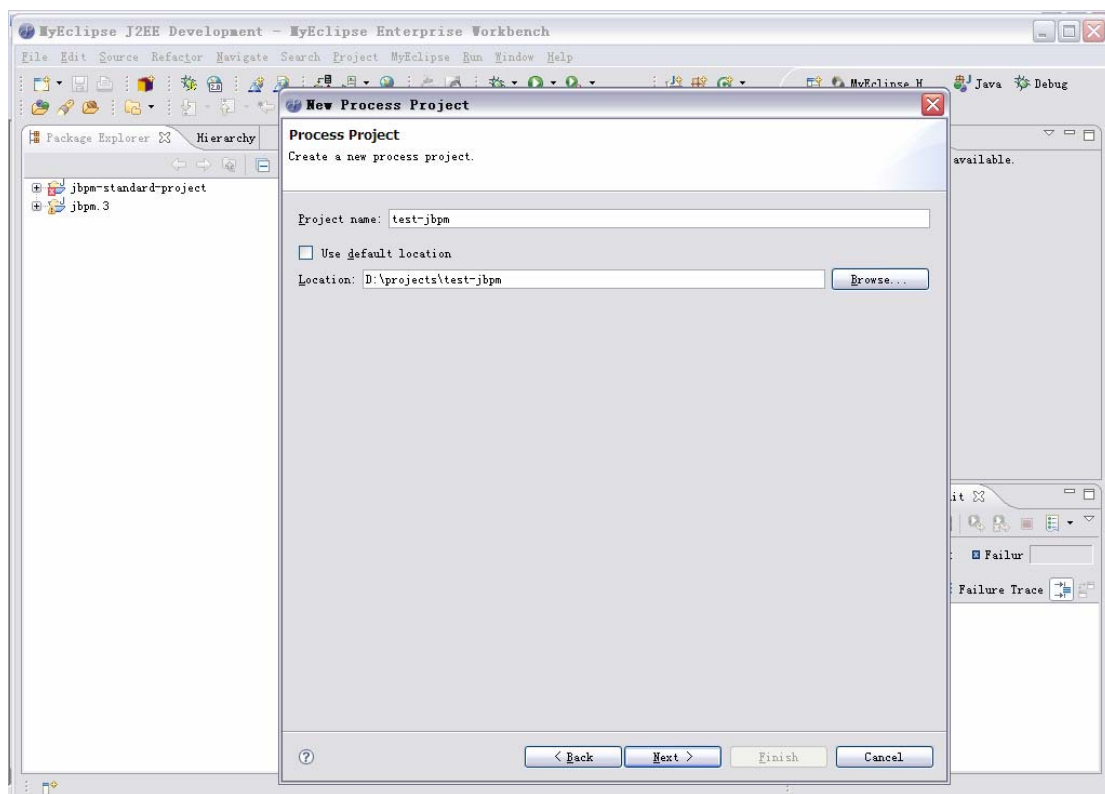
要一个可视化的建模工具来帮助我们提高建模的工作效率，很幸运，JPBM 为我们提供了一个基于 Eclipse 的可视化建模工具。

### 7.1.1 建模工具的安装

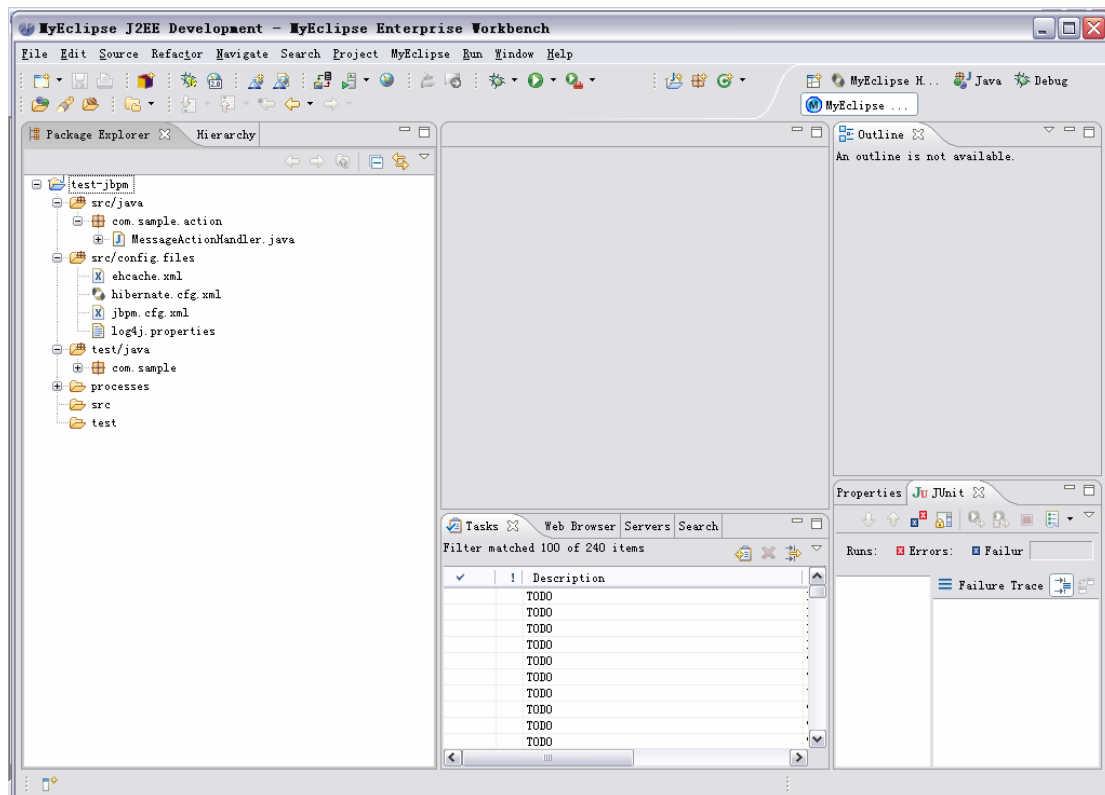
打开我们前面下载的 jbpm-starters-kit-3.1.2 工程包文件，打开其中的 jbpm-designer 目录，我们有两种方式把这个插件安装到 Eclipse 当中，一种是采用 link 的方式，在 jbpm-designer\eclipse\links 目录下已经为我们准备好了 link 文件，还有一种就是把 jbpm-designer\jbpm-gpd-feature\eclipse 目录下的两个文件夹里的内容 copy 到 Eclipse 安装目录下对应的目录里。这里我们采用的是后一种方式。

在命令行模式下，进入 Eclipse 所在目录，输入 eclipse -clean 启动 Eclipse。启动完成后，打开新建窗口，我们可以看到 Jboss jbpm 节点，如下图，点开该节点，我们利用它提供的向导新建一个 JBPM Project 名为 test-jbpm



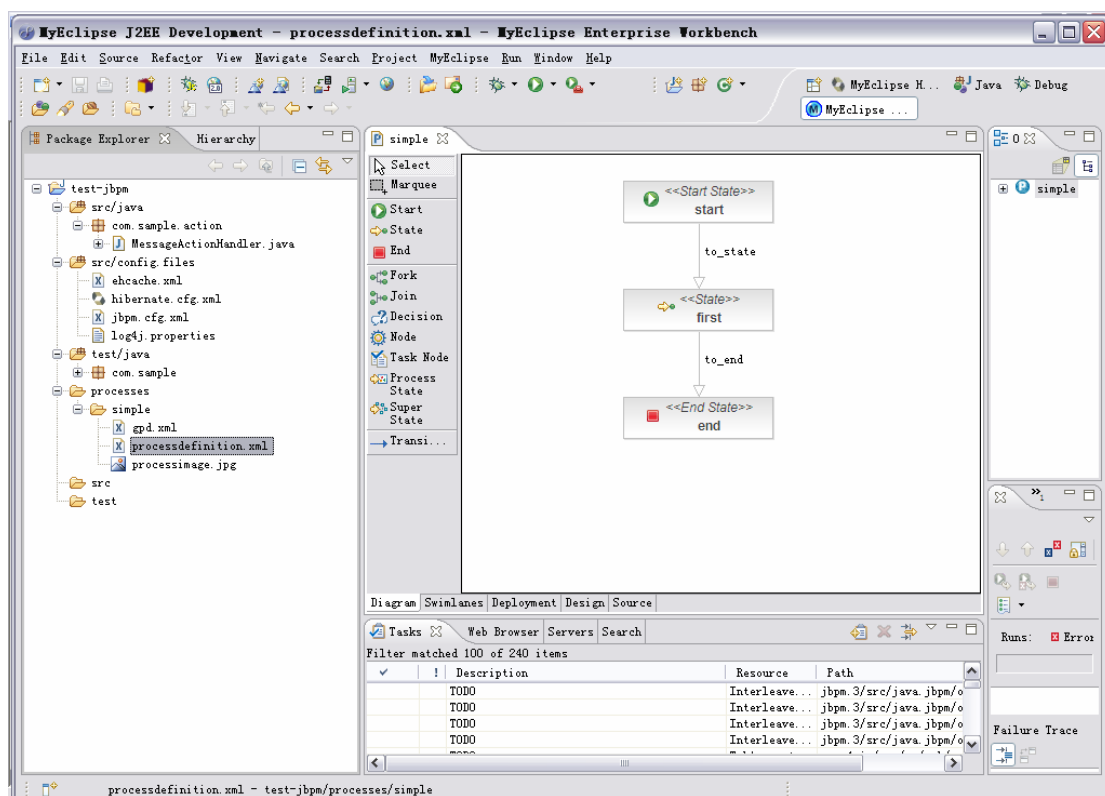


建好的 JPB 工程目录结构图



打开工程的 processes 目录，我们可以看到，JBPM 设计器已经帮我们建好了一个示例流程的模型，双击打开 processdefinition.xml 便可以看到该流程模型的图形化表示，如下

图：



我们的 JBPM 建模工具安装成功。

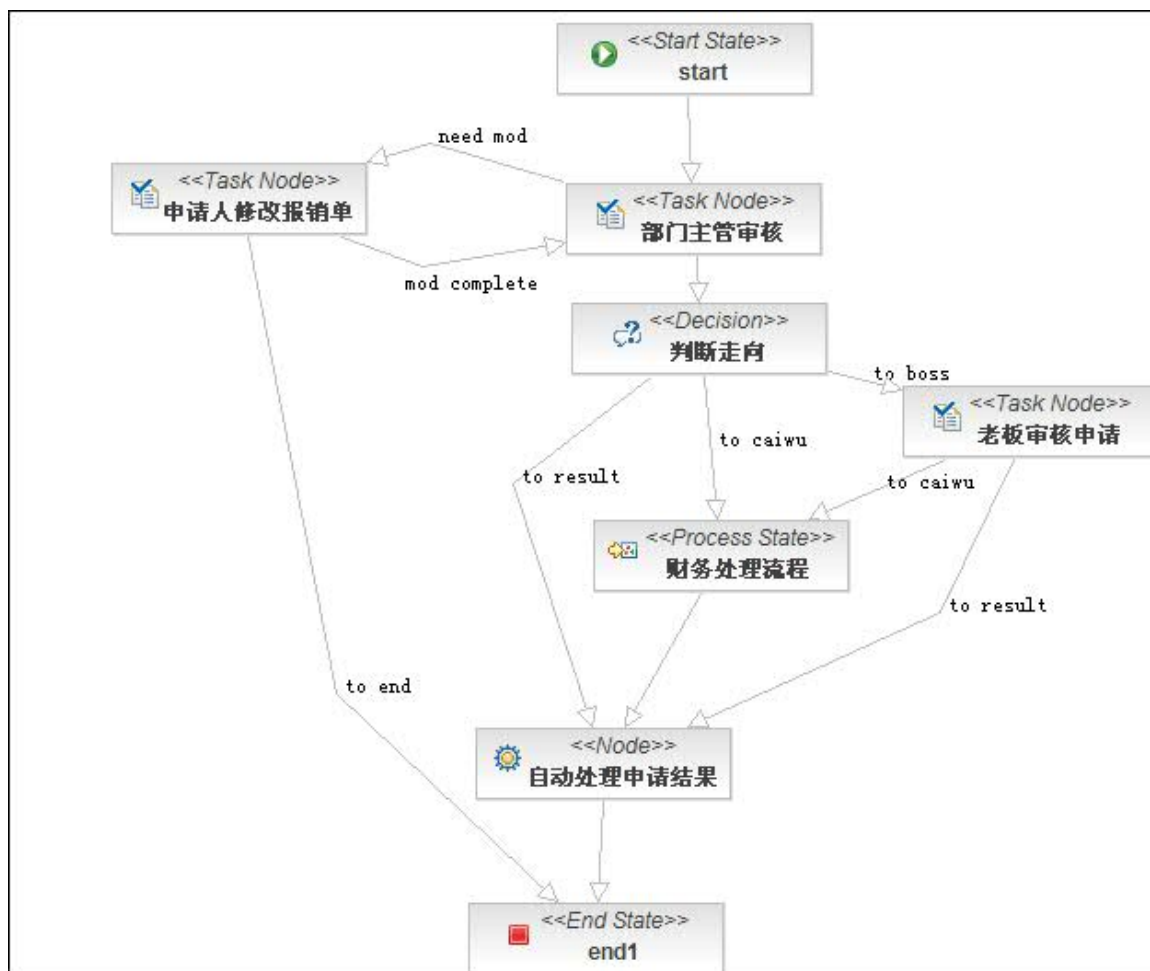
## 7.2 公司报销流程示例

该示例演示的是一个公司报销流程。普通员工可以填写报销单，然后提交主管审批；主管审批可以有三种可能：一是主管可以驳回请求，那么报销人需要重填报销单，或者取消报销操作；二是主管不同意请求，请求直接结束；三是主管同意请求，那又存在两种情况，一是如果报销总费用大于 1000 的话那么会自动转到老板那里，如果小于 1000 就直接进入财务处理子流程，老板审批的话有两种可能，一是同意进入财务处理子流程，二是不同意请求直接结束。

财务处理流程里面只有一个 Node 节点，自动执行一个 Action，并没有做什么特殊处理。这里加上这么一个财务处理子流程目的是介绍一下子流程用法。

### 7.2.1 流程建模

公司报销流程模型图如下：



公司报销流程模型 XML 定义代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<process-definition xmlns="" name="baoxiao">
  <start-state name="start">
    <task name="填写报销单">
      <controller>
        <variable name="baoxiaoId" access="read,write,required"
mapped-name="报销 ID"></variable>
      </controller>
      <assignment
class="demo.workflow.assignment.UserAssignment"></assignment>
    </task>
    <transition name="" to="部门主管审核"></transition>
  
```

```
</start-state>

<task-node name="部门主管审核">

    <task name="主管审核">

        <controller>

            <variable name="baoxiaoId" access="read" mapped-name="报销
ID"></variable>

            </controller>

            <assignment
class="demo.workflow.assignment.ManagerAssignment"></assignment>

        </task>

        <transition name="need mod" to="申请人修改报销单"></transition>

        <transition name="" to="判断走向"></transition>

    </task-node>

    <task-node name="申请人修改报销单">

        <task name="修改报销单">

            <controller>

                <variable name="baoxiaoId" access="read" mapped-name="报销
ID"></variable>

                </controller>

                <assignment
class="demo.workflow.assignment.UserAssignment"></assignment>

            </task>

            <transition name="mod complete" to="部门主管审核"></transition>

            <transition name="to end" to="end1"></transition>

        </task-node>

        <decision name="判断走向">

            <handler class="demo.workflow.action.DecisionProcess"/>

            <transition name="to boss" to="老板审核申请"></transition>

            <transition name="to caiwu" to="财务处理流程"></transition>
```

```
<transition name="to result" to="自动处理申请结果"></transition>

</decision>

<task-node name="老板审核申请">

  <task name="老板审核">

    <controller>

      <variable name="baoxiaoId" access="read" mapped-name="报销
ID"></variable>

      </controller>

      <assignment
class="demo.workflow.assignment.BossAssignment"></assignment>

    </task>

    <transition name="to caiwu" to="财务处理流程"></transition>

    <transition name="to result" to="自动处理申请结果"></transition>

  </task-node>

  <process-state name="财务处理流程">

    <sub-process name="caiwu"/>

    <variable name="baoxiaoId" access="read"></variable>

    <transition name="" to="自动处理申请结果"></transition>

  </process-state>

  <node name="自动处理申请结果">

    <event type="node-enter">

      <action name="action1"
class="demo.workflow.action.ProcessResultAction"></action>

    </event>

    <transition name="" to="end1"></transition>

  </node>

  <end-state name="end1"></end-state>

</process-definition>
```



财务处理子流程的流程图如下：



对应的 XML 代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>

<process-definition
  xmlns="" name="caiwu">
  <end-state name="end1"></end-state>
  <node name="自动财务处理">
    <event type="node-enter">
      <action name="action1"
class="demo.workflow.action.CaiwuProcessAction"></action>
    </event>
    <transition name="" to="end1"></transition>
  </node>
  <start-state name="start">
    <transition name="" to="自动财务处理"></transition>
  </start-state>
</process-definition>
  
```

注：因为子流程和主流程是一样的，所以我们这里没有必要把子流程做的那么复杂，财务子流程没有任务节点，只有一个自动节点，在这个自动节点里只是添加了一个 Action 用来输出从主流程中得到的流程变量。这个 Action 的代码如下：

```

/*
 * CreatePerson:gaojie CreateDate:May 10, 2007 10:41:55 AM
 */
  
```

```

package demo.workflow.action;

import org.jbpm.graph.def.ActionHandler;
import org.jbpm.graph.exe.ExecutionContext;

public class CaiwuProcessAction implements ActionHandler{

    public void execute(ExecutionContext arg0) throws Exception {

        Object baoxiaoId=arg0.getContextInstance().getVariable("baoxiaoId");

        System.out.println("*****财务子流程中报销 ID
(baoxiaoId) :"+baoxiaoId);

    }

}

```

主流程中普通员工用的任务分配类 UserAssignment 代码:

```

/*
 * CreatePerson:gaojie CreateDate:May 10, 2007 10:13:05 AM
 */
package demo.workflow.assignment;
import org.jbpm.graph.exe.ExecutionContext;
import org.jbpm.taskmgmt.def.AssignmentHandler;
import org.jbpm.taskmgmt.exe.Assignable;
import demo.common.Constants;
public class UserAssignment implements AssignmentHandler{

    public void assign(Assignable arg0, ExecutionContext arg1) throws Exception {

        String
issueUser=(String)arg1.getContextInstance().getVariable(Constants.ISSUE_USER);

```

```
        arg0.setActorId(issueUser);  
    }  
}
```

主流程中主管任务分配类 ManagerAssignment 代码如下:

```
/*  
 * CreatePerson:gaojie CreateDate:May 10, 2007 10:34:11 AM  
 */  
package demo.workflow.assignment;  
  
import java.util.List;  
import org.hibernate.Session;  
import org.jbpm.graph.exe.ExecutionContext;  
import org.jbpm.taskmgmt.def.AssignmentHandler;  
import org.jbpm.taskmgmt.exe.Assignable;  
import demo.domain.TbUser;  
  
public class ManagerAssignment implements AssignmentHandler{  
    public void assign(Assignable arg0, ExecutionContext arg1) throws Exception {  
        Session session=arg1.getJbpmContext().getSession();  
  
        String hql="from TbUser u where u.userType=1";  
  
        List userList=session.createQuery(hql).list();  
  
        String[] user=new String[userList.size()];  
  
        for (int i = 0; i < userList.size(); i++) {  
            TbUser u=(TbUser)userList.get(i);  
  
            user[i]=u.getUserName();  
        }  
  
        arg0.setPooledActors(user);  
    }  
}
```

主流程中老板任务分配类 BossAssignment 代码如下:

```
/*
 * CreatePerson:gaojie CreateDate:May 10, 2007 10:34:11 AM
 */
package demo.workflow.assignment;

import java.util.List;
import org.hibernate.Session;
import org.jbpm.graph.exe.ExecutionContext;
import org.jbpm.taskmgmt.def.AssignmentHandler;
import org.jbpm.taskmgmt.exe.Assignable;
import demo.domain.TbUser;

public class BossAssignment implements AssignmentHandler{

    public void assign(Assignable arg0, ExecutionContext arg1) throws Exception {

        Session session=arg1.getJbpmContext().getSession();

        String hql="from TbUser u where u.userType=2";

        List userList=session.createQuery(hql).list();

        String[] user=new String[userList.size()];

        for (int i = 0; i < userList.size(); i++) {

            TbUser u=(TbUser)userList.get(i);

            user[i]=u.getUserName();

        }

        arg0.setPooledActors(user);

    }

}
```

主流程中常量定义类 Constants 代码如下：

```
/*
 * CreatePerson:gaojie CreateDate:May 10, 2007 11:06:32 AM
 */
package demo.common;
```

```
public class Constants {  
  
    public static final String MANAGER_AGREE="主管同意";  
  
    public static final String BOSS_AGREE="老板同意";  
  
    public static final String DISAGREE="不同意";  
  
    public static final String ISSUE_USER="issueUser";  
  
}
```

主流程中在“判断走向”这个节点中的决定流程走向的 Decision 类代码如下：

```
/*  
 * CreatePerson:gaojie CreateDate:May 10, 2007 10:45:29 AM  
 */  
  
package demo.workflow.action;  
  
import java.util.Iterator;  
  
import java.util.List;  
  
import org.hibernate.Session;  
  
import org.jbpm.graph.exe.ExecutionContext;  
  
import org.jbpm.graph.node.DecisionHandler;  
  
import demo.common.Constants;  
  
import demo.domain.TbApprove;  
  
import demo.domain.TbBaoxiaoItem;  
  
/**  
 * 该类是用来处理主管审批完成后，流程该走向哪里。  
 * */  
  
public class DecisionProcess implements DecisionHandler{  
  
    public String decide(ExecutionContext arg0) throws Exception {  
  
        //默认直接进入最后处理结果节点  
  
        String go="to result";  
  
    }  
  
}
```

```
Object baoxiaoId=arg0.getContextInstance().getVariable("baoxiaoId");

Session
session=arg0.getJbpmContext().getSessionFactory().openSession();

String hql="from TbApprove a where a.tbBaoxiao="+baoxiaoId+" order by
a.approveId desc";

TbApprove approve=(TbApprove)session.createQuery(hql).iterate().next();
String result=approve.getApproveResult();

//如果主管审批同意的话，进一步处理
if(result.equals(Constants.MANAGER_AGREE)) {

    hql="from TbBaoxiaoItem b where b.tbBaoxiao="+baoxiaoId+"";

    List ls=session.createQuery(hql).list();

    int amount=0;

    for (Iterator iter = ls.iterator(); iter.hasNext();) {

        TbBaoxiaoItem b = (TbBaoxiaoItem) iter.next();

        amount+=Integer.parseInt(b.getItemMoney());

    }

    //当报销总金额大于 1000 时提交到老板去审批

    if(amount>1000) {

        go="to boss";

    }else{

        //否则直接进入财务处理子流程

        go="to caiwu";

    }

}

session.close();

//返回最终结果

return go;

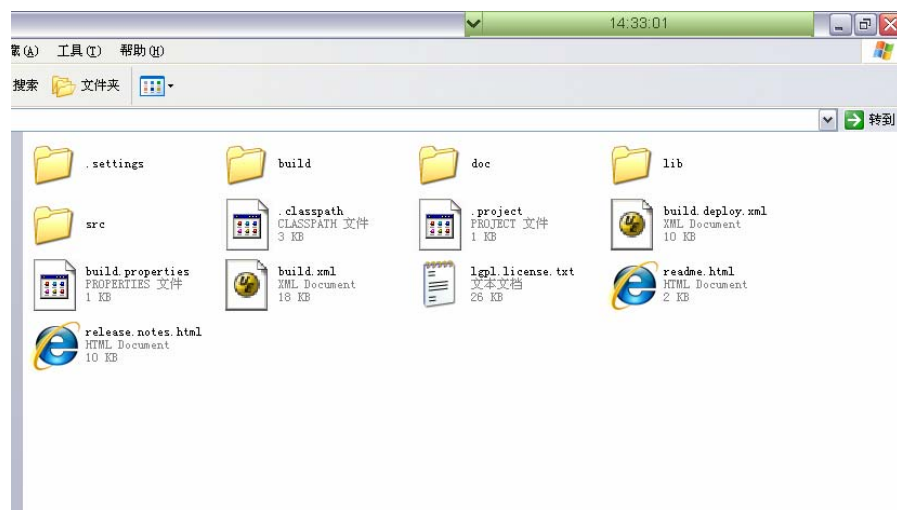
}

}
```

## 7.2.2 流程数据库搭建

我这里采用的是 mssql2000 数据库。

解压 Jbpm-3. x，我这里用的是 Jbpm-3. 1. 2，我放 D 盘根目录下，解压后的文件结构如下：



在 src/resources 目录下添加一个文件夹，比如叫 mssql，然后从边上的 hql 数据库里拷贝 create.db.hibernate.properties 和 identity.db.xml 文件到 mssql 文件夹，修改其中的 create.db.hibernate.properties 中的数据库连接信息，我这边修改如下：

```
# these properties are used by the build script to create
# a hypersonic database in the build/db directory that contains
# the jbpm tables and a process deployed in there

hibernate.dialect=org.hibernate.dialect.SQLServerDialect
hibernate.connection.driver_class=net.sourceforge.jtds.jdbc.Driver
hibernate.connection.url=jdbc:jtds:sqlserver://localhost:1433/jbpm_test
hibernate.connection.username=sa
hibernate.connection.password=
hibernate.show_sql=true
```

这里采用的 MSSQL 的驱动为 apache 的 jtds，所以我们要把 jtds-1.2.jar copy 到 jbpm-3.1.2 目录里的 lib 目录下。

修改 src/config.files/hibernate.cfg.xml 文件，大家知道这个文件是 Hibernate 的数据库配置文件，对照 create.db.hibernate.properties 文件中的数据库连接信息，我们对 hibernate.cfg.xml 的中数据库连接部分做了如下修改：

```
.....

    <!-- jdbc connection properties -->

    <property
name="hibernate.dialect">org.hibernate.dialect.SQLServerDialect</property>

    <property
name="hibernate.connection.driver_class">net.sourceforge.jtds.jdbc.Driver</prop
erty>

    <property
name="hibernate.connection.url">jdbc:jtds:sqlserver://localhost:1433/jbpm_test<
/property>

    <property name="hibernate.connection.username">sa</property>

    <property name="hibernate.connection.password"></property>

.....
```

jbpm\_test 是我们在 MSSQL 里建的一个数据库

接下来我们需要修改根目录下的 build.deploy.xml 文件，这里主要是修改名为 create.db 的 target，修改代码如下：

```
.....

    <target name="create.db" depends="declare.jbpm.tasks, db.clean, db.start"
description="creates a hypersonic database with the jbpm tables and loads the
processes in there">

        <jbpmschema actions="create"

            cfg="${basedir}/src/config.files/hibernate.cfg.xml"

properties="${basedir}/src/resources/mssql/create.db.hibernate.properties"/>

        <loadidentities file="${basedir}/src/resources/mssql/identity.db.xml"
```



```

        cfg="${basedir}/src/config.files/hibernate.cfg.xml"

properties="${basedir}/src/resources/mssql/create.db.hibernate.properties"/>

    <ant antfile="build.xml" target="build.processes" inheritall="false" />

    <deployprocess cfg="${basedir}/src/config.files/hibernate.cfg.xml"

properties="${basedir}/src/resources/mssql/create.db.hibernate.properties">

    <fileset dir="build" includes="*.process" />

    </deployprocess>

    <antcall target="db.stop" />

</target>

.....

```

完成以上工作后，我们就可以利用 ant 来帮助我们构建 JBPM 的数据库表了。

打开 windows 的 dos 窗口，切换到 jbp-3.1.2 目录，运行 ant create.db -buildfile build.deploy.xml 命令即可完成 JBPM 表的构建工作。构建好的表截图如下：

名称	所有者	类型	创建日期
dtproperties	dbo	系统	2007-5-29 16:30:44
JBPM_ACTION	dbo	用户	2007-5-31 11:53:55
JBPM_BYTEARRAY	dbo	用户	2007-5-31 11:53:55
JBPM_BYTELOCK	dbo	用户	2007-5-31 11:53:55
JBPM_COMMENT	dbo	用户	2007-5-31 11:53:55
JBPM_DECISIONCONDITIONS	dbo	用户	2007-5-31 11:53:55
JBPM_DELEGATION	dbo	用户	2007-5-31 11:53:55
JBPM_EVENT	dbo	用户	2007-5-31 11:53:55
JBPM_EXCEPTIONHANDLER	dbo	用户	2007-5-31 11:53:55
JBPM_ID_GROUP	dbo	用户	2007-5-31 11:53:55
JBPM_ID_MEMBERSHIP	dbo	用户	2007-5-31 11:53:55
JBPM_ID_PERMISSIONS	dbo	用户	2007-5-31 11:53:55
JBPM_ID_USER	dbo	用户	2007-5-31 11:53:55
JBPM_LOG	dbo	用户	2007-5-31 11:53:55
JBPM_MESSAGE	dbo	用户	2007-5-31 11:53:55
JBPM_MODULEDEFINITION	dbo	用户	2007-5-31 11:53:55
JBPM_MODULEINSTANCE	dbo	用户	2007-5-31 11:53:55
JBPM_NODE	dbo	用户	2007-5-31 11:53:55
JBPM_POOLEDACTOR	dbo	用户	2007-5-31 11:53:55
JBPM_PROCESSDEFINITION	dbo	用户	2007-5-31 11:53:55
JBPM_PROCESSINSTANCE	dbo	用户	2007-5-31 11:53:55
JBPM_RUNTIMEACTION	dbo	用户	2007-5-31 11:53:55
JBPM_SWIMLANE	dbo	用户	2007-5-31 11:53:55
JBPM_SWIMLANEINSTANCE	dbo	用户	2007-5-31 11:53:55
JBPM_TASK	dbo	用户	2007-5-31 11:53:55
JBPM_TASKACTORPOOL	dbo	用户	2007-5-31 11:53:55
JBPM_TASKCONTROLLER	dbo	用户	2007-5-31 11:53:56
JBPM_TASKINSTANCE	dbo	用户	2007-5-31 11:53:56
JBPM_TIMER	dbo	用户	2007-5-31 11:53:56
JBPM_TOKEN	dbo	用户	2007-5-31 11:53:56
JBPM_TOKENVARIABLEMAP	dbo	用户	2007-5-31 11:53:56
JBPM_TRANSITION	dbo	用户	2007-5-31 11:53:56
JBPM_VARIABLEACCESS	dbo	用户	2007-5-31 11:53:56
JBPM_VARIABLEINSTANCE	dbo	用户	2007-5-31 11:53:56
syscolumns	dbo	系统	2000-8-6 1:29:12

## 7.2.3 构建业务表

在这个例子当中，用户采用外部的用户，在前面介绍的三个 Assignment 中已经体现出来。这里我们涉及到的业务表主要有：用户表（流程中任务的分配）、报销表、报销项目表（与报销表之间有一种主从关系）、审核意见表（对报销的审批历史记录），MSSQL2000 的建库脚本如下：

--用户表

```
create table tb_user(  
    user_id          int primary key identity(1,1),--用户 ID  
    user_name        varchar(30),--用户名  
    user_password     varchar(30),--密码  
    user_type        int --用户类型(0 为普通用户,1 为主管, 2 为老板,3 为财务人员)  
)  
go
```

--报销表

```
create table tb_baoxiao(  
    baoxiao_id       int primary key identity(1,1),  
    baoxiao_title     varchar(30),--报销主题  
    baoxiao_memo      varchar(30),--备注  
    user_id          int,--报销人  
    baoxiao_date      datetime, --报销时间  
    baoxiao_flag      bit --报销状态(0 为未处理, 1 为已处理)  
)  
go
```

--报销项目表

```
create table tb_baoxiao_item(  
    item_id          int primary key identity(1,1),  
    item_name        varchar(30),--项目名称  
    item_money       varchar(100),--项目金额
```

```
    item_memo        varchar(200), --项目备注
    baoxiao_id       int --报销表 ID
)
go
--审核意见表
create table tb_approve(
    approve_id       int primary key identity(1,1),
    user_id          int, --审核人 ID
    baoxiao_id       int, --报销表 ID
    approve_result    varchar(30), --审核结果
    approve_memo      varchar(30), --审核意见
    approve_date      datetime --审核日期
)
go
--为报销表添加外键
alter table tb_baoxiao add CONSTRAINT baoxiao_foreign_key foreign key (user_id)
references tb_user(user_id)
go
--为报销项目表添加外键
alter table tb_baoxiao_item add CONSTRAINT baoxiao_item_foreign_key foreign key
(baoxiao_id) references tb_baoxiao(baoxiao_id)
go
--为审核意见表添加外键
alter table tb_approve add CONSTRAINT approve_user_foreign_key foreign key
(user_id) references tb_user(user_id)
go
alter table tb_approve add CONSTRAINT approve_baoxiao_foreign_key foreign key
(baoxiao_id) references tb_baoxiao(baoxiao_id)
```

```
go
---对用户表进行初始化
Insert into tb_user(user_name,user_password,user_type)
values( 'test' , ' test' ,0)
Go
Insert into tb_user(user_name,user_password,user_type)
values( 'manager' , ' manager' ,1)
Go
Insert into tb_user(user_name,user_password,user_type)
values( 'boss' , ' boss' ,2)
go
```

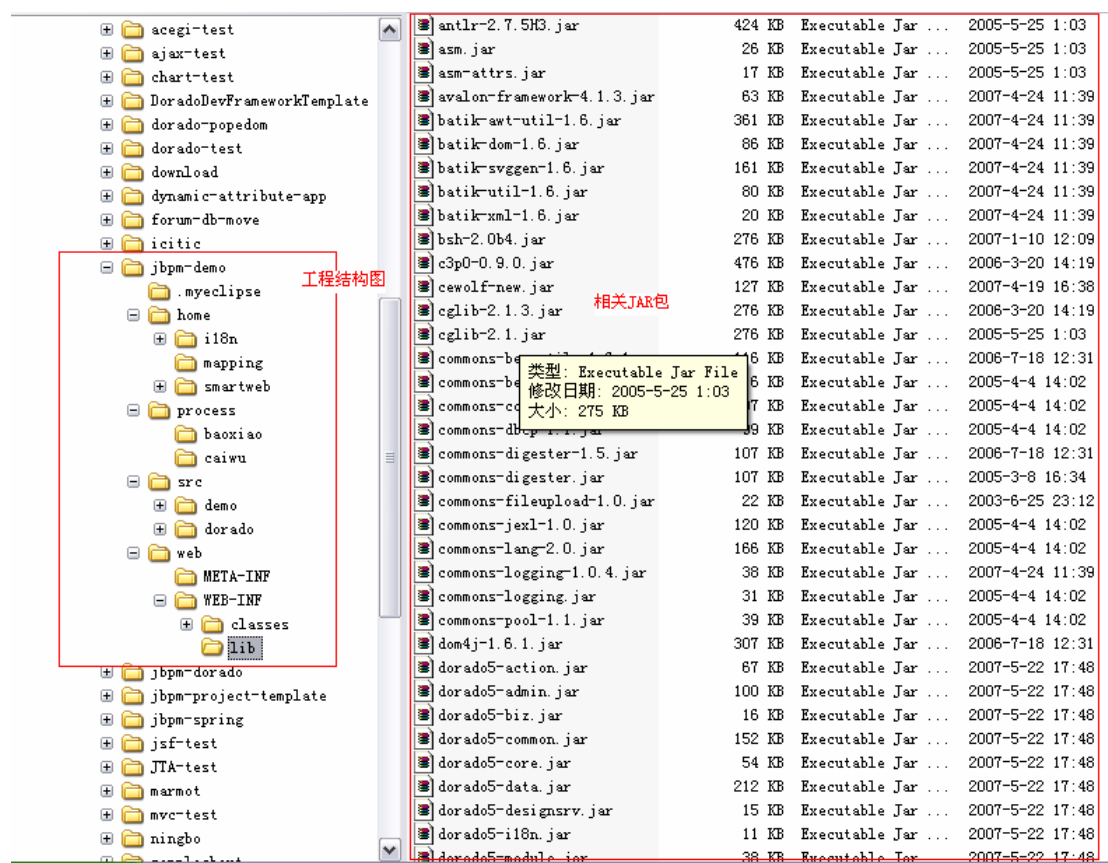
打开 MSSQL2000 的查询分析器，切换到 jbpm\_test 数据库，执行上面这段 SQL，完成我们的业务表的构建工作。

#### 6.2.4 搭建工程

搭建工程我们有一种很简单的办法。我们知道 jbpm-3.1.2 里会自带一个 sample 工程，我们可以把这个工程 copy 一份，在这个工程的基础之上来搭建我们的应用。

为了能快速构建表现层这块，我们采用了目前国内领先的 J2EE 表现层产品 Dorado5 来实现（详情请见 Dorado 的官方网站：<http://www.bstek.com>）。

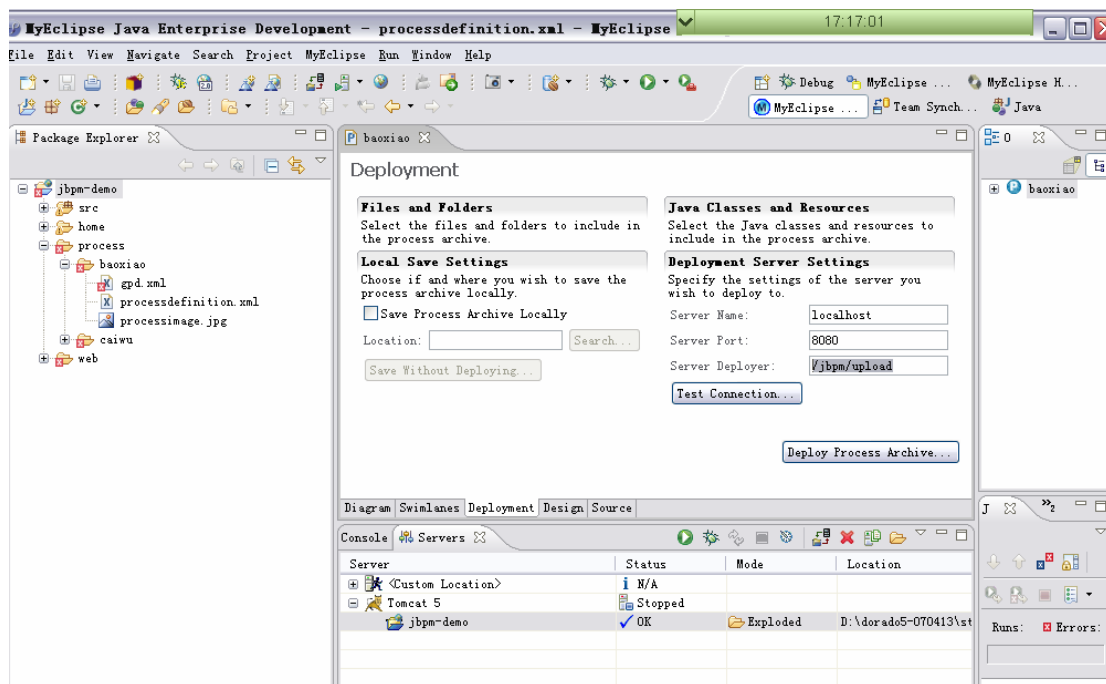
Jbpm 的 Sample 工程与 Dorado 结合后的详细代码请参考附件，也可以与本人联系索取，，拼好后的工程的目录结构如下：



## 7.2.4 报销流程的发布

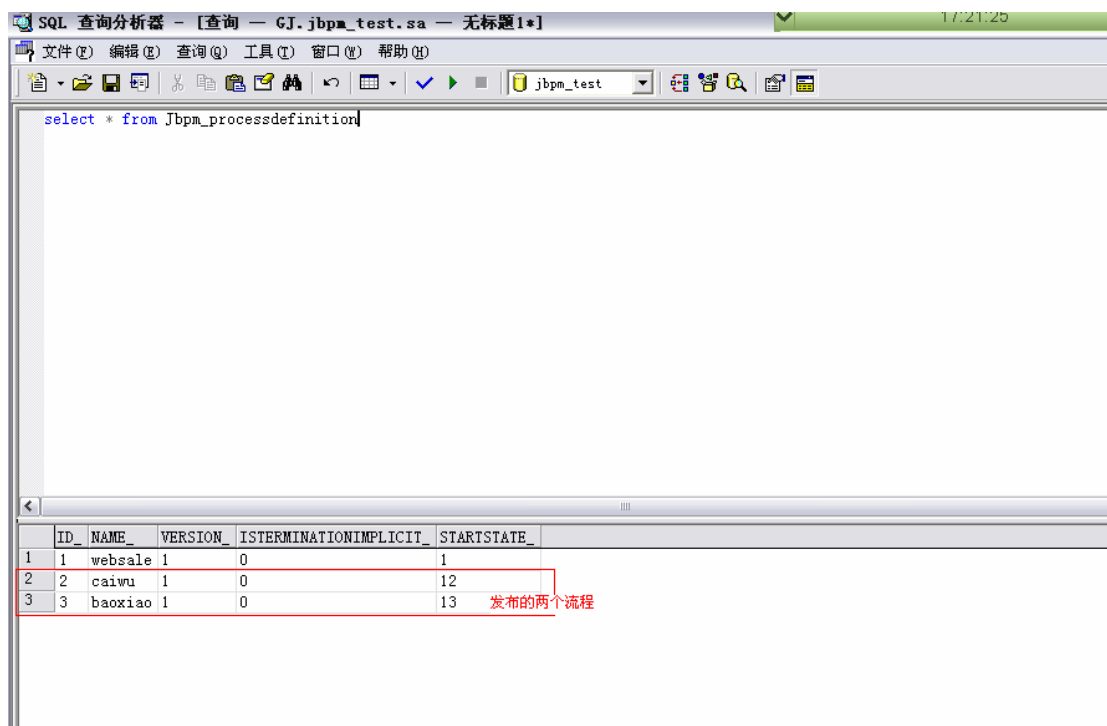
在第五章节我们已经介绍了 JBPM 流程的发布方式，这里我们采用流程引擎 IDE 工具提供的发布方式。

我们当前的工程是在 MyEclipse5.5 的环境下编辑调试的，首先我们要在 MyEclipse5.5 里配置好一个应用服务器，我们这里采用的是 Tomcat5.5。之后将我们前面做的工程导入到 MyEclipse 里，并添加 Web capability, 打开 Server view, 将我们的 JBPM 工程发布到 Tomcat5.5 中并启动 Tomcat5.5。打开我们的财务处理子流程(这里有一点需要注意，在 JBPM 中如果涉及到子流程，一定要先发布子流程再发布主流程，不然可能会造成找不到子流程的错误)，切换到 Deployment 窗口，设置好 server name、server port 和 server deployer 三个属性的值，可以先点击 test connection 按钮看一下连接是否正确，如果没有问题就可以点击 Deploy process archive 按钮来发布我们的流程，如下图：



主流程可以参照子流程的方式发布。

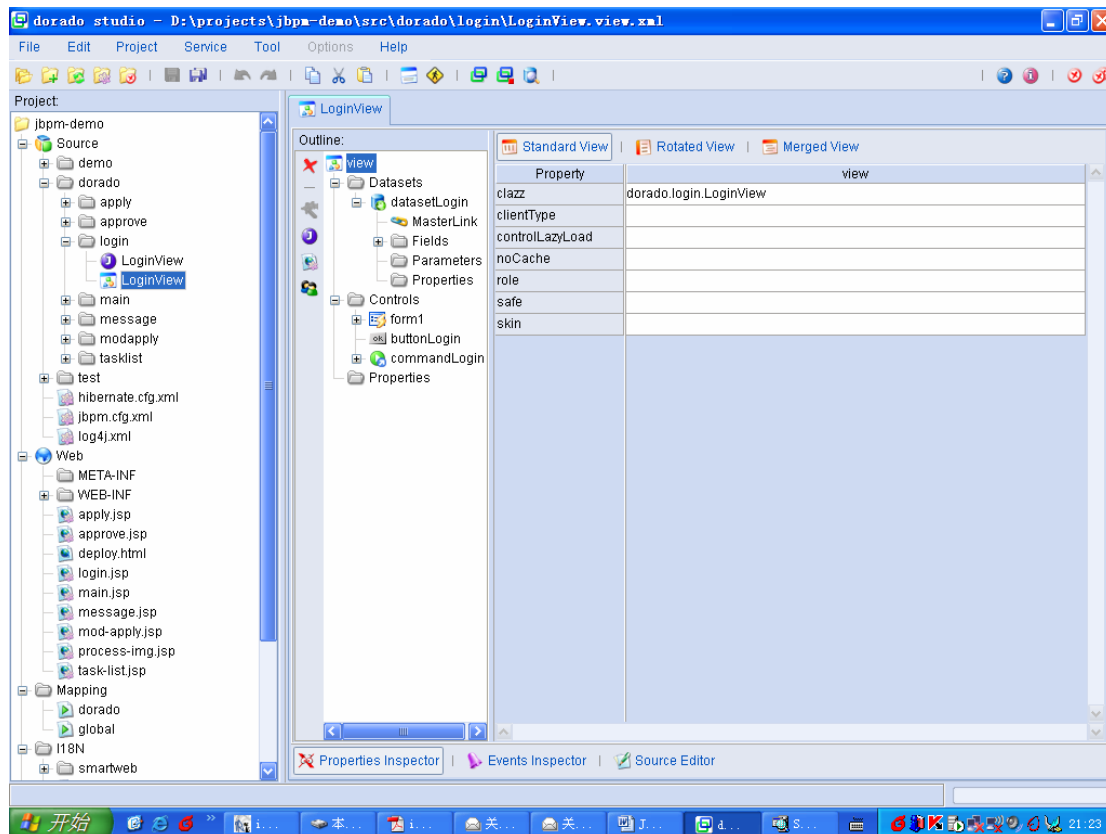
发布完成后我们应该可以在 jibpm\_test 库里 Jbpm\_processdefinition 表里看到我们发布成功的两个流程：



## 7.2.5 应用程序搭建

### 1) 用户登录

用户登录的页面比较简单，利用 Dorado 工具就更加简单。建好的 Dorado ViewModel 如下：



提交处理代码如下：

```
package dorado.login;

import java.util.List;

import org.hibernate.Session;
import org.jbpm.JbpmConfiguration;
import org.jbpm.JbpmContext;

import com.bstek.dorado.common.DoradoContext;
import com.bstek.dorado.data.ParameterSet;
import com.bstek.dorado.view.DefaultViewModel;

import demo.domain.TbUser;

/**
 * LoginView
 */
```

```
public class LoginView extends DefaultViewModel {
    public void login(ParameterSet parameters, ParameterSet outParameters)
        throws Exception {
        String username=parameters.getString("username");
        String userpwd=parameters.getString("userpwd");
        String hql="from TbUser u where u.userName='"+username+"' and
u.userPassword='"+userpwd+"'";
        JbpmConfiguration config=JbpmConfiguration.getInstance();
        JbpmContext context=config.getCurrentJbpmContext();
        Session session=context.getSessionFactory().openSession();

        List ls=session.createQuery(hql).list();

        session.close();
        if(ls.size()>0){
            TbUser user=(TbUser)ls.get(0);

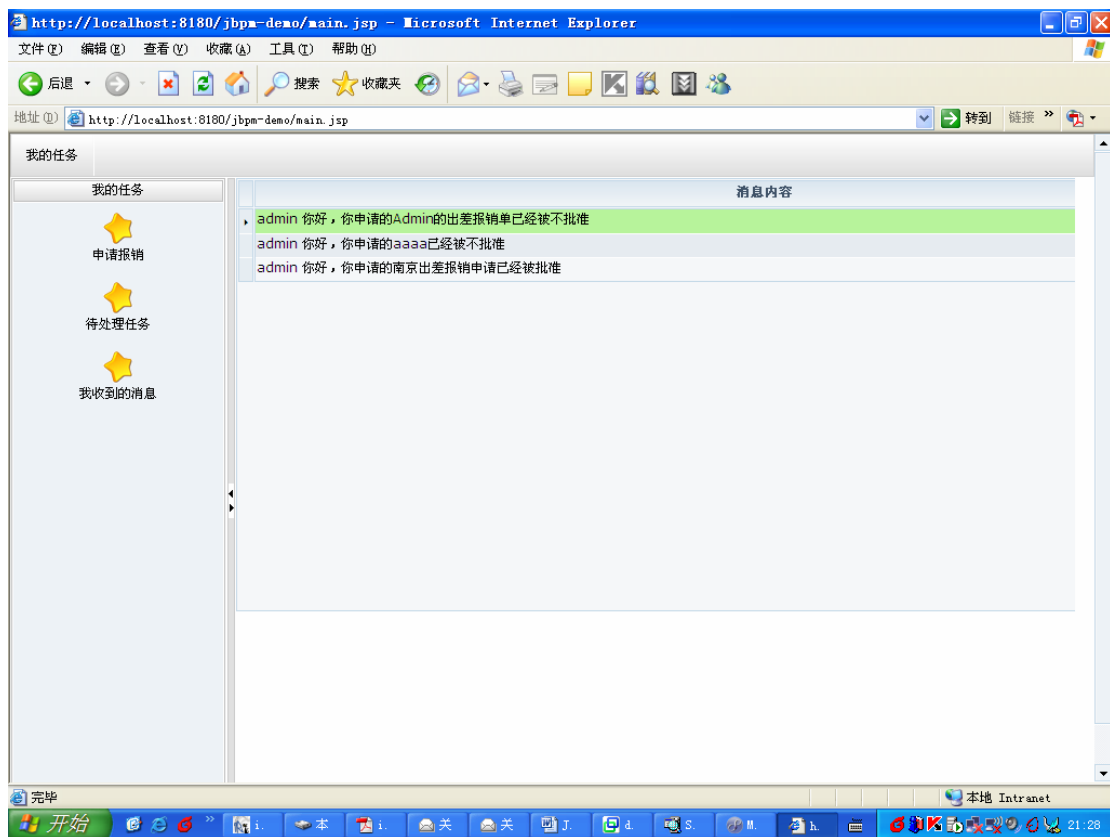
            DoradoContext.getContext().setAttribute(DoradoContext.SESSION, "username",
username);
            DoradoContext.getContext().setAttribute(DoradoContext.SESSION, "userId",
user.getUserId());
            DoradoContext.getContext().setAttribute(DoradoContext.SESSION, "user_type",
user.getUserType());
        } else {
            throw new IllegalArgumentException("用户名密码有误。");
        }
    }
}
```

运行效果如下：





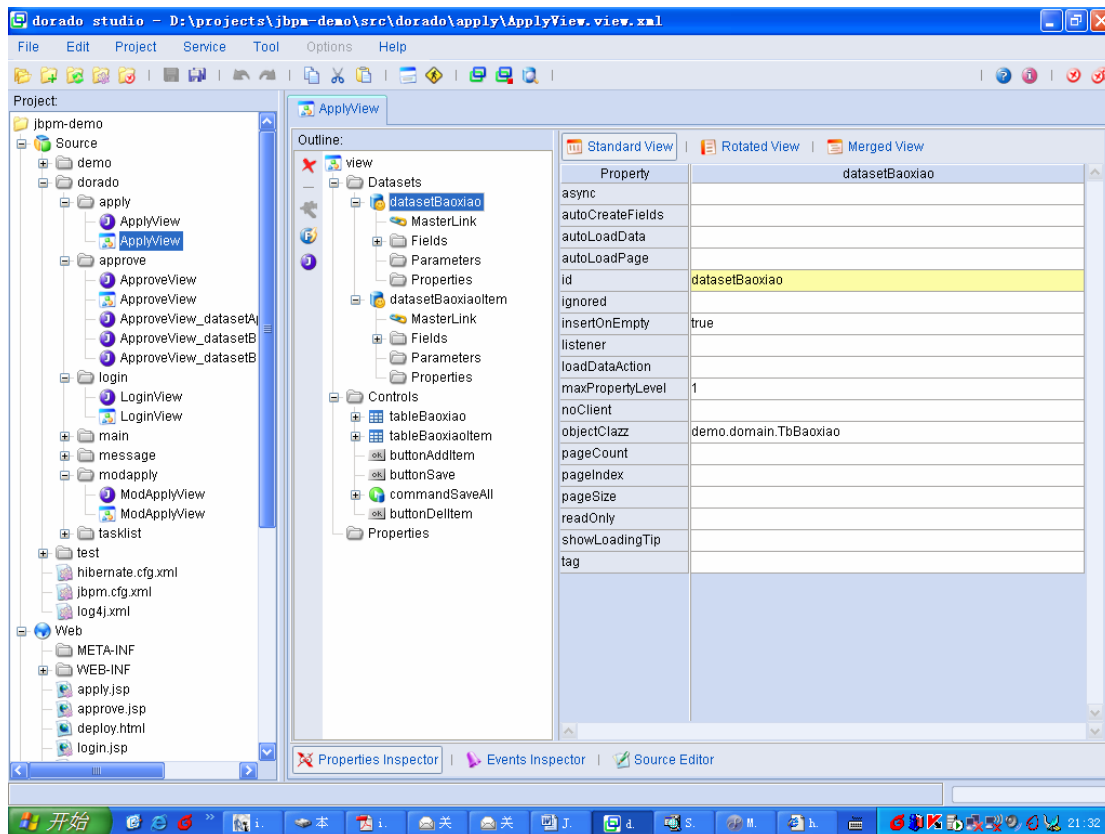
登录成功后既出现操作主界面。



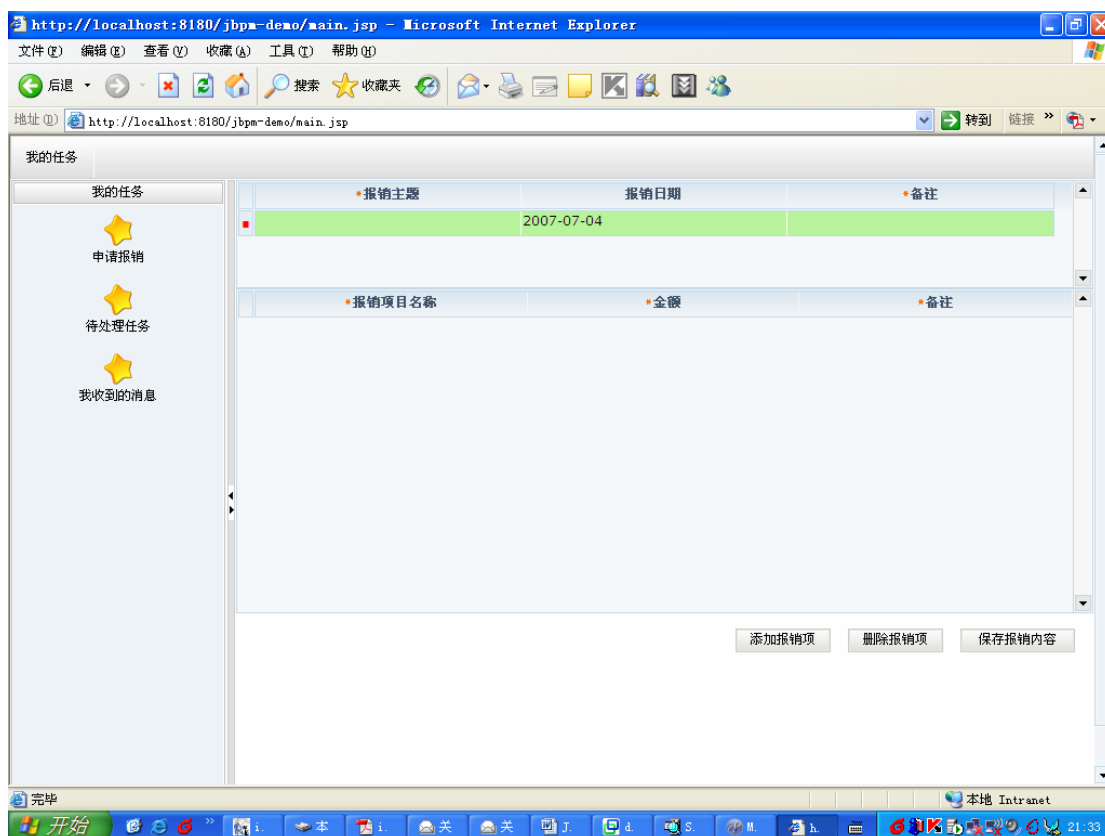
主界面同样采用 Dorado 实现，左边和上边是分级导航和导航菜单。工作区是当前用户收到的消息，消息的传递我们采用 Jbpm 提供的消息功能实现。

## 2) 报销流程的发起与审批（申请报销）

申请报销也既是发起流程，在这里主要涉及到两张表，报销表和报销项目表。这两张表的关系是主从关系，Dorado 的 ViewModel 如下：



运行效果如下：



当点击保存报销内容时，处理代码如下：

```
package dorado.apply;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.Session;
import org.hibernate.Transaction;
import org.jbpm.JbpmConfiguration;
import org.jbpm.JbpmContext;
import org.jbpm.graph.def.ProcessDefinition;
import org.jbpm.graph.exe.ProcessInstance;
import org.jbpm.taskmgmt.exe.TaskInstance;

import com.bstek.dorado.common.DoradoContext;
import com.bstek.dorado.data.Dataset;
import com.bstek.dorado.data.ParameterSet;
import com.bstek.dorado.data.RecordIterator;
import com.bstek.dorado.view.DefaultViewModel;

import demo.common.Constants;
import demo.domain.TbBaoxiao;
import demo.domain.TbBaoxiaoItem;
```

```

import demo.domain.TbUser;

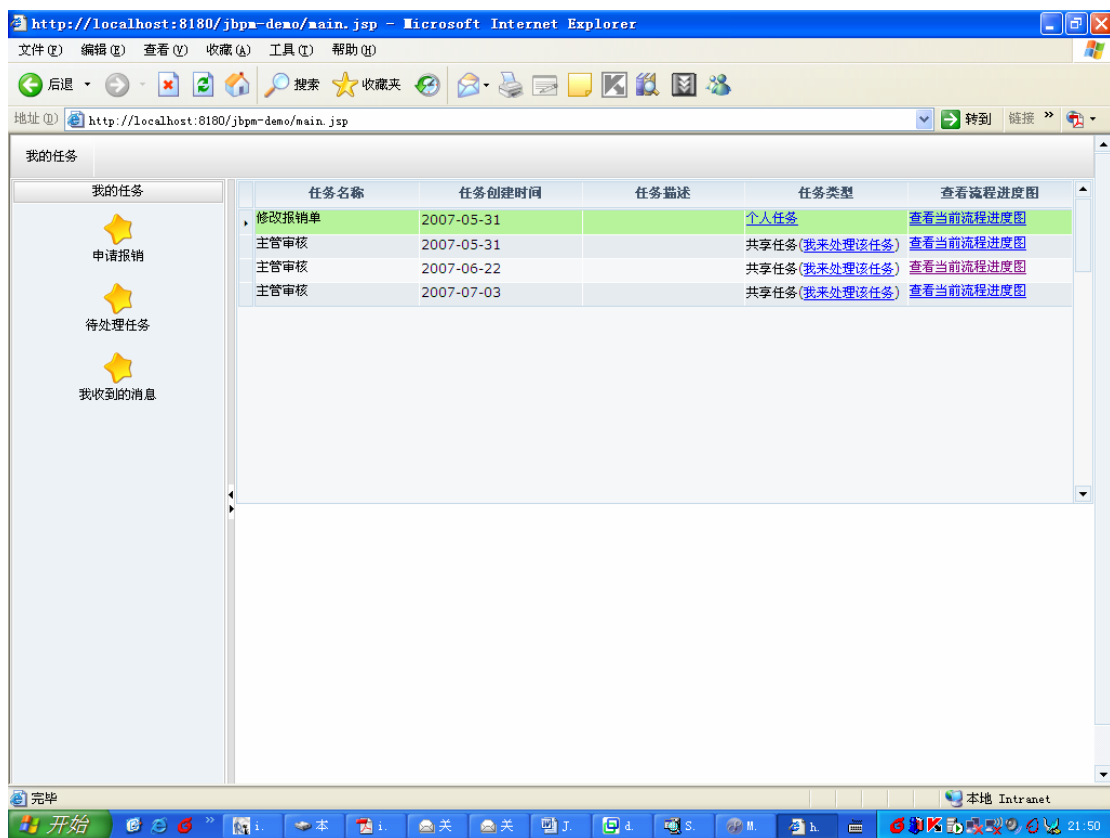
/**
 * ApplyView
 */
public class ApplyView extends DefaultViewModel {
    public void saveAll(ParameterSet parameters, ParameterSet outParameters)
        throws Exception {
        Dataset d=this.getDataset("datasetBaoxiao");
        TbBaoxiao bx=(TbBaoxiao)d.toSingleDO();
        DoradoContext context=DoradoContext.getContext();
        TbUser user=new TbUser();
        user.setUserId((Integer)context.getAttribute(context.SESSION,"userId"));
        bx.setTbUser(user);
        bx.setBaoxiaoFlag(new Byte("0"));
        Dataset dd=this.getDataset("datasetBaoxiaoItem");
        RecordIterator iter=dd.recordIterator();
        Set s=new HashSet();
        while(iter.hasNext()){
            TbBaoxiaoItem item=(TbBaoxiaoItem)dd.toSingleDO(iter.nextRecord());
            item.setTbBaoxiao(bx);
            s.add(item);
        }
        bx.setTbBaoxiaoItems(s);

        JbpmConfiguration config=JbpmConfiguration.getInstance();
        JbpmContext jbpmContext=config.createJbpmContext();
        Session session=jbpmContext.getSessionFactory().openSession();
        Transaction trans=session.beginTransaction();
        try {
            session.save(bx);
            ProcessDefinition
pd=jbpmContext.getGraphSession().findLatestProcessDefinition("baoxiao");
            ProcessInstance pi=pd.createProcessInstance();
            pi.getContextInstance().setVariable(Constants.ISSUE_USER,
context.getAttribute(context.SESSION,"username"));
            TaskInstance ti=pi.getTaskMgmtInstance().createStartTaskInstance();
            ti.setVariable("baoxiaoId", bx.getBaoxiaoId());
            ti.end();
            trans.commit();
        } catch (Exception e) {
            trans.rollback();
            e.printStackTrace();
            throw e;
        }
    }
}

```

```
}finally{  
    session.close();  
    jbpmContext.close();  
  
}  
super.doUpdateData(parameters, outParameters);  
}  
  
}
```

当我们点击“保存报销内容”按钮时就触发上面的处理代码，进行业务数据的保存和流程的开始处理。这样就 manager1 用户就可以在待处理任务里看到当前用户提交的审批任务。



这里有个地方需要说明一下，对于 JBPM 来说，我们可以把任务分给一个用户或者一个用户组（多个用户），分给一个用户就是我们这里把它叫他“个人任务”，分给一组用户的话因为每个用户登录后都可以看到该任务，都可以处理该任务，所以我们把它叫“共享任务”。在这里我们规定，对于个人任务当前用户可以直接去处理，对于共享任务当前用户如果要处理该任务要首先把该任务拉到自己的任务列表里，然后才可以对其进行处理。把任务拉到自己任务列表的方法代码如下：

```
package dorado.tasklist;

import org.jbpm.JbpmConfiguration;
import org.jbpm.JbpmContext;
import org.jbpm.taskmgmt.exe.TaskInstance;

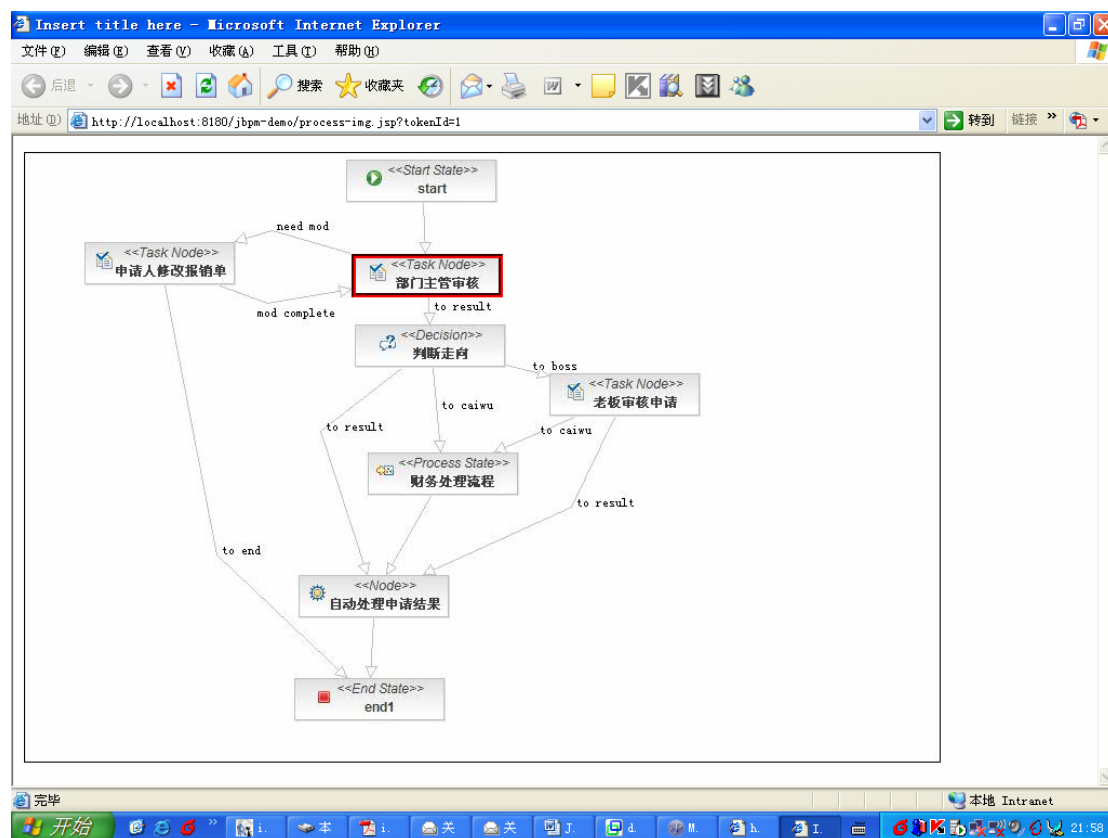
import com.bstek.dorado.common.DoradoContext;
import com.bstek.dorado.data.ParameterSet;
import com.bstek.dorado.view.DefaultViewModel;

/**
 * TaskListView
 */
public class TaskListView extends DefaultViewModel {
    public void changeTask(ParameterSet parameters, ParameterSet outParameters)
        throws Exception {
        JbpmConfiguration config=JbpmConfiguration.getInstance();
        JbpmContext jbpmContext=config.createJbpmContext();
        DoradoContext context=DoradoContext.getContext();
        TaskInstance
ti=jbpmContext.getTaskInstance(parameters.getLong("taskId"));
        String
username=context.getAttribute(context.SESSION,"username").toString();
        ti.setActorId(username);
        jbpmContext.save(ti);
        jbpmContext.close();

    }
}
```

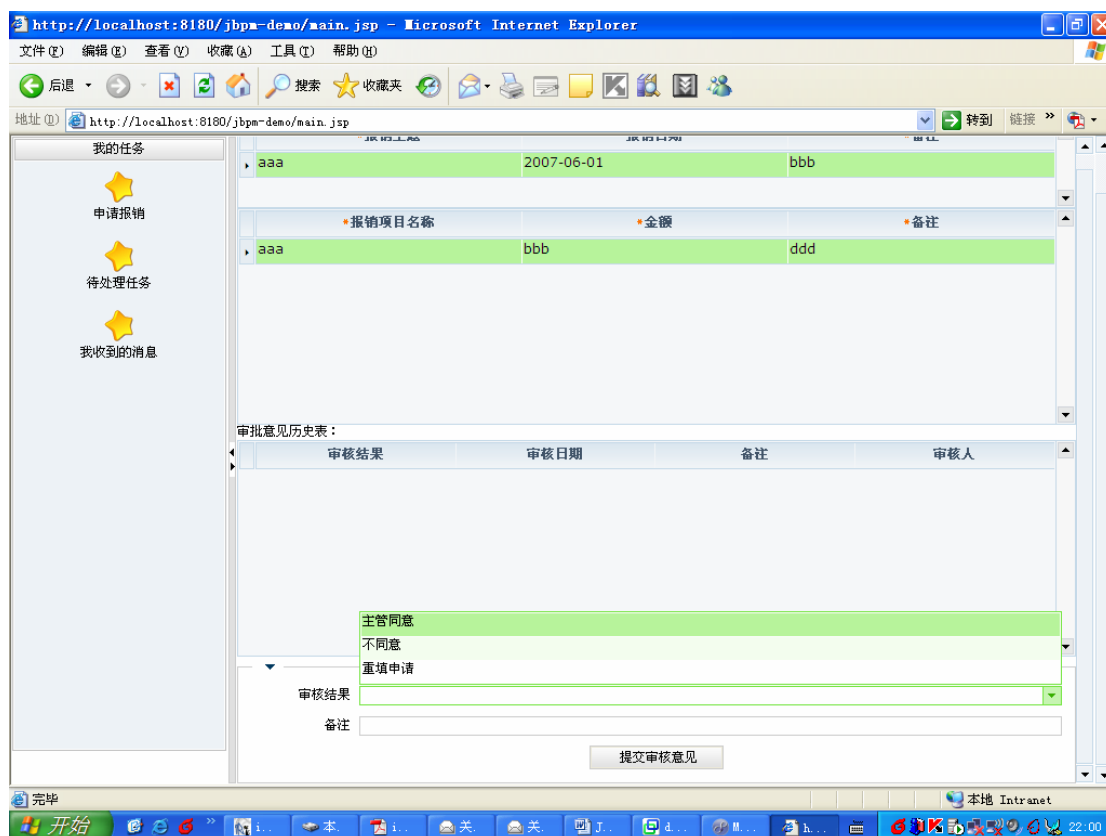
```
}
```

对于当前任务，我们还可以利用 JBPM 提供的流程进度图来查看任务当前在流程中所处的位置，如下图：



把共享任务拉到自己的任务列表后我们就可以针对不同的任务，设定不同的 URL 来做具体的业务页面对其做相应的处理。

如主管审核的操作页面效果如下：



点击“提交审核意见”按钮对应的处理代码如下：

```
package dorado.approve;

import java.util.Date;

import org.hibernate.Session;
import org.hibernate.Transaction;
import org.jbpm.JbpmConfiguration;
import org.jbpm.JbpmContext;
import org.jbpm.taskmgmt.exe.TaskInstance;

import com.bstek.dorado.common.DoradoContext;
import com.bstek.dorado.data.Dataset;
import com.bstek.dorado.data.ParameterSet;
import com.bstek.dorado.view.DefaultViewModel;
```



```
import com.bstek.dorado.view.control.dropdown.ListDropDown;

import demo.domain.TbApprove;

import demo.domain.TbUser;

/**
 * ApproveView
 */
public class ApproveView extends DefaultViewModel {

    protected void initControls() throws Exception {

        super.initControls();

        ListDropDown dd=(ListDropDown)this.getControl("dropdownResult");

        Integer
user_type=(Integer)DoradoContext.getContext().getAttribute(DoradoContext.SESSION, "user_type");

        if(user_type.intValue()==2){

            dd.addItem("老板同意");

        }else{

            dd.addItem("主管同意");

        }

        dd.addItem("不同意");

        dd.addItem("重填申请");

    }

    public void saveTask(ParameterSet arg0, ParameterSet arg1) throws Exception {

        Dataset d = this.getDataset("datasetAdvice");

        TbApprove approve = (TbApprove) d.toSingleDO();

        DoradoContext context = DoradoContext.getContext();

        TbUser user = new TbUser();

        user.setUserId((Integer) context

            .getAttribute(context.SESSION, "userId"));

        approve.setApproveDate(new Date());
```

```
approve.setTbUser(user);

JbpmConfiguration config = JbpmConfiguration.getInstance();

JbpmContext jbpmContext = config.createJbpmContext();

Session session = jbpmContext.getSessionFactory().openSession();

String taskId = arg0.getString("taskId");

Transaction trans=session.beginTransaction();

try {

    session.save(approve);

    trans.commit();

} catch (Exception e) {

    trans.rollback();

    e.printStackTrace();

    throw e;

} finally {

    session.close();

}

TaskInstance ti = jbpmContext.getTaskInstance(Long.valueOf(taskId));

String result = approve.getApproveResult();

if (result.equals("重填申请")) {

    ti.end("need mod");

} else {

    if(result.equals("老板同意")){

        ti.end("to caiwu");

    }else{

        ti.end("to result");

    }

}

jbpmContext.close();

super.doUpdateData(arg0, arg1);
```

```
}  
  
}
```

这时当我们再次查看待处理任务列表时我们会发现刚才处理的任务已从里面消失。其它的业务处理代码我们这里就不再介绍了，因为前面介绍的大体上是差不多的，详细可以去参考用 Dorado 编写的 Jbpm-demo 这个工程。

## 九、写在最后

到这里 JBPM 的教程算是告一段落，由于本人对 JBPM 的了解也只是皮毛，所以文中错误地方在所难免，请见谅。

JBPM目前的最新的版本是 3.2.1，在这个版本里增加了很多新的功能，比如对EJB的支持等，同时也修正了许多BUG，详细情形请参考JBoss网站：<http://www.jboss.org>