jBPM4.3 用户指南

java 技术交流群：38615496

翻译官方文档

2010 年 4 月 16 日

注：本文全部来自互联网，版权归原作者所有

目录

Java 技术交流群: 38615496

第 1 章 导言

最好使用 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 协议和 完整的最终用户协议。

✧ 1.2、下载

可以从 sourceforge 上下载发布包。

http://sourceforge.net/projects/jbpm/files/

✧ 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 实际通常使用的方式。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** : 使用的什么容器? （JBoss, Tomcat, 其他）
- **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（jbpm-4.X.zip）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：用于不同环境的安装脚本
- lib：第三方库和一些特定的 jBPM 依赖库
- src：源代码文件
- jbpm.jar：jBPM 主库归档
- migration：参考开发指南

✧ 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。

如果你之前下载过 jboss-5.0.0.GA.zip，你可以把它放到 ${jbpm.home}/install/downloads 目录下。 否则脚本会为你下载它，但是它会消耗一些时间（与你的网络情况有关）。
eclipse-jee-galileo-win32.zip 也一样（或者
eclipse-jee-galileo-linux-gtk(-x86_64).tar.gz 在 linux 平台下 和
eclipse-jee-galileo-macosx-carbon.tar.gz 在 Mac OSX 平台下）。

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
- 安装 jBPM web 控制台
- 安装 Signavio web 建模器

当这些都完成后，JBoss(或 Tomcat，根据 demo.setup 脚本中的选择)会在后台启动。一旦 eclipse 完成启动，你可以继续执行下面的教程 第 3 章 *流程设计器（GPD）*来开始编码你的 jBPM 业务流程。

或者你可以启动建模流程，通过 Signavio web 设计器。

或者使用 jBPM 控制台。 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

目前存在的问题：现在，对于一些比较慢的机器，在初始化报表时， 控制台的失效时间太短了，所以当你第一次请求报表时， 会出现超时，控制台会崩溃。 注销，然后再次登录，就可以避过这个问题。 这个问题已经提交到 JIRA 中了 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。

要想调用安装脚本，打开命令行，进入${jbpm.home}/install 目录。 使用 ant -p 你可以看到 这里可以使用的所有脚本。脚本的参数都设置了默认值， 可以快速执行，下面列表给出了可用脚本的概况：

- demo.setup.jboss：安装 jboss，把 jbpm 安装到 jboss 中， 启动 jboss，创建 jbpm 数据库表结构，部署实例，加载实例身份认证信息， 安装并启动 eclipse
- demo.setup.tomcat：安装 tomcat，把 jboss 安装到 tomcat 中， 启动 tomcat，创建 jbpm 数据库表结构，部署实例，加载实例身份认证信息， 安装并启动 eclipse
- clean.cfg.dir：删除${jbpm.home}/install/generated/cfg 文件夹。
- create.cfg:创建一个配置在 ${jbpm.home}/install/generated/cfg 下，基于当前的参数。
- create.jbpm.schema：在数据库中创建 jbpm 表

- create.user.webapp 创建一个基本的 webapp 在 ${jbpm.home}/install/generated/user-webapp 中
- delete.jboss：删除安装的 jboss
- delete.tomcat：删除安装的 Tomcat
- demo.teardown.jboss：删除 jbpm 数据库的表并停止 jboss
- demo.teardown.tomcat：停止 tomcat 和 hsqldb 服务器（如果需要）
- drop.jbpm.schema：从数据库中删除 jbpm 的表
- get.eclipse：下载 eclipse 如果它不存在
- get.jboss：下载一个 JBoss AS，已经测试过当前的 jBPM 版本，如果它不存在
- get.tomcat：下载一个 Tomcat，已经测试过当前的 jBPM 版本，如果它不存在
- hsqldb.databasemanager：启动 hsqldb 数据库管理器
- install.eclipse：解压 eclipse，下载 eclipse 如果它不存在
- install.jboss：下载 JBoss 如果它不存在，并解压
- install.jbpm.into.jboss：把 jBPM 安装到 JBoss 中
- install.tomcat：把 tomcat 下载到${tomcat.distro.dir}如果 tomcat 不存在，并解压 tomcat
- install.jbpm.into.tomcat：把 jBPM 安装到 tomcat 中
- install.examples.into.tomcat：部署所有的实例流程
- install.signavio.into.jboss：把 signavio 安装到 jboss 中
- install.signavio.into.tomcat 把 signavio 安装到 tomcat 中
- load.example.identities：读取实例用户和用户组数据到数据库
- reinstall.jboss：删除之前的 jboss 安装，并重新安装 jboss
- reinstall.jboss.and.jbpm：删除之前的 jboss 安装，并重新安装 jboss 并把 jbpm 安装到它里面
- reinstall.tomcat：删除之前的 tomcat 安装，并重新安装 tomcat
- reinstall.tomcat.and.jbpm：删除之前的 tomcat 安装，并重新安装 tomcat 并把 jbpm 安装到它里面
- start.eclipse：启动 eclipse
- start.jboss：启动 jboss，等待到 jboss 启动完，然后让 jboss 在后台运行
- start.tomcat：启动 Tomcat，等待到 Tomcat 启动完，然后让 Tomcat 在后台运行
- stop.jboss：通知 jboss 停止，但是不等到它完成
- stop.tomcat 通知 Tomcat 停止，但是不等到它完成
- upgrade.jbpm.schema：更新数据库中的 jBPM 表到当前版本

要想指定你的配置文件，使用上面的脚本（比如 DB 表结构生成）， 最简单的方法是修改对应的配置文件，在目录${jbpm.home}/install/jdbc。 对应的配置文件会被脚本加载，根据对应的 DB。

下面的参数也可以自定义。

- 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 Tomcat 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。

✧ 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、 数据库

安装脚本也包含了执行数据库的操作 比如创建表，如果你是第一次安装 jBPM， 或者更新数据库使用之前版本的表结构。 删除表也是可选的。

使用任何数据库操作的前提条件是 在${jbpm.home}/install/jdbc 中指定你的连接参数。

2.10.1、 创建或删除表结构

要想创建表结构，执行 create.jbpm.schema 任务 在${jbpm.home}/install 目录下。 作为创建表、约束的一部分，涉及的任务会初始化 JBPM4_PROPERTY 表， 使用当前的引擎版本（key db.version）和 ID 生成器版本 （key next.dbid）。

要想删除表结构，只要执行 drop.jbpm.schema 任务。 注意这个操作会删除 jBPM 表中的 所有数据。

2.10.2、 更新已存在的数据库

要想更新，执行 upgrade.jbpm.schema 任务 在${jbpm.home}/install 目录下。

更新是一个两步操作。前一步是添加额外的表，列或者约束 这些是在新版本中的。 下一步，插入种子数据。

从 4.0 到 4.1，表 JBPM4_VARIABLE 添加了一个新列 CLASSNAME_用来支持设置 流程变量的值的自定义类型，hibernate 的类型映射。 这个列是可为 null 的，因为这个功能在 4.0 中没有支持， 所以没有初始值。

从 4.1 到 4.2，更新过程更有趣一些。

- 一个新表 JBPM4_PROPERTY 被用来保存引擎范围的数据。
- jBPM 版本保存在 JBPM4_PROPERTY 表中 使用 key db.version 用来在未来发布中 精确指定标示符。
- ID 生成策略是完全跨数据库的。 下一个有效的 ID 是通过搜索所有包含主键列的表计算出的， 保存在 key next.dbid 中 在 JBPM4_PROPERTY 表中。
- 流程语言设置为 jpdl-4.0 用于所有已经存在的流程定义，对应 key langid 在表 JBPM4_DEPLOYPROP 中。jPDL 解析器对应 langid 属性来读取流程文档 以此支持向后的兼容。

✧ 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)
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 install/src/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 install/src/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

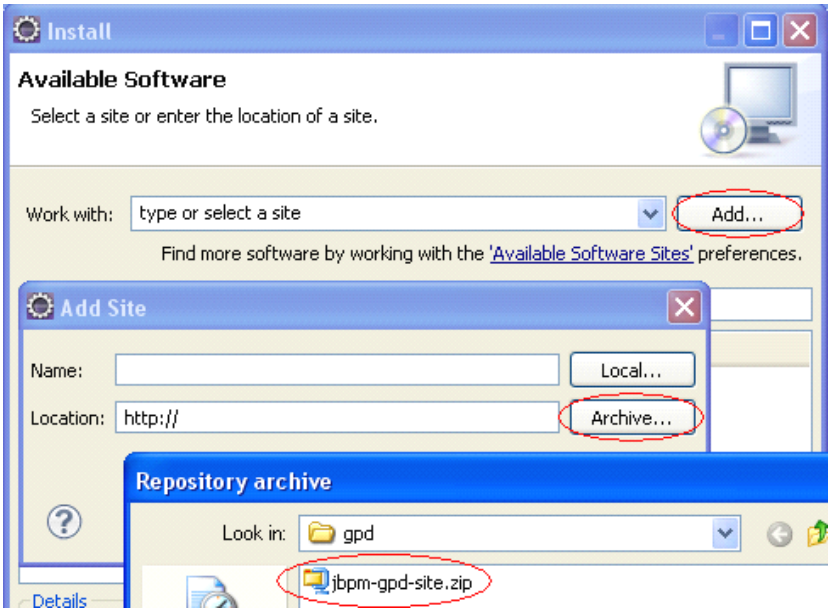

图 2.1. 添加设计器的更新站点

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

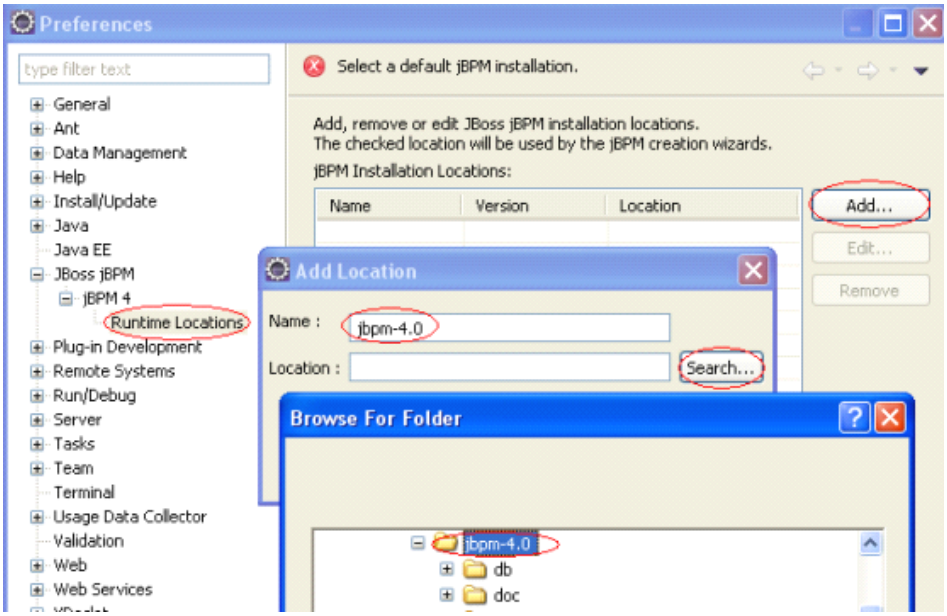

图 2.2. 定义 jBPM 依赖库

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

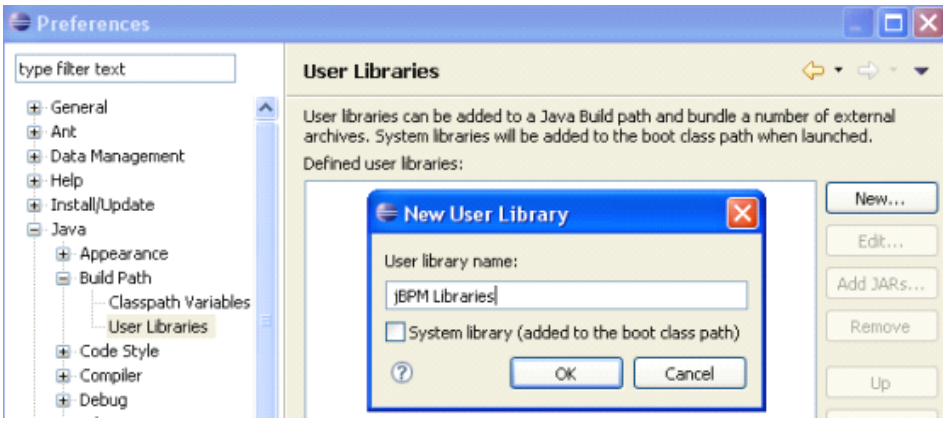

图 2.3. 定义 jBPM 类库

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

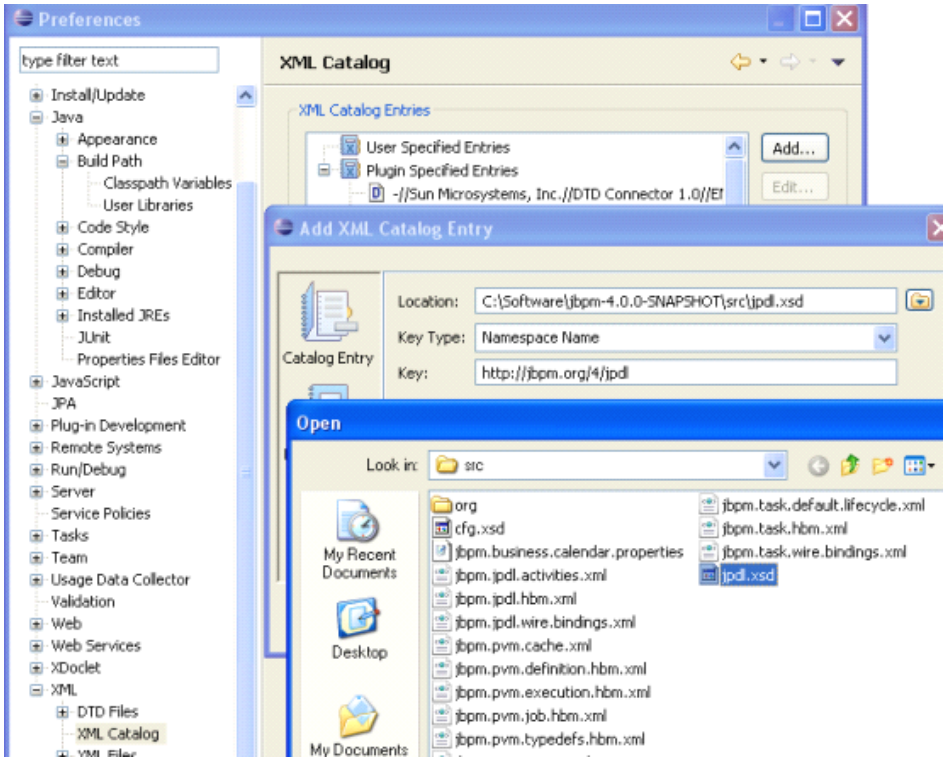

图 2.4. 在目录中添加 jPDL4 模式

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
- 例子项目中的构建文件 build.xml，从包视图拖拽到 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。

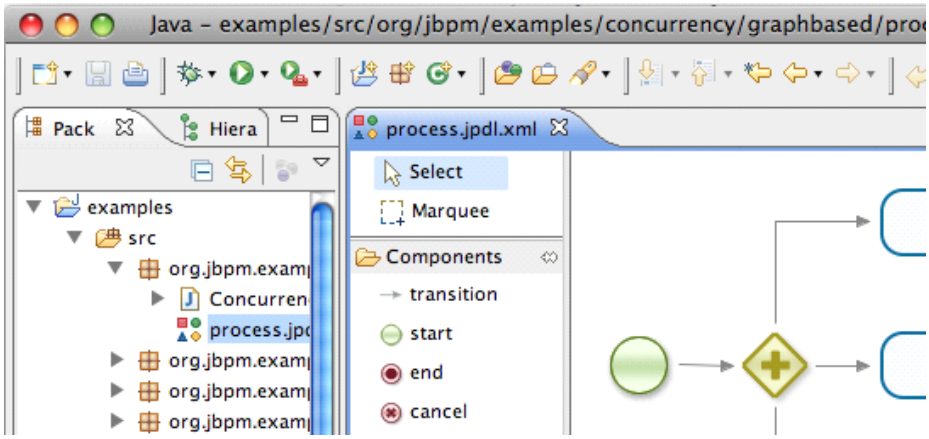

图 3.1. 流程设计器

✧ 3.1、创建一个新的流程文件

Ctrl+N 将打开向导选择器。

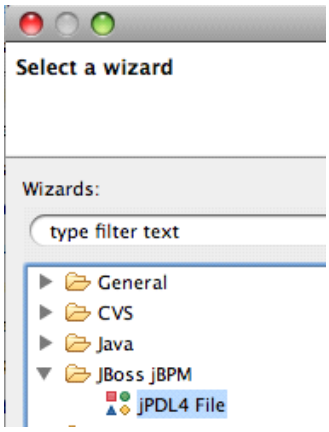

图 3.2. 选择向导对话框

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。

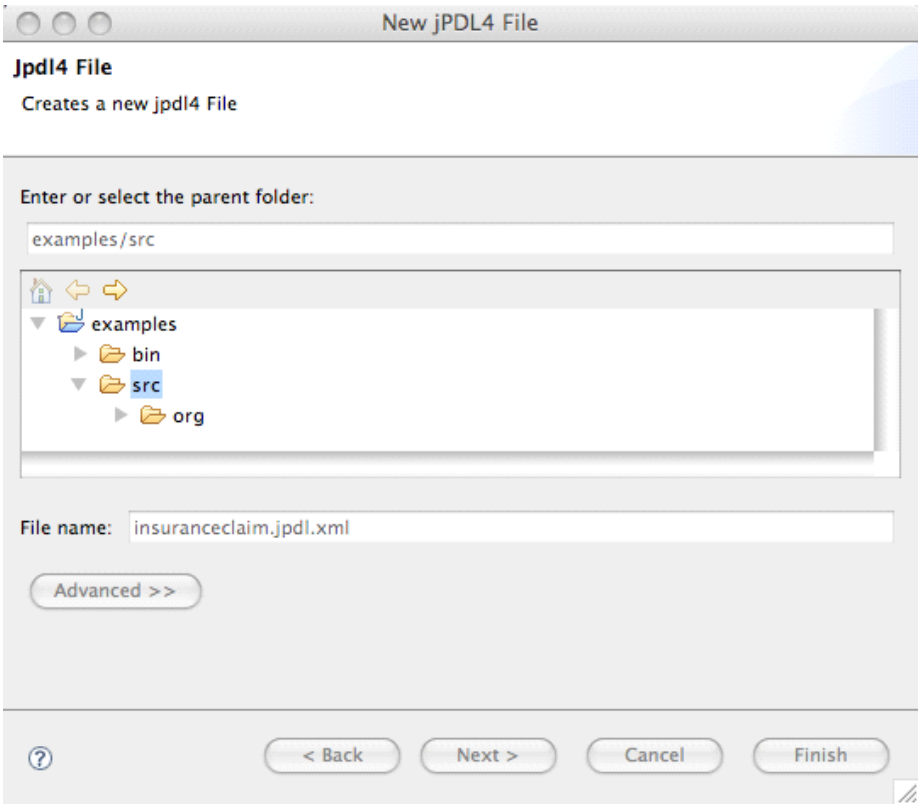

图 3.3. 创建一个新的流程对话框

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。

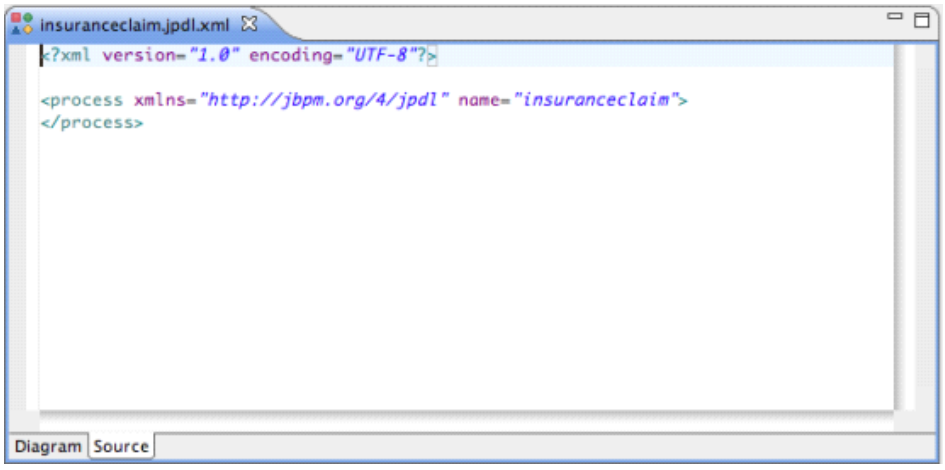

图 3.4. 使用 source 视图编辑 jPDL

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，类， 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

从 4.2 版本开始，jBPM 拥有了一个像 jBPM3 一样的流程类加载器机制。

从流程中引用的类必须至少在下面三种方式之一是 有效的：

- 业务存档中的.class 文件。和 jBPM3 中不同，现在 存档文件的根被用来搜索类资源。 所以当类 com.superdeluxsandwiches.Order 在流程文件中引用时，它会找到，当它在相同的业务归档中 的入门名称 com/superdeluxsandwiches/Order.class 类会被缓存(key 是结合了发布和上下文类加载器)， 所以它应该比 jBPM 3 中执行的更好。
- 在调用 jBPM 的 web 应用中可用的类。 当 jBPM 部署到服务器端的 jboss 或 tomcat 中，jBPM 会找到你的 web 应用或企业应用，调用 jBPM 的类。 这是因为你使用了当前上下文类加载器， 在流程执行过程中查找类时。
- 服务器端可用的类文件。比如像是在 tomcat 和 jboss 的 lib 目录下的 jar。

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。tomcat 下操作相同。参考 install.examples.into.tomcat 和 install.examples.into.jboss 任务。在未来的一个发布版中 我们可能切换到业务存档自身包含的类。

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。

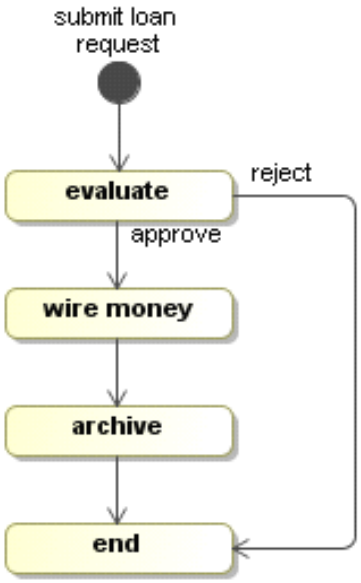

图 5.1. 贷款流程定义示例

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。

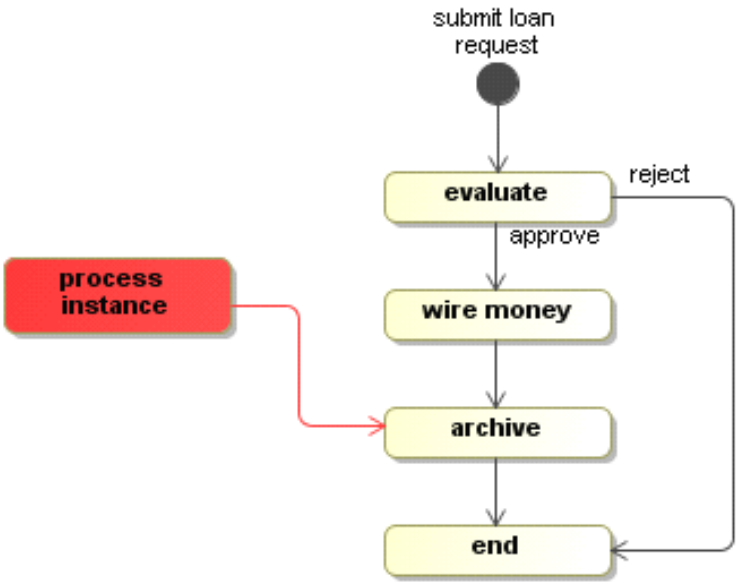

图 5.2. 贷款流程实例的例子

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：

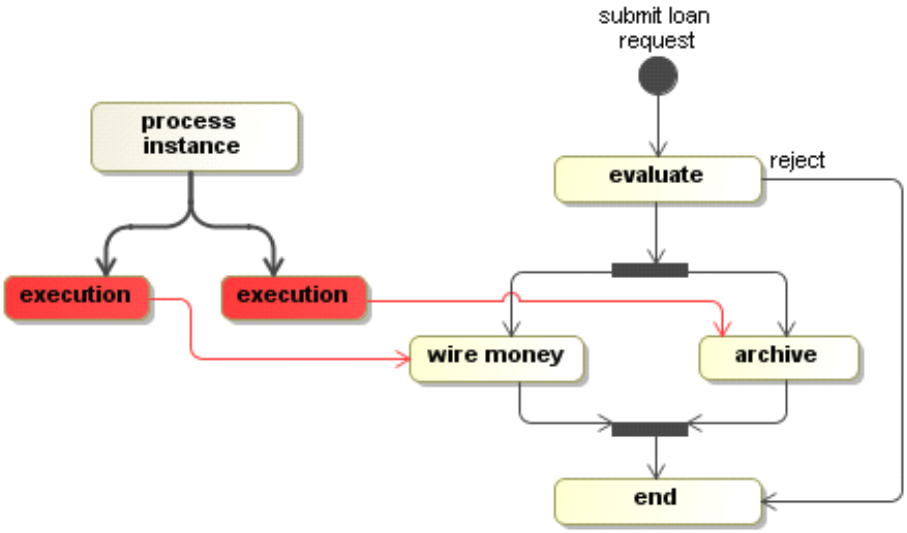

图 5.3. 贷款执行例子

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、 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。

✧ 5.5、启动一个新的流程实例

5.5.1、 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。

5.5.2、 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");
```

5.5.3、 使用 key

我们可以为新启动的流程实例分配一个 key， 这个 key 是用户执行的时候定义的，有时它会作为“业务 key”引用。 一个业务 key 必须在流程定义的 所有版本范围内是唯一的。通常很容易 在业务流程领域找到这种 key。比如，一个订单 id 或者一个保险单号。

```
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();
```

最好使用一个用户定义的 key。特别在你的应用代码中，找到这样的 key 并不困难。提供给一个用户定义的 key，你可以组合流向的 id，而不是执行一个基于流程变量的搜索 - 那种方式太消耗资源了。

5.5.4、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);
```

✧ 5.6、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();
```

要注意上面的解决方式和应用逻辑联系（太）紧密 通过使用真实业务结构的知识。

✧ 5.7、 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);
```

taskSerice 也用来完成任务。

```
taskService.completeTask(taskId);
taskService.completeTask(taskId, variables);
taskService.completeTask(taskId, outcome);
taskService.completeTask(taskId, outcome, variables);
```

这些 API 允许提供一个变量 map，它在任务完成之前作为流程变量添加到流程里。 它也可能提供一个“外出 outcome”，这会用来决定哪个外出转移会被选中。 逻辑如下所示：

如果一个任务拥有一个没用名称的外向转移：

- taskService.getOutcomes() 返回包含一个 null 值集合，。
- taskService.completeTask(taskId) 会使用这个外向转移。
- taskService.completeTask(taskId, null) 会使用这个外向转移。
- taskService.completeTask(taskId, "anyvalue") 会抛出一个异常。

如果一个任务拥有一个有名字的外向转移：

- gtaskService.getOutcomes() 返回包含这个转移名称的集合。
- taskService.completeTask(taskId) 会使用这个单独的外向转移。
- taskService.completeTask(taskId, null) 会抛出一个异常（因为这里没有无名称的转移）。
- taskService.completeTask(taskId, "anyvalue") 会抛出一个异常。
- taskService.completeTask(taskId, "myName") 会根据给定的名称使用转移。

如果一个任务拥有多个外向转移，其中一个转移没有名称，其他转移都有名称：

- taskService.getOutcomes() 返回包含一个 null 值和其他转移名称的集合。
- taskService.completeTask(taskId) 会使用没有名字的转移。
- taskService.completeTask(taskId, null) 会使用没有名字的转移。
- taskService.completeTask(taskId, "anyvalue") 会抛出异常。
- taskService.completeTask(taskId, "myName") 会使用名字为'myName'的转移。

如果一个任务拥有多个外向转移，每个转移都拥有唯一的名字：

- taskService.getOutcomes() 返回包含所有转移名称的集合。
- taskService.completeTask(taskId) 会抛出异常，因为这里没有无名称的转移。
- taskService.completeTask(taskId, null) 会抛出异常，因为这里没有无名称的转移。
- taskService.completeTask(taskId, "anyvalue") 会抛出异常。
- taskService.completeTask(taskId, "myName") 会使用名字为'myName'的转移。

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” 。

✧ 5.8、 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。

✧ 5.9、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。

✧ 5.10. 查询 API

从 jBPM 4.0 开始，一个新的 API 被介绍使用查询系统， 可以覆盖大多数你可以想到的查询。开发者需要编写企业特定查询时 当然也可以使用 Hibenrate。但是对大多数用例来说， 查询 API 是不足够的。 查询可以写成同 ideas 方式，用于主要的 jBPM 概念： 流程实例，任务，发布，历史流程，等等。

比如：

```
List<ProcessInstance> results = executionService.createProcessInstanceQuery()
.processDefinitionId("my_process_definition")
.notSuspended().page(0, 50).list();
```

这个例子返回指定流程定义的所有流程实例， 流程定义不是暂停的。结果支持分页，第一页的前 50 条数据 会被我们获得。

查询任务也可以使用相同的方式完成：

```
List<Task> myTasks = taskService.createTaskQuery()
    .processInstanceId(piId)
    .assignee("John")
    .page(100, 120)
    .orderDesc(TaskQuery.PROPERTY_DUEDATE)
    .list();
```

这个查询会获得指定流程实例，分配给 John 的所有任务，也使用分页，对 duedate 进行逆序查询。

每个服务拥有操作这些统一查询的功能(比如，查询 job 通过 ManagementService，查询完成的 流程实例通过 HistoryService。 可以参考服务的 javadoc 了解这些查询 API 的所有细节。)

第 6 章 jPDL

这章将会解释用来描述流程定义的 jPDL 文件格式。jDPL 是 jBPM 的突出的流程语言。jPDL 的目标是尽量精简和尽可能的开发者友好，在提供所有你期望 从 BPM 流程语言中获得功能的同时。

jPDL 的 schema 文件包含了比这个文档中更多的属性和元素。这个文档解释了 jPDL 中稳定的被支持的部分。 试验性的、不支持的 jPDL 特性可以在开发者指南中找到。

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/4.3/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，	如果省略，key 中的 非字母和非数字的	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

属性	类型	默认值	是否必须	描述
	下划线	字符会被替换为 下划线。		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。	可选	流程的版本号

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/4.3/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/4.3/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/4.3/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。 配置属性和 decision 的 handler 的内容元素 可以在第 6.7 节 “用户代码”中找到。

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 6.7. 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/4.3/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/4.3/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 6.8. 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 6.9. 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/4.3/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。

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 6.10. 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 6.11. 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。

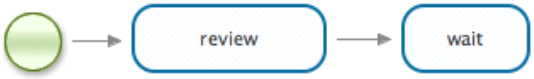

图 6.11. 任务候选人示例流程

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。

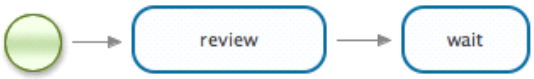

图 6.12. 任务分配处理器的示例流程

```
<process name="TaskAssignmentHandler" xmlns="http://jbpm.org/4.3/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 6.12. 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 6.13. 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

属性	类型	默认值	是否必填	描述
				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。

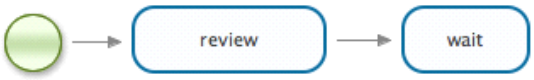

图 6.13. 任务泳道示例流程

任务泳道示例是下面这个流程文件：

```
<process name="TaskSwimlane" xmlns="http://jbpm.org/4.3/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 6.14. 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 6.15. 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 6.16. 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 6.17. 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 6.18. 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 6.19. 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		'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

属性	类型	默认值	是否必填	描述
	串			量中。
lang	字符串	juel	可选	表达式解析时使用的脚本语言。

表 6.20. 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’其中之一必须指定值	子流程中需要获取的 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 6.21. 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。

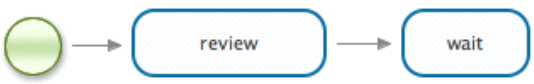

图 6.14. 子流程文档示例流程

```
<process name="SubProcessDocument" xmlns="http://jbpm.org/4.3/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/4.3/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。

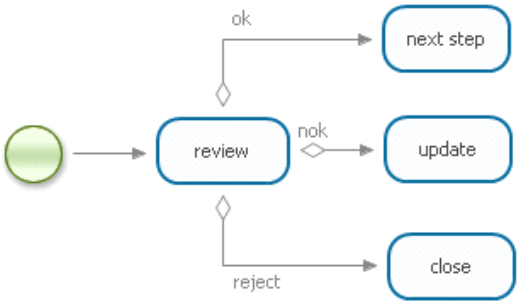

图 6.16. 子流程文档示例流程

```
<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/4.3/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("result", "ok");
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。

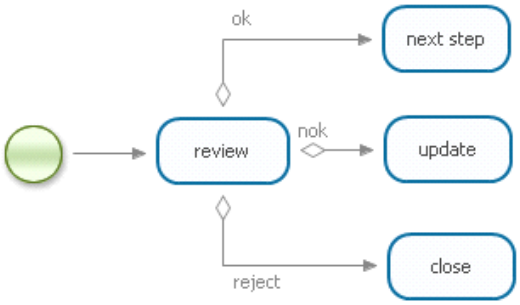

图 6.18. 子流程文档实例流程，对于外出活动

```
<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：

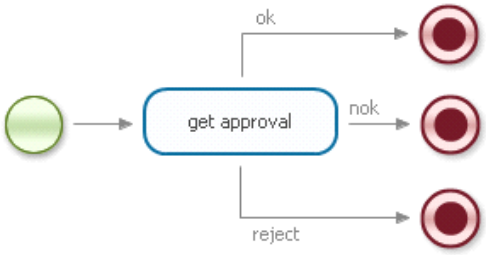

图 6.19. 子流程重审示例流程，为外出活动

```
<process name="SubProcessReview" xmlns="http://jbpm.org/4.3/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/4.3/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 6.22. 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。参考第 6.7.2 节 “用户代码类加载器” 来获得类加载的信息。用户代码对象会被延迟加载， 并作为流程定义的一部分进行缓存。
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 6.23. 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：

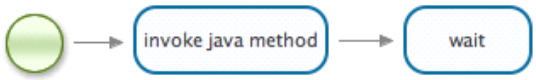

图 6.20. java 任务（task）

```
<process name="Java" xmlns="http://jbpm.org/4.3/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 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 6.24. 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。

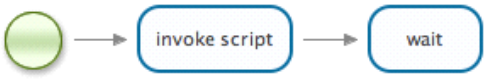

图 6.21. 流程处理的 script.expression 示例

```
<process name="ScriptExpression" xmlns="http://jbpm.org/4.3/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 6.25. 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 6.26. script text 元素

元素	个数	描述
text	1	包含 script 脚本的文本

例如：

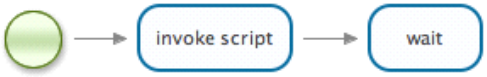

图 6.22. 流程处理的 script text 示例

```
<process name="ScriptText" xmlns="http://jbpm.org/4.3/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， 并将返回的结果保存到流程处理的变量中。

表 6.27. 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 6.28. 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/4.3/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 6.29. mail 属性

属性	类型	默认值	是否必须	描述
template	字		否	引用配置文件中的一个 mail-template 元素。 如果

属性	类型	默认值	是否必须	描述
	符串			没找到， 必须使用子元素在内部指定。

表 6.30. 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/4.3/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 6.31. 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 6.32. 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， 这是因为它们对实现技术细节更感兴趣。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 6.33. 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 6.34. on 元素：

元素	个数	描述

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 6.35. 事件监听器属性：

event-listener 是用户代码所以它可以 像第 6.7 节 “用户代码” 中一样进行配置。

任何自动活动（包括 event-listener）在事件中， 可以指定下面的额外属性：

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/4.3/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，作为流程变量：

```
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 6.36. 任意活动属性：

属性	类型	默认值	是否必填	描述

属性	类型	默认值	是否必填	描述
continue	{sync | async | exclusive}	sync	可选	

- **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/4.3/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

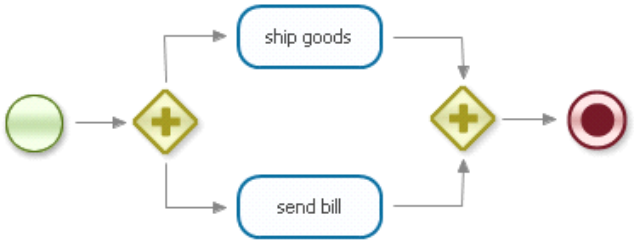

图 6.26. 异步分支实例流程

```
<process name="AsyncFork" xmlns="http://jbpm.org/4.3/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

jPDL 流程语言中的大量元素都引用一个对象， 它的一个接口方法将被调用。这一章介绍了通用属性和元素， 对这些用户编码对象 进行初始化和配置。

- custom
- event-listener
- task 中的 assignment-handler
- decision 中的 handler
- transition 中的 condition

6.7.1. 用户代码配置

表 6.37. 属性：

属性	类型	默认值	是否必填?	描述
class	类名		{class|expr} 其中之一是必须的	全类名。初始化只会进行一次， 用户对象会被作为流程定义的一部分进行缓存。
expr	表达式		{class|expr} 其中之一是必须的	表达式的值会当做目标对象被获得。 表达式会在每次使用时被执行。换句话说， 执行的结果值不会被缓存。

表 6.38. 用户代码配置元素：

元素	数目	描述
field	0..*	描述一个配置值，在用户类使用之前 注入到成员变量中。
property	0..*	描述一个配置值，在用户类使用之前 通过一个 setter 方法进行注入。

表 6.39. field 和 property 的属性：

属性	类型	默认值	是否必填?	描述
name	string		**必填**	field 或 property 的名称。

表 6.40. field 和 property 包含的元素：

field 和 property 元素 都拥有一个子元素， 表示将被注入的值。

元素	数目	描述

元素	数目	描述
string	0..1	a java.lang.String
int	0..1	a java.lang.Integer
long	0..1	a java.lang.Long
float	0..1	a java.lang.Float
double	0..1	a java.lang.Double
true	0..1	Boolean.TRUE
false	0..1	Boolean.FALSE
object	0..1	会通过反射初始化的对象

表 6.41. 基本类型 string, int, long, floatand double 的属性：

属性	类型	默认值	是否必填？	描述
value	text		**必填**	text 值会被解析成期望的类型

6.7.2. 用户代码类加载器

流程定义被缓存了。默认情况，所有用户代码对象 都会作为流程定义的一部分被缓存。 对于所有通过类名引用的对象， 都会在解析期间被初始化。这就意味着，这些对象 不允许保存非无状态化的数据（比如可以改变）。 实际上因为这些对象实际都常常是不会改变的。 如果你需要使用“动态”数据在你的用户代码中，你通常可以 使用流程变量（或调用 Environment.get(xxx)）。

表达式中引用的对象 是动态计算的。

开发只能拿也解释了未支持的属性， 来预防用户对象的缓存。

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。

✧ 7.3. 更新持久化流程变量

(jBPM 4.3 中新添加的功能)

在 customs, event-handlers 和其他 用户代码中，你可以获取流程变量。当一个流程变量作为 持久化的对象被保存时，你可以直接更新反序列化的对象， 而不需要再次进行保存。jBPM 会管理反序列化的流程变量 如果出现了修改就会自动更新它们。比如（参考报 org.jbpm.examples.serializedobject）， 查看一个 custom 活动行为内的代码片段：

```
public class UpdateSerializedVariables implements ActivityBehaviour {

  public void execute(ActivityExecution execution) {
    Set<String> messages = (Set<String>) execution.getVariable("messages");
    messages.clear();
    messages.add("i");
    messages.add("was");
    messages.add("updated");
  }
}
```

当事务提交时，用户代码会被调用， 更新消息集合会自动被更新到数据库中。

当从 DB 读取以序列化格式保存的流程变量时， jBPM 会监控反序列化的对象。在事务提交之前，jBPM 会反序列化并自动更新变量，如果必要的话。 jBPM 会忽略更新反序列化对象，如果其他对象

被设置到那个作用域 （这可能是其他类型）。jBPM 也会略过更新 那些数值没有发生变化的的反序列化对象。这些检测会查看 如果一个对象被修改了，基于计算再次序列化对象的字节数组， 和从数据库中原来读取的字节数组进行比较。

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。

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 Configuration 配置

✧ 9.1. 工作日历

如果希望自定义工作日历， 需要导入的默认工作日历配置 使用你自定义的工作日历进行替换。比如。

```
<jbpm-configuration>

  <import resource="jbpm.default.cfg.xml" />
  ...

  <process-engine-context>
    <business-calendar>
      <monday    hours="9:00-18:00"/>
      <tuesday   hours="9:00-18:00"/>
      <wednesday hours="9:00-18:00"/>
      <thursday  hours="9:00-18:00"/>
      <friday    hours="9:00-18:00"/>
      <holiday period="01/02/2009 - 31/10/2009"/>
    </business-calendar>
  </process-engine-context>

</jbpm-configuration>
```

✧ 9.2. Email

TODO 这里需要补充如何自定义 jbpm.mail.properties

附录 A. 修改日志

修订历史

修订 jBPM-4.3 2009-12-29

1. revision: 5849 ~ 5949
2. 第 6 章 *jPDL* 使用新的命名空间：http://jbpm.org/4.3/jpdl
3. 第 5.7 节 “TaskService 任务服务” 添加对任务完成执行转移的介绍。
4. 第 7 章 *Variables 变量* 添加“更新持久化流程变量”。

修订 jBPM-4.2 2009-11-01

1. revision: 5613 ~ 5835
2. 第 6 章 *jPDL* 使用新的命名空间：http://jbpm.org/4.2/jpdl
3. 第 6.7 节 “用户代码” 介绍如何使用用户自定义代码。
4. 第 2.10 节 “数据库” 加强了介绍数据库创建，更新部分的步骤。
5. 第 1 章 *导言* 修改介绍章节中的小问题。
6. 第 2.4 节 “安装脚本” 详细介绍安装脚本。
7. 第 5.5.3 节 “使用 key” 介绍业务 key。
8. 第 5.10 节 “查询 API” 介绍查询 api。
9. 删除 Email 一章。
10. 第 4.2 节 “部署 java 类” 介绍发布业务类。

修订 jBPM-4.1 2009-09-01

1. revision: 5288 ~ 5600
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。 第 9 章 *Configuration 配置*。
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 ???。

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 5.6 节 “执行等待的流向”

补充： 第 5.8 节 “HistoryService 历史服务”

补充： 第 5.9 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。