jBPM4.1用户指南

作者: JBoss jBPM Teams

翻译: family168.com

版本: 4.1

本版本由贩卖你168发布

http://www.family168.com

本版本由满江红合作推广. www.redsaga.com

2009年9月3日

.... v
1. 导言 1
1.1. 许可证与最终用户许可协议 1
1.2. 下载 1
1.3. 源码 1
1.4. 什么是jBPM 1
1.5. 文档内容 1
1.6. 从jBPM 3升级到jBPM 4 1
1.7. 报告问题 1
2. 安装配置 3
2.1. 发布 3
2.2. 必须安装的软件 3
2.3. 快速上手 3
2.4. 安装脚本 4
2.5. 依赖库和配置文件 5
2.6. JBoss 5
2.7. Tomcat 5
2.8. Signavio基于web的流程编辑器 5
2.9. 用户web应用 5
2.10. Database 6
2.11. 流程设计器（GPD） 6
2.11.1. 获得eclipse 6
2.11.2. 在eclipse中安装GPD插件 6
2.11.3. 配置jBPM运行时 7
2.11.4. 定义jBPM用户库 7
2.11.5. 在目录中添加jPDL4模式 8
2.11.6. 导入示例 9
2.11.7. 使用ant添加部分文件 10
3. 流程设计器（GPD） 11
3.1. 创建一个新的流程文件 11
3.2. 编辑流程文件的源码 12
4. 部署业务归档 14
4.1. 部署流程文件和流程资源 14
4.2. 部署java类 15
5. 服务 16
5.1. 流程定义，流程实例和执行 16
5.2. ProcessEngine流程引擎 18
5.3. Deploying a process部署流程 18
5.4. 卸载已发布的流程定义 20
5.5. 删除流程定义 20
5.6. 启动一个新的流程实例 20
5.6.1. 最新的流程实例 20
5.6.2. 指定流程版本 20
5.6.3. 使用key 20
5.6.4. 使用变量 21
5.7. 执行等待的流向 21
5.8. TaskService任务服务 22
5.9. HistoryService历史服务 22

5.10. ManagementService管理服务 23
6. jPDL 24
6.1. process流程处理 24
6.2. 控制流程Activities活动 25
6.2.1. start启动 25
6.2.2. State状态节点 25
6.2.2.1. 序列状态节点 26
6.2.2.2. 可选择的状态节点 26
6.2.3. decision决定节点 27
6.2.3.1. decision决定条件 28
6.2.3.2. decision expression唯一性表达式 29
6.2.3.3. Decision handler决定处理器 30
6.2.4. concurrency并发 32
6.2.5. end结束 34
6.2.5.1. end process instance结束流程处理实例 34
6.2.5.2. end execution结束流向 34
6.2.5.3. end multiple多个结束 34
6.2.5.4. end State结束状态 35
6.2.6. task 36
6.2.6.1. 任务分配者 37
6.2.6.2. task候选人 38
6.2.6.3. 任务分配处理器 40
6.2.6.4. 任务泳道 41
6.2.6.5. 任务变量 43
6.2.6.6. 在任务中支持e-mail 43
6.2.7. sub-process子流程 44
6.2.7.1. sub-process变量 46
6.2.7.2. sub-process外出值 48
6.2.7.3. sub-process外向活动 49
6.2.8. custom 51
6.3. 原子活动 52
6.3.1. java 52
6.3.2. script脚本 55
6.3.2.1. script expression脚本表达式 55
6.3.2.2. script 文本 56
6.3.3. hql 57
6.3.4. sql 58
6.3.5. mail 59
6.4. Common activity contents通用活动内容 60
6.5. Events事件 60
6.5.1. 事件监听器示例 61
6.5.2. 事件传播 63
6.6. 异步调用 63
6.6.1. 异步活动 64
6.6.2. 异步分支 65
6.7. 用户代码 66
7. Variables变量 68
7.1. 变量作用域 68
7.2. 变量类型 69

8. Scripting脚本 70
9. Indentity身份认证 71
10. 支持邮件 72
10.1. 生产者 72
10.1.1. 默认生产者 72
10.2. 模板 73
10.3. 服务器 73
10.3.1. 多服务器 73
10.4. 扩展点 74
10.4.1. 自定义生产者 74
10.4.1.1. 例子：自定义附件 74
A. 修改日志 76

表 1. 联系方式

译者	联络
康爱媛	kayzhan168@gmail.com
徐会生	xyz20003@gmail.com

表 2. 版本变化

版本	日期	作者	说明
jBPM4 beta1	2009年4月9日	Tom Baeyens	延迟了9天发布
jBPM4 beta2	2009年5月11日	Tom Baeyens	延迟了11天发布
jBPM4 CR1	2009年6月5日	Tom Baeyens	延迟了5天发布
jBPM4 GA	2009年7月10日	Tom Baeyens	延迟了10天发布
jBPM4.1	2009年9月1日	Tom Baeyens	与roadMap同一天发布

第 1 章 导言

最好使用firefox [http://www.mozilla.com/firefox/]浏览这份教程。 在使用internet explorer的时候会有一些问题。

1.1. 许可证与最终用户许可协议

jBPM是依据GNU Lesser General Public License（LGPL） 和JBoss End User License Agreement（EULA）中的协议发布的， 请参考 完整的LGPL协议 [http://docs.jboss.com/jbpm/lgpl.html]和 完整的最终用户协议 [http://docs.jboss.com/jbpm/JBossORG-EULA.txt]。

1.2. 下载

可以从sourceforge上下载发布包。
http://sourceforge.net/project/showfiles.php?group_id=70542&package_id=268068

1.3. 源码

可以从jBPM的SVN仓库里下载源代码。
https://anonsvn.jboss.org/repos/jbpm/jbpm4/

1.4. 什么是jBPM

jBPM是一个可扩展、灵活的流程引擎， 它可以运行在独立的服务器上或者嵌入任何Java应用中。

1.5. 文档内容

在这个用户指南里， 我们将介绍在持久执行模式下的jPDL流程语言。 持久执行模式是指流程定义、流程执行以及流程历史都保存在关系数据库中， 这是jBPM通常使用的方式。

这个用户指南介绍了jBPM中支持的使用方式。 开发指南介绍了更多的、高级的、定制的、 没有被支持的选项。

1.6. 从jBPM 3升级到jBPM 4

没办法实现从jBPM 3到jBPM 4的升级。 可以参考开发指南来获得更多迁移的信息。

1.7. 报告问题

在用户论坛或者我们的支持门户报告问题的时候， 请遵循如下模板：

```
=== 环境 ==============================
```

```
- jBPM Version : 你使用的是哪个版本的jBPM?
- Database : 使用的什么数据库以及数据库的版本
- JDK : 使用的哪个版本的JDK? 如果不知道可以使用'java -version'查看版本信息
- Container : 使用的哪个版本的JDK? 如果不知道可以使用'java -version'查看版本信息
- Configuration : 你的jbpm.cfg.xml中是只导入了jbpm.jar中的默认配置,
   还是使用了自定义的配置?
- Libraries : 你使用使用了jbpm发布包中完全相同的依赖库的版本?
   还是你修改了其中一些依赖库?

=== Process ==================================
这里填写jPDL流程定义

=== API ======================================
这里填写你调用jBPM使用的代码片段

=== Stacktrace ==============================
这里填写完整的错误堆栈

=== Debug logs ==============================
这里填写调试日志

=== Problem description =========================
请保证这部分短小精悍并且切入重点。比如，API没有如期望中那样工作。
或者，比如，ExecutionService.SignalExecutionById抛出了异常。
```

聪明的读者可能已经注意到这些问题已经指向了可能导致问题的几点原因：） 特别是对依赖库和配置的调整都很容易导致问题。 这就是为什么我们在包括安装和使用导入实现建议配置机制时花费了大量的精力。 所以，在你开始在用户手册覆盖的知识范围之外修改配置之前，一定要三思而行。 同时在使用其他版本的依赖库替换默认的依赖库之前， 也一定要三思而行。

第 2 章 安装配置

2.1. 发布

只需要把jBPM下载下来，然后解压到你的硬盘上的什么地方。 你将看到下面的子目录：

- `doc`：用户手册，javadoc和开发指南
- `examples`：用户手册中用到的示例流程
- `install`：安装脚本
- `lib`：第三方库和一些特定的jBPM依赖库
- `src`：源代码
- `jbpm.jar`：jBPM主库归档

2.2. 必须安装的软件

jBPM需要JDK （标准java）5或更高版本。
http://java.sun.com/javase/downloads/index.jsp

为了执行ant脚本，你需要1.7.0或更高版本的apache ant：
http://ant.apache.org/bindownload.cgi

2.3. 快速上手

这个范例安装是最简单的方式开始使用jBPM。 这一章介绍了完成范例安装的步骤。

如果你已经预先下载了jboss-5.0.0.GA.zip，并在你的`${jbpm.home}/install/downloads`目录下 创建了一个 downloads 目 录 ， 并 把 zip 文 件 放 在 那 里 。 否 则 脚 本 会 自 动 为 你 下 载 它。 对于 eclipse-jee-galileo-win32.zip 的 操 作 也 是 一 样的 （ 或 者 linux 上 的 eclipse-jee-galileo-linux-gtk(-x86_64).tar.gz ，Mac OSX 的 eclipse-jee-galileo-macosx-carbon.tar.gz）。

打开命令控制台，进入目录`${jbpm.home}/install`。 然后运行

```
ant demo.setup.jboss
```

或者

```
ant demo.setup.tomcat
```

这将

- 把JBoss安装到`${jbpm.home}/jboss-5.0.0.GA`目录

- 把jBPM安装到JBoss中。
- 安装hsqldb，并在后台启动。
- 创建数据库结构
- 在后台启动JBoss
- 根据示例创建一个examples.bar业务归档，把它发布到jBPM数据库中
- 从${jbpm.home}/install/src/demo/example.identities.sql，读取用户和组。
- 安装eclipse到${jbpm.home}/eclipse
- 启动eclipse

当这些都完成之后，JBoss会在后台运行起来。当eclipse启动完成后， 你就可以继续下面的教程了第 3 章 流程设计器（GPD）。

然后你可以访问jBPM控制台 [http://localhost:8080/jbpm-console] 你可以使用下面用户之一进行登陆：

表 2.1. 示例控制台用户：

用户名	密码
alex	password
mike	password
peter	password
mary	password

已知的限制：当前，控制台的超时现对于报表初始化显得太紧张了。 所以的那个你地方次访问报表时，请求就会超时，控制台就会崩溃。 这时可以进行注销，然后再次登录，就不会出问题了。 这个问题已经提到JIRA中了 JBPM-2508 [https://jira.jboss.org/jira/browse/JBPM-2508]

2.4. 安装脚本

jBPM下载包中包含了一个install目录， 目录中有一个ant的build.xml文件，你可以使用它来 把jBPM安装到你的应用环境中。

最好严格按照这些安装脚本， 进行安装和发布jBPM配置文件。我们可以自定义jBPM配置文件， 但这是不被支持的。

要想执行安装脚本，打开控制台，进入 ${jbpm.home}/install目录。通过ant -p 你可以看到有哪些脚本可以使用。

这些脚本的参数都使用了已经配置好的默认值。

要想指定你的jdbc配置，来生成数据库表结构，直接访问数据库， 像是发布和生成JBoss的数据源，

最简单的办法是在 ${jbpm.home}/install/jdbc目录下更新合适的配置文件。 合适的配置文件将被与数据库对应的脚本读取。

按照配置你可能希望进行一些自定义的配置：

- database ： 默认值是hsqldb。 可选值为mysql， oracle和postgresql
- jboss.version ： 默认值是5.0.0.GA。 可选值是5.1.0.GA

如果想要自定义这些值，只需要像这样使用-D

```
ant -Ddatabase=postgresql demo.setup.jboss
```

作为可选方案，你可以在${user.home}/.jbpm4/build.properties 中设置自定义的参数值

2.5. 依赖库和配置文件

我们提供了自动安装jBPM的ant脚本。 这些脚本会将正确的依赖库和正确的配置文件 为你安装到正确的位置。如果你想在你的应用中创建自己的jBPM， 可以参考开发指南获得更多信息。

2.6. JBoss

install.jbpm.into.jboss任务会把jBPM安装到你的JBoss 5中。 进入安装目录下，执行ant -p可以获得更多信息。 这个安装脚本会把jBPM安装为一个JBoss的服务， 因此所有应用都可以使用同一个jBPM的流程引擎。

可以指定-Djboss.home=PathToYourJBossInstallation 来修改你的JBoss的安装路径。

在JBoss中，ProcessEngine可以通过JNDI获得， new InitialContext().lookup("java:/ProcessEngine")， 相同的流程引擎可以通过Configuration.getProcessEngine()获得。

2.7. Tomcat

install.jbpm.into.tomcat任务会把jBPM安装到 你的JBoss 5中。

2.8. Signavio基于web的流程编辑器

使用install.signavio.into.jboss和 install.signavio.into.tomcat任务可以将Signavio基于web 的流程编辑器安装到JBoss或Tomcat中。

signavio的web应用腺癌还不是默认安装脚本的一部分， 因为它需要依赖JDK 6。

在应用启动之后

2.9. 用户web应用

如果你希望把jBPM部署为你的web应用的一部分，可以使用 create.user.webapp这个安装任务。 这会创建一个包含jBPM的web应用，在${jbpm.home}/install/generated/user-webapp目录下。

如果你在JBoss上或其他包含jta.jar的应用服务器上部署了你的应用， 你需要把${jbpm.home}/install/generated/user-webapp/WEB-INF/lib/jta.jar删除。

2.10. Database

安装脚本也包含了 像用来创建和删除数据库结构的操作。

在你的数据库中创建表结构：

- 首先，更新${jbpm.home}/install/jdbc中的数据库配置文件。
- 然后在${jbpm.home}/install目录下执行ant create.jbpm.schema

2.11. 流程设计器（GPD）

图形化流程设计器（GPD）使用Eclipse作为其平台， 这一节的内容将介绍如何获得和安装Eclipse，并把GPD插件安装到eclipse上。

2.11.1. 获得eclipse

你需要Eclipse3.5.0

使用实例安装 或手工下载eclipse： Eclipse IDE for Java EE Developers (163 MB) [http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/galileo]

eclipse的传统版本无法满足要求， 因为它没有XML编辑器。 Eclipse的Java开发者版也可以工作。

2.11.2. 在eclipse中安装GPD插件

使用Eclipse软件升级（Software Update）机制安装设计器是非常简单的。 在gpd目录下有一个gpd/jbpm-gpd-site.zip文件， 这就是更新站点（archived update site）的 压缩包。

在Eclipse里添加更新站点的方法：

- 帮助 --> 安装新软件...
- 点击 添加...
- 在 添加站点 对话框中，单击 压缩包...
- 找到 gpd/jbpm-gpd-site.zip 并点击 '打开'
- 点击 确定 在 添加站点 对话框中，会返回到 '安装'对话框
- 选择出现的 jPDL 4 GPD 更新站点

- 点击 下一步.. 然后点击 完成
- 接受协议
- 当它询问的时候重启eclipse

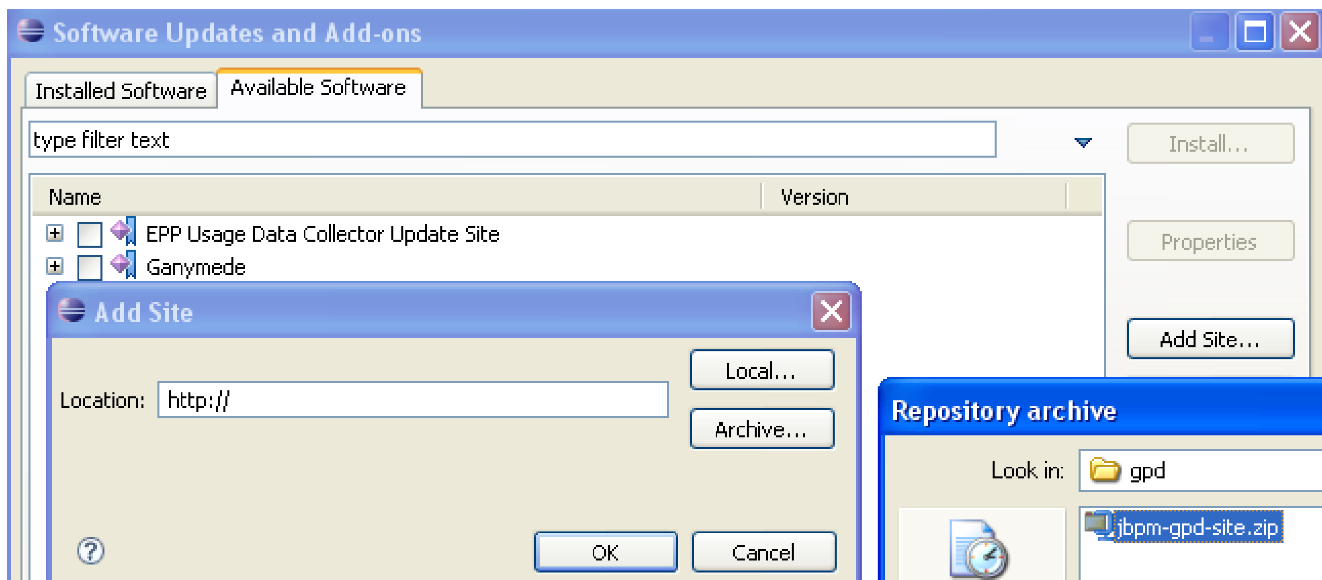

图 2.1. 添加设计器的更新站点

2.11.3. 配置jBPM运行时

- 点击 Window --> Preferences
- 选择 JBoss jBPM --> jBPM 4 --> Runtime Locations
- 点击 Add...
- 在 Add Location 对话框中，输入一个名字，比如 jbpm-4.0 然后点击 Search...
- 在 Browse For Folder 对话框中，选择你的jbpm根目录，然后点击 OK
- 点击 OK 在 Add Location 对话框中

图 2.2. 定义jBPM依赖库

2.11.4. 定义jBPM用户库

这一节演示如何在你的工作空间定义一个用户库， 用来放置jBPM的库文件。 如果你创建一个新工程， 只需要将用户库全部添加到build path下

- 点击窗口 --> 属性（Windows --> Preferences）

- 选择Java --> 创建路径 --> 用户类库（Java --> Build Path --> User Libraries）
- 点击新建（New）
- 类型名字jBPM Libraries
- 点击添加JARs（Add JARs...）
- 找到jBPM安装程序下的lib目录
- 选择lib下的所有jar文件并点击打开（Open）
- 选择jBPM Libraries作为入口
- 重新点击添加JARs（Add JARs）
- 在jBPM的安装程序的根目录下选择jbpm.jar文件
- 点击打开（Open）
- 在jbpm.jar下选择源码附件（Source attachment）作为入口
- 点击编辑（Edit）
- 在源码附件的配置（Source Attachment Configuration）对话框中，点击目录（External Folder...）
- 找到jBPM安装程序下的src目录
- 点击选择（Choose）
- 点击两次'确定'（Ok）会关闭所有对话框

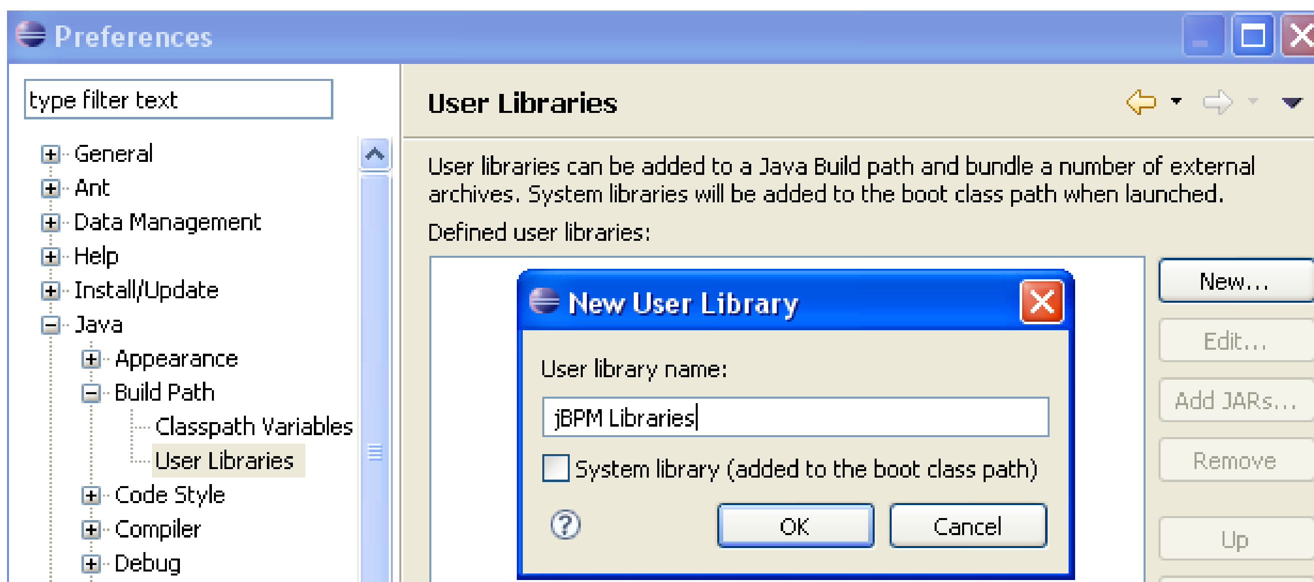

图 2.3. 定义jBPM类库

2.11.5. 在目录中添加jPDL4模式

如果你想直接编辑XML源码， 最好是在你的XML目录中指定一下模式（schema）， 这样当你在编辑流程源码的时候，可以更好的帮助你编写代码。

- 点击窗口 --> 属性（Windows --> Preferences）
- 选择XML --> 目录（XML --> CataLog）
- 点击添加（Add）
- 添加XML目录（Add XML Catalog Entry）的窗口打开
- 点击map-icon的图标下面的按钮并选择文件系统（File System）
- 在打开的对话框中， 选择jBPM安装目录下src文件夹中jpdl.xsd文件
- 点击打开（Open）并且关闭所有的对话框

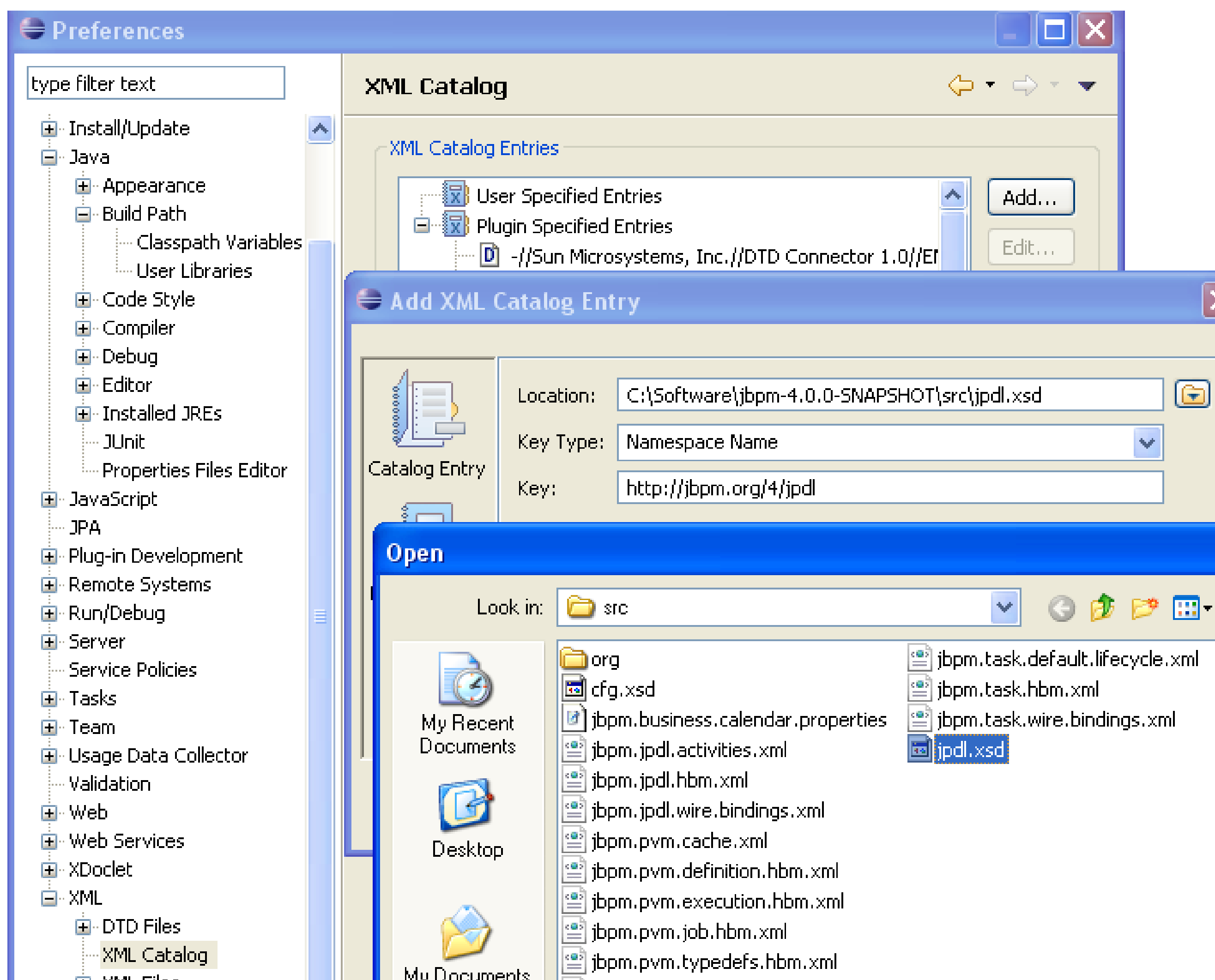

图 2.4. 在目录中添加jPDL4模式

2.11.6. 导入示例

这一节我们会在Eclipse的安装程序下 导入示例工程

- 选择文件 --> 导入（File --> Import）
- 选择正常 --> 工作区中已有的工程（General --> Existing Projects into Workspace）
- 点击下一步（Next）
- 点击浏览去选择一个根目录（Browse）
- 通向jBPM安装程序的根目录
- 点击好（Ok）
- 示例工程会自动找到并且选中
- 点击完成（Finish）

在配置了jBPM用户依赖库也导入了实例后， 所以的例子可以作为JUnit测试运行了。在一个测试上右击， 选择'Run As’ --> 'JUnit Test’。

设置完成，现在你可以开始享受这个最酷的Java流程技术。

2.11.7. 使用ant添加部分文件

你可以使用eclipse和ant整合来处理流程的发布。 我们会告诉你它是如何在例子里工作地。然后你可以把这些复制到你的项目中。 首先，打开ant视图。

- 选择Window --> Show View --> Other... --> Ant --> Ant
- 然后把构建文件build.xml拖拽到例子仙姑中，从包视图到ant视图。

第 3 章 流程设计器（GPD）

这一章我们讲述了怎样使用流程设计器， 在安装流程设计器和配置好例子之后， 你会看到jPDL流程文件都有一个对应的特殊图标， 在包视图的下面双击某一个这种图标文件，就会在流程设计器中打开一个jPDL流程文件。

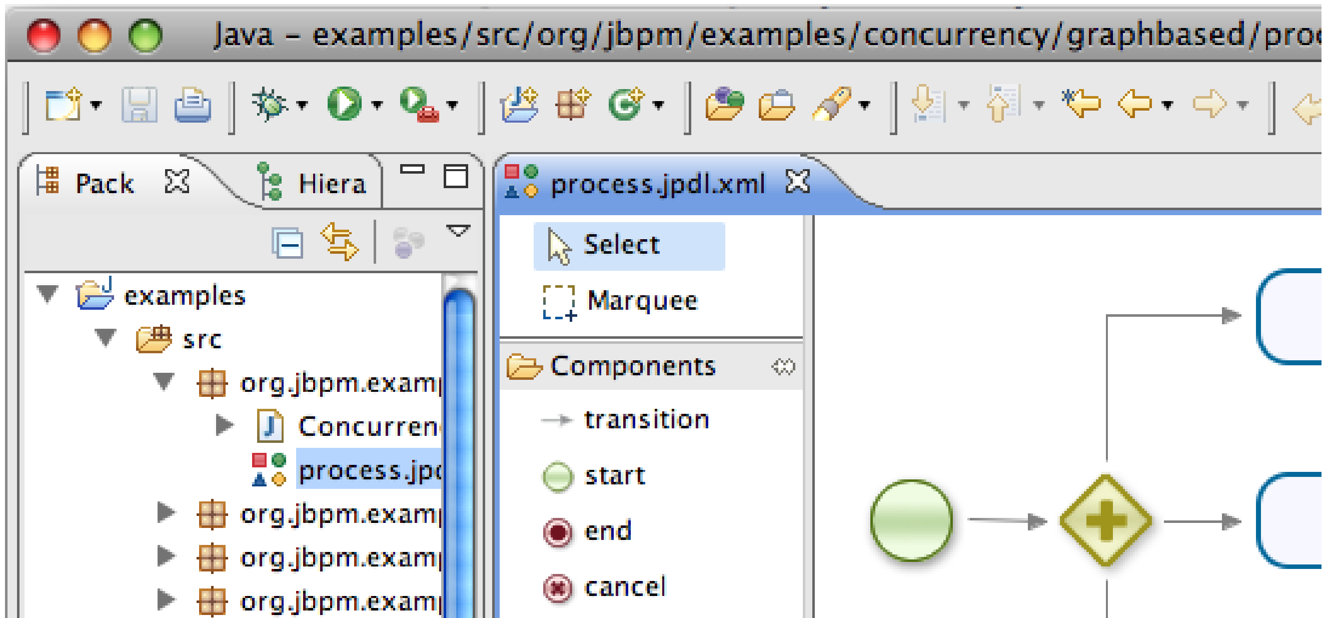

图 3.1. 流程设计器

3.1. 创建一个新的流程文件

Ctrl+N将打开向导选择器。

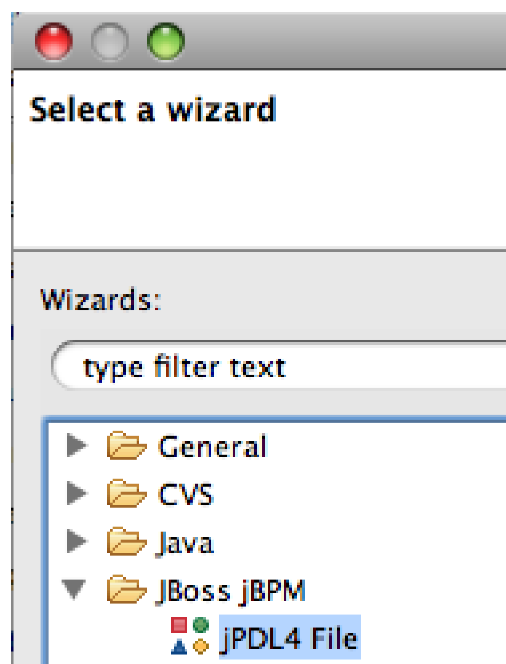

图 3.2. 选择向导对话框

选择 jBPM --> jPDL 4 文件（File）. 点击下一步（ Next >）. 然后新的jPDL 4文件（New jPDL 4 File），就会打开向导。

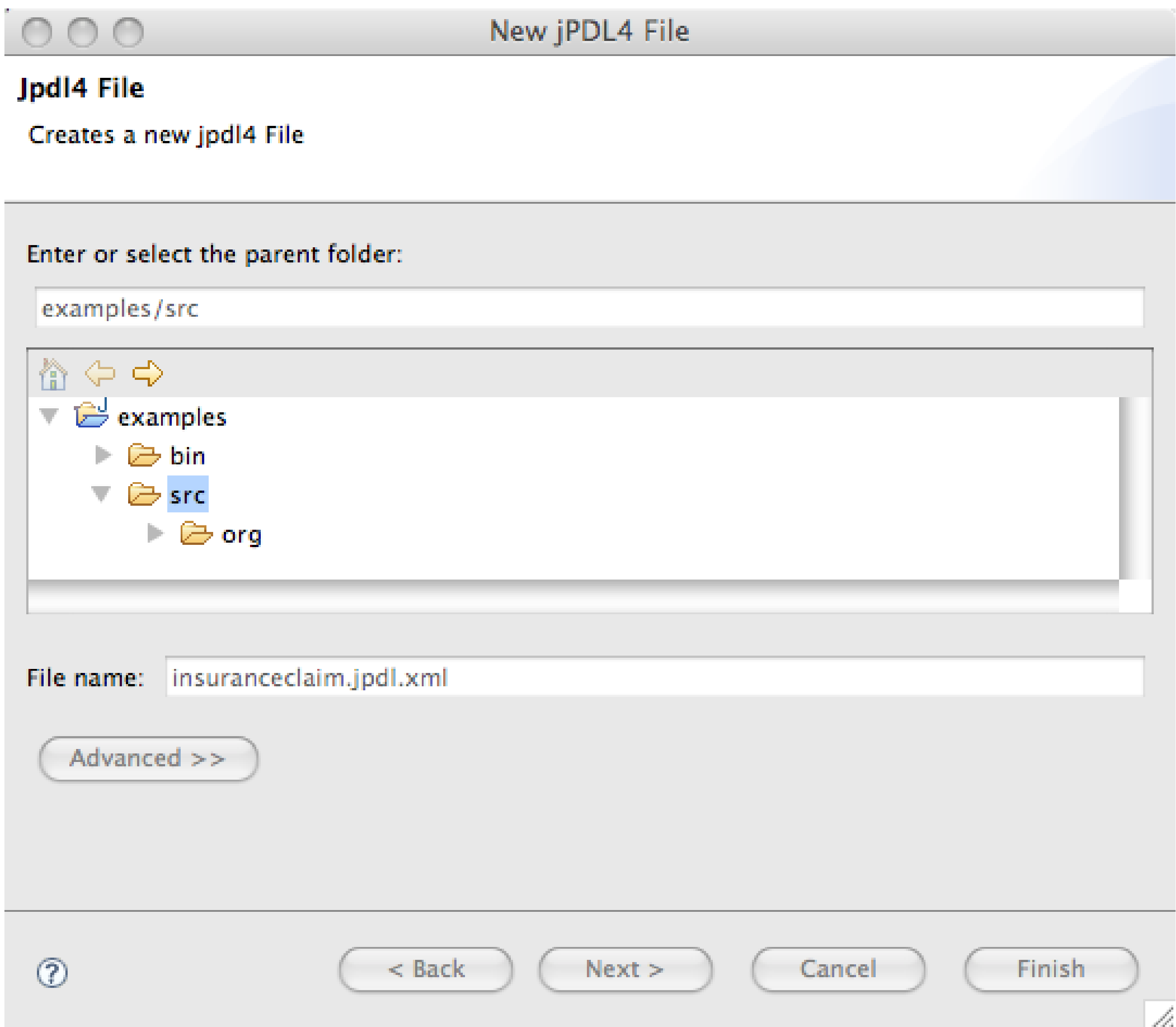

图 3.3. 创建一个新的流程对话框

选择上一级目录，输入一个文件名字并点击'完成'（Finish）， 你便创建了第一个jPDL流程文件。

3.2. 编辑流程文件的源码

GPD里有一个可以修改XML内容的'Source'标签。 可以在标签里直接进行编辑，当你切换到图形时，图形视图会反映出刚才进行的修改。

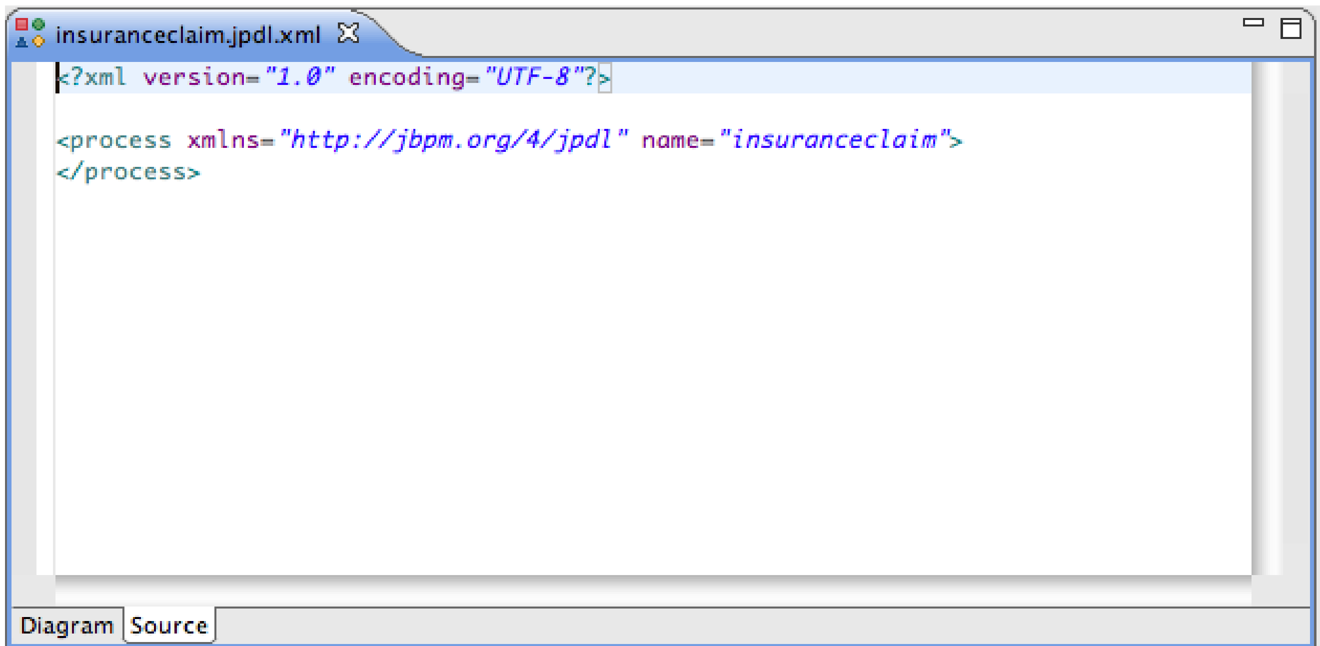

图 3.4. 使用source视图编辑jPDL

第 4 章 部署业务归档

业务归档是一系列文件的集合 分发在一个jar格式的文件里。。 业务归档中的文件可以使jPDL流程文件，表单， 流程图和其他流程资源。

4.1. 部署流程文件和流程资源

流程文件和流程资源必须 部署到流程资源库里 并保存到数据库中。

这儿有一个jBPM的ant任务来部署业务流程归档 (org.jbpm.pvm.internal.ant.JbpmDeployTask)。JbpmDeployTask可以部署 单独的流程文件和流程归档。 它们通过JDBC连接直接部署到数据库中。 所以在你部署流程之前 需要保证数据库正在运行。

创建和部署流程归档的例子 可以在发布包的examples目录下找到ant脚本（build.xml）。 让我们看一下相关部分。 首先，path用来声明包含jbpm.jar和它的所有依赖库。

```
<path id="jbpm.libs.incl.dependencies">
  <pathelement location="${jbpm.home}/examples/target/classes" />
  <fileset dir="${jbpm.home}">
    <include name="jbpm.jar" />
  </fileset>
  <fileset dir="${jbpm.home}/lib" />
</path>
```

你使用的数据库的JDBC驱动jar应该也包含在path中。 MySQL, PostgreSQL和HSQLDB的驱动都包含在发布包中。 但是oracle的驱动你必须从oracle网站上单独下载， 因为我们没有被允许重新分发这个文件。

当一个业务归档被发布时，jBPM扫描 业务归档中所有以.jpdl.xml结尾的文件。 所以那些文件会被当做jPDL流程解析，然后可以用在运行引擎中。 业务归档中所有其他的资源也会作为资源 保存在部署过程中，然后可以通过 RepositoryService类中的 InputStream getResourceAsStream(long deploymentDbid, String resourceName);访问。

为了创建一个业务归档， 可以使用jar任务。

```
<jar destfile="${jbpm.home}/examples/target/examples.bar">
      <fileset dir="${jbpm.home}/examples/src">
        <include name="**/*.jpdl.xml" />
        ...
      </fileset>
    </jar>
```

在jbpm-deploy被使用之前，它需要像这样进行声明：

```
<taskdef name="jbpm-deploy"
           classname="org.jbpm.pvm.internal.ant.JbpmDeployTask"
         classpathref="jbpm.libs.incl.dependencies" />
```

然后可以像这样使用ant任务

```

  <jbpm-deploy file="${jbpm.home}/examples/target/examples.bar" />
```

表 4.1. jbpm-deploy属性：

属性	类型	默认值	是否必填	描述
file	文件		可选	被部署的文件。.xml结尾的文件会被当做流程文件部署。 ar结尾，比如.bar或.jar，的文件 会被当做业务归档部署。
cfg	文件	jbpm.cfg.xml	可选	指向jbpm配置文件，它应该放在jbpm-deploy定义的classpath下。

表 4.2. jbpm-deploy元素

元素	数目	描述
fileset	0..*	被部署的文件，表示成一个简单的ant的fileset。 .xml结尾的文件会被当做流程文件部署。 ar结尾，比如.bar或.jar，的文件会被当做业务归档部署。

4.2. 部署java类

jBPM 4还没有像jBPM 3一样实现流程的类加载机制。

在你的流程文件中引用到的所有类， 无论是直接还是间接的，都需要放在 你的jBPM安装的classpath下。

在实例中，一个包含了所有类的examples.jar被创建了， 并把它放在了JBoss服务器配置的 lib目录下。

如果你希望使用流程类加载器，这样用户定义的类可以发布到业务架构 并从业务架构中读取，请投票支持 JIRA issue JBPM-2200 [https://jira.jboss.org/jira/browse/JBPM-2200]。 我们会按照投票的数量和用例来决定优先级， 你可以添加评论，来解释为何你需要这个功能。

第 5 章 服务

5.1. 流程定义，流程实例和执行

一个流程定义式对过程的步骤的描述。 比如，一个保险公司可以有一个贷款流程定义 描述公司如何处理贷款请求 的步骤的描述。

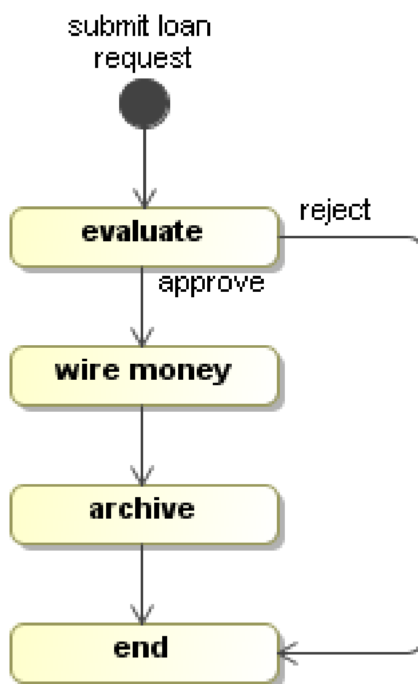

图 5.1. 贷款流程定义示例

流程实例代表着流程定义的特殊执行例子， 例如：上周五John Doe提出贷款买船， 代表着一个贷款流程定义的流程实例。

一个流程实例包括了所有运行阶段， 其中最典型的属性就是跟踪当前节点的指针。

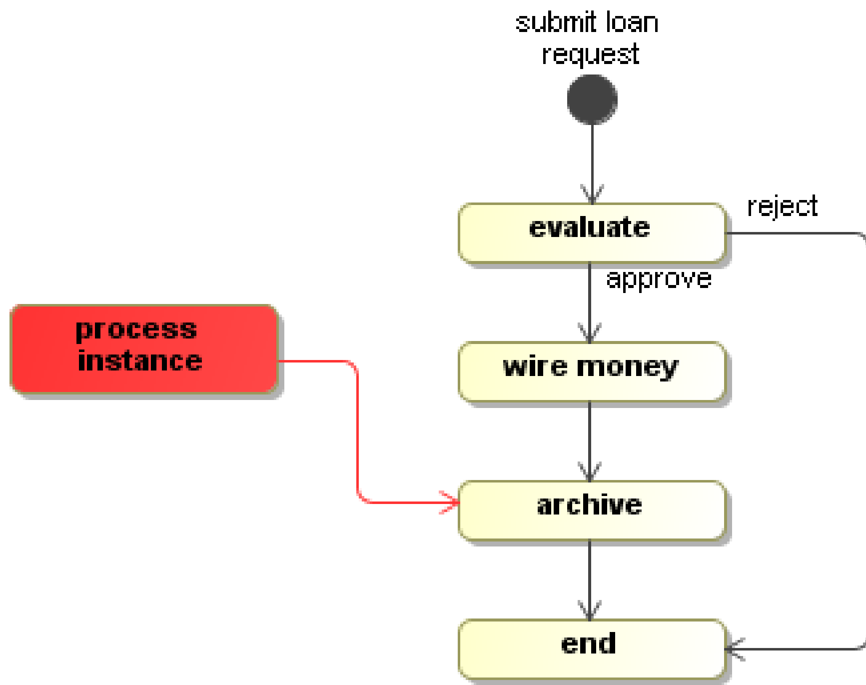

图 5.2. 贷款流程实例的例子

假设汇款和存档可以同时执行， 那么主流程实例就包含了2个 用来跟踪状态的子节点：

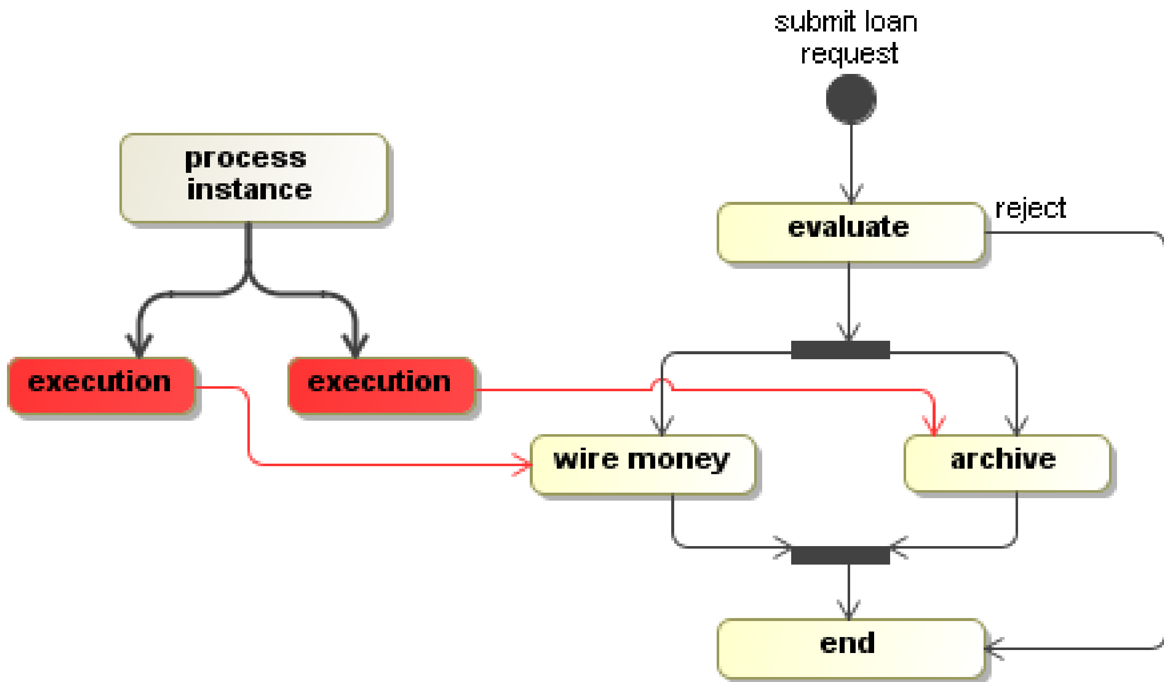

图 5.3. 贷款执行例子

一般情况下，一个流程实例是一个执行树的根节点， 当一个新的流程实例启动时，实际上流程实例就处于根节点的位置， 这时只有它的"子节点"才可以被激活。

使用树状结构的原因在于， 这一概念只有一条执行路径， 使用起来更简单。 业务API不需要了解流

程实例和执行之间功能的区别。 因此， API里只有一个执行类型来引用流程实例和执行。

5.2. ProcessEngine流程引擎

在jBPM内部通过各种服务相互作用。 服务接口可以从ProcessEngine中获得， 它是从Configuration构建的。

流程引擎是线程安全的，它可以保存在静态变量中， 甚至JNDI中或者其他重要位置。 在应用中，所有线程和请求都可以使用同一个流程引擎对象， 现在就告诉你怎么获得流程引擎。

这章中涉及到的代码和下一章中关于流程部署的代码， 都来自org.jbpm.examples.services.ServicesTest 例子。

```
ProcessEngine processEngine = new Configuration()
      .buildProcessEngine();
```

上面的代码演示了如何通过classpath根目录下 默认的配置文件jbpm.cfg.xml创建一个ProcessService。 如果你要指定其他位置的配置文件， 请使用setResource()方法：

```
ProcessEngine processEngine = new Configuration()
      .setResource("my-own-configuration-file.xml")
      .buildProcessEngine();
```

还有其他setXxxx()方法可以获得配置内容， 例如：从InputStream中、 从xml字符串中、从InputSource中、 从URL中或者从文件（File）中。

我们可以根据流程引擎得到 下面的服务：

```
RepositoryService repositoryService = processEngine.getRepositoryService();
ExecutionService executionService = processEngine.getExecutionService();
TaskService taskService = processEngine.getTaskService();
HistoryService historyService = processEngine.getHistoryService();
ManagementService managementService = processEngine.getManagementService();
```

在配置中定义的这些流程引擎（ProcessEngine）对象， 也可以根据类型processEngine.get(Class<T>) 或者根据名字processEngine.get(String)来获得。

5.3. Deploying a process部署流程

RepositoryService包含了用来管理发布资源的所有方法。 在第一个例子中，我们会使用RepositoryService从classpath中部署一个流程资源。

```
String deploymentid = repositoryService.createDeployment()
    .addResourceFromClasspath("org/jbpm/examples/services/Order.jpdl.xml")
    .deploy();
```

通过上面的addResourceFromClass方法， 流程定义XML的内容可以从文件， 网址，字符串，输入流或zip输入流中获得。

每次部署都包含了一系列资源。每个资源的内容都是一个字节数组。 jPDL流程文件都是以.jpdl.xml作

为扩展名的。 其他资源是任务表单和java类。

部署时要用到一系列资源， 默认会获得多种流程定义和其他的归档类型。 jPDL发布器会自动识别后缀名是.jpdl.xml 的流程文件。

在部署过程中，会把一个id分配给流程定义。 这个id的格式为{key}-{version}， key和version之间使用连字符连接。

如果没有提供key， 会在名字的基础自动生成。 生成的key会把所有不是字母和数字的字符替换成下划线。

同一个名称只能关联到一个key， 反之亦然。

如果没有为流程文件提供版本号， jBPM会自动为它分配一个版本号。 请特别注意那些已经部署了的名字 相同的流程文件的版本号。 它会比已经部署的同一个key的流程定义 里最大的版本号还大。 没有部署相同key的流程定义的版本号会分配为1。

在下面第1个例子里，我们只提供了流程的名字，没有提供其他信息：

```
<process name="Insurance claim">
...
</process>
```

假设这个流程是第一次部署， 下面就是它的属性：

表 5.1. 没有key值的属性流程

Property	Value	Source
name	Insurance claim	process xml
key	Insurance_claim	generated
version	1	generated
id	Insurance_claim-1	generated

第2个例子我们将演示如何通过设置流程的key 来获得更短的id。

```
<process name="Insurance claim" key="ICL">
...
</process>
```

这个流程定义的属性就会像下面这样：

表 5.2. 有key值属性的流程

Property	Value	Source
name	Insurance claim	process xml
key	ICL	process xml
version	1	generated

Property	Value	Source
id	ICL-1	generated

5.4. 卸载已发布的流程定义

TODO

5.5. 删除流程定义

删除一个流程定义会把它从数据库中删除。

```
repositoryService.deleteDeployment(deploymentId);
```

如果在发布中的流程定义还存在活动的流程实例， 这个方法就会抛出异常。

如果希望级联删除一个发布中流程定义的 所有流程实例， 可以使用deleteDeploymentCascade。

5.6. 启动一个新的流程实例

5.6.1. 最新的流程实例

下面是为流程定义启动一个新的流程实例的最简单也是 最常用的方法：

```
ProcessInstance processInstance = executionService.startProcessInstanceByKey("ICL");
```

上面service的方法会去查找 key为ICL的最新版本的流程定义， 然后在最新的流程定义里启动流程实例。

当insurance claim流程部署了一个新版本， startProcessInstanceByKey方法会自动切换到 最新部署的版本。

5.6.2. 指定流程版本

换句话说，你如果想根据特定的版本启动流程实例， 便可以使用流程定义的id启动流程实例。如下所示：

```
ProcessInstance processInstance =
    executionService.startProcessInstanceById("ICL-1");
```

5.6.3. 使用key

我们可以为新启动的流程实例分配一个key， key是用户执行的时候定义的。 一个流程定义里的所有key必须都是唯一的。 在企业的流程里很容易在领域模型里找到唯一的key。 例如：序列号或保险编号。

```
ProcessInstance processInstance =
    executionService.startProcessInstanceByKey("ICL", "CL92837");
```

key可以用来创建流程实例的id， 格式为{process-key}.{execution-id}。 所以上面的代码会创建一个id为 ICL.CL92837的流向 （execution）。

如果没有提供用户定义的key，数据库就会把主键作为key。 这样可以使用如下方式获得id：

```
ProcessInstance processInstance =
    executionService.startProcessInstanceByKey("ICL");
String pid = processInstance.getId();
```

我们推荐使用用户定义的key， 特别是在你的应用代码里，你可以得到有效的key。 根据用户定义的key，你可以直接使用流向的id， 而不用根据流程的变量进行查询。

5.6.4. 使用变量

当一个新的流程实例启动时就会提供一组对象参数。 将这些参数放在variables变量里， 然后可以在流程实例创建和启动时使用。

```
Map<String,Object> variables = new HashMap<String,Object>();
variables.put("customer", "John Doe");
variables.put("type", "Accident");
variables.put("amount", new Float(763.74));

ProcessInstance processInstance =
    executionService.startProcessInstanceByKey("ICL", variables);
```

5.7. 执行等待的流向

当使用一个state活动时，执行（或流程实例） 会在到达state的时候进行等待，直到一个signal（也叫外部触发器）出现。 signalExecution方法可以被用作这种情况。 执行通过一个执行id（字符串）来引用。

在一些情况下，到达state的执行会是流程实例本身。 但是这不是一直会出现的情况。在定时器和同步的情况， 流程是执行树形的根节点。所以我们必须确认你的signal 作用在正确的流程路径上。

获得正确的执行的比较好的方法是给state活动分配一个事件监听器， 像这样：

```
<state name="wait">
  <on event="start">
    <event-listener class="org.jbpm.examples.StartExternalWork" />
  </on>
  ...
</state>
```

在事件监听器StartExternalWork中，你可以执行那些需要额外完成的部分。 在这个时间监听器里，你也可以通过execution.getId()获得确切的流程id。 那个流程id，在额外的工作完成后， 你会需要它来提供给signal操作的：

```
executionService.signalExecutionById(executionId);
```

这里有一个可选的（不是太推荐的）方式，来获得流程id， 当流程到达state活动的时候。 只可能通过这种方式获得执行id，如果你知道哪个jBPM API调用了之后， 流程会进入state活动：

```
// assume that we know that after the next call
// the process instance will arrive in state external work

ProcessInstance processInstance =
  executionService.startProcessInstanceById(processDefinitionId);
// or ProcessInstance processInstance =
//   executionService.signalProcessInstanceById(executionId);

Execution execution = processInstance.findActiveExecutionIn("external work");
String executionId = execution.getId();
```

5.8. TaskService任务服务

TaskService的主要目的是提供对任务列表的访问途径。 例子代码会展示出如何为id为johndoe 的用户获得任务列表

```
List<Task> taskList = taskService.findPersonalTasks("johndoe");
```

一般来说，任务会对应一个表单，然后显示在一些用户接口中。 表单需要可以读写与任务相关的数据。

```
long taskId = task.getId();

Set<String> variableNames = taskService.getVariableNames(taskId);
variables = taskService.getVariables(taskId, variableNames);

variables = new HashMap<String, Object>();
variables.put("category", "small");
variables.put("lires", 923874893);
taskService.setVariables(taskId, variables);
```

完成任务

```
taskService.completeTask(taskId);
```

任务可以拥有一批候选人。候选人可以是用户也可以是用户组。 用户可以接收自己是候选人的任务。接收任务的意思是用户会被设置为被分配给任务的人。 在那之后，其他用户就不能接收这个任务了。

人们不应该在任务做工作， 除非他们被分配到这个任务上。 用户界面应该显示表单，并允许用户完成任务， 如果他们被分配到这个任务上。对于有了候选人，但是还没有分配的任务， 唯一应该暴露的操作就是“接收任务”。

更多的任务见第 6.2.6 节 “task” 。

5.9. HistoryService历史服务

在流程实例执行的过程中，会不断触发事件。 从那些事件中，运行和完成流程的历史信息会被收集到历史表中。 HistoryService提供了 对那些信息的访问功能。

如果想查找某一特定流程定义的所有流程实例， 可以像这样操作：

```
List<HistoryProcessInstance> historyProcessInstances = historyService
  .createHistoryProcessInstanceQuery()
  .processDefinitionId("ICL-1")
  .orderAsc(HistoryProcessInstanceQuery.PROPERTY_STARTTIME)
  .list();
```

单独的活动流程也可以作为HistoryActivityInstance 保存到历史信息中。

```
List<HistoryActivityInstance> histActInsts = historyService
    .createHistoryActivityInstanceQuery()
    .processDefinitionId("ICL-1")
    .activityName("a")
    .list();
```

也可以使用简易方法avgDurationPerActivity和 choiceDistribution。 可以通过javadocs获得这些方法的更多信息。

5.10. ManagementService管理服务

管理服务通常用来管理job。可以通过javadocs获得这些方法的更多信息。 这个功能也是通过控制台暴露出来。

第 6 章 jPDL

这章将会解释用来描述流程定义的 jPDL文件格式。schema文档也可以作为 这些信息的快速参考。

下面是一个jPDL流程文件的例子：

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

<process name="Purchase order" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="Verify supplier" />
  </start>

  <state name="Verify supplier">
    <transition name="Supplier ok" to="Check supplier data" />
    <transition name="Supplier not ok" to="Error" />
  </state>

  <decision name="Check supplier data">
    <transition name="nok" to="Error" />
    <transition name="ok" to="Completed" />
  </decision>

  <end name="Completed" />

  <end name="Error" />

</process>
```

6.1. process流程处理

顶级元素（element）是流程处理定义。

表 6.1. process流程处理的属性

属性	类型	默认值	是否必须	描述
name名称	文本		必须	在与用户交互时，作为流程名字显示的一个名字或是标签。
key键	字母或数字，下划线	如果省略，key中的非字母和非数字的字符会被替换为下划线。	可选（optional）	用来辨别不同的流程定义。 拥有同一个key的流程会有多个版本。 对于所有已发布的流程版本，key-name这种组合都必须是完全一样的。
version版本	整型	比已部署的key相	可选	流程的版本号

属性	类型	默认值	是否必须	描述
		同的流程版本号高1， 如果还没有与之相同的key的流程被部署，那么版本就从1开始。		

表 6.2. process流程的元素

元素	个数	描述
description描述	0个或1个	描述文本
activities活动	至少1个	流程中会有很多活动， 至少要有1个是启动的活动。

6.2. 控制流程Activities活动

6.2.1. start启动

说明一个流程的实例从哪里开始。 在一个流程里必须有一个开始节点。 一个流程必须至少拥有一个开始节点。 开始节点必须有一个向外的流向，这个流向会在流程启动的时候执行。

已知的限制：直到现在， 一个流程处理只能有一个启动节点（start）。

表 6.3. start启动的属性

属性	类型	默认值	是否必须	描述
name名称	文本		可选	活动的名字，在启动活动没有内部的转移（transition）时， name名称是可选的。

表 6.4. start启动的元素

元素	个数	描述
transition转移	1	向外的转移

6.2.2. State状态节点

一个等待状态节点。 流程处理的流向会在外部触发器调用提供的API之前一直等待。 状态节点和其他

的活动不一样， 它没有其他任何属性或元素。

6.2.2.1. 序列状态节点

让我们看一个用序列连接状态 和转移的例子。

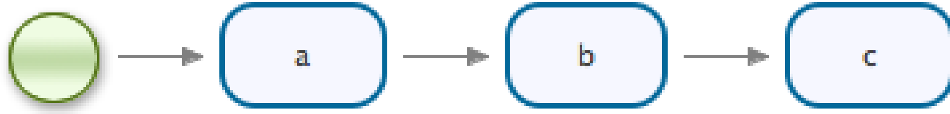

图 6.1. 序列状态节点

```
<process name="StateSequence" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="a" />
  </start>

  <state name="a">
    <transition to="b" />
  </state>

  <state name="b">
    <transition to="c" />
  </state>

  <state name="c" />

</process>
```

下列代码将启动一个流向：

```
ProcessInstance processInstance =
    executionService.startProcessInstanceByKey("StateSequence");
```

创建的流程处理实例会停留在状态节点a的位置， 使用signalExecution方法就会触发 一个外部触发器。

```
Execution executionInA = processInstance.findActiveExecutionIn("a");
assertNotNull(executionInA);

processInstance = executionService.signalExecutionById(executionInA.getId());
Execution executionInB = processInstance.findActiveExecutionIn("b");
assertNotNull(executionInB);

processInstance = executionService.signalExecutionById(executionInB.getId());
Execution executionInC = processInstance.findActiveExecutionIn("c");
assertNotNull(executionInC);
```

6.2.2.2. 可选择的状态节点

在第2个状态节点的例子里， 我们将演示如何使用状态节点实现 路径的选择。

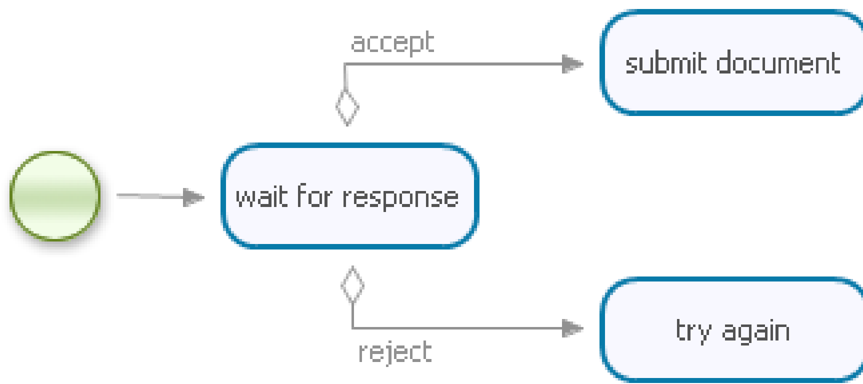

图 6.2. 状态节点中的选择

```
<process name="StateChoice" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="wait for response" />
  </start>

  <state name="wait for response">
    <transition name="accept" to="submit document" />
    <transition name="reject" to="try again" />
  </state>

  <state name="submit document" />

  <state name="try again" />

</process>
```

让我们在这个流程处理定义里启动一个新的流程实例。

```
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("StateChoice");
```

现在，流向到达wait for response状态节点了。 流向会一直等待到外部触发器的出现。 这里的状态节点拥有多个向外的转移， 外部触发器将为向外的转移提供不同的信号名（signalName）， 下面我们将提供accept信号名（signalName）：

```
String executionId = processInstance
    .findActiveExecutionIn("wait for response")
    .getId();

processInstance = executionService.signalExecutionById(executionId, "accept");

assertTrue(processInstance.isActive("submit document"));
```

流向会沿着名字是accept的向外的转移继续进行。 同样，当使用reject作为参数触发signalExecutionXxx方法时。流 向会沿着名字是reject的向外的转移 继续进行。

6.2.3. decision决定节点

在多个选择中选择一条路径。也可以当做是一个决定。 一个决定活动拥有很多个向外的转移。当一个流向到达一个决定活动时， 会自动执行并决定交给哪个向外的转移。

一个决定节点应该配置成下面三个方式之一。

6.2.3.1. decision决定条件

decision中会运行并判断每一个transition里的判断条件。 当遇到一个嵌套条件是true或者没有设置判断条件的转移， 那么转移就会被运行。

表 6.5. exclusive.transition.condition 属性

属性	类型	默认值	是否必须?	描述
expr	expression		required必须	将被运行的 指定脚本
lang	expression language	从脚本引擎配置里得到的默认代表性语言（default-expression-language）	可选	指定expr中执行的脚本语言的种类

例子：

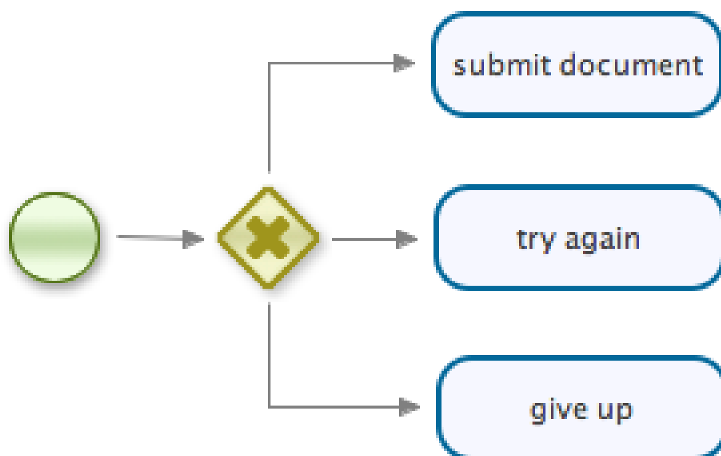

图 6.3. 流程处理的决定条件例子

```
<process name="DecisionConditions" >

  <start>
    <transition to="evaluate document" />
  </start>

  <decision name="evaluate document">
```

```
    <transition to="submit document">
      <condition expr="#{content=="good"}" />
    </transition>
    <transition to="try again">
      <condition expr="#{content=="not so good"}" />
    </transition>
    <transition to="give up" />
  </decision>

  <state name="submit document" />

  <state name="try again" />

  <state name="give up" />

</process>
```

在使用good content启动一个流程之后

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("content", "good");
ProcessInstance processInstance =
    executionService.startProcessInstanceByKey("DecisionConditions", variables);
```

submit document活动会变成活动的

```
assertTrue(processInstance.isActive("submit document"));
```

参考实例中的单元测试，了解更多的场景。

6.2.3.2. decision expression唯一性表达式

decision表达式返回类型为字符串的 向外转移的名字。

表 6.6. 决定属性

属性	类型	默认值	是否必须?	描述
expr	expression		required必须	将被运行的指定脚本
lang	expression language	从脚本引擎配置里得到的默认指定的脚本语言（default-expression-language）	可选	指定expr中执行的脚本语言的 种类。

例子：

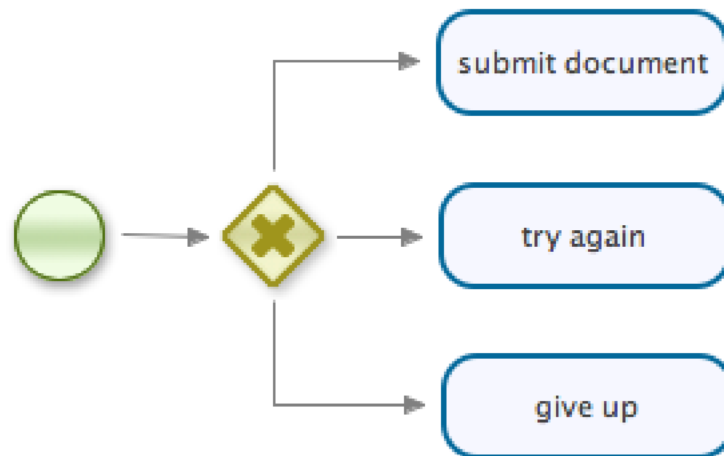

图 6.4. 流程处理的决定表达式例子

```
<process name="DecisionExpression" xmlns="http://jbpm.org/4.0/jpdl">

  <start >
    <transition to="evaluate document"/>
  </start>

  <decision name="evaluate document" expr="#{content}" >
    <transition name="good" to="submit document"  />
    <transition name="bad"  to="try again"  />
    <transition name="ugly" to="give up"  />
  </decision>

  <state name="submit document"  />
  <state name="try again"  />
  <state name="give up"  />

</process>
```

当你使用good content启动一个新的流程实例，代码如下：

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("content", "good");
ProcessInstance processInstance =
    executionService.startProcessInstanceByKey("DecisionExpression", variables);
```

然后新流程会到达submit document活动。

参考实例中的单元测试，获得其他场景。

6.2.3.3. Decision handler决定处理器

唯一性管理是继承了DecisionHandler接口的java类。 决定处理器负责选择 向外转移。

```
public interface DecisionHandler {
   String decide(OpenExecution execution);
}
```

这个handler被列为decision的子元素。

表 6.7. decision.handler 属性

属性	类型	默认值	是否必须?	描述
class	classname		required必须	handler的完整类名

下面是一个决定使用DecisionHandler的流程处理例子：

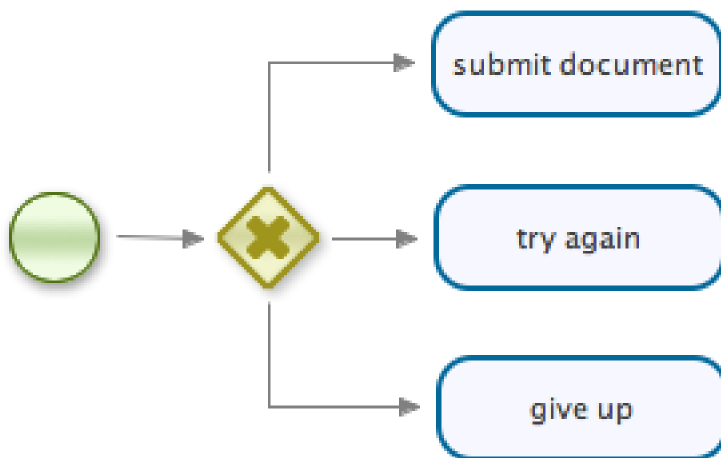

图 6.5. 流程处理的exclusive管理例子

```
<process name="DecisionHandler">

  <start>
    <transition to="evaluate document" />
  </start>

  <decision name="evaluate document">
    <handler class="org.jbpm.examples.decision.handler.ContentEvaluation" />
    <transition name="good" to="submit document" />
    <transition name="bad" to="try again" />
    <transition name="ugly" to="give up" />
  </decision>

  <state name="submit document" />

  <state name="try again" />

  <state name="give up" />

</process>
```

下面是ContentEvalation类：

```
public class ContentEvaluation implements DecisionHandler {

  public String decide(OpenExecution execution) {
    String content = (String) execution.getVariable("content");
    if (content.equals("you're great")) {
      return "good";
    }
    if (content.equals("you gotta improve")) {
      return "bad";
    }
    return "ugly";
  }
}
```

当你启动流程处理实例， 并为变量content提供值you're great时， ContentEvalation就会返回字符串good， 流程处理实例便会到达Submit document活动。

6.2.4. concurrency并发

使用fork和join活动， 可以模拟流向（executions）的汇合。

表 6.8. join属性：

属性	类型	默认值	是否必须?	描述
multiplicity	integer	传入转移的数目	可选	在这个join活动之前需要到达的执行的数目， 然后一个执行 会沿着join的单独的外向转移向外执行。
lockmode	{none, read, upgrade, upgrade_nowait, write}	upgrade	optional	hibernate的锁定模式，应用在上级执行， 来防止两个还没到达join的同步事务看到对方， 这会导致死锁。

例子：

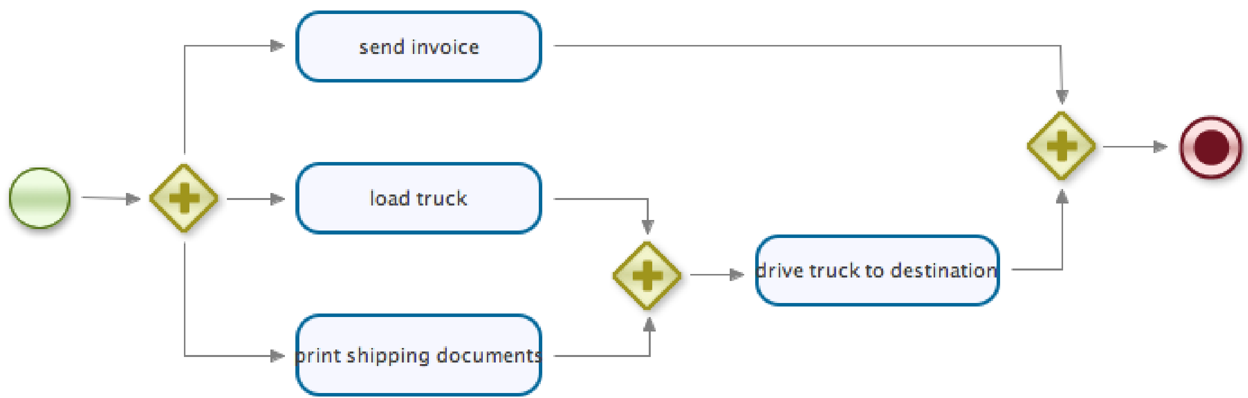

图 6.6. 流程处理的并发例子

```
<process name="ConcurrencyGraphBased" xmlns="http://jbpm.org/4.0/jpdl">

   <start>
      <transition to="fork"/>
   </start>

   <fork name="fork">
      <transition to="send invoice" />
      <transition to="load truck"/>
      <transition to="print shipping documents" />
   </fork>

   <state name="send invoice" >
      <transition to="final join" />
   </state>

   <state name="load truck" >
      <transition to="shipping join" />
   </state>

   <state name="print shipping documents">
      <transition to="shipping join" />
   </state>

   <join name="shipping join" >
      <transition to="drive truck to destination" />
   </join>

   <state name="drive truck to destination" >
      <transition to="final join" />
   </state>

   <join name="final join" >
      <transition to="end"/>
   </join>

   <end name="end" />

</process>
```

6.2.5. end结束

结束流向

6.2.5.1. end process instance结束流程处理实例

默认情况下，结束活动会终结已完成流程处理实例。 因此在流程处理实例中， 仍然在活动的多个并发（concurrent）流向（concurrent） 也会结束。

图 6.7. 结束活动

```
<process name="EndProcessInstance" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="end" />
  </start>

  <end name="end" />

</process>
```

新的流程处理实例一创建便会直接结束。

6.2.5.2. end execution结束流向

只有流向到达结束（end）活动时会结束流程处理实例， 并且其他并发流向会放弃活动。 我们可以设置属性ends="execution" 来达到这种状况。

表 6.9. end execution属性

属性	类型	默认值	是否必须	描述
ends	{processinstance|execution}	processinstance	optional可选	流向路径到达end活动 整个流程处理实例就会结束。

6.2.5.3. end multiple多个结束

一个流程处理可以有多个end events， 这样就很容易显示出流程处理实例的不同结果。示例：

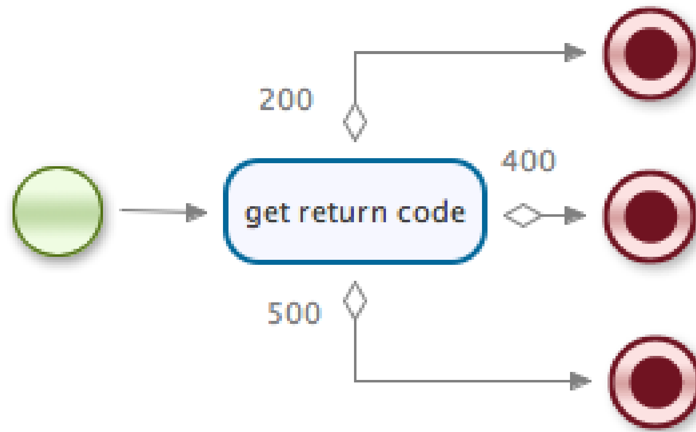

图 6.8. 多个end events

```
<process name="EndMultiple" xmlns="http://;jbpm.org/4/jpdl">

  <start>
    <transition to="get return code" />
  <start>

  <state name="get return code">
    <transition name="200" to="ok"/>
    <transition name="400" to="bad request"/>
    <transition name="500" to="internal server error"/>
  </state>

  <end name="ok"/>
  <end name="bad request"/>
  <end name="internal server error"/>

</process>
```

如果你启动一个流向并使用下面的代码将它执行到get return code等待状态， 流向便会以bad request的end 活动（event）结束

```
ProcessInstance processInstance = executionService.startProcessInstanceByKey("EndMultiple");
String pid = processInstance.getId();
processInstance = executionService.signalExecutionById(pid, "400");
```

同样地，使用值为200或者500就会让流向（execution） 分别以ok或者internal server error的end events结束。

6.2.5.4. end State结束状态

流向（execution）可以以不同的状态结束。可以用其他的方式列出流程处理实例的结果。 可以用end event的状态属性或者end-cancel 和end-error表示。

表 6.10. end execution 属性

属性	类型	默认值	是否必须	描述
state	String		可选	状态分配给流向

参考下面流程的例子。

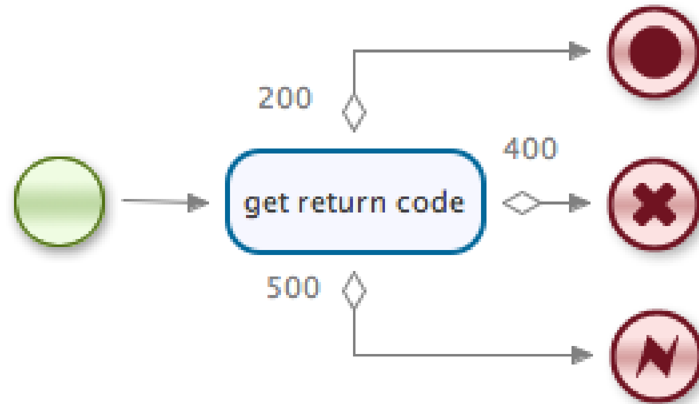

图 6.9. 不同的结束状态

```
<process name="EndState" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
     <transition to="get return code"/>
  </start>

  <state name="get return code">
    <transition name="200" to="ok"/>
    <transition name="400" to="bad request" />
    <transition name="500" to="internal server error"/>
  </state>

  <end name="ok" state="completed"/>
  <end-cancel name="bad request"/>
  <end-error name="internal server error"/>

</process>
```

这时，如果我们启动一个流向并使用下面的代码将流向执行到get return code等待状态， 流向会以取消状态（cancel state）结束。

```
TODO (复制代码片段)
```

和上面一样，使用值为200或500会让流向 分别以comleted或者error states结束。

6.2.6. task

在任务组件中，为一个人创建一个任务。

6.2.6.1. 任务分配者

一个简单的任务会被分配给一个指定的用户

表 6.11. 任务属性：

属性	类型	默认值	是否必填	描述
assignee	表达式		可选	用户id引用的用户负责完成任务。

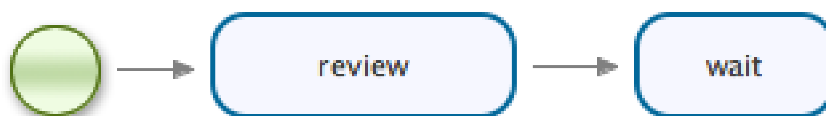

图 6.10. 任务分配者示例流程

```
<process name="TaskAssignee">

  <start>
    <transition to="review" />
  </start>

  <task name="review"
        assignee="#{order.owner}">

     <transition to="wait" />
  </task>

  <state name="wait" />

</process>
```

这个流程演示了任务分配的两个方面。第一， assignee用来指示用户， 负责完成任务的人。分配人是一个任务中的字符串属性 引用一个用户。

第二，这个属性默认会当做表达式来执行。 在这里任务被分配给#{order.owner}。 这意味着首先使用order这个名字查找一个对象。 其中一个查找对象的地方是这个任务对应的流程变量。 然后getOwner()方法会用来 获得用户id， 引用的用户负责完成这个任务。

这就是我们例子中使用到得Order类：

```
public class Order implements Serializable {

  String owner;

  public Order(String owner) {
    this.owner = owner;
```

```
  }

  public String getOwner() {
    return owner;
  }

  public void setOwner(String owner) {
    this.owner = owner;
  }
}
```

当一个新流程实例会被创建， 把order作为一个流程变量分配给它。

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("order", new Order("johndoe"));
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("TaskAssignee", variables);
```

然后johndoe的任务列表可以像下面这样获得。

```
List<Task> taskList = taskService.findPersonalTasks("johndoe");
```

注意也可以使用纯文本， assignee="johndoe"。 在这里，任务会被分配给johndoe。

6.2.6.2. task候选人

任务可能被分配给一组用户。 其中的一个用户应该接受这个任务并完成它。

表 6.12. 任务属性：

属性	类型	默认值	是否必填	描述
candidate-groups	表达式		可选	一个使用逗号分隔的组id列表。 所有组内的用户将会成为这个任务的候选人。
candidate-users	表达式		可选	一个使用逗号分隔的用户id列表。 所有的用户将会成为这个任务的候选人。

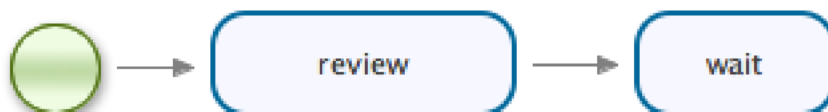

图 6.11. 任务候选人示例流程

这是一个使用任务候选人的示例流程：

```
<process name="TaskCandidates">

  <start>
    <transition to="review" />
  </start>

  <task name="review"
        candidate-groups="sales-dept">

     <transition to="wait" />
  </task>

  <state name="wait"/>

</process>
```

在启动之后，一个任务会被创建。这个任务不显示在任何人的个人任务列表中。 下面的任务列表会是空的。

```
taskService.getAssignedTasks("johndoe");
taskService.getAssignedTasks("joesmoe");
```

但是任务会显示在所有sales-dept组成员的 分组任务列表中。

在我们的例子中，sales-dept有两个成员：johndoe和joesmoe

```
identityService.createGroup("sales-dept");

identityService.createUser("johndoe", "johndoe", "John", "Doe");
identityService.createMembership("johndoe", "sales-dept");

identityService.createUser("joesmoe", "joesmoe", "Joe", "Smoe");
identityService.createMembership("joesmoe", "sales-dept");
```

所以在流程创建后， 任务会出现在johndoe和joesmoe用户的分组任务列表中。

```
taskService.findGroupTasks("johndoe");
taskService.findGroupTasks("joesmoe");
```

候选人必须接受一个任务，在他们处理它之前。 这会表现为两个候选人在同一个任务上开始工作。 分组任务列表中，用户接口必须只接受对这些任务的“接受”操作。

```
taskService.takeTask(task.getDbid(), "johndoe");
```

当一个用户接受了一个任务，这个任务的分配人就会变成当前用户。 任务会从所有候选人的分组任务列表中消失， 它会出现在用户的已分配列表中。

用户只允许工作在他们的个人任务列表上。 这应该由用户接口控制。

简单的，candidate-users属性 可以用来处理用逗号分隔的一系列用户id。 candidate-users属性可以和其他分配选项结合使用。

6.2.6.3. 任务分配处理器

一个AssignmentHandler可以通过编程方式来计算 一个任务的分配人和候选人。

```
public interface AssignmentHandler extends Serializable {

  /** sets the actorId and candidates for the given assignable. */
  void assign(Assignable assignable, OpenExecution execution) throws Exception;
}
```

Assignable是任务和泳道的通用接口。 所以任务分配处理器可以使用在任务， 也可以用在泳道中（参考后面的内容）。

assignment-handler是任务元素的一个子元素。 它指定用户代码对象。所以assignment-handler的属性和元素 都来自第 6.7 节 “用户代码”

让我们看一下任务分配的例子流程。

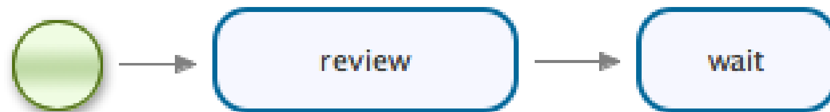

图 6.12. 任务分配处理器的示例流程

```
<process name="TaskAssignmentHandler" xmlns="http://jbpm.org/4.0/jpdl">

  <start g="20,20,48,48">
    <transition to="review" />
  </start>

  <task name="review" g="96,16,127,52">
    <assignment-handler class="org.jbpm.examples.task.assignmenthandler.AssignTask">
      <field name="assignee">
        <string value="johndoe" />
      </field>
    </assignment-handler>
    <transition to="wait" />
  </task>

  <state name="wait" g="255,16,88,52" />

</process>
```

引用的类AssignTask看起来像这样：

```
public class AssignTask implements AssignmentHandler {

  String assignee;

  public void assign(Assignable assignable, OpenExecution execution) {
    assignable.setAssignee(assignee);
  }
```

```
}
```

请注意，默认AssignmentHandler实现可以使用使用流程变量 任何其他Java API可以访问资源，像你的应用数据库来计算 分配人和候选人用户和组。

启动一个TaskAssignmentHandler的新流程实例 会立即让新流程实例运行到任务节点。 一个新review任务被创建，在这个时候 AssignTask的分配处理器被调用。这将设置johndoe为分配人。 所以John Doe将在他自己的任务列表中找到这个任务。

6.2.6.4. 任务泳道

一个流程中的多任务应该被分配给同一个用户或换选人。 一个流程中的多任务可以分配给一个单独的泳道。 流程实例将记得换选人和用户，在泳道中执行的第一个任务。 任务序列在同一个泳道中将被分配给 这些用户和候选人。

一个泳道也可以当做一个流程规则。 在一些情况下， 这可能与身份组件中的权限角色相同。 但是实际上它们并不是同一个东西。

表 6.13. 任务属性：

属性	类型	默认值	是否必填	描述
swimlane	泳道(字符串)		可选	引用一个定义在流程中的泳道

泳道可以被声明在流程元素中：

表 6.14. 泳道属性：

属性	类型	默认值	是否必填	描述
name	泳道(字符串)		必填	泳道名称。 这个名称将被任务泳道属性中引用。
assignee	表达式		可选	用户id引用的用户负责完成这个任务。
candidate-groups	表达式		可选	一个使用逗号分隔的组id列表。 所有组中的人将作为这个任务的这个泳道中的 候选人。
candidate-users	表达式		可选	一个使用逗号分隔的用户id列表。所有的用户将作为这个任务的这个泳道中的 候选人。

图 6.13. 任务泳道示例流程

任务泳道示例是下面这个流程文件：

```
<process name="TaskSwimlane" xmlns="http://jbpm.org/4.0/jpdl">

  <swimlane name="sales representative"
            candidate-groups="sales-dept" />

  <start>
    <transition to="enter order data" />
  </start>

  <task name="enter order data"
        swimlane="sales representative">

    <transition to="calculate quote"/>
  </task>

  <task
      name="calculate quote"
      swimlane="sales representative">
  </task>

</process>
```

在这个例子中，我们在身份组件中 创建了下面的信息：

```
identityService.createGroup("sales-dept");

identityService.createUser("johndoe", "johndoe", "John", "Doe");
identityService.createMembership("johndoe", "sales-dept");
```

在启动一个新流程实例后，用户johndoe将成为 enter order data的一个候选人。还是像上一个流程候选人例子一样， John Doe可以像这样接收任务：

```
taskService.takeTask(taskDbid, "johndoe");
```

接收这个任务将让johndoe成为任务的分配人。 直到任务与泳道sales representative关联， 分配人johndoe也会关联到泳道中 作为分配人。

接下来，John Doe可以像下面这样完成任务：

```
taskService.completeTask(taskDbid);
```

完成任务会将流程执行到下一个任务， 下一个任务是calculate quote。 这个任务也关联着泳道。因此， 任务会分配给johndoe。 初始化分配的候选人用户和候选人组也会从泳道复制给任务。 这里所指的

用户johndoe 会释放任务，返回它给其他候选人。

6.2.6.5. 任务变量

任务可以读取，更新流程变量。 稍后任务可以选择定义任务本地流程变量。 任务变量是任务表单的一个很重要的部分。 任务表单显示来自任务和流程实例的数据。 然后从用户一侧录入的数据会转换成设置的任务变量。

获得任务变量就像这样：

```
List<Task> taskList = taskService.findPersonalTasks("johndoe");

Task task = taskList.get(0);
long taskDbid = task.getDbid();

Set<String> variableNames = taskService.getVariableNames(taskDbid);

Map<String, Object> variables = taskService.getVariables(taskDbid, variableNames);
```

设置任务变量就像这样：

```
variables = new HashMap<String, Object>();
variables.put("category", "small");
variables.put("lires", 923874893);

taskService.setVariables(taskDbid, variables);
```

6.2.6.6. 在任务中支持e-mail

可以为分配人提供一个提醒， 当一个任务添加到他们的列表时，以及在特定的时间间隔进行提醒。 每个email信息都是根据一个模板生成出来的。模板可以在内部指定， 或者在配置文件中的process-engine-context部分指定。

表 6.15. task元素

元素	数目	描述
notification	0..1	让一个任务被分配的时候发送一个提醒消息。 如果没有引用模板，也没有提供内部的模板，mail会使用task-notification名字的模板。
reminder	0..1	根据指定的时间间隔发送提醒信息。 如果没有引用模板，也没有提供内部模板， mail会使用task-reminder名字的模板。

表 6.16. notification属性

属性	类型	默认值	是否必填	描述
continue	{sync | async | exclusive}	sync	可选	指定在发送提醒邮件后， 是不是产生一个异步执行。

表 6.17. reminder属性：

属性	类型	默认值	是否必填	描述
duedate	持续时间（纯字符串或包含表达式）		必填	在reminder email发送前的延迟时间。
repeat	持续时间（纯字符串或包含表达式）		可选	在一个序列reminder email发送后延迟的时间
continue	{sync | async | exclusive}	sync	可选	指定在发送提醒邮件后， 是不是产生一个异步执行。

这里有一个基本的例子，可以获得默认的模板。

```
<task name="review"
      assignee="#{order.owner}"
     <notification/>
     <reminder duedate="2 days" repeat="1 day"/>
</task>
```

6.2.7. sub-process子流程

创建一个子流程实例然后等待直到它完成。 当子流程实例完成，子流程中的流向就会 继续。

表 6.18. 子流程属性：

属性	类型	默认值	是否必填	描述
sub-process-id	字符串		这个或sub-process-key是必填的	根据id获得子流程。 这意味着引用了一个流程定义的指定版本。
sub-process-key	字符串		这个或sub-process-id是必须的	根据key获得子流程。 这意味着引用了一个指定了key的流程定义的最新版本。 流程定义的最新版本会在每次活动执行的

属性	类型	默认值	是否必填	描述
				时候进行查找。
outcome	表达式		当指定outcome-value时必填	当子流程结束的时候执行表达式。值用来映射向外的流向。添加outcome-value元素到sub-process活动的外出流向中。

表 6.19. sub-process元素：

元素	多重	描述
parameter-in	0..*	声明一个变量，传递给子流程实例， 在创建它时。
parameter-out	0..*	定义一个变量，在子流程结束时设置到上级执行中。

表 6.20. parameter-in属性：

属性	类型	默认值	是否必填	描述
subvar	字符串		必填	已经赋值的子流程变量的名称。
var	字符串		'var'或'expr'其中之一必须指定值	上级流程环境中的变量名。
expr	字符串		'var'或'expr'其中之一必须指定值	这个表达式将会在super流程环境中被解析。 结果值会被设置到子流程变量中。
lang	字符串	juel	可选	表达式解析时使用的脚本语言。

表 6.21. parameter-out属性：

属性	类型	默认值	是否必填	描述
var	字符串		必填	上级流程环境中需要设置的 变量名。
subvar	字符串		'subvar'或'expr'	子流程中需要获取

属性	类型	默认值	是否必填	描述
			其中之一必须指定值	的 变量名。
expr	字符串		’subvar’或’expr’其中之一必须指定值	这个表达式将会在sub流程环境下被解析。 结果值会被设置到上级流程变量中。
lang	字符串	juel	可选	表达式解析时使用的脚本语言。

表 6.22. 对外变量映射的额外transition元素：

元素	多重	描述
outcome-value	0..1	如果outcome与值匹配， 就会在子流程结束时进入这个流向。这个值是由一个子元素指定的。

6.2.7.1. sub-process变量

这个SubProcessVariables示例场景将展示子流程获得基本工作方式， 如何向子流程中反馈信息，当它启动时， 如果从子流程中导出信息，当它结束时。

上级流程调用一个需要重审的文档。

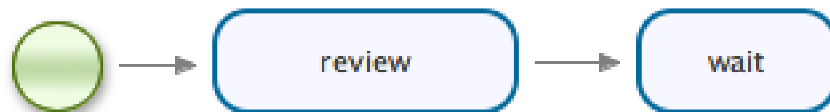

图 6.14. 子流程文档示例流程

```
<process name="SubProcessDocument" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="review" />
  </start>

  <sub-process name="review"
               sub-process-key="SubProcessReview">

    <variable name="document" init="#{document}" />
    <out-variable name="reviewResult" init="#{result}" />

    <transition to="wait" />
  </sub-process>
```

```
  <state name="wait"/>

</process>
```

重审流程是一个可以对所有类型的重审工作重用的流程。

图 6.15. 子流程重审示例流程

```
<process name="SubProcessReview" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="get approval"/>
  </start>

  <task name="get approval"
        assignee="johndoe">

    <transition to="end"/>
  </task>

  <end name="end" />

</process>
```

文档流程被启动，并授予一个文档变量：

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("document", "This document describes how we can make more money...");

ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("SubProcessDocument", variables);
```

然后上级流程会到达子流程节点。 一个子流程会被创建并关联到上级流程中。 当SubProcessReview流程实例启动时， 它到达了task。 一个任务会为johndoe创建。

```
List<Task> taskList = taskService.findPersonalTasks("johndoe");
Task task = taskList.get(0);
```

我们可以看到文档已经被通过， 从上级流程实例到子流程实例：

```
String document = (String) taskService.getVariable(task.getDbid(), "document");
assertEquals("This document describes how we can make more money...", document);
```

然后我们在任务上设置一个变量。这一般都是通过一个表单来完成。 但是这是我们将演示如何使用编程方式完成。

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("result", "accept");
taskService.setVariables(task.getDbid(), variables);
```

完成这个任务，会导致子流程实例结束。

```
taskService.completeTask(task.getDbid());
```

当子流程结束时，上级流程会被signal标记（不是notify提醒）。 首先result变量会从子流程复制到父流程的reviewResult变量中。 然后上级流程会继续， 并离开review活动。

6.2.7.2. sub-process外出值

在SubProcessOutcomeValueTest示例中， 子流程实例变量的值被用来选择sub-process活动 的外出流向。

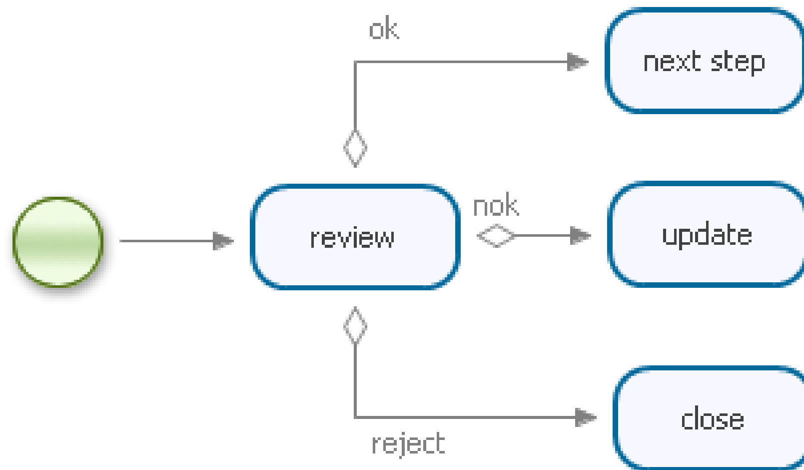

图 6.16. 子流程文档示例流程

```
<process name="SubProcessDocument">

  <start>
    <transition to="review" />
  </start>

  <sub-process name="review"
               sub-process-key="SubProcessReview"
               outcome="#{result}">

    <transition name="ok" to="next step" />
    <transition name="nok" to="update" />
    <transition name="reject" to="close" />
  </sub-process>

  <state name="next step" />
  <state name="update" />
  <state name="close" />

</process>
```

这个SubProcessReview和上面的 子流程变量示例相同：

图 6.17. 子流程复审示例流程，为外向变量

```
<process name="SubProcessReview" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="get approval"/>
  </start>

  <task name="get approval"
        assignee="johndoe">

    <transition to="end"/>
  </task>

  <end name="end" />

</process>
```

一个新文档实例会像通常一样启动：

```
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("SubProcessDocument");
```

任务被获得到johndoe的任务列表中

```
List<Task> taskList = taskService.findPersonalTasks("johndoe");
Task task = taskList.get(0);
```

然后result变量被设置， 任务完成。

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put(quot;resultquot;, quot;okquot;);
taskService.setVariables(task.getId(), variables);
taskService.completeTask(task.getDbid());
```

在这个场景中，ok流向被获取在上级流程中 在子流程复审活动外。 这个例子测试用例也展示了其他场景。

6.2.7.3. sub-process外向活动

一个流程可以有多个结束节点。在SubProcessOutcomeActivityTest示例中， 结果的结束节点被用来选择sub-process活动的 外出流向。

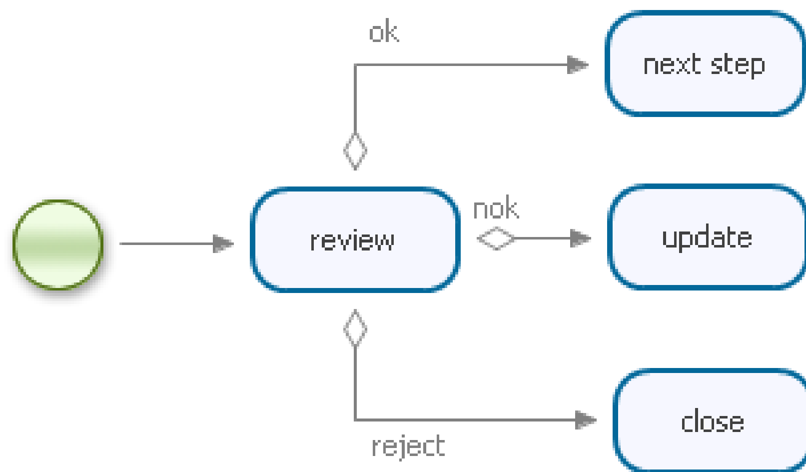

图 6.18. 子流程文档实例流程，对于外出活动

```
<process name="SubProcessDocument">

  <start>
    <transition to="review" />
  </start>

  <sub-process name="review"
               sub-process-key="SubProcessReview">

    <transition name="ok" to="next step" />
    <transition name="nok" to="update" />
    <transition name="reject" to="close" />
  </sub-process>

  <state name="next step" />
  <state name="update" />
  <state name="close" />

</process>
```

这个SubProcessReview现在拥有多个结束活动：

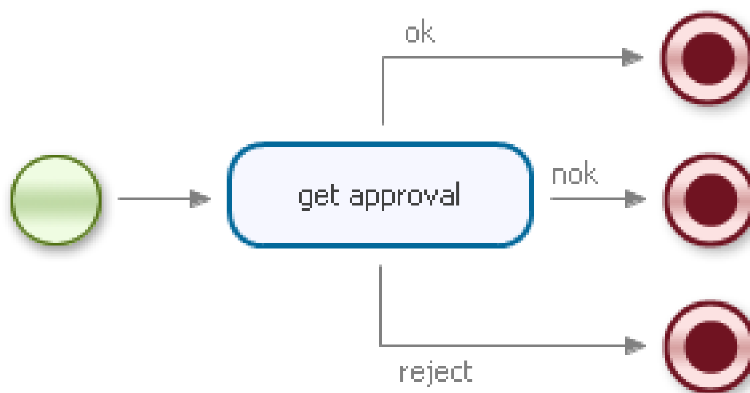

图 6.19. 子流程重审示例流程，为外出活动

```
<process name="SubProcessReview" xmlns="http://jbpm.org/4.0/jpdl">
```

```
  <start>
    <transition to="get approval"/>
  </start>

  <task name="get approval"
        assignee="johndoe">

    <transition name="ok" to="ok"/>
    <transition name="nok" to="nok"/>
    <transition name="reject" to="reject"/>
  </task>

  <end name="ok" />
  <end name="nok" />
  <end name="reject" />

</process>
```

一个新文档流程实例像通常一样被启动：

```
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("SubProcessDocument");
```

任务被获取到johndoe的任务列表中

```
List<Task> taskList = taskService.findPersonalTasks("johndoe");
Task task = taskList.get(0);
```

任务会在ok结束。

```
taskService.completeTask(task.getDbid(), "ok");
```

这将导致子流程结束在ok结束活动。 上级节点会通过ok流向 进入next step。

这个示例测试用例也展示了其他场景。

6.2.8. custom

调用用户代码，实现一个自定义的活动行为。

一个自定义活动引用了用户代码。参考第 6.7 节 “用户代码” 获得特定属性和元素的更多信息。让我们看这个例子：

```
<process name="Custom" xmlns="http://jbpm.org/4.0/jpdl">

  <start >
    <transition to="print dots" />
  </start>

  <custom name="print dots"
        class="org.jbpm.examples.custom.PrintDots">

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

```
  </custom>

  <end name="end" />

</process>
```

这个自定义活动行为类PrintDots 演示了它有可能去控制流向，当实现了自定义活动行为时。 在这种情况下PrintDots 活动实现将在打印点后在活动中暂停 直到出现一个signal。

```
public class PrintDots implements ExternalActivityBehaviour {

  private static final long serialVersionUID = 1L;

  public void execute(ActivityExecution execution) {
    String executionId = execution.getId();

    String dots = ...;

    System.out.println(dots);

    execution.waitForSignal();
  }

  public void signal(ActivityExecution execution,
                     String signalName,
                     Map<String, ?> parameters) {
    execution.take(signalName);
  }
}
```

6.3. 原子活动

6.3.1. java

java任务。流程处理的流向会执行 这个活动配置的方法。

表 6.23. java 属性

属性	类型	默认值	是否必须	描述
class	classname		'class' 或 'expr' 之一必须指定	完全类名。这个类将被实例化， 然后对象会在方法调用之后被处理。
expr	表达式		'class' 或 'expr' 之一必须指定	这个表达式返回方法被调用 产生的目标对象。
method	methodname		必须	调用的方法名
var	variablename		可选	返回值存储的 变量名

表 6.24. java 元素

元素	个数	描述
field	0..*	在方法调用之前给成员变量注入配置值
arg	0..*	方法参数

思考下面的例子：

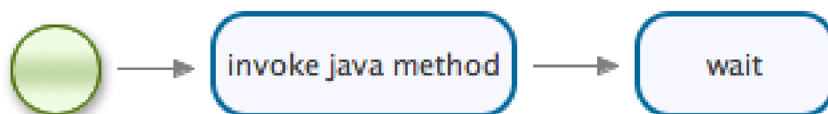

图 6.20. java 任务（task）

```
<process name="Java" xmlns="http://jbpm.org/4.0/jpdl">

  <start >
    <transition to="greet" />
  </start>

  <java name="greet"
        class="org.jbpm.examples.java.JohnDoe"
        method="hello"
        var="answer"
        >

    <field name="state"><string value="fine"/></field>
    <arg><string value="Hi, how are you?"/></arg>

    <transition to="shake hand" />
  </java>

  <java name="shake hand"
        expr="#{hand}"
        method="shake"
        var="hand"
        >

    <arg><object expr="#{joesmoe.handshakes.force}"/></arg>
    <arg><object expr="#{joesmoe.handshakes.duration}"/></arg>

    <transition to="wait" />
  </java>

  <state name="wait" />

</process>
```

调用的类：

```
public class JohnDoe {

  String state;
  Session session;

  public String hello(String msg) {
    if ( (msg.indexOf("how are you?")!=-1)
         && (session.isOpen())
       ) {
      return "I'm "+state+", thank you.";
    }
    return null;
  }
}
```

```
public class JoeSmoe implements Serializable {

  static Map<String, Integer> handshakes = new HashMap<String, Integer>();
  {
    handshakes.put("force", 5);
    handshakes.put("duration", 12);
  }

  public Map<String, Integer> getHandshakes() {
    return handshakes;
  }
}
```

```
public class Hand implements Serializable {

  private boolean isShaken;

  public Hand shake(Integer force, Integer duration) {
    if (force>3 && duration>7) {
      isShaken = true;
    }

    return this;
  }

  public boolean isShaken() {
    return isShaken;
  }
}
```

第一个java活动greet指定了，在它执行期间，一个 org.jbpm.examples.java.JohnDoe类的实例会被初始化 这个类的hello方法会被调用，并获得调用的返回对象。 名为answer的变量会获得调用的结果。

上面的类展示了它包含名字为state和session的两个fields， 在整个流向中field指定的values和arg 这两个配置元素会被调用。 流程处理实例预期的结果是流程处理的变量answer的值为 字符串I'm fine,thank you.。

第二个java活动叫做shake hand。 它会处理#{hand}表达式， 把调用的结果对象作为目标对象。在这个对象上， shake方法会被调用。这两个参数会各自被 表达式#{joesmoe.handshakes.force}和#{joesmoe.handshakes.duration}计算。结果对象 是一个hand的修改版本，而var="hand"回导致修改hand，

通过覆盖老hand的变量值。

6.3.2. script脚本

script脚本活动会解析一个script脚本。任何一种符合JSR-223 [https://scripting.dev.java.net/] 规范 的脚本引擎语言都可以在这里运行。 脚本引擎的配置会在下面解释：

下面有2种方式详细说明如何使用脚本：

6.3.2.1. script expression脚本表达式

script脚本提供expr属性。 这个短小的符号在属性里比在文本元素里表达更简单。 如果没有指定语言（lang）会使用 默认的表达式语言（default-expression-language）。

表 6.25. script脚本表达式属性

属性	类型	默认值	是否必须	描述
expr	text		必须	执行表达式的文本
lang	脚本语言名字定义在 第 8 章 Scripting脚本	默认的表达式语言定义在第 8 章 Scripting脚本	可选	表达式指定的语言
var	variablename		可选	返回值存储的 变量名。

在下一个例子中，我们会看到script脚本如何 使用表达式活动和返回结果怎样存储在variable变量里 。

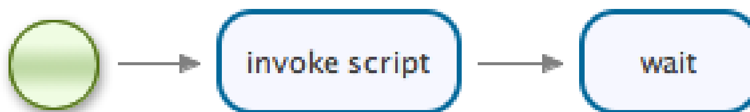

图 6.21. 流程处理的script.expression示例

```
<process name="ScriptExpression" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="invoke script" />
  </start>

  <script name="invoke script"
          expr="Send packet to #{person.address}"
          var="text">

    <transition to="wait" />
  </script>

  <state name="wait"/>
```

```
</process>
```

这个例子使用了person类，代码如下：

```
public class Person implements Serializable {

  String address;

  public Person(String address) {
    this.address = address;
  }

  public String getAddress() {
    return address;
  }

  public void setAddress(String address) {
    this.address = address;
  }
}
```

当为这个流程处理启动一个流程处理实例时， 我们提供一个的person的地址属性的变量。

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("person", new Person("Honolulu"));

executionService.startProcessInstanceByKey("ScriptText", variables);
```

然后script脚本活动中的整个流向， 变量中将包含'Send packet to Honolulu'

6.3.2.2. script 文本

第2种方式是用text元素指定script脚本。 当script text有多行的时候这种方式更方便。

表 6.26. script text属性

属性	类型	默认值	是否必须	描述
lang	脚本语言名字定义在第 8 章 Scripting脚本	默认的表达式语言定义在第 8 章 Scripting脚本	可选	表达式指定的语言
var	variablename		可选	返回值存储的 变量名。

表 6.27. script text元素

元素	个数	描述
text	1	包含script脚本的文本

例如：

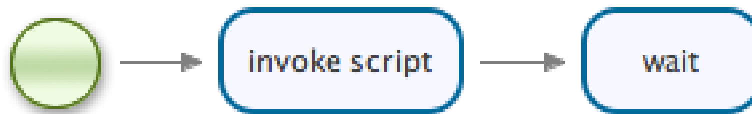

图 6.22. 流程处理的script text示例

```
<process name="ScriptText" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="invoke script" />
  </start>

  <script name="invoke script"
          var="text">
    <text>
      Send packet to #{person.address}
    </text>
    <transition to="wait" />
  </script>

  <state name="wait"/>

</process>
```

这个流程处理的整个流向要求和上面的script脚本表达式一样。

6.3.3. hql

使用hql活动，我们可以在database中执行HQL query， 并将返回的结果报仇呢到流程处理的变量中。

表 6.28. hql属性

属性	类型	默认值	是否必须	描述
var	variablename		可选	存储结果的变量名
unique	{true,false}	false	可选	值为true是指从uniqueResult()方法中获得hibernate query的结果。默认值是false。值为false的话会使用list()方法得到结果。

表 6.29. hql元素

元素	个数	描述
query	1	HQL query
parameter	0..*	query的参数

例如：

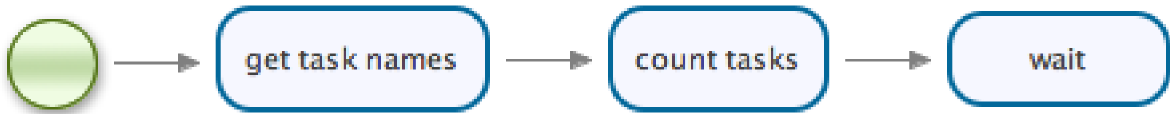

图 6.23. 流程处理的hql例子

```
<process name="Hql" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="get process names" />
  </start>

  <hql name="get process names"
       var="activities with o">
    <query>
      select activity.name
      from org.jbpm.pvm.internal.model.ActivityImpl as activity
      where activity.name like :activityName
    </query>
    <parameters>
      <string name="activityName" value="%o%" />
    </parameters>
    <transition to="count activities" />
  </hql>

  <hql name="count activities"
       var="activities"
       unique="true">
    <query>
      select count(*)
      from org.jbpm.pvm.internal.model.ActivityImpl
    </query>
    <transition to="wait" />
  </hql>

  <state name="wait"/>

</process>
```

6.3.4. sql

sql活动和hql活动十分相似， 唯一不同的地方就是 使用session.createSQLQuery(...)。

6.3.5. mail

通过使用mail活动，流程作者 可以指定一个邮件信息的内容，一次发送给多个收件人。 每个email信息都是从一个模板生成的。 模板可能指定在元素内部，或者在配置文件的 process-engine-context部分指定。

表 6.30. mail属性

属性	类型	默认值	是否必须	描述
template	字符串		否	引用配置文件中的一个mail-template元素。 如果没找到， 必须使用子元素在内部指定。

表 6.31. mail元素

元素	个数	描述
from	0..1	发件者列表
to	1	主要收件人列表
cc	0..1	抄送收件人列表
bcc	0..1	密送收件人列表
subject	1	这个元素的文字内容会成为消息的主题
text	0..1	这个元素的文字内容会成为消息的文字内容
html	0..1	这个元素的文字内容会成为消息的HTML内容
attachments	0..1	附件可以指定URL，classpath资源或 本地文件

示例使用方法：

```
<process name="InlineMail" xmlns="http://jbpm.org/4.0/jpdl">
  <start>
    <transition to="send birthday reminder note" />
  </start>
  <mail name="send birthday reminder note">
    <to addresses="johnDoe@some-company.com" />
    <subject>Reminder: ${person} celebrates his birthday!</subject>
    <text>Do not forget: ${date} is the birthday of ${person} </text>
    <transition to="end" />
  </mail>
```

```
  <state name="end"/>
</process>
```

在安装后的默认配置中包含一个jbpm.mail.properties， 它是为了指定jBPM使用的邮件服务器的。 如果想要使用其他邮件服务器，而不是localhost， 可以修改配置文件中的 mail.smtp.host。

参考开发者指南， 以获得更多mail的配置和使用方式。（尚未支持）

6.4. Common activity contents通用活动内容

除非在上面指定其他的元素，否则所有的活动都会包含 这些内容模板：

表 6.32. common activity属性

属性	类型	默认值	是否必须	描述
name	any text		必须	activity活动的名字

表 6.33. common activity元素

元素	个数	描述
transition	0..*	向外的转移

6.5. Events事件

事件指定流程中的特定点，那里注册了一系列的时间监听器。 当一个流程通过这一点时，事件监听器就会被提醒。 事件和监听器不会显示在流程的图形视图中。 一个事件会被流程定义中的一个元素触发，比如流程定义， 一个活动或一个流向。

事件监听器接口看起来就像这样：

```
public interface EventListener extends Serializable {

  void notify(EventListenerExecution execution) throws Exception;

}
```

所有的自动活动可以作为 事件监听器来使用。

为了给一个流程或一个活动分配一系列的事件监听器， 使用on元素来为事件监听器分组并指定事件。 on可以内嵌到process 或任何活动的子节点。

为了分配一系列的事件监听器给流向的take事件， 只需要包含事件监听器，直接在transition 元素中。

表 6.34. on属性：

属性	类型	默认值	是否必填	描述
event	{start ｜ end}		必填	事件名称

表 6.35. on元素：

元素	个数	描述
event-listener	0..*	一个事件监听器实现对象。
任何自动活动	0..*	

表 6.36. 事件监听器属性：

属性	类型	默认值	是否必填	描述
propagation	{enabled ｜ disabled ｜ true ｜ false ｜ on ｜ off}	disabled	可选	指定事件监听器应该也被传播的事件调用。
continue	{sync ｜ async ｜ exclusive}	sync	可选	指定execution是否应该被异步执行，在事件监听器执行之前，可以参考第 6.6 节 “异步调用”

6.5.1. 事件监听器示例

让我们看一个使用了事件监听器的示例流程：

图 6.24. 事件监听器示例流程

```
<process name="EventListener" xmlns="http://jbpm.org/4.0/jpdl">

  <on event="start">
    <event-listener class="org.jbpm.examples.eventlistener.LogListener">
      <field name="msg"><string value="start on process definition"/></field>
    </event-listener>
  </on>
```

```
  <start>
    <transition to="wait"/>
  </start>

  <state name="wait">
    <on event="start">
      <event-listener class="org.jbpm.examples.eventlistener.LogListener">
        <field name="msg"><string value="start on activity wait"/></field>
      </event-listener>
    </on>
    <on event="end">
      <event-listener class="org.jbpm.examples.eventlistener.LogListener">
        <field name="msg"><string value="end on activity wait"/></field>
      </event-listener>
    </on>
    <transition to="park">
      <event-listener class="org.jbpm.examples.eventlistener.LogListener">
        <field name="msg"><string value="take transition"/></field>
      </event-listener>
    </transition>
  </state>

  <state name="park"/>

</process>
```

LogListener将维护一系列日志， 在静态属性中：

```
public class LogListener implements EventListener {

  // value gets injected from process definition
  String msg;

  public void notify(EventListenerExecution execution) {
    List<String> logs = (List<String>) execution.getVariable("logs");
    if (logs==null) {
      logs = new ArrayList<String>();
      execution.setVariable("logs", logs);
    }

    logs.add(msg);

    execution.setVariable("logs", logs);
  }
}
```

下一步，我们启动一个新流程实例。

```
ProcessInstance processInstance = executionService.startProcessInstanceByKey("EventListener");
```

然后流程实例执行到等待活动。 所以我们提供了一个singal，那将导致它执行到结束。

```
Execution execution = processInstance.findActiveExecutionIn("wait");
executionService.signalExecutionById(execution.getId());
```

这个日志消息队列会像这样：

```
[start on process definition,
 start on activity wait,
 end on activity wait,
 take transition]
```

6.5.2. 事件传播

事件是从活动和转移传播到外部的活动， 最终传播到流程定义。

默认情况下，事件监听器只对当前的事件监听器订阅的元素 所触发的事件起作用。 但是，通过指定propagation="enabled"， 事件监听器也可以对所有在这个元素包含中的事件起作用。

6.6. 异步调用

每次对于ExecutionService.startProcessInstanceById(...) 或ExecutionService.signalProcessInstanceById(...)的调用 会让流程执行在发起的线程中（客户端）。换句话说， 那些方法将只会流程执行到达一个等待状态后才能返回。

这种默认的行为有很多优点：用户系统的事务可以很容易的就传递给jBPM， 这样jBPM的数据库更新操作就可以在用户的事务环境中完成了。 其次，当流程执行过程中某些操作出错的时候， 客户也可以获得一个异常。通常来说， 这些在流程的两个等待状态之间需要完成的工作量都是比较小的。 即便在两个等待状态之间有很多个自动的活动节点需要执行。 所以在大多数情况下， 最好是在一个单独的事务中执行所有这些工作。这也就解释了jPDL的默认行为， 它会在客户端的线程中同步执行流程的所有工作。

在一些情况下，你不想等待所有的自动活动都完成后再返回响应， jPDL允许在事务边界上进行良好的控制。 在流程中的不同环境中，可以使用异步调用这种方式。 异步调用一般用在以下环节，异步调用会提交事务， jBPM方法调用会立即返回。jBPM会启动一个新事务， 以异步方式继续执行其他的自动流程工作。 jBPM在内部使用异步消息来完成这些工作。

当使用异步调用时，一个异步消息会被作为当前事务的一部分发送出去。 然后原始调用方法，像是startProcessInstanceById(...) 或signalProcessInstanceById(...)会直接返回。 当异步消息被提交执行时，它会启动一个新事务， 在流程离开的地方重新开始执行。

表 6.37. 任意活动属性：

属性	类型	默认值	是否必填	描述
continue	{sync | async | exclusive}	sync	可选	用来表明在活动被执行前， 是否使用异步调用。

- sync (默认值) 作为当前事务的一部分，继续执行元素。
- async 使用一个异步调用（又名安全点）。 当前事务被提交，元素在一个新事务中执行。 事务性的异步消息被jBPM用来 实现这个功能。

- exclusive 使用一个异步调用（又名安全点）。 当前事务被提交，元素在一个新事务中执行。 事务性的异步消息被jBPM用来 实现这个功能。唯一性消息不会被同步执行。 jBPM会确认对同一个流程实例，具有唯一性的定时计划 不会同时执行，即使你的jBPM配置了 多个异步消息执行器（比如jobExecutor）在不同的系统中运行。 这可以用来防止乐观锁失败， 如果多个，有潜在冲突可能的job被安排在同一个事务中。

让我们来看一些例子。

6.6.1. 异步活动

图 6.25. 异步活动实例流程

```
<process name="AsyncActivity" xmlns="http://jbpm.org/4.0/jpdl">

  <start>
    <transition to="generate pdf"/>
  </start>

  <java name="generate pdf"
        continue="async"
        class="org.jbpm.examples.async.activity.Application"
        method="generatePdf" >
    <transition to="calculate primes"/>
  </java>

  <java name="calculate primes"
        continue="async"
        class="org.jbpm.examples.async.activity.Application"
        method="calculatePrimes">
    <transition to="end"/>
  </java>

  <end name="end"/>

</process>
```

```
public class Application {

  public void generatePdf() {
    // assume long automatic calculations here
  }

  public void calculatePrimes() {
    // assume long automatic calculations here
  }
}
```

```
ProcessInstance processInstance =
     executionService.startProcessInstanceByKey("AsyncActivity");
String processInstanceId = processInstance.getId();
```

如果不使用异步调用，这将是一个完全自动的流程， 流程会在在startProcessInstanceByKey方法中 从头执行到尾。

可使用了continue="async"后 执行只会执行到generate pdf活动。 然后一个异步调用消息会被发送，startProcessInstanceByKey方法就会返回。

在一个通常的配置中，job执行器会自动获得消息并执行它。 当时在测试环境下，对于这些例子我们希望控制这些消息什么时候被执行， 所以就没有配置job执行器。 因此我们必须像下面这样手工执行job：

```
Job job = managementService.createJobQuery()
  .processInstanceId(processInstanceId)
  .uniqueResult();
managementService.executeJob(job.getDbid());
```

这会获得流程，直到它执行calculate primes活动， 然后另一个异步消息又会 被发送。

然后这个消息又会被获得，然后当消息被执行时， 那个事务就会将流程执行到结束为止。

6.6.2. 异步分支

图 6.26. 异步分支实例流程

```
<process name="AsyncFork" xmlns="http://jbpm.org/4.0/jpdl">

  <start >
    <transition to="fork"/>
  </start>

  <fork >
    <on event="end" continue="exclusive" />
    <transition />
    <transition />
  </fork>

  <java class="org.jbpm.examples.async.fork.Application" >
    <transition />
  </java>

  <java class="org.jbpm.examples.async.fork.Application" >
    <transition />
  </java>

  <join >
    <transition to="end"/>
  </join>

  <end />

</process>
```

```
public class Application {

  public void shipGoods() {
    // assume automatic calculations here
  }

```

```
  public void sendBill() {
    // assume automatic calculations here
  }
}
```

通过在fork的end事件上使用异步调用 （<on event="end" continue="exclusive" />）， 每个沿着分支转移生成的执行都会 使用异步方式继续执行。

exclusive这个值被用来 将两个来自分支的异步调用的job结果进行持久化。 各自的事务会分别执行ship goods和send bill， 然后这两个执行都会达到join节点。 在join节点中，两个事务会同步到一个相同的执行上（在数据库总更新同一个执行）， 这可能导致一个潜在的乐观锁失败。

```
ProcessInstance processInstance = executionService.startProcessInstanceByKey("AsyncFork");
String processInstanceId = processInstance.getId();

List<Job> jobs = managementService.createJobQuery()
  .processInstanceId(processInstanceId)
  .list();

assertEquals(2, jobs.size());

Job job = jobs.get(0);

// here we simulate execution of the job,
// which is normally done by the job executor
managementService.executeJob(job.getDbid());

job = jobs.get(1);

// here we simulate execution of the job,
// which is normally done by the job executor
managementService.executeJob(job.getDbid());

Date endTime = historyService
  .createHistoryProcessInstanceQuery()
  .processInstanceId(processInstance.getId())
  .uniqueResult()
  .getEndTime();

assertNotNull(endTime);
```

6.7. 用户代码

当一个接口方法被调用时， 大量的元素在jPDL流程语言中引用了一个对象。

- event-listener
- custom
- java
- task中的assignment-handler
- decision中的handler

所有的对象都是通过类名引用， 它们会在解析时初始化。这些对象必须是线程安全的。 这是没问题的，因为这些对象是不变的。

表达式中引用的对象 是动态计算的。

表 6.38. 属性：

属性	类型	默认值	是否必填	描述
class	类名		class和expr其中之一是必须的	全类名
expr	表达式		class和expr其中之一是必须的	全类名。

表 6.39. 子元素：

元素	个数	描述
field	0..*	描述一个配置值，被直接注入到字段中 在用户类使用之前。
property	0..*	描述一个配置值，通过一个setter方法注入 在用户对象使用之前。

表 6.40. field属性：

属性	类型	默认值	是否必填	描述
name	字符串		必填	字段名

在field或property注入时， 可以使用很多不同值的类型，比如string, object, map, list, ref等等。可以icankaojpdl的schema文档获得更多信息。

第 7 章 Variables变量

流程变量在流程外部，通过ExecutionService提供的方法进行访问：

- ProcessInstance startProcessInstanceById(String processDefinitionId, Map<String, Object> variables);
- ProcessInstance startProcessInstanceById(String processDefinitionId, Map<String, Object> variables, String processInstanceKey);
- ProcessInstance startProcessInstanceByKey(String processDefinitionKey, Map<String, ?> variables);
- ProcessInstance startProcessInstanceByKey(String processDefinitionKey, Map<String, ?> variables, String processInstanceKey);
- void setVariable(String executionId, String name, Object value);
- void setVariables(String executionId, Map<String, ?> variables);
- Object getVariable(String executionId, String variableName);
- Set<String> getVariableNames(String executionId);
- Map<String, Object> getVariables(String executionId, Set<String> variableNames);

在流程中可以通过Execution接口，传递给用户代码，比如 ActivityExecution和EventListenerExecution：

- Object getVariable(String key);
- void setVariables(Map<String, ?> variables);
- boolean hasVariable(String key);
- boolean removeVariable(String key);
- void removeVariables();
- boolean hasVariables();
- Set<String> getVariableKeys();
- Map<String, Object> getVariables();
- void createVariable(String key, Object value);
- void createVariable(String key, Object value, String typeName);

jBPM没有自动检测变量值变化的机制。 比如，从实例变量中获得了一个序列化的集合，添加了一个元素， 然后你就需要把变化了的变量值准确的保存到DB中。

7.1. 变量作用域

默认情况下，变量创建在顶级的流程实例作用域中。 这意味着它们对整个流程实例中的所有执行都是可见的，可访问的。 流程变量是动态创建的。意味着，当一个变量通过任何一个方法设置到流程中，整个变量就会被创建了。

每个执行都有一个变量作用域。声明在内嵌执行级别中的变量， 可以看到它自己的变量和声明在上级执行中的变量，这时按照正常的作用域规则。 使用execution接口中的createVariable方法，ActivityExecution和EventListenerExecution可以创建流程的局部变量。

在未来的发布中，我们可能添加在jPDL流程语言中声明的变量。

7.2. 变量类型

jBPM支持下面的Java类型，作为流程变量：

- java.lang.String
- java.lang.Long
- java.lang.Double
- java.util.Date
- java.lang.Boolean
- java.lang.Character
- java.lang.Byte
- java.lang.Short
- java.lang.Integer
- java.lang.Float
- byte[] (byte array)
- char[] (char array)
- hibernate entity with a long id
- hibernate entity with a string id
- serializable

为了持久化这些变量，变量的类型会按照这个列表中的例子进行检测。 第一个匹配的类型， 会决定变量如何保存。

第 8 章 Scripting脚本

只有jUEL被配置成了脚本语言。 jUEL是通用表达式语言的一个实现。如果想获得 更多如何使用UEL的细节，可以参考 JEE 5 教程，通用表达式语言一章[http://java.sun.com/javaee/5/docs/tutorial/doc/bnahq.html]。

如果想配置其他脚本语言， 请参考开发者指南（尚未支持）。

第 9 章 Indentity身份认证

jBPM的identity组件是以JBoss IDM [http://www.jboss.org/community/docs/DOC-13258]为基础的，配置如下：

```
<jbpm-configuration xmlns="http://jbpm.org/xsd/cfg">

  <process-engine-context>
    ...
    <identity-service />
    ...
  </process-engine-context>

  <transaction-context>
    ...
    <identity-session realm="realm://jbpm-identity" />
  </transaction-context>

</jbpm-configuration>
```

如果想替换identity组件，保持identity-service声明，实现org.jbpm.session.IdentitySession并在你的transation context里配置identity session，如下：

```
<jbpm-configuration xmlns="http://jbpm.org/xsd/cfg">
    ...
    <transition-context>
        ...
        <object class="your.package.YourIdentitySession" />
    </transition>
</jbpm-configuration>
```

第 10 章 支持邮件

jBPM 4利用JavaMail API来为业务流程作者们 实现高级的邮件服务。

10.1. 生产者

生产者用来为jBPM创建邮件信息。 所有邮件生产者都会实现org.jbpm.pvm.internal.email.spi.MailProducer接口。 一个默认的邮件生产者可以用来在外部指定邮件所需的地址。

10.1.1. 默认生产者

默认的邮件生产者可以根据模板生成包含文本，HTML和附件的内容。 模板可以在内部提供或者 在jBPM配置的流程引擎环境章节中。 模板可能包含了表达式，通过脚本引擎执行的表达式。 可以参考脚本章节。

下面列出了使用内置模板的邮件方式。

```
<mail name="rectify" language="juel">                              (1)
  <from addresses='winston@minitrue' />                            (2)
  <to addresses='julia@minitrue, obrien@miniluv'/>                 (3)
  <cc users='bigbrother'/>
  <bcc groups='thinkpol, innerparty'/>
  <subject>Part ${part} Chapter ${chapter}</subject>               (4)
  <text>times ${date} reporting bb dayorder doubleplusungood       (5)
    refs ${unpersons} rewrite fullwise upsub antefiling</text>
  <html><table><tr><td>times</td><td>${date}</td>                  (6)
    <td>reporting bb dayorder doubleplusungood
    refs ${unpersons} rewrite fullwise upsub antefiling</td>
    </tr></table></html>
  <attachments>                                                    (7)
    <attachment url='http://www.george-orwell.org/1984/3.html'/>
    <attachment resource='org/example/pic.jpg'/>
    <attachment file='${user.home}/.face'/>
  </attachments>
</mail>
```

1. 这里指定了在模板中使用的表达式所用的脚本语言。 如果没有指定，会假设使用默认的表达式语言。

2. 信息发送者的列表。 发送者可以直接写成他们的邮箱地址或者使用身份模块里的定义。

3. 信息接收者的列表，使用以下的分类：To （发送给），CC（抄送）和BCC（密送）。 像发送者一样，接收者可以直接写成他们的邮箱地址 或者使用身份模块里的定义。

4. subject元素中的字符会用来 作为邮件的标题。

5. text元素中的字符会用来作为 邮件的纯文本内容。

6. html元素中的节点会用来作为 邮件的HTML内容。

7. 附件可以指定为绝对URL网址， classpath下的资源或本地文件。

注意，模板中的每个部分都可以使用脚本表达式。

如果希望得到复杂的邮件或自己生成附件， 参考扩展点：自定义邮件。

10.2. 模板

邮件模板可以使用流程定义中的信息。 模板被放在你配置文件的流程引擎环境部分。 所以内置模板的可用元素，像写在上一章中的那些信息， 都可以用来扩展模板。参考下面的代码片段。

```
<jbpm-configuration>
<process-engine-context>
  <mail-template name="rectify-template">
    <!-- same elements as inline template -->
  </mail-template>
</process-engine-context>
</jbpm-configuration>
```

每个模板必须有一个唯一的名字。邮件节点可以通过`template`属性来引用模板， 像下面这样。

```
<mail name="rectify" template="rectify-template />
```

10.3. 服务器

邮件服务器是配置在配置文件中的。 `mail-server`元素定义了一个SMTP邮件服务器来发送邮件信息。 因为jBPM用户使用JavaMail来发送邮件，所有JavaMail所支持的属性都可以在jBPM中用到。 使用`session-properties`这个子元素， SMTP属性必须像下面这样提供出来。

可以参考Sun的JavaMail API获得更多被支持的属性的信息： Sun SMTP 属性 [http://java.sun.com/products/javamail/javadocs/com/sun/mail/smtp/package-summary.html].

```
<jbpm-configuration>
<transaction-context>
  <mail-session>
    <mail-server>
      <session-properties>
        <property name="mail.smtp.host" value="localhost" />
        <property name="mail.smtp.port" value="2525" />
        <property name="mail.from" value="noreply@jbpm.org" />
      </session-properties>
    </mail-server>
  </mail-session>
</transaction-context>
</jbpm-configuration>
```

如果在外发的信息中没有配置"From"属性， 会使用`mail.from`中配置的参数。

10.3.1. 多服务器

对多个SMTP服务器的支持已经添加到jBPM 4中，来获得更广泛的服务器组织结构。 比如，对于那些既有内部邮箱 又有外部邮箱的公司就很有用。

为了安装多SMTP邮件服务器，需要在配置文件中定义多个邮件服务器， 像下面这样。其中的address-filter标签定义了 每个邮件服务器的域。地址过滤器是一个正则表达式 用来决定一个邮件地址应该被哪个服务器处理。

可以参考Sun的Pattern API获得更多正则表达式支持的信息： Sun正则表达式 [http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html].

```
<jbpm-configuration>
<transaction-context>
  <mail-session>
    <mail-server>
      <address-filter>
        <include>.+@jbpm.org</include>
      </address-filter>
      <session-properties>
        <property name="mail.smtp.host" value="int.smtp.jbpm.org" />
        <property name="mail.from" value="noreply@jbpm.org" />
      </session-properties>
    </mail-server>
    <mail-server>
      <address-filter>
        <exclude>.+@jbpm.org</exclude>
      </address-filter>
      <session-properties>
        <property name="mail.smtp.host" value="ext.smtp.jbpm.org" />
        <property name="mail.from" value="noreply@jbpm.org" />
      </session-properties>
    </mail-server>
  </mail-session>
</transaction-context>
</jbpm-configuration>
```

地址过滤器使用以下的逻辑来获得地址。

- 被包含了， 而且没有被排除的地址。
- 没有指定为被包含的地址， 默认为被包含。
- 没有指定为被排除地址， 默认为没有被排除。

10.4. 扩展点

10.4.1. 自定义生产者

jBPM 4允许我们创建自己的邮件生产者来指定一个组织所需的特定邮件。 为了实现这个功能，用户必须实现 org.jbpm.pvm.internal.email.spi.MailProducer接口。 produce方法将返回一个或多个Message对象， 这些信息会被MailSession发送出去。

10.4.1.1. 例子：自定义附件

在运行时生成自定义附件，在jBPM 4里很容易实现。 通过扩展默认的邮件生产者，或者实现你自己的MailProducer接口， 附件可以生成， 并在运行时添加到邮件信息中。

下面是一个如何扩展MailProducerImpl的例子， 用来向每个外发的邮件中添加额外的附件。

```
public class CustomMailProducer extends MailProducerImpl {

  protected void addAttachments(Execution execution, Multipart multipart) {
    // have default mail producer create attachments from template
    super.addAttachments(execution, multipart);

    // create a body part to carry the content
    BodyPart attachmentPart = new MimeBodyPart();

    // set content provided by an arbitrary data handler
    attachmentPart.setDataHandler(...);

    // attach content
    multipart.addBodyPart(attachmentPart);
  }
}
```

附录 A. 修改日志

修订历史

修订 jBPM-4.1　　　　　　　2009-09-01

1. revision: 5288 ~ 5592
2. ch09-Identity.xml改为ch09-Configuration.xml，不再有identity的介绍了，改成了对工作日历的配置说明。???。
3. 在 第 1 章 导言 中添加“报告问题”部分的内容。
4. 在 第 2 章 安装配置 中重写了 第 2.4 节 “安装脚本”。
5. 在 第 2 章 安装配置 中添加 了进行数据库结构升级的内容，但是又被注释掉了。

修订 jBPM-4.0　　　　　　　2009-07-10

1. 第 2 章 安装配置

   ???，补充内容

   第 2.6 节 “JBoss”，补充内容
2. 第 6 章 jPDL

   第 6.2.6.6 节 “在任务中支持e-mail”，补充内容

   第 6.3.5 节 “mail”，删除重复的template参数

   第 6.3.5 节 “mail”，补充实例

   第 6.5 节 “Events事件”，因为api修改了，所以要修改例子的代码

   第 6.5 节 “Events事件”，补充内容

   第 6.6 节 “异步调用”，补充内容

   6.7. timer定时器，删除

   第 6.7 节 “用户代码”，补充内容
3. 第 7 章 Variables变量，添加内容
4. 第 8 章 Scripting脚本，重写
5. ch10-JBossIntegration，删除

修订 CR1　　　　　　　2009-06-05

1. 重构

   原第 3 章 流程设计器（GPD）改名为GraphicalProcessDesigner

添加：第 4 章 部署业务归档。

原第四章，改为 第 5 章 服务。

原第五章，改为 第 6 章 jPDL。

原第六章，改为 第 7 章 Variables变量。

原第七章，改为 第 8 章 Scripting脚本。

原第八章，改为 ch09-identity。

原第九章，改为 （ch10-jbossintegration）。

原第十章，改为 第 10 章 支持邮件。

2. 第 1 章 导言

添加：从jBPM 3升级到jBPM 4。

3. 第 2 章 安装配置

添加：必须安装的软件。

4. 第 5 章 服务

添加：必须安装的软件。

重写：第 5.7 节 “执行等待的流向”

补充： 第 5.9 节 “HistoryService历史服务”

补充： 第 5.10 节 “ManagementService管理服务”

5. 第 6 章 jPDL

添加：第 6.6 节 “异步调用”。

添加：第 6.2.6.6 节 “在任务中支持e-mail”。

添加：第 6.3.5 节 “mail”。

添加：第 6.2.8 节 “custom”。

在第 6.2.4 节 “concurrency并发”中添加了一个join属性表。

在第 6.2.7 节 “sub-process子流程”中添加parameter-in和parameter-out的属性表。

修改第 6.3.1 节 “java”的例子。

把super-state改成group

把getAssignedTasks()改成getPersonalTasks()

把getTakableTasks()改成getGroupTasks()

删除原6.3.3.esb活动

修订 0.0.1　　　　　　　　　　2009-03-05

1. 初稿完成。