[**jBPM-jPDL学习笔记—框架设计简介(一)**](http://linliangyi2007.javaeye.com/blog/176340)

**关键字: jbpm jpdl 框架设计 工作流**

**相关资料：**

[《jBPM学习笔记(V3.2环境部署)》](http://linliangyi2007.javaeye.com/../../blog/176339)

[《jBPM学习笔记(流程设计与控制)》](http://linliangyi2007.javaeye.com/../../blog/176345)

**概述:**

         jBPM（JBoss Business Process Management业务流程管理），它是覆盖了业务流程管理、工作流、服务协作等领域的一个开源的、灵活的、易扩展的可执行流程语言框架。

         jPDL是构建于jBPM框架上的流程语言之一。在jPDL中提供了任务（tasks）、待处理状态 （wait states）、计时器（timers）、自动处理（automated actions）…等术语，并通过图型化的流程定义，很直观地描述业务流程。

         jPDL可以部署于Java语言的任何应用环境下，JavaSE Application(Swing/SWT)、JavaEE Application(Application Server)

**jPDL分发包组成**

目前jPDL的最新版本是3.2.2，您可以在以下地址：<http://labs.jboss.com/jbossjbpm/jpdl_downloads/>

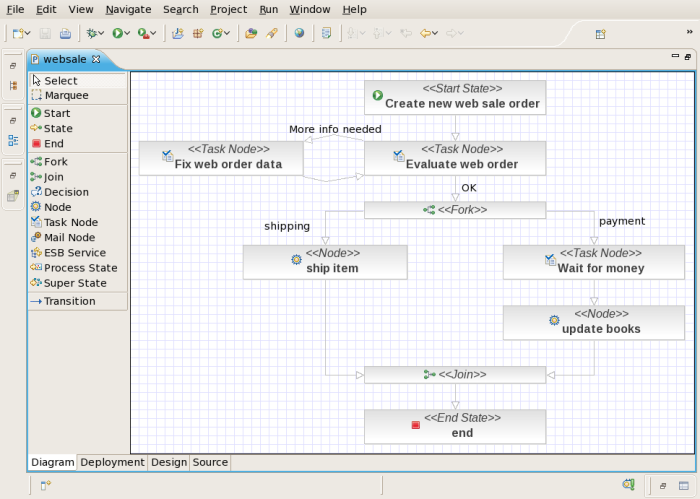
获得官方下载包。下载包分为两种，一种是标准包(jbpm-jpdl-3.2.2.zip)只有20M左右；一种是套件包(jbpm-jpdl-suite-3.2.2.zip)将近80M。两者最大的区别是套装包自带一个已经配置好的jboss服务，下载解压缩后，可直接运行。

**jPDL的核心程序包(jbpm-jpdl.jar)**

它是一个典型的jar文件，它涵盖对流程定义和运行时流程实例上下文环境的操作API。在最简化的业务流程样例中，jPDL将流程定义存储与XML格式的定义文件中，在运行时载入内存，在内存里记录完整的业务实例和执行过程的环境变量，并完成全部操作。显然，在实际应用中，流程信息必须被持久化的存储于数据库中。jPDL使用了Hibernate作为其对象持久化的API。

**Graphical Process Designer插件(jbpm-jpdl-designer-3.1.2.zip)**

它是一个基于Eclipse的图型编辑插件。使用它可以对业务流程进行可视化的定制，而无须手动编辑XML文件。该插件还提供了流程定义上传功能，可以将新定义的流程发布到在线运行的jPDL系统上。



**jPDL身份包(jbpm-identity.jar)**

         它是jPDL的一个轻量级扩展包，简单实现了组Group、用户User、许可权Permission等用户身份认证和鉴权机制。如果您的应用环境只需要简单的用户认证鉴权管理，那么它十分适合您。

**jPDL Web控制台(jbpm-console.war**)

         它是一个标准Web应用（使用了JSF技术），提供了对jPDL平台执行情况的监控程序（对jPDL数据库的监控）。管理员可以通过该程序方便的维护jPDL的数据库，管理流程定义和流程实例的执行情况。同时，它还提供了一个简单的Getting started DEMO，方便初学者了解jPDL。

**其他相关目录**

          在解压后的目录中，还有一些文件是很有用的

**/config** 该目录存放了jPDL的默认配置文件 ，包括hibernate.cfg.xml 、jbpm.cfg.xml（jPDL的主配置文件，默认是空的）、jbpm.mail.templates.xml（邮件模板配置文件）、log4j.properties（日志配置文件）

**/db**该目录存放了jPDL针对不同数据库的表初始化sql文件。这里要说的是，jbpm.jpdl.mysql.sql在MySql下批量运行时，会出现错误提示，原因是语句后缺少”**;**”号，需要修订后才能使用。

**/** **designer（套件包中才有）**该目录存放了Graphical Process Designer插件

**/examples**该目录存放了jPDL的简单范例

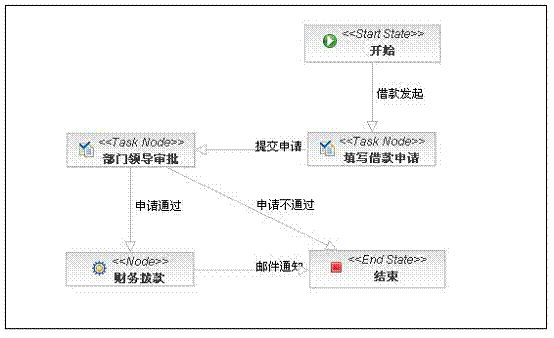
**/doc**该目录存放jPDL 的相关说明文档，包括用户指南、Java API文档等

**/server （套件包中才有）**该目录存放了配置好的jBoss和控制台程序，Windows下双击start.bat可立即执行。数据库使用HSQL，存放于/server/server/jbpm/data下。

**jPDL业务流程模型**

**理解现实生活中的业务流程**

       下面我们通过一个简单的部门借款流程来了解现实中的业务流程概念，流程图如下：



上图表达了一个抽象的流程定义（Process Definition）。之所以称之为抽象，是因为它没有表示确切的执行者。当上述的借款人被具体化为“张三”，部门主管就成为张三的部门经理“李四”，而财务则明确为“王五”的时候，这个流程定义就被具体化，成为了一个流程实例（Process Instance）。此外，过程中的相关人员我们称之为参与者（Actor）；过程中需要参与者介入的环节称之为任务（Task）；每个任务在流程实例中的具体化称之为任务实例（Task Instance）；从一个任务（结点）到另一个任务（结点）的转化过程叫做流转(Transition**)；**而在流程中，由程序预先设定的行为如发送邮件，我们称之为活动（Action）。         通过上述样例，我们了解了业务流程中一些专有名词和概念。接下来，我们要从程序设计的角度来讲述jPDL中定义的流程对象模型。

         在jPDL中，对流程的建模被划分为两大部分，一是针对“流程定义”的静态模型部分；二是针对具体“流程实例”的运行时动态模型。在这一章节，我们将详细的讲述jPDL的对象及对应的数据库设计。

**静态的流程定义模型**

|  |  |
| --- | --- |
| **编号** | PD-001 |
| **对象** | 流程定义实体（Process Definition） |
| **描述** | 流程定义实体是对一个流程抽象的对象化定义，如图-002。一套系统中，用户可以定义保存多个流程定义实体，如：报销流程定义、请假流程定义、人事录用流程定义等。每个流程定义还可以有多个不同的版本，如:针对同样的报销流程可以有修订前和修订后的两个流程定义，同时存储于jPDL数库中。用户可以通过流程名称和版本号获取相应的流程定义。在默认请况下，系统启用最新的流程定义。 |
| **Java对象** | org.jbpm.graph.def.ProcessDefinition |
| **数据库表** | JBPM\_PROCESSDEFINITION该表存储流程定义的通用信息，如：流程名称、版本号 |
| **表关联说明** | JBPM\_PROCESSDEFINITION表中，每条记录有自己的数据库流水号      ID\_JBPM\_PROCESSDEFINITION的外键（Foreign Keys）：   * startstate\_ 同JBPM\_NODE（流程结点）表关联。此外键指向流程定义中的起始结点ID |

|  |  |
| --- | --- |
| **编号** | PD-002 |
| **对象** | 流程结点（Node） |
| **描述** | 流程结点是对流程中的过程环节/行为的抽象对象化定义，如图-002中的每个方框即代表一个结点。结点有两个主要职责：一，实现某个指定行为，这在jBPM中就是执行一段制定的Java代码；二，转递、维持流程的延续，直至达到最终结点。在jPDL的结点设计中，系统开放了充分的用户行为模型，程序员可以基于这些行为接口，实现自定义的业务功能。在jPDL中，系统预定义了一系列的Node类型，同时也允许程序员开发定义自己的Node类型。 |
| **Java对象** | org.jbpm.graph.def.Node |
| **数据库表** | JBPM\_NODE该表存储结点的定义信息。 |
| **表关联说明** | JBPM\_NODE表中，每条记录有自己的数据库流水号    ID\_JBPM\_NODE的外键（Foreign Keys）：   * processdefinition\_ - 此外键说明该Node从属的流程定义对象ID * subprocessdefinition\_- 此外键指定了一个由该Node发起的子流程。在主流程离开当前结点之前，该子流程必须完成执行过程。action\_ - 此外键指定该结点动作类的ID * superstate\_ - 该外键是一个自关联键，指向结点的上级父结点ID。一个子流程中的多个结点从属于同一个父流程结点。 * decisiondelegation\_ – 该外键指定了结点委派的判定类ID（所谓委派判定类是指根据用户的业务逻辑，判定流程的下一个流向的Java Class） |

**Node类型祥解：**

**任务结点（task-node）**任务结点是代表由人介入的一个或多个任务。因此当流程运行到一个任务结点时，会生成“任务实例对象（**task instances**）”，并添加到参与人的任务列表中，之后结点会处于等待状态，直到参与人完成他们的任务，并激活流程继续向下执行。

**状态结点（state）**

状态结点是一个典型的等待状态。同任务结点不同的是，状态结点不会向任务列表添加任务实例。当业务进程需要等待外部系统的干预时，这种结点是很有用的。假设如下情况：在进入该结点时，通过node-enter事件向外部系统发送一个消息，然后结点进入等待状态；当外部系统完成处理，并回送一个消息，这将导致触发一个token.signal()方法的运行，该方法重新激活正在等待的流程继续下行。

**判定结点（decision）**判定节点的作用就同它的命名一样，用来决定业务流程的走向。有两个不同裁决模式，两者的区别在“谁”来做决定：是由流程内部的变量，还是由外部实体来提供决定的依据。当需要对流程执行方向做判定时，就要使用“判定结点（decision）”。有两种方法来指定判定条件。最简单的是在转向（transitions）中添加条件元素，条件可以是能返回boolean值的EL表达式或者beanshell脚本。在运行过程中，判定结点将首先轮训有条件设定的转向（leaving transitions），轮训的顺序是按照XML文件中指定的。当找到第一个条件返回为true的转向时，该出口将被选中。如果所有的表换中的条件判定都是false，则选择XML文件中排在第一位的转向作为出口。还有一种途径是在判定结点上定义一个返回转向名称的表达式，通过表达式计算返回的名称，决定选择哪个transition.另一方式是在结点上设定“处理（handle）”元素。在结点上指定一个实现了DecisionHandler接口的Java处理类，该类通过返回选定的transition的名称来决定流程的出口方向。

当判定结点的出口是由外部程序来给出的时候，建议使用多个transition或者具有等待状态的结点。可以通过外部的触发器结束一个等待状态并提供一个transition的判定。

**分支结点（fork）**

分支结点的作用是将单个执行流程分裂成多个并发的执行流程。默认的行为是为每个子流程生成一个子令牌，并建立子令牌和主流程根令牌之间的父子关系。

**合并结点（join）**

         相对于fork结点的分支，join结点将分支收拢。默认的行为模式是当所有的分支（由同一个fork衍生出来的分支）都到达该结点的时候，join结点将结束这些分支上的子token，并通过token上的父子关系找到上一级流程的token，将此token通过唯一的transition传播下去。如果只有分支中的部分token到达时，join结点将处于等待状态。

**普通结点（node）**普通类结点主要用于提供用户定制自己的程序代码。普通结点拥有一个action子元素，当流程到达该结点时，这个action就会被执行。可以通过实现ActionHandler接口来执行你想要的任何代码。此外普通结点也一样要负责流程的延续。

在流程图上，普通结点用来表达一个用户关心的、与业务相关的处理逻辑；相比而言Action（下文中将会提到）则允许添加业务逻辑以外的程序处理，这些程序处理在流程图上是不可见的，也是业务流程分析所不用关心的。

|  |  |
| --- | --- |
| **编号** | PD-003 |
| **对象** | 流程转向（Transitions） |
| **描述** | 流程转向是描述流程中从一个结点到另一个结点的状态转换过程，因此一个转向一定有一个源结点和一个目标结点。 在jPDL中transition的命名是通产是唯一的，结点依靠transition的命名来区别到下一结点的路径，当一个Node中存在有多个同名的transition的时候，第一个transition将会被选中。结点转向的过程中，排在transition列表第一位置的即是默认的transition。 |
| **Java对象** | org.jbpm.graph.def.Transition |
| **数据库表** | JBPM\_TRANSITION该表存储流程定义中的转向对象。 |
| **表关联说明** | JBPM\_TRANSITION表中，每条记录有自己的数据库流水号    ID\_JBPM\_TRANSITION的外键（Foreign Keys）：   * processdefinition\_ - 该外键指向transition所属的流程定义对象ID * from\_ - 该外键指向transition的源结点ID * to\_ -该外键指向transition的目标结点ID |

|  |  |
| --- | --- |
| **编号** | PD-004 |
| **对象** | 动作（Actions） |
| **描述** | Actions是指一系列的在流程事件中运行的Java代码。流程图是软件需求的传达的重要手段，但它只是软件需求的一个投影，隐藏了很多技术实现的细节。Actions则是向流程图添加技术实现细节的一种机制，它可以很好的补充和修饰流程图。这意味着在不改变流程图结构的情况下，可以将Java的代码与之关联。Actions通过事件（Events）同流程绑定，常用的主要事件包括：进入结点、离开结点、进行转向。请注意，同Events关联的Actions和处于Node中的Actions是有不同的。处于Events中的Actions是通过事件触发执行的，它是典型的观察者模式，是无法影响流程控制的流向。而处于Node中的Action则要承担起流程传递的责任。此外，Actions是可以命名的。可以通过Actions的命名在任何地方引用该Actions。命名的Actions可以作为主流程定义的公用子元素。这个功能可以提高对Actions定义的复用。 |
| **Java对象** | org.jbpm.graph.def.Action |
| **数据库表** | JBPM\_ACTION该表存储流程定义中的动作对象。 |
| **表关联说明** | JBPM\_ACTION表中，每条记录有自己的数据库流水号ID\_    JBPM\_ACTION的外键（Foreign Keys）：   * processdefinition\_ - 指明action所属的流程定义ID * event\_ - 指明action绑定的event ID * actiondelegation\_ - 指明action的委派对象ID * exceptionhandler\_ - 指定action的异常处理ID * referencedaction\_ - 指向当前action引用的action的ID * timeraction\_ - 指向当前action引用的计时器action |

|  |  |
| --- | --- |
| **编号** | PD-005 |
| **对象** | 事件（Events） |
| **描述** | 事件表示流程执行中的某个特定的时刻。在流程执行的过程中，通过jBPM的引擎触发事件，这通常发生在jbpm计算后续状态的时候。事件总是和流程中的元素绑定，这些元素包括：流程定义（process definition）、流程结点（node）、流程转向（transition）和任务（task）。不同的元素会触发不同的事件，拿node元素来说，有 node-enter事件和node-leave事件。事件是action的钩子，一个事件可以回调一系列的action。当jBPM引擎触发事件的时候，事件中绑定的action就会被执行。在jBPM中，事件模型是可传播的。一个子元素触发的事件，将逐层向上传播到顶层的流程定义元素。这样的设计使得事件可以被集中化处理。 |
| **Java对象** | org.jbpm.graph.def. Event |
| **数据库表** | JBPM\_EVENT该表存储流程定义中的事件对象，这些事件与相关的action绑定。 |
| **表关联说明** | JBPM\_EVENT表中，每条记录有自己的数据库流水号ID\_     * JBPM\_EVENT的外键（Foreign Keys）： * processdefinition\_  - 指向event所属的流程定义ID * node\_  - 指向event所属的node ID * transition\_  - 指向event所属的transition ID * task\_  -  指向event所属的任务ID |

|  |  |
| --- | --- |
| **编号** | PD-006 |
| **对象** | 任务（Task） |
| **描述** | 任务作为流程定义中的一部分，定义了如何创建任务实例（Task instances）以及任务分配。任务可以在任务结点（Task-Node）和流程定义（process-definition）两个地方定义。通常会在一个任务结点中定义一个或多个的任务。在这种情况下，任务结点表示在流程中等待用户处理的交互操作。业务流程会停留在任务处，等待用户完成相应的操作之后继续。任务也可在流程定义中声明。通过任务名，可以找到相应的任务定义，并引用他们。实际上任何有命名的任务都可以通过名字被引用。在整个的流程定义中，任务名必须是唯一的。此外，还可以给任务设定优先级，这些优先级属性将成为任务实例化时的默认设置。任务实例可以在之后的运行中改变优先级。. |
| **Java对象** | org.jbpm.taskmgmt.def.Task |
| **数据库表** | JBPM\_TASK该表存储流程定义中声明的任务对象。 |
| **表关联说明** | JBPM\_TASK表中，每条记录有自己的数据库流水号ID\_    JBPM\_TASK的外键（Foreign Keys）：   * processdefinition\_  -指向task所属的流程定义ID * taskmgmtdefinition\_ - 指向task引用的TaskMgmtDefintion对象ID * tasknode\_ - 指向task所属的任务结点ID * startstate\_ -指向task所属流程的起始结点ID * assignmentdelegation\_ - 指定task委派处理的class对象的ID * swimlane\_ - 指定task要授权的swimlane（角色）ID * taskcontroller\_ - 指定一个task赋值的委托类 |

上述我们剖析了jPDL业务流程定义中最重要的几个静态对象，说明了它们的概念、作用、行为特征及关联关系。在正式的运行环境中，这些对象定义被jPDL持久化于数据库中，当用户发起一个业务流程时，被系统实例化。在下面的小节中，我们要讲述流程被实例化后的数据模型。

**具体化的流程实例模型**

本章节介绍流程实例中最重要的三个对象：流程实例（Process Instance）、令牌（Token）、任务实例（Task Instance）.

|  |  |
| --- | --- |
| **编号** | PI-001 |
| **对象** | 流程实例（Process Instance） |
| **描述** | 流程实例是流程定义的运行时状态，它记录了一个流程运行的起始时间、结束时间等状态信息 |
| **Java对象** | org.jbpm.graph.exe.ProcessInstance |
| **数据库表** | JBPM\_PROCESSINSTANCE该表存储运行时的流程对象信息。 |
| **表关联说明** | JBPM\_PROCESSINSTANCE表中，每条记录有自己的数据库流水号    ID\_JBPM\_PROCESSINSTANCE的外键（Foreign Keys）：   * processdefinition\_ - 该流程实例对应的流程定义的ID * roottoken\_ - 该流程实例引用的跟令牌 * superprocesstoken\_ - 如果当前流程是某个流程实例的子流程，则指向该父流程的ID |

|  |  |
| --- | --- |
| **编号** | PI-002 |
| **对象** | 令牌（Token） |
| **描述** | 令牌代表一个流程执行的路径，它维护着流程定义中各个结点的指针，同时记录了流程当前执行的位置。 |
| **Java对象** | org.jbpm.graph.exe.Token |
| **数据库表** | JBPM\_TOKEN该表存储运行时令牌对象信息，它表示当前流程执行到的位置。 |
| **表关联说明** | JBPM\_TOKEN表中，每条记录有自己的数据库流水号ID\_    JBPM\_TOKEN的外键（Foreign Keys）：   * processinstance\_  -  指向该令牌所属的流程实例的ID。 * subprocessinstance\_ - 指向该令牌所属的子流程实例ID。 * node\_  - 令牌所在（所指向的）的当前节点ID. * parent\_ - 如果当前令牌是一个分支流程或子流程的令牌，则该值指向其父令牌ID。 |

|  |  |
| --- | --- |
| **编号** | PI-003 |
| **对象** | 任务实例（Task Instance） |
| **描述** | 一个任务实例对象可以分配给指定的操作者处理，当任务完成后，将触发流程继续下行。任务实例的生命周期很简单，生成实例-->开始处理-->任务结束。从弹性设计考虑出发，任务的分配不属于实例生命周期的一部分，任务是否被分配是不会影响其生命周期过程的。当流程进入一个任务结点的时候，结点上任务被实例化；用户通过访问系统的界面组件，获取自己的任务列表，可以进行任务委派、处理或直接结束任务。任务实例记录着任务创建的时间、结束时间，这些信息可以通过TaskInstance对象中的getter方法获取。一个任务实例的结束时间标识它的完成，在后绪的任务查询中，该实例不会被取出，但实例的记录仍保留于数据库中。 |
| **Java对象** | org.jbpm.taskmgmt.exe.TaskInstance |
| **数据库表** | JBPM\_ TASKINSTANCE该表存储运行时任务实例信息。 |
| **表关联说明** | JBPM\_ TASKINSTANCE表中，每条记录有自己的数据库流水号ID\_    JBPM\_ TASKINSTANCE的外键（Foreign Keys）：   * task\_ - 指向定义该实例的任务ID * token\_ - 指向任务实例对应的令牌ID * swimlaneinstatnce\_  - swimlane 实例，表示处理该任务的角色 * taskmgmtinstance\_  -  管理该任务的模块实例ID |

**jPDL语言主要XML元素大纲**

本章节将介绍jPDL语言中，主要几个XML元素的属性及相关描述。

**<process-definition>(流程定义)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **name** | 属性 | 可选 | 流程的名称 |
| **swimlane** | 元素 | [0..\*] | 在流程中使用的swimlanes 。swimlanes是过程中的用户角色，可以用于任务的委派分配 |
| **start-state** | 元素 | [0..1] | 流程的起始状态结点 |
| **{end-state|state|node| task-node|process-state| super-state|fork|join|decision}** | 元素 | [0..\*] | 流程定义中出现的各类结点 |
| **event** | 元素 | [0..\*] | 流程中的事件，可看成是action的容器 |
| **{action|script|create-timer |cancel-timer}** | 元素 | [0..\*] | 全局定义的action，可以在events和transitions中通过命名引用 |
| **task** | 元素 | [0..\*] | 全局定义的task，可以在action中使用命名来引用. |
| **exception-handler** | 元素 | [0..\*] | 异常处理类，用来处理委派类抛出的异常. |

**<node>(节点)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **{action|script|create-timer |cancel-timer}** | 元素 | 1 | 定制的结点动作 |
| **common node elements** |  |  | 参考common node elements |

**<common node elements>(****通用结点元素)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **name** | 属性 | 必须 | 结点名称 |
| **async** | 属性 | { true | false }, false is the default | 设置成true，则结点以异步方式运行 |
| **transition** | 元素 | [0..\*] | 离开结点的转向.每个转向必须有一个可区别的名字。最多只允许一个转向是没有命名。第一个转向指定为默认的转向，当没有给结点指定转向时，将使用默认转向。 |
| **event** | 元素 | [0..\*] | 支持的事件类型（node-enter | node-leave） |
| **exception-handler** | 元素 | [0..\*] | 针对委派类抛出异常的处理 |
| **timer** | 元素 | [0..\*] | 指定一个计数器监视结点上的执行过程 |
|  |  |  |  |

**<start-state>(起始状态结点)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **name** | 属性 | 可选 | 起始结点名称 |
| **task** | 元素 | [0..1] | 此任务会创建一个流程实例或者捕获流程的初始化者 |
| **event** | 元素 | [0..\*] | 支持结点的事件：{node-leave} |
| **transition** | 元素 | [0..\*] | 流程转向，必须有唯一命名 |
| **exception-handler** | 元素 | [0..\*] | 对委派类异常的处理者 |

**<end-state>(终止状态结点)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **name** | 属性 | 必须 | 终止结点名称 |
| **event** | 元素 | [0..\*] | 支持的结点事件：{node-enter} |
| **exception-handler** | 元素 | [0..\*] | 对委派类异常的处理者 |

**<task-node>(任务结点)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **signal** | 属性 | 可选 | 值类型有：{unsynchronized|never|first|first-wait|last|last-wait}, 默认为last. 该属性定义了任务完成对流程延续的效果。 |
| **create-tasks** | 属性 | 可选 | 值类型有：{yes|no|true|false},默认值为true，也可以设置为false。用户可以在结点的node-enter事件中定义自己action，由action来生成任务，并且将此属性设置为false。 |
| **end-tasks** | 属性 | 可选 | 值类型有：{yes|no|true|false},默认值为false. 当设置为true的时候，所有未结束任务在node-leave 事件中会被关闭。 |
| **task** | 属性 | [0..\*] | 当流程执行到当前结点时，将生成任务列表中的任务。 |
| **common node elements** |  |  | 参考common node elements |

**<fork>(分叉)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **common node elements** |  |  | 参考 common node elements |

**<join****>(合并)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **common node elements** |  |  | 参考 common node elements |

**<decision>(判定结点)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **handler** | 元素 | handler或者transition conditions二者必选其一 | 实现org.jbpm.jpdl.Def.DecisionHandler接口的Java类名 |
| **transition conditions** |  | handler或者transition conditions二者必选其一 | 判定结点将按顺序遍历每个leaving transitions中的条件判定。找到的第一个判定为true的leaving transitions将被选中。如果所有leaving transitions中的判定均为false，则选择第一个默认的leaving transitions. |
| **common node elements** |  |  | See common node elements |

**<event>(事件)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **type** | 属性 | 必须 | 事件的类型表明事件在元素中发生的位置。 |
| **{action|script| create-timer|cancel-timer}** | 元素 | [0..\*] | 与事件绑定的动作列表。 |

**<transition>(转向)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **name** | 属性 | 可选 | 转向的名称，每个转向必须有一个可区分的名字。 |
| **to** | 属性 | required | 目标节点的带层次的名称。如： phase one/invitemurphy |
| **condition** | 属性/元素 | optional | 条件表达式或条件属性 |
| **{action|script| create-timer| cancel-timer}** | 元素 | [0..\*] | 进行转向时执行的动作。动作不需要放置于事件中，因为在转向（transition）中只有一个地方可以运行动作。 |
| **exception-handler** | 元素 | [0..\*] | 对委派类异常的处理者 |

**<action>(动作)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **name** | 属性 | 可选 | 动作的名称。当动作被赋予名称的时候，它可以从流程定义中查找获取。这对运行时action和action的一次性声明是很有用的。 |
| **class** | 属性 | 此属性同 ref-name 属性及expression属性，三者只能选其一 | 实现了org.jbpm.graph.def.ActionHandler 接口的Java类名 |
| **ref-name** | 属性 | 此属性同 class属性及expression属性，三者只能选其一 | 所引用的action的名称。当该属性被指定的时候，action标签中的内容将被忽略。 |
| **expression** | 属性 | 此属性同 class属性及ref-name属性，三者只能选其一 | jPDL表达式，代替相应的处理方法。 |
| **accept-propagated-events** | 属性 | 可选 | 可选值：{yes|no|true|false}.默认值为 yes | true. 是否接受来至action绑定元素的子元素提交的触发事件。设为false | no，表示只接受action绑定的元素自身的事件 |
| **config-type** | 属性 | 可选 | 对ActionHandler实现类的初始化方式，可选值有： {field|bean|constructor|configuration-property}. |
| **async** | 属性 | true |false | 默认为false，它意味着action与流程在同一线程中执行。如果设为true，将向命令执行器发送一则消息，并且action会以异步的方式在分离的事务（另一线程）中处理。 |
|  | {内容} | optional | action标签体中的内容，作为初始化参数用来初始化ActionHandler的实现类. |

**<variable>(变量)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **name** | 属性 | 必须 | 过程变量名 |
| **access** | 属性 | 可选 | 访问约束。可选值为：read, write and required. 默认是read, write该属性使用逗号分隔访问修饰列表。 |
| **mapped-name** | 属性 | 可选 | 变量映射名，默认同变量名。 |

**<handler>(处理器)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **expression** | 属性 | 此属性同class属性互斥 | jPDL表达式。该表达式返回字符串结果，与leaving transitions的命名相匹配 |
| **class** | 属性 | 此属性同expression属性互斥 | 实现接口org.jbpm.graph.node.DecisionHandler的Java类的完整类名. |
| **config-type** | 属性 | 可选 | 对DecisionHandler实现类的初始化方式，可选值有： {field|bean|constructor|configuration-property}. |
|  | {内容} | 可选 | handler标签体中的内容，作为初始化参数用来初始化DecisionHandler的实现类. |

**<task>(任务)**

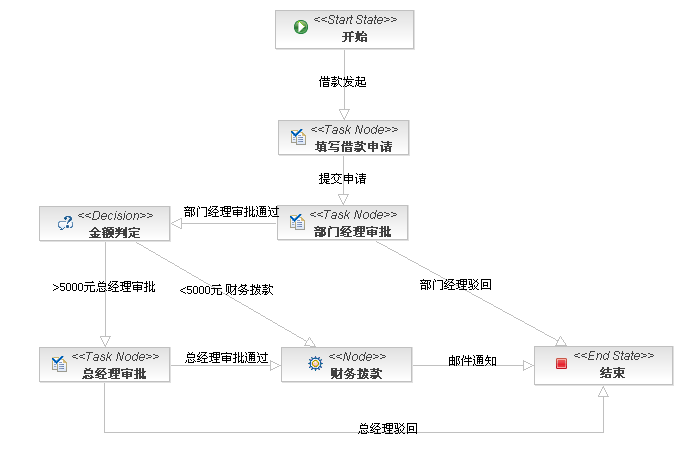
|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **name** | 属性 | 可选 | task的名称。该名称可以被查找和引用task对象。 |
| **blocking** | 属性 | 可选 | 可选值为：{yes|no|true|false}, 默认值为false.如果该属性被设置为true，则在没有完成任务前，流程不能离开此结点如果是true，那么token对象的signal方法将促使流程向下继续。大多情况下，强制中断流程会有客户操作界面发起。 |
| **signalling** | 属性 | 可选 | 可选值为：{yes|no|true|false}, 默认值为true.如果设置为false，则任务将失去触发流程继续下行的能力。 |
| **duedate** | 属性 | 可选 | 任务的持续时间。可以用绝对时间或工作日两种计量方式表示。 |
| **swimlane** | 属性 | 可选 | 对swimlane的引用.当一个swimlane 被赋予任务的时候，assignment属性将被忽略。 |
| **priority** | 属性 | 可选 | 优先级。可选值：{highest, high, normal, low, lowest}. 另一种表达方式可以用整数，如：{highest=1, lowest=5) |
| **assignment** | 元素 | 可选 | 为task委派一个参与者。 |
| **event** | 元素 | [0..\*] | 支持的事件类型有：  {task-create|task-start|task-assign|task-end}. 特别要指出的是，我们在task-assign 事件上为任务实例添加了一个非持久化的属性previousActorId |
| **exception-handler** | 元素 | [0..\*] | 对委派类异常的处理者 |
| **timer** | 元素 | [0..\*] | 为任务的执行过程指定一个计时器，监视任务的执行。这里我们可以利用 cancel-event 同计时器挂钩来做一些特定的事。默认情况下，cancel-event会结束任务，但我们可以将它定制为任务分配或者任务开始。 |
| **controller** | 元素 | [0..1] | 指定如果将过程变量转换成任务表单参数以提供给用户交互界面 |

**<assignment>(分配)**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **多样性** | **描述** |
| **expression** | 属性 | 可选 | 由于历史原因，这里的expression不是指jPDL 表达式，而是指jBPM 身份组件中的表达式，详细参考[Section 11.11.2, “Assignment expressions”](http://docs.jboss.com/jbpm/v3/userguide/taskmanagement.html#assignmentexpressions). 注意，该属性必须依赖于jbpm身份组件 |
| **actor-id** | 属性 | 可选 | 参与者ID. 该属性可以与pooled-actors属性联合使用。可以设计一个表达式来替代；也可以设定一个固定值； 又或者指定一个属性或方法，返回一个诸如**actor-id="myVar.actorId"**的字符窜,这里 myVar是一个任务实例对象，它将调用方法getActorId 返回真实的actor-id |
| **pooled-actors** | 属性 | 可选 | 一个由逗号分隔的参与者ID列表.一个固定的列表，诸如： pooled-actors="chicagobulls, pointersisters". 又或者使用表达式返回。还可以引用一个属性或方法，返回字符串数组变量、集合变量以及用逗号分隔的字窜来表达参与者集合。 |
| **class** | 属性 | 可选 | 实现 org.jbpm.taskmgmt.def.AssignmentHandler接口的Java类全名 |
| **config-type** | 属性 | 可选 | 对AssignmentHandler实现类的初始化方式，可选值有： {field|bean|constructor|configuration-property}. |
|  | {内容} | 可选 | 作为初始化参数用来初始化AssignmentHandler的实现类. |

[**jBPM-jPDL学习笔记——流程设计与控制**](http://linliangyi2007.javaeye.com/blog/176345)

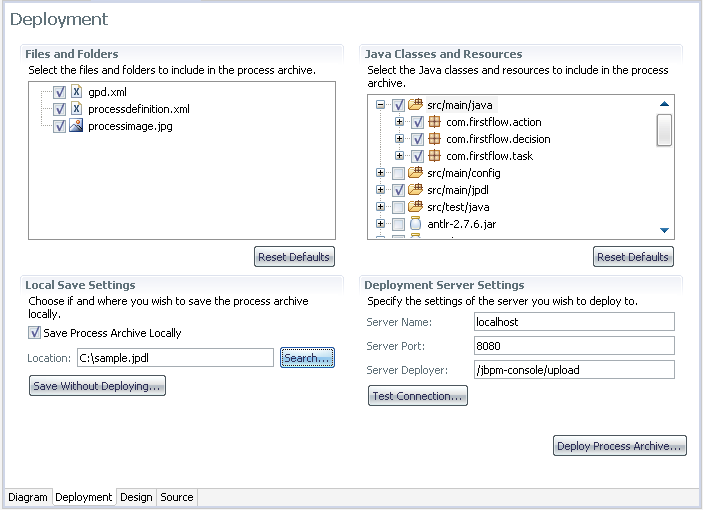
**关键字: jbpm jpdl 流程设计与控制 样例代码 api**

**相关资料：**   
[《jBPM学习笔记(V3.2环境部署)》](http://linliangyi2007.javaeye.com/blog/176339)   
[《jBPM学习笔记(框架设计简介)》](http://linliangyi2007.javaeye.com/blog/176340)   
**背景**   
本片文章，我们将从业务流程的设计开始，通过带领大家完成一个完整工作流的程序设计，来学习jPDL的使用。   
  
**业务流程设计**   
这里我们实现一个相对简化的公司借款申请流程。流程图如下：   
   
  
在jPDL中，与流程设计相关的文件有三个：processdefinition.xml、gdp.xml、processimage.jpg。其中processdefinition.xml是流程定义的描述文件；gpd.xml是对图形界面呈现的XML描述；而processimage.jpg则是对图形界面的快照。下面我们将展示本样例的流程定义文件。   
 **流程定义描述**

**processdefinition.xml文件**

**引用**

<?xml version="1.0" encoding="UTF-8"?>   
  
<process-definition   xmlns ="urn:jbpm.org:jpdl-3.2"  name="simple">   
   <start-state name="开始">   
      <transition name="借款发起" to="填写借款申请">   
<action name="Action\_StartProcess" class="com.firstflow.action.StartProcessActionHander"></action>   
</transition>   
   </start-state>   
   <task-node name="填写借款申请">   
<task name="Task\_AssignToInitiator">   
<assignment class="com.firstflow.task.NewApplicationAssignmentHandler"></assignment>   
</task>   
<transition to="部门经理审批" name="提交申请">   
<action name="Action\_SubmitApply" class="com.firstflow.action.SubmitApplyActionHandler"></action>   
</transition>   
</task-node>   
  
<task-node name="部门经理审批">   
<task name="Task\_ManagerApprove">   
<assignment class="com.firstflow.task.DepartmentManagerApproveAssignmentHandler"></assignment>   
</task>   
<transition to="金额判定" name="部门经理审批通过">   
<action name="Task\_ManagerApproved" class="com.firstflow.action.ManagerApprovedActionHandler"></action>   
</transition>   
<transition to="结束" name="部门经理驳回">   
<action name="Action\_ManagerDisapprove" class="com.firstflow.action.ManagerDisapproveActionHandler"></action>   
</transition>   
</task-node>   
  
<node name="财务拨款">   
<action name="Action\_AccountantProcess" class="com.firstflow.action.AccountantProcessActoinHandler"></action>   
<transition to="结束" name="邮件通知">   
<action name="Action\_Mail" class="com.firstflow.action.SendMailActionHandler"></action>   
</transition>   
</node>   
  
<decision name="金额判定">   
<handler class="com.firstflow.decision.MoneyCheckDecisionHandler"></handler>   
<transition to="总经理审批" name="&gt;5000元总经理审批"></transition>   
<transition to="财务拨款" name="&lt;5000元 财务拨款"></transition>   
</decision>   
  
<task-node name="总经理审批">   
<task name="Task\_PresidentApprove">   
<assignment class="com.firstflow.task.PresidentApproveAssignmentHandler"></assignment>   
</task>   
<transition to="财务拨款" name="总经理审批通过">   
<action name="Action\_PresidentApproved" class="com.firstflow.action.PresidentApprovedActionHandler"></action>   
</transition>   
<transition to="结束" name="总经理驳回">   
<action name="Action\_PresidentDisapproved" class="com.firstflow.action.PresidentDisapprovedActionHandler"></action>   
</transition>   
</task-node>   
   <end-state name="结束"></end-state>   
</process-definition>

在样例流程中，除了开始和结束结点外，我们定义了三种类型的结点：   
  
**任务结点<task-node>**   
任务结点是一个需要人工参与的结点类型。当流程进入结点时，会生成相应的任务实例(TaskInstatnce)，并通过委派接口AssignmentHandler或jBPM表达式将任务委派给一个或多个特定的角色或参与者。结点自身进入等待状态，直到任务被参与者完成或者跳过，流程继续。   
  
**判定结点<decision>**   
判定结点的设计目标是根据上下文环境和程序逻辑，判定流程转向。通过指定一个实现DecisionHandlder接口的Java委派类或jBPM表达式，来返回转向（transition）的字符窜类型的名称（可以是中文哦），来达到决定流程方向的功能。   
  
**普通结点<node>**   
普通结点也可以定义相应的处理任务，通过定义相应的ActioinHandler类。同任务结点不同的是，普通结点定义的任务是由流程自动执行的，无须人工干预。   
  
**三种结点都可定义结点事件（event）：**   
node-enter，该事件在流程进入结点时触发   
node-leave，该事件在流程离开节点是触发   
可以在事件上挂接ActioinHandler接口的实现类来完成一些特定的功能。   
  
**三种节点都可以定义异步处理方式（async属性）：**   
异步处理意味着每个结点的事务处理是通过消息机制分离的，不再同一线程中统一调用执行。而是由消息监听线程从消息队列中取得消息体来运行相应得程序。   
此外我们定义了结点间的转向（transition），用来记录和处理状态的变迁。每个转向中，可以委派一个或多个的ActioinHandler接口实现类，负责处理节点变迁时的上下文状态变更及回调用户定义的处理程序。   
  
**流程的程序接口说明**   
  
**动作处理接口（ActioinHandler）**   
**接口方法：**void execute( ExecutionContext executionContext ) throws Exception   
该接口是jPDL中最常用的一个回调接口。从它的接口方法可以发现，它仅仅暴露了流程执行上下文变量ExecutionContext。用户程序通过ExecutionContext来了解流程的执行状态，并通过改变ExecutionContext中的属性值来影响流程的执行。   
ActioinHandler接口可以在所有能包含事件(event)、动作(action)元素的地方被回调。   
  
**判定处理接口（DecisionHandlder）**   
**接口方法：**String decide(ExecutionContext executionContext) throws Exception   
判定接口仅适用于判定节点(decision)中。从它的接口方法可以看出，方法要返回一个字符串型的结果，这个结果必须和判定节点拥有的转向(transition)集合中的一条转向名称相匹配。   
在DecisionHandlder的接口方法中一样能访问到ExecutionContext变量，这为判定提供了执行上下文的根据。当然，如果有必要，用户也可以在该接口中改变ExecutionContext中的变量值。   
  
**委派处理接口（AssignmentHandler）**   
**接口方法：**void assign(Assignable assignable, ExecutionContext executionContext) throws Exception;   
委派处理接口被用户任务元素(task)的委派(assignment)子元素中，它的职责很明确，就是将任务分配给指定的人员或角色。   
在AssignmentHandler接口的方法中，Assignable变量通常指任务实例(TaskInstance)。通过将ExecutionContext和TaskInstance两个变量都暴露给接口方法，用户就可以根据流程上下文情况，来决定要将指定的任务分配个谁。   
  
**流程的部署**   
  
用户使用jPDL的流程设计器定义业务流程，当然，你也可以直接用文档编辑器直接编辑processdefinition.xml定义文件。定义文档是可以直接被ProcessDefinition类载入使用的，但在正式运行的系统中，流程定义信息更多是使用关系型数据库来存储。从流程定义文件将数据导入流程数据库的过程，我们称之为流程部署。   
jPDL的流程部署文件包含processdefinition.xml的定义部分和Java处理器的代码部分，这些文件可以被一起打包成.jpdl的zip格式包而后上传服务器端。这个过程可以在流程设计器界面的“deployment”标签页中操作：   
   
这里我们着重要讲述的是接受部署文件上载的服务器端配置。在jBPM3.2的包中带着一个jPDL的管理控制台web应用，默认名字为jbpm-console。该应用带有接受流程定义包部署的程序，但不是最小化的。实际上完成流程部署功能的，只是jbpm-jpdl.jar核心包中的一个servlet类：org.jbpm.web.ProcessUploadServlet . 完成这个Servlet的成功部署，需要以下工作：   
  
1. 配置web.xml，将servlet配置成启动时加载，如下：

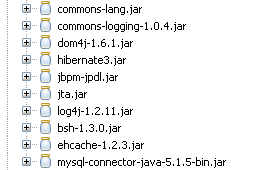
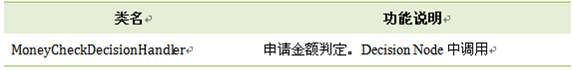
**引用**

<web-app>   
<servlet>   
  <servlet-name>GDP Deployer Servlet</servlet-name>   
  <servlet-class>org.jbpm.web.ProcessUploadServlet</servlet-class>   
  <load-on-startup>1</load-on-startup>   
  </servlet>   
    
<servlet-mapping>   
  <servlet-name>GDP Deployer Servlet</servlet-name>   
  <url-pattern>/upload/\*</url-pattern>   
  </servlet-mapping>   
    
</web-app>

2. 建立流程定义存储数据库表：   
Demo中，我们使用的数据库是MySQL的，在E:\Java\tools\jbpm-jpdl-3.2.2\db\目录下有个jbpm.jpdl.mysql.sql数据库脚本文件。但我们不能直接导入该文件, 会提示有错误, 应为该文件的SQL语句末尾少了分号，在批量执行时，MySQL报错。需要在每一行SQL的末尾添加一个分号，这样就可以用source命令导入了。   
  
3. 配置Hibernate.cfg.xml   
由于jBPM的数据库持久化是依靠Hibernate进行的，因此需要配置Hibernate.cfg.xml使其适应我们的MySQL环境

**引用**

<!-- hibernate dialect -->   
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>   
    <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>   
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/linly</property>   
    <property name="hibernate.connection.username">linly</property>   
    <property name="hibernate.connection.password">coffee</property>   
    <property name="hibernate.cache.provider\_class">org.hibernate.cache.HashtableCacheProvider</property>

4. Import需要的jar包   
这里的jar包包括三部分：jbpm的核心包；Hibernate及其支撑包；MySQL的JDBC驱动包。   
   
  
到此，我们的配置工作完成，这是实现jBPM流程部署服务端的最小化应用配置。   
  
**流程控制及API使用**   
  
**样例程序中的Handler接口实现**   
  
下面，根据上述的接口分类，列出样例程序中的类名及相应的功能说明，具体可参考源代码。   
**动作处理接口（ActioinHandler）**   
   
这里要提到一个很重要的区别，就是作用于Node上的ActoinHandler和作用于Transition上的ActoinHandler是有不同的。区别在于，Node上的ActoinHandler在结束业务逻辑处理后，必须调用executionContext.leaveNode();或executionContext.leaveNode(transition)来保证流程向下执行；而作用于Transition上的则不需要。   
  
**判定处理接口（DecisionHandlder）**   
   
  
**委派处理接口（AssignmentHandler）**   
   
  
**流程测试剖析**   
  
本章节，我们将给大家剖析两个流程测试类。一个是简单的基于内存模型的流程测试FirstFlowProcessTest；一个是更贴近实用的，基于MySQL数据库操作的标准测试案例。通过对这两个测试例程的分析，来直观的学习如何通过Java API操作jPDL。   
  
**简单流程测试案例**   
**测试案例类：FirstFlowProcessTest.java**

**Java代码**

1. **public** **class** FirstFlowProcessTest **extends** TestCase {
2. ProcessDefinition pdf ;
3. ProcessInstance pi;
5. **public** **void** test4000YuanApplication() **throws** Exception {
6. deployProcessDefinition();
7. createProcessInstance("linly");
8. submitApplication(4000);
9. approveByManager(**true**);
10. checkTasks();
11. }
13. **public** **void** test6000YuanApplication() **throws** Exception {
14. deployProcessDefinition();
15. createProcessInstance("linly");
16. submitApplication(6000);
17. approveByManager(**true**);
18. approveByPresident(**true**);
19. checkTasks();
20. }
22. **public** **void** test7000YuanApplication() **throws** Exception {
23. deployProcessDefinition();
24. createProcessInstance("linly");
25. submitApplication(7000);
26. approveByManager(**true**);
27. approveByPresident(**false**);
28. checkTasks();
29. }
31. /\*\*
32. \* 部署流程定义
33. \* @throws Exception
34. \*/
35. **protected** **void** deployProcessDefinition() **throws** Exception{
36. System.out.println("==FirstFlowProcessTest.deployProcessDefinition()==");
37. pdf = ProcessDefinition.parseXmlResource("firstflow/processdefinition.xml");
38. assertNotNull("Definition should not be null", pdf);
39. }
40. /\*\*
41. \* 生成流程实例
42. \*/
43. **protected** **void** createProcessInstance(String user){
44. System.out.println("==FirstFlowProcessTest.createProcessInstance()==");
45. assertNotNull("Definition should not be null", pdf);
46. //生成实例
47. pi = pdf.createProcessInstance();
48. assertNotNull("processInstance should not be null", pi);
49. //设置流程发起人
50. pi.getContextInstance().createVariable("initiator", user);
51. //触发流程转向
52. pi.signal();
53. }
54. /\*\*
55. \* 填写提交申请单
56. \* @param money
57. \*/
58. **protected** **void** submitApplication(**int** money){
59. System.out.println("==FirstFlowProcessTest.submitApplication()==");
60. TaskInstance ti = (TaskInstance)pi.getTaskMgmtInstance().getTaskInstances().iterator() .next() ;
61. System.out.println("ti.actor = " + ti.getActorId());
62. ContextInstance ci = ti.getContextInstance();
63. ci.setVariable("money",**new** Integer(money));
64. ti.end();
65. }
66. /\*\*
67. \* 部门经理审批
68. \* @param pass
69. \*/
70. @SuppressWarnings("unchecked")
71. **protected** **void** approveByManager(**boolean** pass){
72. System.out.println("==FirstFlowProcessTest.approveByManager()==");
73. Iterator<TaskInstance> it = pi.getTaskMgmtInstance().getTaskInstances().iterator();
74. **for**( ;it.hasNext(); ){
75. TaskInstance ti = it.next();
76. **if**(ti.getActorId().equals("DepartmentManager")){
77. List<Transition> transitions = ti.getToken().getNode().getLeavingTransitions();
78. **for**(Transition t : transitions){
79. System.out.println("----Transition" + t.getName());
80. }
81. assertEquals("DepartmentManager",ti.getActorId());
82. **if**(pass){
83. ti.end("部门经理审批通过");
84. }**else**{
85. ti.end("部门经理驳回");
86. }
87. **return**;
88. }
89. }
90. }
92. /\*\*
93. \* 总经理审批
94. \* @param pass
95. \*/
96. @SuppressWarnings("unchecked")
97. **protected** **void** approveByPresident(**boolean** pass){
98. System.out.println("==FirstFlowProcessTest.approveByPresident()==");
100. Iterator<TaskInstance> it = pi.getTaskMgmtInstance().getTaskInstances().iterator();
102. **for**( ;it.hasNext(); ){
104. TaskInstance ti = it.next();
106. **if**(ti.getActorId().equals("President")){
107. List<Transition> transitions = ti.getToken().getNode().getLeavingTransitions();
108. **for**(Transition t : transitions){
109. System.out.println("----Transition" + t.getName());
110. }
111. assertEquals("President",ti.getActorId());
112. **if**(pass){
113. ti.end("总经理审批通过");
114. }**else**{
115. ti.end("总经理驳回");
116. }
117. **return**;
118. }
119. }
120. }
121. /\*\*
122. \* 打印任务记录
123. \*/
124. @SuppressWarnings("unchecked")
125. **protected** **void** checkTasks(){
126. System.out.println("==FirstFlowProcessTest.checkTasks()==");
127. Collection<TaskInstance> coll = pi.getTaskMgmtInstance().getTaskInstances();
128. System.out.println("====Process has task:====");
129. **for**(TaskInstance ti : coll){
130. System.out.println("=="+ti.getName()+"==");
131. System.out.println("=="+ti.getActorId()+"==");
132. System.out.println("=="+ti.getVariables().toString() +"==");
133. }
134. System.out.println("====end====");
135. }

该案例是在没有数据库支持的情况下，对报销流程进行运行测试，测试逻辑如下：   
  
**1. 加载流程定义**

**Java代码**

1. ProcessDefinition.parseXmlResource("firstflow/processdefinition.xml")

代码说明：   
在没有数据库存储的情况下，流程定义通过ProcessDefinition类直接从processdefinition.xml文件中解析加载。   
  
**2. 实例化流程对象**

**Java代码**

1. //生成实例
2. pi = pdf.createProcessInstance();
3. assertNotNull("processInstance should not be null", pi);
4. //设置流程发起人
5. pi.getContextInstance().createVariable("initiator", user);
6. //触发流程转向
7. pi.signal();

代码说明：   
在获得流程定义的实例后，可以用它生成流程实例，使用如下的语句：   
pi = pdf.createProcessInstance();   
流程实例拥有自己的ContextInstance环境变量对象。它实际上是一个HashMap，以key-value方式记录了流程的上下文变量值，代码中的   
pi.getContextInstance().createVariable("initiator", user);就是向环境变量中添加一个key为initiator的对象。   
每个流程实例都拥有自己Token令牌对象，主流程有自己的RootToken，子流程也拥有自己的子Token。父流程的Token和子流程的Token相互关联，形成Token树。   
Token对象表示流程运行的当前位置（运行到哪个节点了）。通过对Token对象的signal()方法调用，可以使流程向下运行。代码中的pi.signal();实际上是间接调用了pi.getRootToken().signal();它使得新建的流程继续向下个节点（即借款申请单填写）进发。   
  
**3. 员工发起借款申请**

**Java代码**

1. /\*\*
2. \* 填写提交申请单
3. \* @param money
4. \*/
5. **protected** **void** submitApplication(**int** money){
6. System.out.println("==FirstFlowProcessTest.submitApplication()==");
8. TaskInstance ti = (TaskInstance)pi.getTaskMgmtInstance().getTaskInstances().iterator() .next()
9. System.out.println("ti.actor = " + ti.getActorId());
10. ContextInstance ci = ti.getContextInstance();
11. ci.setVariable("money",**new** Integer(money));
12. ti.end();

代码说明：   
在借款流程发起后，流程进入了申请单填写阶段。这个阶段是个人工的任务，需要用户的介入。因此，对于要借款的用户而言，首先是获取填写申请单的任务实例：   
     TaskInstance ti = (TaskInstance)pi.getTaskMgmtInstance().getTaskInstances().iterator() .next()           
在这个测试类中，由于没有数据库。对流程实例的引用是依靠了类的全局标量pi。这里通过pi获取全部的任务列表。实际上有且仅有一个任务，就是我们刚刚发起的申请单填写任务。   
接下来，我们获取流程的上下文变量，将申请借款的数额记录在上下文变量中ContextInstance ci = ti.getContextInstance();   
    ci.setVariable("money",new Integer(money));   
最后，我们要结束当前任务，告诉流程继续下行，调用ti.end()；这个方法的本质依然是调用了token.signal()，它选择了一个默认的transition进行转向。这里要说明的是signal方法有多态的实现signal(Transition  transition)，是可以指定具体转向参数的。   
  
**4. 部门领导审批申请**

**Java代码**

1. /\*\*
2. \* 部门经理审批
3. \* @param pass
4. \*/
5. @SuppressWarnings("unchecked")
6. **protected** **void** approveByManager(**boolean** pass){
7. System.out.println("==FirstFlowProcessTest.approveByManager()==");
8. Iterator<TaskInstance> it = pi.getTaskMgmtInstance().getTaskInstances().iterator();
9. **for**( ;it.hasNext(); ){
10. TaskInstance ti = it.next();
11. **if**(ti.getActorId().equals("DepartmentManager")){
12. List<Transition> transitions = ti.getToken().getNode().getLeavingTransitions();
13. **for**(Transition t : transitions){
14. System.out.println("----Transition" + t.getName());
15. }
16. assertEquals("DepartmentManager",ti.getActorId());
17. **if**(pass){
18. ti.end("部门经理审批通过");
19. }**else**{
20. ti.end("部门经理驳回");
21. }
22. **return**;
23. }
24. }
25. }

代码说明：   
这里，流程进入了部门经理审批阶段。由于没有数据库支持，我们只能采取遍历任务列表，并比对委派者ID的方式来确定委派给部门经理的任务实例。（在后面的基于数据库的标准案例中，我们会看到如果根据用户的ID来获取分配给指定用户的任务）   
ti.getActorId().equals("DepartmentManager") // 比对任务的委派人。   
ti.getToken().getNode().getLeavingTransitions();//获取任务在当前节点上的所有转向。   
这里我们要特别指出的是ti.end("部门经理审批通过")和ti.end("部门经理驳回")这实际上调用token.signal(transition);来完成任务的转向，从而使流程继续。   
  
**5. 总经理审批申请**

**Java代码**

1. /\*\*
2. \* 总经理审批
3. \* @param pass
4. \*/
5. @SuppressWarnings("unchecked")
6. **protected** **void** approveByPresident(**boolean** pass){
7. System.out.println("==FirstFlowProcessTest.approveByPresident()==");
8. Iterator<TaskInstance> it = pi.getTaskMgmtInstance().getTaskInstances().iterator();
10. **for**( ;it.hasNext(); ){
11. TaskInstance ti = it.next();
12. **if**(ti.getActorId().equals("President")){
13. List<Transition> transitions = ti.getToken().getNode().getLeavingTransitions();
14. **for**(Transition t : transitions){
15. System.out.println("----Transition" + t.getName());
16. }
17. assertEquals("President",ti.getActorId());
18. **if**(pass){
19. ti.end("总经理审批通过");
20. }**else**{
21. ti.end("总经理驳回");
22. }
23. **return**;
24. }
25. }
26. }

代码说明：   
此步代码同“部门经理审批”代码相似，不作更多说明。   
  
**标准流程测试案例**   
  
该案例模拟了标准运行环境中，基于关系型数据库的jBPM系统是如何执行流程的。   
测试案例类：FirstFlowProcessDBTest.java

**Java代码**

1. **public** **class** FirstFlowProcessDBTest {
2. /\*
3. \* 初始化jBPM配置
4. \* 包含对Hibernate的数据库初始化
5. \*/
6. **static** JbpmConfiguration jbpmConfiguration = JbpmConfiguration.getInstance();
7. **public** **static** **void** main(String[] args){
8. FirstFlowProcessDBTest test = **new** FirstFlowProcessDBTest();
9. test.test4000YuanApplication();
10. test.test6000YuanApplication();
11. test.test7000YuanApplication();
12. }
14. **public** **void** test4000YuanApplication(){
15. ProcessInstance pi = createProcessInstance("linly");
16. submitApplication("linly" , 4000);
17. approveByManager(**true**);
18. checkTasks(pi);
19. }
20. **public** **void** test6000YuanApplication() {
21. ProcessInstance pi = createProcessInstance("linly");
22. submitApplication("linly" , 6000);
23. approveByManager(**true**);
24. approveByPresident(**true**);
25. checkTasks(pi);
26. }
27. **public** **void** test7000YuanApplication() {
28. ProcessInstance pi = createProcessInstance("linly");
29. submitApplication("linly" , 7000);
30. approveByManager(**true**);
31. approveByPresident(**false**);
32. checkTasks(pi);
33. }
34. /\*\*
35. \* 生成流程实例
36. \*/
37. **protected** ProcessInstance createProcessInstance(String user){
38. System.out.println("==FirstFlowProcessTest.createProcessInstance()==");
39. JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
40. **try** {
41. GraphSession graphSession = jbpmContext.getGraphSession();
42. /\*
43. \* 从数据库获取指定的流程定义
44. \*/
45. ProcessDefinition pdf  = graphSession.findLatestProcessDefinition("simple");
46. //生成流程实例
47. ProcessInstance pi = pdf.createProcessInstance();
48. //设置流程发起人
49. pi.getContextInstance().createVariable("initiator", user);
50. //触发流程转向
51. pi.signal();
52. /\*
53. \* 保存流程实例
54. \*/
55. jbpmContext.save(pi);
56. **return** pi;
57. }**finally**{
58. jbpmContext.close();
59. }
60. }
62. /\*\*
63. \* 填写提交申请单
64. \* @param money
65. \*/
66. @SuppressWarnings("unchecked")
67. **protected** **void** submitApplication(String actorId , **int** money){
68. System.out.println("==FirstFlowProcessTest.submitApplication()==");
69. JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
70. **try** {
71. /\*
72. \*根据操作者ID，获取属于该操作者的任务集
73. \*/
74. List<TaskInstance> taskInstances = jbpmContext.getTaskList(actorId);
75. **for**(TaskInstance ti : taskInstances){
76. System.out.println("ti.name = " + ti.getName());
77. System.out.println("ti.actor = " + ti.getActorId());
78. ContextInstance ci = ti.getContextInstance();
79. ci.setVariable("money",**new** Integer(money));
80. ti.end();
81. }
82. }**finally**{
83. jbpmContext.close();
84. }
85. }
87. /\*\*
88. \* 部门经理审批
89. \* @param pass
90. \*/
91. @SuppressWarnings("unchecked")
92. **protected** **void** approveByManager(**boolean** pass){
93. System.out.println("==FirstFlowProcessTest.approveByManager()==");
94. JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
95. **try**{
96. List<TaskInstance> taskInstances = jbpmContext.getTaskList("DepartmentManager");
97. **for**(TaskInstance ti : taskInstances){
98. System.out.println("ti.name = " + ti.getName());
99. System.out.println("ti.actor = " + ti.getActorId());
100. **if**(pass){
101. ti.end("部门经理审批通过");
102. }**else**{
103. ti.end("部门经理驳回");
104. }
105. }
106. }**finally**{
107. jbpmContext.close();
108. }
109. }
111. /\*\*
112. \* 总经理审批
113. \* @param pass
114. \*/
115. @SuppressWarnings("unchecked")
116. **protected** **void** approveByPresident(**boolean** pass){
117. System.out.println("==FirstFlowProcessTest.approveByPresident()==");
118. JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
119. **try**{
120. List<TaskInstance> taskInstances = jbpmContext.getTaskList("President");
121. **for**(TaskInstance ti : taskInstances){
122. System.out.println("ti.name = " + ti.getName());
123. System.out.println("ti.actor = " + ti.getActorId());
124. **if**(pass){
125. ti.end("总经理审批通过");
126. }**else**{
127. ti.end("总经理驳回");
128. }
129. }
130. }**finally**{
131. jbpmContext.close();
132. }
133. }
135. /\*\*
136. \* 打印任务记录
137. \*/
138. @SuppressWarnings("unchecked")
139. **protected** **void** checkTasks(ProcessInstance pi){
140. System.out.println("==FirstFlowProcessTest.checkTasks()==");
141. JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
142. **try**{
143. List<TaskInstance> coll = jbpmContext.getTaskMgmtSession().findTaskInstancesByProcessInstance(pi);
144. System.out.println("====Process has task:====");
145. **for**(TaskInstance ti : coll){
146. System.out.println("=="+ti.getName()+"==");
147. System.out.println("=="+ti.getActorId()+"==");
148. System.out.println("=="+ti.getVariables().toString() +"==");
149. }
150. System.out.println("====end====");
151. }**finally**{
152. jbpmContext.close();
153. }
154. }
155. }

相对于简单流程测试案例，标准流程的业务是相同的。它们的不同点在于：简单流程通过XML载入流程定义，并使用类的全局变量来保存流程实例的引用；而标准流程则是通过数据库载入流程定义，并使用数据库的会话来维护流程的运行。   
  
**1. 从数据库载入流程定义**

**Java代码**

1. JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();  ¬

从jBPM配置环境中获取jBPM上下文实例，jbpmContext是对jbpm持久层操作API及上下文环境的封装，它根据jBPM的配置文件生成。      
¬

**Java代码**

1. GraphSession graphSession = jbpmContext.getGraphSession();

生成对流程图操作的持久层会话   
¬

**Java代码**

1. ProcessDefinition pdf  = graphSession.findLatestProcessDefinition("simple");

从数据库获取命名为“simple”的流程定义。   
  
**2. 新建流程实例，并存入数据库持久化**   
¬

**Java代码**

1. ProcessInstance pi = pdf.createProcessInstance();

根据流程定义生成实例。   
¬

**Java代码**

1. pi.getContextInstance().createVariable("initiator", user);
2. pi.signal();

设置实例的上下文变量，激活实例执行进程。   
¬

**Java代码**

1. jbpmContext.save(pi);

保存实例到数据库，持久化实例。   
  
**3. 从数据库获取属于指定操作者的任务集**

**Java代码**

1. JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
2. **try**{
3. List<TaskInstance> taskInstances = jbpmContext.getTaskList("DepartmentManager");
4. **for**(TaskInstance ti : taskInstances){
5. System.out.println("ti.name = " + ti.getName());
6. System.out.println("ti.actor = " + ti.getActorId());
7. **if**(pass){
8. ti.end("部门经理审批通过");
9. }**else**{
10. ti.end("部门经理驳回");
11. }
12. }
13. }**finally**{
14. jbpmContext.close();
15. }

通过JbpmContext对象，从数据库获取指定操作者的任务集合：

**Java代码**

1. List<TaskInstance> taskInstances = jbpmContext.getTaskList("DepartmentManager");

注意，在每次的JbpmContext对象使用完毕后，一定要执行jbpmContext.close()，其本质是要释放数据库的操作连接。