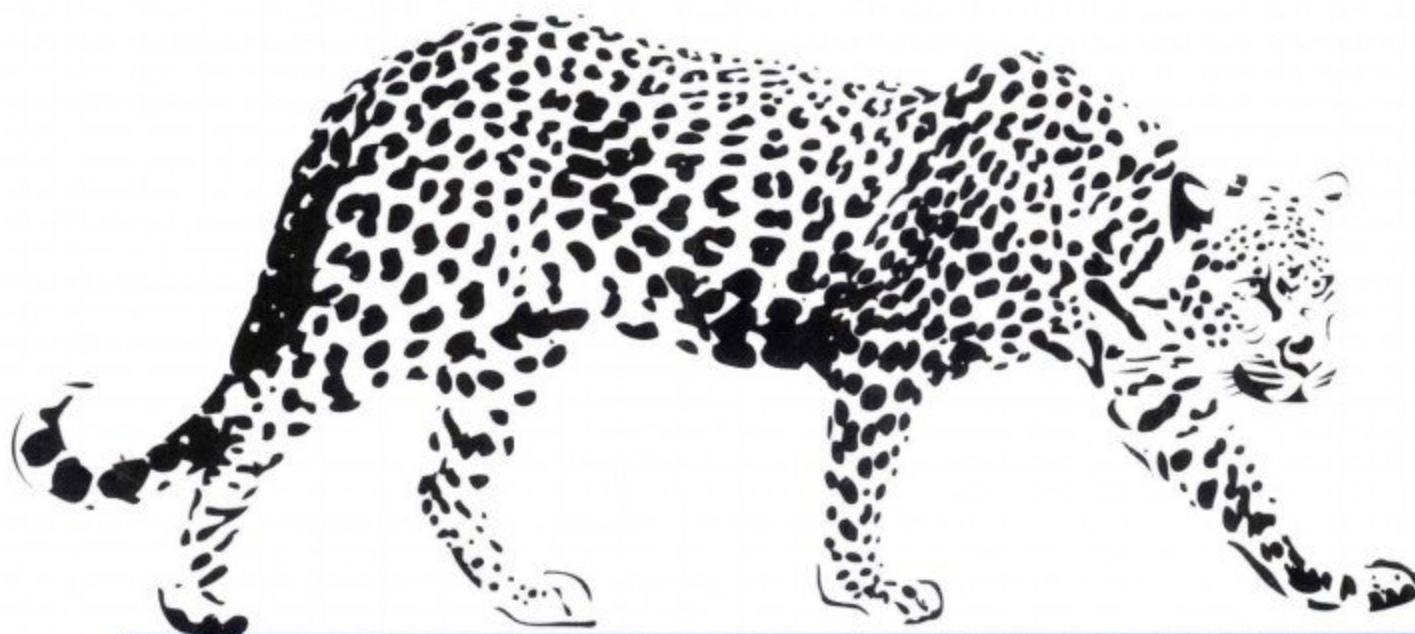


- 完全解析jBPM4应用开发技术
- 9位工作流业内专家联袂推荐

Broadview
www.broadview.com.cn



jBPM4

工作流 应用开发指南

● 胡奇 编著

 电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

jBPM4 workflow 应用开发指南

jBPM作为历史最悠久、功能最强大的开源工作流引擎一直拥有着国内外广泛的使用群体。2005年我在美国时因为第一次在大型企业应用中使用了jBPM与Seam技术，还得到了jBPM团队所在的JBoss公司2006年度世界创新奖。2007年回国与本书作者成为同事后，我们一起为国内大量客户进行了有关工作流引擎的咨询工作。那时候本书作者就开始不断总结客户所面对的问题与需求，并一直想把jBPM真正结合本土业务特色发扬光大。如今本书终于问世，过去一直缠绕着众多国内程序员的一些如回退、会签、自由流等“中国特色”的问题都通过jBPM4有了明快的解决之道。多年来jBPM工作组与本书作者共同的厚积薄发恰如其时地为国内程序员在实际工作中快速应用开源技术又提供了一个宝贵的武器。

马 越

就职于中通软联信息科技有限公司，创始人兼首席架构师

<http://www.jbosschina.org> 社区创始人；2006年 Red Hat/JBoss 世界创新大奖获得者
美国10年大型商业软件构架经验；第一个将 Seam 及 jBPM 框架引入大型企业应用的开拓者



责任编辑：董 英
封面设计：李 玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。



上架建议：编程语言/Java开发

ISBN 978-7-121-11791-6



9 787121 117916 >

定价：59.00元

jBPM4 workflow应用开发指南

胡奇 编著

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING



内 容 简 介

随着在现代企业管理中对于信息化、流程化的深度挖掘,具有业务流程管理的技术和思想已经成为致力于全面掌控企业级应用系统人士“日常生活、居家旅行”的必备素质。

本书分两篇。第一篇介绍 workflow 管理技术的概念、起源和发展历程,开源 workflow 选型,以及 jBPM——这个迄今为止最成功的 Java 开源 workflow 项目的“前世今生”。此外,本篇还可以帮助读者快速上手 jBPM4、使用 jBPM4 开发企业流程应用,包括安装和配置 jBPM4、使用 jBPM 图形化流程设计器(GPD)设计流程、把流程部署到服务器上去、使用 jBPM4 Service API 控制流程、掌握 jBPM 流程定义语言、流程变量、流程脚本。第二篇主要涉及基于 jBPM4 这个强大的应用程序框架打造属于自己独特业务的“企业流程管理平台”,包括 jBPM4 扩展研发先决条件、深入 jPDL 和 jBPM Service API、升级 jBPM3 到 jBPM4、流程虚拟机原理、jBPM4 的设计思想、按需而配 jBPM4、异步工作执行器、深入 jBPM4 电子邮件支持、系统日志、jBPM4 与 Spring 框架集成、jBPM4 与 JBoss 应用服务器集成、中国特色 workflow 的 jBPM 实现。

本书结构条理清晰,实践例程与理论思想紧密结合,翔实易懂,由浅入深,具有很强的参考性和实用性。

本书适合所有掌握 JavaEE (Java 企业级版本) 开发技术的人员——无论您是技术开发者、项目实施者、系统架构师,还是流程分析师、业务方案顾问,本书都适合您。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

jBPM4 工作流应用开发指南 / 胡奇编著. —北京: 电子工业出版社, 2010.10

ISBN 978-7-121-11791-6

I. ①j… II. ①胡… III. ①企业管理—管理信息系统—指南 IV. ①F270.7-62

中国版本图书馆 CIP 数据核字(2010)第 175629 号

责任编辑: 董 英

印 刷: 北京东光印刷厂

装 订: 三河市皇庄路通装订厂

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16

印张: 22.75

字数: 549 千字

印 次: 2011 年 4 月第 2 次印刷

印 数: 5 001~6 500 册 定价: 59.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线: (010) 88258888。

本书作者从2005年即开始在国内最大的企业应用软件集团之一负责 workflow 引擎内核及其周边系统的创造性研发工作，并因此获得了2006年度北京市中关村科技园经济技术创新标兵等荣誉。

随着对 workflow 管理系统研究开发和实践应用的深入，不仅从自主研发的 workflow 管理系统中获得了大量“深入骨髓”的第一手体验，而且由于工作原因从多个国内、国外的工作流产品中吸取、借鉴了众多的第一线应用实战经验，这不仅涵盖了多种商业工作流产品，也包括众多开源工作流产品。

因为工作的机会，作者曾作为 RedHat JBoss 产品应用架构师有幸深入地“解剖”了 jBPM 系列产品并且为国内多个 jBPM 应用项目提供咨询、培训等服务。作者发现在许多优秀的工作流产品中已经实现了的设计、计划实现的创意以及许多让人忍不住“击节叫好”的思想，都已经被 jBPM 系列产品做到了！当前，在 Red Hat JBoss 项目组工程师、架构师们的不懈努力下，jBPM 已经发布了第4个大版本，最新的 jBPM4 进一步克服了 jBPM3 的固有缺陷（这在书中会多次提到），并且更加“变本加厉”地增强和优化了 jBPM——这个世界上首屈一指的开源工作流产品的功能。

因此，作者“忍不住”、也“不得不”将 jBPM4 这个优秀工作流产品的应用、开发技巧以及自己对 workflow 技术的经验、体会编写成书。从某种程度上来说，本书也是作者多年“workflow 职业生涯”的一个里程碑和总结。

- 对于想快速入门的企业流程开发人员来说：
 - 本书将使您快速了解什么是 workflow、BPM 和 jBPM，以及它们的发展历程。当然，重要的是使您明白它们能为企业信息化做什么，开发人员该如何抉择。
 - 本书内容从下载、安装、配置 jBPM4 到流程设计、程序开发、单元测试、应用部署、调试运行，直至管理、监控、优化、扩展，使您掌握利用 jBPM4 开发企业流程的全生命周期过程。
 - 本书全面系统地为您介绍 jBPM4 的 Service API、活动和支持行为，通过手把手的实例练习把 jBPM4 企业流程架构的思维植入您的脑中。
- 对于已经在使用 jBPM（包括 jBPM3）开发的、并渴望把 jBPM4 “玩弄于股掌之

间”的研发者来说：

- 抛弃不尽如人意的、稍显啰唆的 jBPM3，来尽情拥抱架构更为优雅、更为易用、更适于扩展的 jBPM4 吧！为什么不呢？
- 本书为您精选了近年来众多企业在“实际”使用 jBPM 过程中遇到的最棘手、最纠结的问题，这包括业内人士所谓的“中国特色工作流”的经典问题，根据作者的经验，这些问题导致了大量 jBPM 项目选型、实施的流产乃至失败……本书将以 jBPM4 设计者的理论思想为前提，提出解决思路、方法乃至实战例程。
- 本书不失为一条企业级应用架构师的进阶之路，因为 jBPM4 不仅仅是一个平台（Platform），更是一个框架（Framework），它的代码、它的设计无处不蕴含着世界顶级企业应用架构大师的精华思想。同样，因为作者对 jBPM 设计理念的认同、优雅架构的努力追求，所以将力求在介绍中为您起到抛砖引玉的效果。
- 本书亦可作为 jBPM4 的工具手册在您的计算机旁伴随您进行企业流程研发之旅。

需要强调的是，本书中作者在介绍 jBPM4 的新功能时，会经常性地提到 jBPM3 与 jBPM4 的差异，以帮助广大 jBPM3 的“铁杆用户”消除思维定势，快速把握新版本的变化、从根本思想上“升级”到 jBPM4。同时也能让第一次使用 jBPM4 的用户体会到 jBPM 系列产品发展的“沧桑历程”，知其然（新功能）亦知其所以然（为什么要变更或加强以前的功能）。

本书由胡奇编著，参与具体工作的还包括王斌、万雷、赵楠、周华樟、朱诚、刘利平、李伟、袁学东、刘浩、杨建平、于洪丹、黄胜、田洪铭、陈培帅、黄北军。业内专家 PCCW Solutions, senior consultant, Cyril；盛大在线公司架构师，胡长城；搜狐福州分公司 Java 架构师，林良益；中通软联信息科技有限公司创始人兼首席架构师，马越；GrapeCity 集团技术管理组高级架构师，王瑜；北京天大天科科技发展有限公司首席架构师，吴俊；百度公司软件工程师，徐会生；TIBCO 中国开发中心开发部门经理，赵亮；上述 8 位同志对本书提出了非常诚恳的意见和建议，本书能够面世与他们的辛勤工作和专业建议密不可分。同时我还要感谢我的太太王少玲女士，她在本书的写作过程中给了我极大的支持和鼓励，并参与了第一章的校稿工作。最后将本书献给我刚出生的女儿胡清扬小朋友，愿她健康成长，好好学习，天天向上。

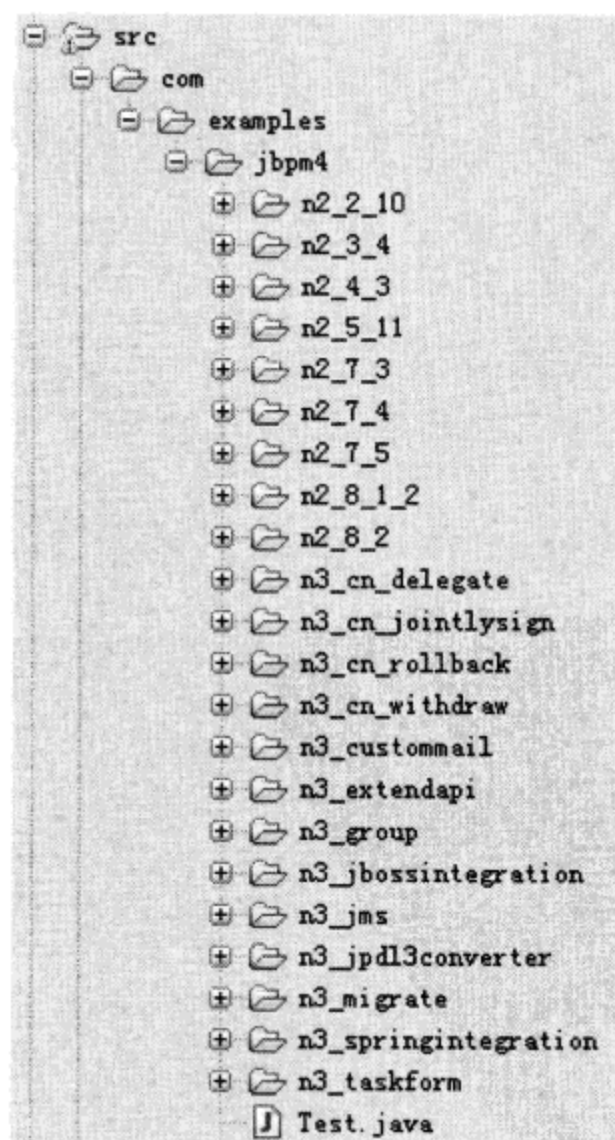
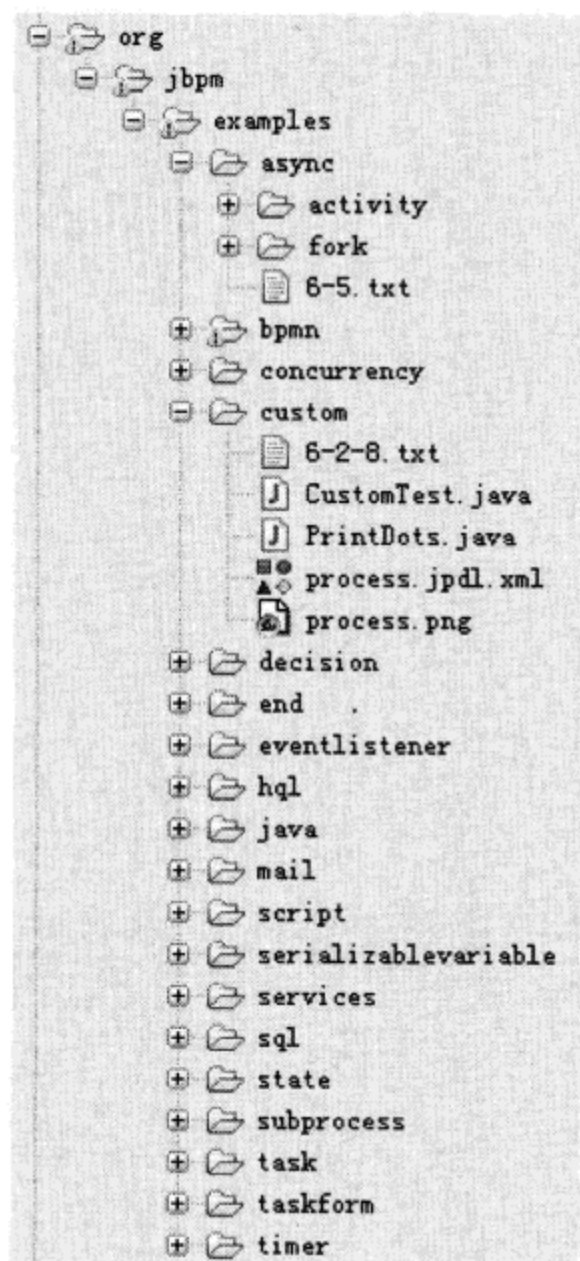
如果您对本书的内容有任何问题或反馈，欢迎通过电子邮件联系我们：
sharepub@126.com。

文件使用说明

读者可以先去下面这个地址下载一个压缩文件 jbpm-4.3.zip:

<http://sourceforge.net/projects/jbpm/files/jBPM%204/jbpm-4.3/jbpm-4.3.zip/download>

这个 jbpm-4.3.zip 解压缩后有 137.9MB 大小, 这是 jBPM4.3 的源代码、文档、依赖库、示例代码等所有官方的东西。因为是遵循 LGPL 开源协议的, 读者可以直接下载该文件。解压之后, 我们可以看到文件中有个 examples 目录, 为官方的示例代码, 结构如下(左)图所示。



本书第 6 章的大部分示例代码来自于此, 而上(左)图中的 6-5.txt 和 6-2-8.txt 正

是加入的用来表明此代码是第几章节所引用的。我们再通过 www.dozan.cn 网址下载本书的配书文件，解压缩到本地电脑。从文件目录结构可以看出，我们对 `examples` 目录做了扩充（因为要依赖官方 `examples` 的环境），在配书代码中我们还加入了本书其他章节的示例代码（具体路径为 `examples\src\com\examples\jbpm4`），如上（右）图所示。`n2_2_10` 表示第 2 章第 10 节的代码。`n3_cn_delegate` 表示中国特色工作流“代理人”部分的代码，为了方便起见，我们直接以面对的问题来命名，这样读者可以有针对性地查找并解决问题。

提示：本书涉及代码可到博文视点公司网站（www.broadview.com.cn）下载。



许久不曾买书，这本《jBPM4 workflow应用开发指南》是我迄今为止最期待的一本书。本书循序渐进地介绍了 jBPM，从 workflow 的基础知识开始，到最后的中国特色 workflow 的 jBPM 实现，让人不睹不快。可以说，它是一本很好的 jBPM 参考手册，能够帮助更多的人来学习和掌握 jBPM。

陈勇

就职于北京拓尔思信息技术股份有限公司，java 开发工程师

本人从六七年前就开始接触 workflow 系统的设计与开发，一共参与过三套 workflow 系统的设计与开发过程，应用涵盖了 OA 系统、行政审批系统、ERP 系统。在这么多年的 workflow 系统的设计与开发过程中，感触颇多，其中有如下两点是体会最深的：

1. 从最开始只关注一个个具体业务流程在自己设计的 workflow 系统中怎么实现，到后来渐渐关注 workflow 系统对企业整个业务流程重组（BPR）的巨大推动效应和企业应用集成（EIP）的强有力支持，这个过程我用了五六年。《jBPM4 workflow应用开发指南》的作者在第 1 章就对这点进行了深入的阐述，让读者在十来分钟的时间内快速完成了这个认识的转变，可以使后续的工作流系统的设计与开发者站在更高的起点去设计与开发适合于自己项目的工作流管理系统。

2. 在中国进行 workflow 系统的开发尤其不易，特别是政府与国企使用的 workflow 系统。用户提出的许多“非常有理”的特色功能往往会让 workflow 系统的设计与开发者焦头烂额，难以在 workflow 系统的规范性与用户要求的“灵活性”之间加以取舍。很显然《jBPM4 workflow应用开发指南》作者的实战经验非常丰富，明确地在本书中对中国特色 workflow 的 jBPM 实现进行了详细的说明。

所以《jBPM4 workflow应用开发指南》这本书在理论认识与设计思想、具体分析与

细节实现这两点上有比较好的平衡，充分体现了作者在工作流管理系统设计与开发上的功力。

Cyril

就职于 PCCW Solutions, senior consultant

终于盼来胡奇的新作《jBPM4 workflow应用开发指南》出版上市。在作者刚刚着手开始写书的时候，我们俩就针对大纲内容有过探讨，当时定的基调就是“实用”。

“实用”的本质就是“实战经验的总结和抽象”。特别对于 workflow应用来讲，看似就是几个节点的连接和组合，但是实际却是变化多样的，不同的行业领域或应用场景之间的差异性太大，而需求又特别繁杂。这也是为什么很少有一款 workflow产品能够适应所有行业。有的 workflow产品主要面向政务办公中的公文处理流程；有的则面向电信业务的高性能处理流程；有的则面向应用系统过程集成的处理流程；有的则面向制造业中高度自动化控制流程……

在这种情况下， workflow产品要想在“简单易用”和“灵活扩展”之间找到平衡就非常难。但 Tom 先生（jBPM 的创造者之一）却完成了这个壮举，缔造了精美小巧的 jBPM。在 jBPM 中，除了过程调度这一核心模块无法扩展以外，几乎流程相关的所有点都可以扩展活动类型、活动执行方式、事件行为、Spring 整合、存储等。所以开发者可以很容易基于 jBPM 构建出有自己特色的流程框架，以满足特殊业务领域模式。本书中，胡奇用了几个章节的内容，为读者诠释了如何基于 jBPM 构建属于自己的 workflow平台。

提到 jBPM4，就不能不提到 PVM（Process Virtual Machine, 流程虚拟机）。PVM 是新一代流程引擎架构设计理念，而这是由 Tom 先生在 jBPM4 中首创的，现在这种 PVM 设计理念在越来越多的开源和商业 workflow或 BPM 产品中体现出来，比如 Orchestra BPM, Oralse BPM Suite 11g, Activiti BPM 等。在书中，胡奇为大家详细诠释了 PVM 的原理和 jBPM4 的实现，值得参考阅读。

jBPM 本身是一款开源的 workflow和 BPM 组件，但如何真正结合实际情况需求和场景，把 jBPM 运用好却是个很大的挑战。在现实中，开发人员的实施经验不足、jBPM 专业开发人员稀少、指导参考资源匮乏，对项目实施质量和进度影响颇多，甚至导致部分项目失败。而在本书中，胡奇不仅对 jBPM 做了深入透彻的介绍，并分享了自己

多年的 jBPM 成功实施经验，为大家的工作流应用项目实施提供了宝贵经验借鉴，很值得参考学习。

胡长城

就职于盛大在线公司，负责盛大开放平台的整体架构，近十年的工作流研发实施咨询经验

翻开这本书的目录，从对工作流理论发展的理解到 jBPM4 的设计思想，从 Step By Step 的图示化安装指南到特定的“中国式企业应用”场景案例，从简单的基于 Web 页面设计器的流程定制，到与 Spring, JMS 的复杂整合。对读者而言，该书做到了一步一步带入门，而后逐渐展开视野、深入机理，使人“知 jBPM 然，也知其所以然”。

作者对工作流模型，尤其是 BPM 的理解，功力十分深厚。他以适合于中国学生理解的思维方式，为读者解读了 jBPM 设计，展现了 jBPM4 的 API 的全景视图。更难得的是，书中的“中国特色工作流的 jBPM 实现”这一章非常精准地直击目前中国企业信息化过程中在工作流系统上经常遇到的老大难问题，让人读后拍手称快。

不是所有的书籍都能从工作流的基础概念讲述到开发中遇到的现实问题的，这本书做到了。它既适合刚刚接触流程引擎的初学者入门，又适合于对 jBPM3/4 有一定使用基础、需要深入学习的在职从业人员。它是本人所读到的国内相关书籍中对 jBPM4 描述最为全面、最为深入的作品之一。

林良益

就职于搜狐福州分公司，Java 架构师，负责网络搜索与 SNS 底层框架设计
开源项目《IKAnalyzer 中文分词器》《IKExpression 表达式解析器》作者

jBPM 作为历史最悠久，功能最强大的开源工作流引擎一直拥有着国内外广泛的使用群体。2005 年我在美国时因为第一次在大型企业应用中使用了 jBPM 与 Seam 技术，还得到了 jBPM 团队所在的 JBoss 公司 2006 年度世界创新奖。2007 年回国与本书作者成为同事后我们一起为国内大量客户进行了有关工作流引擎的咨询工作。那时候本书

作者就开始不断总结客户所面对的问题与需求，并一直想把 jBPM 真正结合本土业务特色发扬光大。如今本书终于问世，过去一直缠绕着众多国内程序员的一些如回退、会签、自由流等“中国特色”的问题都通过 jBPM4 有了明快的解决之道。多年来 jBPM 工作组与本书作者共同的厚积薄发恰如其时地为国内程序员在实际工作中快速应用开源技术又提供了一个宝贵的武器。

马越

就职于中通软联信息科技有限公司，创始人兼首席架构师

<http://www.jbosschina.org> 社区创始人；2006 年 Red Hat/JBoss 世界创新大奖获得者

美国 10 年大型商业软件构架经验；第一个将 Seam 及 jBPM 框架引入大型企业应用的开拓者

熟悉企业应用开发的人都知道，工作流是企业应用开发中的一个核心概念。从办公自动化领域的公文审批到跨企业的异构系统集成，背后都可以见到工作流的身影。jBPM 作为一套开源工作流产品，已在各个行业得到了广泛深入的应用，其工业标准的品质也已成为开源世界中的范例之一。

本书作者具有坚实的工作流理论基础，同时又具备丰富的基于 jBPM 的系统设计和开发经验。更难能可贵的是，作者能够很好地将理论知识与软件开发实践结合起来，通过对理论的讲述切入实践，然后通过对实践的讲解进一步巩固理论，进而引发读者思考如何根据理论进一步扩展实践。与此同时，本书还就国内工作流应用的特点，结合 jBPM 产品给出了自己的解决方案。这些来自开发一线的实践积累和经验总结，对于读者来说无疑具有宝贵的价值。

另一方面，针对某一软件产品的介绍性书籍往往容易被写成一本介绍如何使用该产品的操作指南，冷冰冰的操作步骤描述，再加上直接从外文资料上截取下来的翻译内容，让人阅读起来觉得了无生趣。而本书作者可以说在提高阅读体验方面花了很大心力。语言朴实，描述准确，包括设计演示场景和示例的时候，也是尽量考虑到读者的知识背景和阅读体验。介绍理论的时候不显得枯燥，描述操作的时候不觉得烦琐，无论是初次涉及工作流概念的新手，还是有一定工作流经验的设计师，都能从本书中有所收获。

首先十分感谢作者给我这个机会在他的作品即将问世之前做一些感想，也正好让我能在忙碌中抽空回顾一下这么多年在技术平台方面走过的路以及在 Workflow 方面的点点滴滴。因为本书是介绍 jBPM 的专业书籍，所以我谈不上给些什么评论，只是将对作者的印象和对书中内容引发的一些思考和回味分享给大家。

首先就要从我跟作者如何相识的初次合作谈起了。记得那应该是 2005 年，我们经历了两年多搭建面向 ERP 项目的研发平台初具规模，正在 10 多个大型项目中服务。当时从底层技术框架、代码生成工具、组织权限等诸多技术问题都得到了充分的考虑和解决。当然肯定少不了对 Workflow 业务的解决。当时我们选用的 WfMC 的 XPD L 语法，从引擎、设计器、监控器、模拟器、组织模型全套都实现了。最初因某些项目原因采用的 .NET，后迁移到 Java 平台下，并且以 Hibernate 做存储，也是采用的微内核和状态机的设计思路。提供了一整套的 API（类似 jBPM 等主流工作流引擎提供的方式）。但在几个大项目实际使用中发现这种纯粹 API 二次开发的方式，基于封闭工作流系统无法高效地支撑业务快速开发及流程仿真重组。

对业务系统快速实施有价值的是整合了组织定义、流程定义、业务表单定义、业务数据权限定义工具等全集成的定制系统，不用通过编写代码业务需求人员通过拖曳即可完成各种层级集团应用的定义，并适当设置一定的业务脚本即可完成业务系统的定义。所以我们在 2005 年 6 月启动了这件事情，基于已有的组织权限、工作流基础上加入业务表单定义及整合的设计定义功能。然后在公司内部到处挖人，我习惯于以技术 geek 的方式在公司内部到处诱人，都是先通过非正式的方式从技术角度找到有想法感兴趣的人，然后再用各种手段搞到这个人加入。

记得那个时候一个意气风发、满怀抱负的年轻小伙子给我留下了深刻的印象。他刚刚做完一个电信的项目，爱好非常广泛，JS 很好，也会 Flash，想法特别多，手特别快。所以我就通过各种手段搞到这个家伙加入我们，那就是年轻时的作者。很快他先搞定了业务表单定义的技术攻关，可能当时我们是最早使用 XmlHttpRequest 的（Ajax 的核心），然后基于 Workflow 如何接收表单传递过来的数据，然后接手了整个 Workflow 的优化及实现各种变态的中国特色的工作流需求。我们在 10 月左右发布了一个完整的

版本，支持了上面讲到的那些功能。现在回头看看大家都很神奇，10 个人不到 3 个月就搞定了这么复杂的东西，也许换到今天开源技术如此成熟可能都未必能做到。之后这个家伙就一定要坚持在专业上坚持以 Workflow 为方向不断成为专家。

再一晃就是多年以后的事情了，我也从搭建 ERP 研发平台到互联网 SNS 大型数据平台又转回到专属行业 ERP 产品平台。不论管理大型团队多少年，始终秉承着对技术的痴迷，从未放弃对任何技术细节的关注，也越来越推崇中庸、推崇妥协。

这期间我们有过断断续续的联系，也曾经为一些点子一起疯狂、一起兴奋，不过终究没有找到合适的机会再次合作。收到作者的邀请时，十分地受宠若惊，当时手头的事情也特别多。而且类似的书我也见得特别多，不外乎就是英翻中，所以就根本没放在心上。直到后来拿到作品仔细阅读了一番，渐渐地能体会和重现这个家伙几年的历程——一直在专业（Workflow）的道路上孜孜以求。

当然文中难免有大量资料的复制，但是能花大量心血组织起来，如此翔实全面地介绍已实属不易。而且加上了这家伙从当年我们一起做流程系统时的经验和他在 jBPM 系列产品应用实施上多年的咨询经历。

最近我也在平台中的多处集成了 jBPM 用于解决传统审批流，跨业务域集成 BPM，以及 ETL 流等，当然不会孤立使用 Workflow，肯定是结合 ESB、MDA、规则引擎等一起使用。一直想给大家从概念到 Workflow 的历史、再到如何运用 jBPM 解决问题等进行一个全面的讲解，正好拿到这本书，省了我很大的力气。

这几年 IT 业内浮躁风气、快餐文化盛行，用过 Spring、Struts、jBPM 就敢说自己是架构师。而很少有人能够踏踏实实潜心研究各种技术的根源，为什么要这么设计，如何扩展更为有效。从我们做平台的人的角度来看简单会用没什么，关键是能理解精髓并扩展或改造才是本事。

像作者这样致力于在国内传播 Workflow 的模型，让更多的人少走弯路，快速上手，并不断将多年研究的心得和成果同大家分享的人太少了，国内太少这种人了。也许这本书可以为不少人提供了解 jBPM 的素食套餐。尤其本书中“第 20 章 中国特色工作流的 jBPM 实现”十分具有实战意义，是特别值得国内使用 jBPM 做应用的人们参考的最佳实践。

笔触于此，抬眼已很晚了，一气呵成就先写这么多。让我们一起读完全书后再回来一起思考和回味下面这两个问题吧！

- 有想法、专注、愿意分享的年轻人，我们应该有怎样的 IT 人生？
- 业内规范的演化进步，我们又会遭遇怎样的 IT 进程？

吴俊

就职于北京天大天科科技发展有限公司，首席架构师

很高兴向大家推荐《jBPM4 工作流应用开发指南》这本书，jBPM 是目前国内公司使用最多的开源工作流引擎，不过虽然越来越多公司选择将 jBPM 加入自己的项目或产品中，却发现总是被同样的问题绊住手脚。这本书为尚未接触过工作流领域的同仁们开启了大门，它以浅显易懂的文字，循序渐进地介绍了流程领域的应用知识，并提供了实例辅助大家学习，最后还特别针对国内的特殊流程需求提出了对应的解决方案，帮助大家快速越过这些障碍。

徐会生

就职于百度公司，软件工程师

<http://www.family168.com/> 站长

对于企业级 IT 应用系统而言重要的内容是业务活动流程、人员组织架构、业务数据标准定义。工作流软件帮助企业应用开发者方便地处理实现以上三个部分的内容，可以使用图形化流程编排的方式定义业务流程，定义企业内部组织结构，人员角色和角色关联的工作任务，在流程中连接企业内部的其他系统，让标准的业务数据流在企业内部的各个系统中流转，完成企业运作的各项工作。

类比编程实现函数的数据处理流程，使用工作流软件工具编排实现业务处理流程是企业应用开发者必须具备的一项技能。随着企业应用系统复杂度的不断提升，只提三尺编程之剑，是无法纵横于软件江湖的。每一个在企业应用领域的技术工作者都需要更先进、更强大的技术来武装自己。工作流是企业应用中非常重要的技术，了解这门技术，包括概念、功能、发展过程、基本原理、技术标准、产品工具的使用、开发技能和经验共享等，就是一项紧迫的任务了。

学习需要深入本质，举一反三。我们不需要学习所有主流 workflow 软件，选择一个规模适度（不要像 IBM 软件套件那么让人生畏）、应用广泛、代码开源的产品来切入，这是个明智的选择。本书适合于想学习 workflow 技术，并且将使用 jBPM 来做应用开发的读者阅读学习。对于从事 workflow 产品开发工作的开发者来说也可以从第 1、第 12、第 13、第 20 章读到一些自己感兴趣的内容。如果说 jBPM 是开发企业 workflow 系统的利器，那么只有开发者熟练掌握了这个工具才真正有用。本书就是个很好的选择，它可以帮助你了解学习和掌握这门技术及相关工具。不只局限于对工具使用的介绍，本书将会告诉你这项技术的方方面面，开启一扇步入 workflow 领域的大门，引导你获悉内功本质，熟练操作招式，在之后的企业应用开发过程中把这项技术用好、用准。

赵亮

就职于 TIBCO 中国开发中心，开发部门经理



第一篇 jBPM workflow 开发基础	1
第 1 章 workflow 基础	2
1.1 workflow 概念	2
1.1.1 workflow 管理思想之于企业现代化管理	2
1.1.2 workflow 技术在企业中的应用	5
1.1.3 如何从一个开发者的角度看 workflow 技术	6
1.2 workflow 管理系统的发展历程	9
1.2.1 workflow 管理系统参考模型	11
1.2.2 BPM	15
1.3 开源 workflow 选型	16
1.4 jBPM	19
1.4.1 jBPM 前世今生	19
1.4.2 关于 jBPM4 您需要知道的	19
1.5 小结	23
第 2 章 安装和配置 jBPM4	24
2.1 jBPM4 安装先决条件	24
2.2 快速开始吧	26
2.3 安装脚本详解	27
2.3.1 关于配置文件	30
2.3.2 关于依赖库	31
2.4 安装到 JBoss	31
2.5 安装到 Tomcat	32
2.6 基于 Web 的 Signavio 流程设计器	33
2.6.1 jBPM Web 流程设计器简介	33
2.6.2 独立安装 Signavio	34
2.6.3 配置 Signavio	34
2.7 用户自定义 jBPM Web 应用程序	35
2.8 安装 jBPM 数据库	35
2.8.1 新数据库安装	36
2.8.2 升级旧的数据库	36
2.9 安装图形化流程设计器 (GPD)	37

2.9.1	获取 Eclipse	37
2.9.2	在 Eclipse 中安装 GPD 插件	37
2.9.3	配置 jBPM 运行环境	38
2.9.4	添加 jPDL4 Schema 校验	41
2.9.5	导入和使用范例	41
2.10	例程: jBPM HelloWorld	43
2.11	小结	45
第 3 章	使用 jBPM 图形化流程设计器设计流程	46
3.1	创建一个新流程	47
3.2	编辑流程定义源	49
3.3	例程: 设计一个“复杂的”业务流程	49
3.4	小结	53
第 4 章	把流程部署到服务器上去	54
4.1	部署流程定义和资源文件	54
4.2	部署流程 Java 类的 3 个方法	57
4.3	例程: 部署业务流程定义	58
4.4	小结	61
第 5 章	使用 jBPM4 Service API 控制流程	62
5.1	流程定义、流程实例和执行的概​​念	62
5.2	流程引擎 API	64
5.3	利用 API 部署流程	67
5.4	通过 API 删除已部署的流程	69
5.5	使用 API 发起新的流程实例	69
5.5.1	发起流程实例的常规方法	70
5.5.2	指定业务键发起流程实例	70
5.5.3	指定变量发起流程实例	71
5.6	唤醒一个等待状态的执行	71
5.7	任务服务 API	72
5.8	历史服务 API	75
5.9	管理服务 API	76
5.10	查询服务 API	77
5.11	例程: 利用 jBPM Service API 完成流程实例	78
5.12	小结	80
第 6 章	掌握 jBPM 流程定义语言	81
6.1	process (流程)	82
6.2	流转控制活动	84
6.2.1	start (开始活动)	85

6.2.2	state (状态活动)	86
6.2.3	decision (判断活动)	89
6.2.4	fork-join (分支/聚合活动)	97
6.2.5	end (结束活动)	102
6.2.6	task (人工任务活动)	107
6.2.7	sub-process (子流程活动)	120
6.2.8	自定义活动	132
6.3	自动活动	134
6.3.1	java (Java 程序活动)	135
6.3.2	script (脚本活动)	139
6.3.3	hql (Hibernate 查询语言活动)	144
6.3.4	sql (结构化查询语言活动)	147
6.3.5	mail (邮件活动)	149
6.4	事件	153
6.4.1	事件监听	155
6.4.2	事件传播	157
6.4.3	处理异常事件	159
6.5	异步执行	160
6.5.1	异步活动	162
6.5.2	异步分支/聚合	164
6.6	用户代码	166
6.6.1	用户代码的定义	166
6.6.2	用户代码的类加载	168
6.7	小结	170
第 7 章	流程变量	171
7.1	变量作用域	173
7.2	变量类型	174
7.3	变量的自动更新和序列化	175
7.4	例程: 用变量去控制一个流程的运行	177
7.5	小结	179
第 8 章	流程脚本	182
8.1	Java 统一表达式语言	182
8.1.1	语法特点	183
8.1.2	值和方法表达式	184
8.1.3	隐式对象	187
8.1.4	运算符和保留字	188
8.1.5	一些经典 EL 表达式的例子	190
8.2	例程: 用脚本去控制一个流程的运行	192

8.3 小结	194
第二篇 定制属于自己的流程——深入 jBPM4 扩展研发	195
第 9 章 jBPM4 扩展研发先决条件	196
9.1 深入应用 jBPM4 所需要知道的	196
9.1.1 如果您的业务基于复杂的规则，在 jBPM 中加入 Drools 吧	196
9.1.2 抉择，是否使用 BPEL	197
9.2 Maven 仓库和 Java 依赖库	199
9.3 小结	200
第 10 章 深入 jPDL 和 jBPM Service API	201
10.1 timer（定时器）能为您做什么	201
10.1.1 持续时间表达式	202
10.1.2 工作日历	202
10.1.3 定时转移	204
10.1.4 定时事件	205
10.1.5 工作日历定时	207
10.1.6 定时重复	208
10.2 使用 group 活动编组流程	209
10.3 如何在活动中调用 EJB 方法	214
10.4 使用 jms 活动	215
10.4.1 模拟 JMS 服务	217
10.4.2 JMS 文本消息	219
10.4.3 JMS Object 消息	220
10.4.4 JMS Map 消息	222
10.5 历史会话监听链	223
10.6 自定义 Web 任务表单	225
10.6.1 基本思路	225
10.6.2 表单格式	226
10.7 流程实例的自动迁移	228
10.7.1 简单的流程实例迁移	230
10.7.2 终止流程实例运行的迁移	232
10.7.3 应用活动映射的迁移	234
10.7.4 自定义迁移处理器	236
10.8 小结	239
第 11 章 升级 jBPM3 到 jBPM4	240
11.1 你所要知道的升级局限性	241
11.2 流程定义转换工具	242

11.2.1	命令行执行	242
11.2.2	Java 编码执行	243
11.3	jBPM3 到 jBPM4 的语义变更及翻译	244
11.4	小结	246
第 12 章	流程虚拟机原理	247
12.1	PVM 的架构	247
12.2	PVM 的实现	250
12.3	小结	253
第 13 章	jBPM4 的设计思想	254
13.1	API 设计	254
13.1.1	活动 API	256
13.1.2	事件监听 API	256
13.2	执行环境设计	257
13.3	命令设计	258
13.4	服务设计	259
13.5	历史流程处理原理	262
13.6	数据持久化设计	263
13.6.1	jBPM4 流程定义资源和实例运行时数据表	264
13.6.2	jBPM4 流程历史数据表	265
13.6.3	jBPM4 身份认证数据表	266
13.6.4	jBPM4 引擎属性数据表	267
13.7	例程：扩展 jBPM4 的 API 满足客户化的需求	268
13.8	小结	270
第 14 章	按需而配 jBPM4	272
14.1	配置文件设计概要	273
14.2	配置工作日历	274
14.3	配置身份认证组件（组织适配器）	274
14.4	小结	277
第 15 章	异步工作执行器	278
15.1	设计原理	278
15.2	配置使用	280
15.3	小结	281
第 16 章	深入 jBPM4 电子邮件支持	282
16.1	电子邮件的产生	282
16.2	电子邮件服务器	285
16.3	电子邮件扩展	287

16.4 小结.....	289
第 17 章 系统日志	290
17.1 配置日志.....	290
17.2 日志输出级别	292
17.3 Java Logging API 日志	292
17.4 利用持久化层日志进行调试.....	294
17.5 小结.....	295
第 18 章 jBPM4 与 Spring 框架集成.....	296
18.1 集成的目标	297
18.2 为集成配置 jBPM4.....	297
18.3 为集成配置 Spring.....	299
18.4 使用.....	301
18.5 测试.....	302
18.6 小结.....	302
第 19 章 jBPM4 与 JBoss 应用服务器集成	303
19.1 流程定义打包部署	303
19.2 在 JBoss 企业级编程模型中使用 jBPM4.....	304
19.3 小结.....	306
第 20 章 中国特色工作流的 jBPM 实现	307
20.1 退回.....	308
20.2 取回.....	313
20.3 会签.....	318
20.4 委派.....	326
20.5 自由流	331
20.6 小结.....	332
附录 A jBPM 术语.....	334

第一篇

jBPM workflow 开发基础

随着企业信息化的深入，越来越多的企业流程需求已经无法用普通的 OA（Office Automation，办公自动化）系统来满足，工作流（Workflow）/ BPM（Business Process Management，业务流程管理）系统逐渐“流行”起来，在 ERP（Enterprise Resource Planning，企业资源计划）、CRM（Customer Relationship Management，客户关系管理）、EAI（Enterprise Application Integration，企业应用集成）等经典企业应用领域，工作流技术的应用已经深深扎根、不可缺少。甚至在互联网领域，我们也能逐渐发现工作流技术的“入侵”，例如内容审核流程、注册引导流程……可以说，在当前的时代，不了解工作流，就不好意思自称 IT 人；不懂得工作流的 IT 技术人，在自己的职业发展中就永远会有一块“短板”、“盲区”。

本篇除了讲解工作流的入门知识外，还以 jBPM4.3 版本为基础，重点介绍目前 jBPM4 宣布稳定支持的功能及应用方法，并辅以大量的实战例程，帮助读者快速掌握 jBPM4 框架的安装、配置、部署，以及基于 jPDL4 流程定义语言的业务流程设计、开发，使读者迅速成为一名合格的 jBPM4 应用开发者。

数字图书馆
PDG

本章将为您开宗明义地介绍 workflow 这门科学，使您了解这门“很有前途的”技术的概念、发展历程以及目前的状况，并给您一个选择 jBPM 的理由。

1.1 workflow 概念

workflow 管理联盟（即 WfMC，这个组织在后面会介绍）对于“workflow”这个概念的经典定义为：全部或者部分由计算机支持或自动处理的业务过程。

workflow 管理系统（Workflow Management System, WFMS）是这样一个软件包：它通过执行经过计算的流程定义去支持一批专门设定的业务流程。它被用来支持定义、管理和执行 workflow。

因此，对于我们来说，workflow 管理系统的目标是：管理工作的流程以确保工作在正确的时间被期望的人员所执行——在自动化进行的业务过程中“插入”人工的执行和干预，可以说正是 workflow 管理系统的价值所在，也是 workflow 系统开发者的主要工作内容。

1.1.1 workflow 管理思想之于企业现代化管理

提示：workflow 经常与“业务过程重组（BPR, Business Process Re-engineering）”联系在一起。BPR 是关于企业（组织）核心业务过程的评估、分析、模拟、定义，以及其后的操作实现。尽管不是所有的 BPR 都是采用 workflow 技术实现的，但 workflow 技术是实现 BPR 的最佳方法。这主要是因为 workflow 技术提供了业务过程逻辑与 IT 操作支持的分离，从而可以通过修改过程规则来重定义业务过程。当然，workflow 技术并不只在 BPR 中采用。

专业分工及组织分层机制是西方国家大规模工业化改造成功的前提。在托福勒的《第三次浪潮》一书中，对“大就是好”的大规模生产时代进行了详尽的描述，并预言该时代将终结。但十多年过去了，大企业并未消失，而是采用了 BPR 及其他先进思想使自己获得了新生。

“铁路警察，各管一段”式的专业分工、精细化的组织机构、职能部门制度是造成企业（特别是大型企业）僵化的主要原因。企业僵化主要有如下特征：

- 每个员工取悦的是自己的“上司”，因为上司掌握员工的地位、薪酬，每个员工可以冷落顾客，但丝毫不敢怠慢“领导”。
- 职能部门以专业划分，在企业中形成一个个的利益中心，部门之间的边界极为明显，在一项业务涉及多个部门时，若发生利益冲突，各部门可以把全公司利益放到一边，维护自己的利益。协调内部矛盾耗去了大量的企业精力。
- 为了加强“内部管理”，企业建立大量制度及审批手续，但几乎找不到几条是为了更好、更快地向顾客提供优质服务的条款，基本上全部是监督内部职工的。层层审批、众多领导“签字”的制度，大大降低了企业的运行效率，也是推卸责任的最好方式。
- 所有员工追求的是“当官”，一旦“升迁”，地位、名义、薪酬均将“旧貌换新颜”，否则“人微言轻”，一切名义、待遇无从谈起。在此情况下，员工要么跳槽，要么混日子，这是现代企事业单位官职重叠的原因之一。
- 公文旅行、文牍主义存在于各个企业，对公文、报告、表格的检查、校对及控制是企业工作极其重要的基础工作，可以压倒一切。大量的人力、物力投放在此？他们都忘了企业的真正生存目的是什么！

当然，以上是属于较为严重的情况，在国内企业中比较常见。而西方一些大型企业最常见的问题是各种制度均已健全，但已出现老化，甚至有的已严重阻碍企业的发展，增大了企业的运营成本，使企业失去了竞争力。

事实上，一些西方企业已经意识到此类问题，尝试了各种解决办法，在 BPR 理论刚出现时，就立即进行实践，甚至到了“狂热”的地步，许多企业获得了新生，如 IBM、HP、沃尔玛、宝洁、711、福特等。

实施“BPR/ workflow 管理”的主要原则有三个：

- 1) 以顾客为中心——全体员工建立以顾客而不是“上司”为服务中心的原则。顾客可以是外部的，如零售商业企业，柜台营业员直接面对的是真正的顾客；可以是内部的，如商场的理货员，他的顾客是卖场的柜台小组。每个人的工

作质量由他的“顾客”做出评价，而不是“领导”。

- 2) 企业的业务以“流程”为中心，而不以一个专业职能部门为中心进行——一个流程是一系列相关职能部门配合完成的，体现于为顾客创造有益的服务。对“流程”运行不利的障碍将被铲除，职能部门的意义将被减弱，多余的部门及重叠的“流程”将被合并。
- 3) “流程”的改进必须具有显效性——改进后的流程必须提高效率，消除浪费，缩短时间，提高顾客满意度和公司竞争力，达到降低整个企业运营成本的目标。

最终，“BPR/ workflow管理”可能对企业的各方面产生如下积极影响：

- **对组织机构的影响**——业务流程管理对企业的“冲击”是巨大的，实施后企业的职能部门数量和级别会压缩，企业的组织机构不再是“多级管理”，而是呈现“扁平化”趋势。以专业技术组织的职能部门仍将存在，但部门之间的“边界”大大淡化。部门经理的权力有限，一般只是制定战略、培训及管理人員，员工的直接服务对象是“顾客”，而不是“上司”。
- **流程团队（Process Team）在企业中体现出重要的地位**——按照一定的流程组成的团队活跃在企业经济活动中，这个“流程团队”可以是临时的，也可以是永久的。一个流程团队可以跨越许多专业部门，例如在一个计算机公司内，为了一个项目，可以由市场部、销售部、技术部、维修部、财务部等多部门共同组成一个临时的流程团队。这样，企业可以以一个整体面向用户，避免了在销售时，同一公司的不同部门络绎不绝地出现在同一个用户面前；而在维护系统时，用户则不知道去找谁。同样的，在一个商场内，对于某类商品的进货业务，可以由商品部、采购部、财务部、小组、库房等组成一个永久的流程团队，用以提高商品进货的效率和商品的适销度。
- **对人事管理及考核、薪酬制度的冲击**——由于采用“流程”为工作重点，对以官本位为基础的专业职能及人事管理体制，产生了极其猛烈甚至是残酷无情的冲击，分析并量化工作流程将是一项复杂及崭新的挑战，对各级管理人员的评定将不再是主观的判断，整个流程的执行结果将是人员考核、薪酬评定的新标准。
- **对员工的积极要求**——在运作中，员工将分为具有领导及沟通能力的“流程领导者”和各类应用专家，每个人可以根据自身特点选择自己的发展方向，这样只要认真努力，自然会拥有名誉及地位。例如在微软公司的项目组中，一个级别较低的 PM（项目经理）可以领导一个技术级别等同于比尔·盖茨的技术专家。在此情况下，每个人追求的可能将不再是各级“经理”、“处长”

等职务，而是各种“专家”等称号。

- **对企业管理方式的冲击**——国内有些企业有个误区，提起加强管理，就是制定出数大本《××企业管理制度汇编》，然后监督执行。但我们同样可以看到，许多管理制度健全并严格执行的国营企业仍然被市场无情地抛弃了，这就是只重局部管理、不看整体流程所造成的，可以称之为“见木不见林”。我们可以在事后埋怨体制，但事实上，整体流程的僵化往往是企业自己套在脖子上的绳索。

因此，真正的工作流管理思想对于企业的改造是全面彻底的，大部分僵化体制将被打破、重组。也许您要问，企业能够直面这样的现实吗？但无论如何，只有重视顾客、关心流程、高效响应的企业，才能生存在今后的市场中。

1.1.2 workflow 技术在企业中的应用

workflow 技术是一项快速发展的信息化技术，各种行业企业都在逐渐采用 workflow 管理系统。

workflow 技术的主要特点是过程的自动化处理（这些过程包含人与以机器交互为基础的人工活动）。

因此，目前 workflow 技术广泛应用于办公室环境中，例如保险、银行和行政管理等。其价值主要体现在：

- 1) 协助涉及多人（或多个部门）相关任务的工作执行。
 - 大部分 workflow 管理系统都有一个方便的机制，来生成和处理执行任务的电子表单，使各个部门人员能方便地通过这种机制实现交互，“参与”到业务流程中来。
 - 对于专注于 ISO 或者 CMM 认证的组织，通过这种方式使用 workflow 管理系统能够显著地提升“规范化流程”的运转速度，从而提高生产率。
 - 不用将业务过程用文字的形式记录在纸上——workflow 管理系统能使用户方便地通过流程建模实现业务过程的定义以及自动化执行。
- 2) 作为企业应用集成（Enterprise Application Integration, EAI）的平台。
 - 对于大型企业来说，各种各样的异构应用和数据运行在企业内部。
 - workflow 管理系统并不是专门的业务系统，但 workflow 管理系统和专门业务系统是互补的。
 - 大部分 workflow 管理系统具有结合专门业务应用、构建统一 EAI 平台的能力。

3) 嵌入式 workflow 引擎。

- “专门业务应用”将指定业务领域的相关业务流程固化在信息系统中。开发专门业务应用的公司可以考虑将 workflow 引擎嵌入到他们的专门业务信息系统中。
- 在这种情况下，workflow 引擎作为一个应用组件，对于应用的最终用户是不可见的，应用的最终用户也完全不需要知道 workflow 的存在。将 workflow 引擎嵌入到应用中的主要目的是为了加强应用的扩展性和系统的可维护性。

综上所述，引入 workflow 管理技术对于企业业务过程的提升主要表现在：

- 提高运转效率——业务流程在“自动化运行”过程中会暴露出一些业务流程中不必要的步骤。
- 较好的流程控制——使得大家执行标准的工作方法和跟踪审计成为可能。
- 改进客户服务——因为流程的一致性控制，提高了对客户响应的可预见性。
- 使企业变得“灵活”——方便企业业务流程重组。
- 促进业务改进——使得专注于流程的业务趋向于流畅和简单。

1.1.3 如何从一个开发者的角度看 workflow 技术

前面说了那么多思想、业务之类的东西，无非是想说明 workflow 技术对于企业有什么样的好处，但对于一个开发者，坦白地说就是一个写代码的人，workflow 管理对于企业业务过程改进的提升也许离他们有点远了。使用 workflow 技术体系开发软件系统对开发者又有何好处呢？

举个例子说明。

现在我们来看一个简单的业务——订货流程：

- 1) 客户提交采购订单。
- 2) 业务员执行订单处理。
- 3) 如果缺货，转工厂生产。
- 4) 仓库出货。
- 5) 物流发货。

整个流程如图 1-1 所示。

不使用 workflow 技术，从头开始开发这个订购流程的业务系统，我们需要：

- 每个活动节点都要开发交互界面和后台处理程序。
- 每次活动的流转都需要硬性判断下一步活动节点及其处理人。

- 每次操作都需要维护业务数据和流程的一些相关数据。

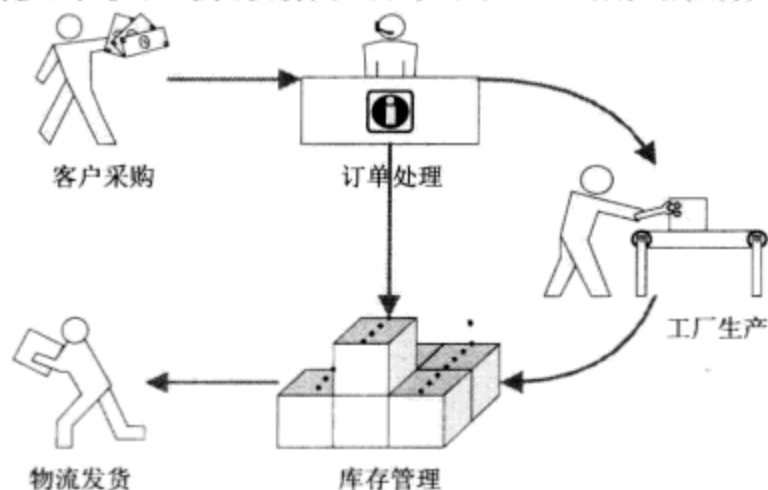


图 1-1 订货流程示意图

- 一旦业务流程变更，就需要大量地更改程序，甚至重新开发以适应新的需求。
- 监视、控制、分析流程处理情况的应用还需要单独开发，且成本不低。

结果这个业务系统可能是如图 1-2 所示的情况，请注意这还不包括监视、控制、分析流程的部分。

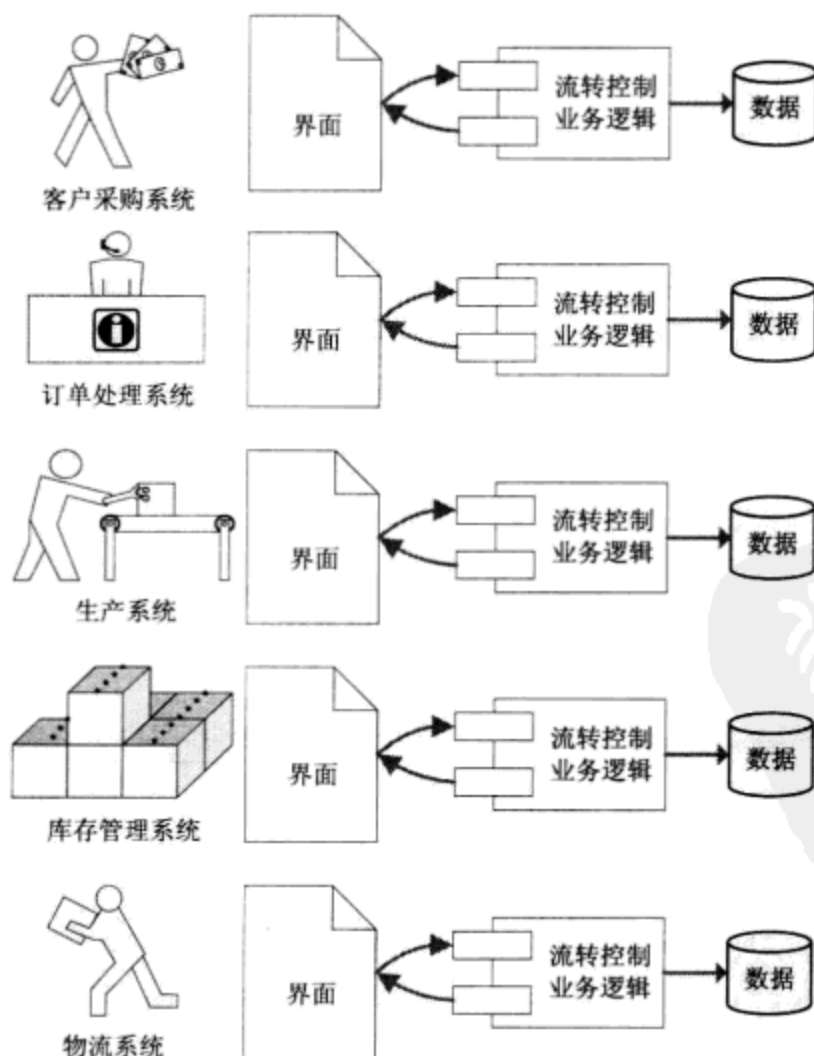


图 1-2 不使用 workflow 技术实现订货业务流程

下面我们看看使用 workflow 技术实现上述的订货流程将会是一种什么情况，如图 1-3 所示。

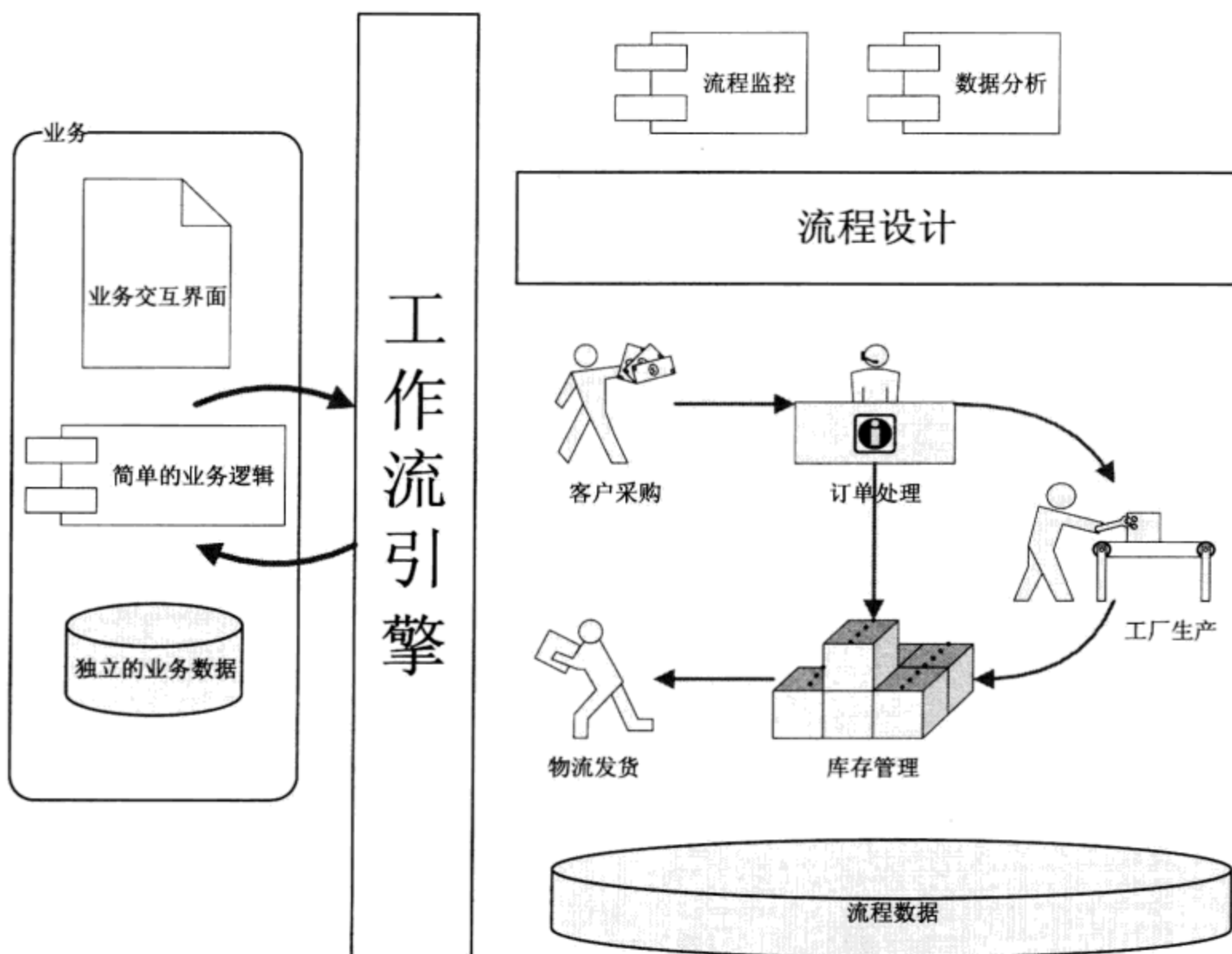


图 1-3 使用 workflow 技术实现订货业务流程

很明显，位于右侧的工作流管理系统接管了所有订货业务在流程方面的定义和执行，这包括：

- 使用专门的“流程数据”系统，维护所有涉及流程流转的数据。
- 提供“流程设计”工具，帮助用户定义订货流程的模型，这一般都是基于图形界面可视化的。
- 工作流引擎作为工作流管理的核心，负责解释流程定义、管理流程数据、计算和驱动流程实例的运行……其作用如同大脑之于人体。
- 工作流引擎提供众多 API（Application Programming Interface，应用编程接口）供客户端应用程序或外部业务系统调用，将特定的“业务（例如：订货）”纳入“流程”的管理和控制之中，从而实现工作流管理和业务操作的完美结合。

- workflow引擎还提供众多 API 供流程的“增值”系统使用，例如流程监控系统可以使用 workflow引擎提供的 API 去监视流程的执行过程、挂起和恢复流程实例的运行；流程数据分析系统可以使用 workflow引擎提供的 API 分析出工作完成的效率、业务流程的瓶颈等结果，以便重组流程、优化业务。

综上所述，引入 workflow技术对于技术开发来说，有如下好处：

- 降低开发风险——通过使用诸如活动、流转、状态、行为这样的术语，使得业务分析师和开发人员使用同一种语言交谈成为可能。优秀的流程设计建模工具，甚至能使开发人员不必将用户需求转化成详细设计文档。
- 流程实现的集中统一——应对业务流程经常变化的情况，使用 workflow技术的最大好处是使业务流程的实现代码，不再散落在各式各样的业务系统中。
- 加速开发——开发者不用再关注流程的参与者、活动节点的衔接、流转控制……因为这些工作很多被 workflow框架接管了。因而开发者开发起来更快、代码出错更少、系统更加容易维护。
- 提升对迭代开发的支持——如果系统中业务流程部分被硬编码，就不容易更改，需求分析师就会花费很大的精力在开发前的业务分析中，并且希望一次成功。但可悲的是，在任何软件项目开发中，这都很少能实现。 workflow管理系统使得业务流程很容易部署和重新编排，业务流程相关的应用开发可以以一种“迭代/渐进”的方式推进，也就是说 workflow技术在某种程度上支持“需求分析不必一次完全成功”。

1.2 workflow管理系统的发展历程

如果说数据库系统（Database Systems）的发展历程像受人尊敬的智者讲述条理清晰的故事，那么 workflow（Workflow）的发展历程就像一群乳臭未干的小子们在大谈各自的“哲理”。

之所以这样比喻是想指出， workflow管理系统相对于数据库系统还处于技术曲线上的发展阶段。这是一个激动人心的阶段，但不是一个成熟的阶段。为了证明这一点，我们可以拿 workflow管理系统和关系型数据库系统（RDBMS）做一个对比：当在软件开发团队中谈论关系型数据库系统时，例如 Oracle，SQL Server 等，大部分技术人员会有一个清晰的概念，在您和他们交流的时候，他们会通过轻微的点头表示认可或理解您所说的。可当您使用 workflow术语和他们讨论 workflow时，他们很可能会摇头表示其他意见，因为很多人对 workflow术语都有不同的理解。

形成这种状况的原因之一是，在“早期”的工作流中使用了过多的概念，至今尚未完全统一。在这个领域中有大量存在差异的规范和工具，当然，它们相互之间有重叠并且会相互参考引证。

图 1-4 演绎了 workflow 管理系统从无到有的“进化”历程。

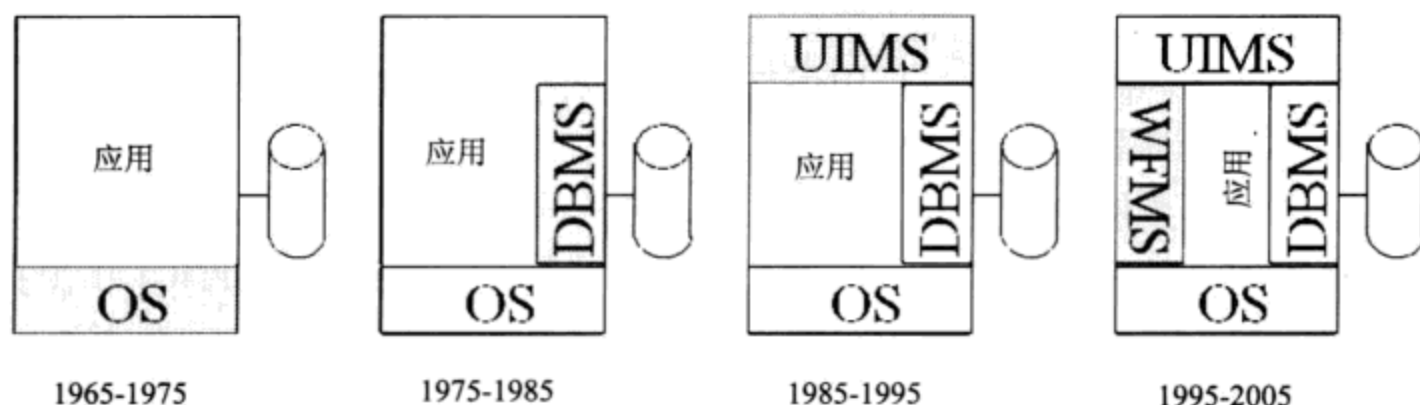


图 1-4 workflow 管理系统的“进化”历程

通过上面这张图，我们了解到企业应用软件的发展走过了这样一个历程：

- 1) 最初的企业应用直接架设在操作系统（Operation System, OS）之上，应用的程序和数据混合在一起，耦合之紧，超乎想象。这大约是在 1965—1975 年。
- 2) 后来，人们发现程序和数据混合在一起，维护起来太困难了——您能想象把所有的数据记录都硬编码在程序中吗？于是，人们把数据库管理系统从应用系统中剥离出来，自成一派。这大约是在 1975—1985 年。
- 3) 再后来，随着企业应用复杂度的增加，人们对人机交互的需求越来越高，架构师们发现把应用程序的逻辑和用户界面（User Interface, UI）也分离是个不错的主意。这样底层的开发者可以专注于业务逻辑的实现，而前端的 UI 工程师可以全力去追求交互体验，同时一套业务逻辑的实现还可以与不同的 UIMS（User Interface Management System，用户界面管理系统）相匹配，应用的灵活性得到了提升，同时成本也能下降。这大约是在 1985—1995 年。
- 4) 随着时代的发展，企业应用有了相对独立的 DBMS 和 UIMS，越来越能适应变更频繁、复杂化、大型化的需要了……这时候，瓶颈又出现了，如同本章前几节所描述的，现代企业对于业务流程的需求正在发生以下变化。
 - 变得更重要（业务流程化）。
 - 越来越需要适应频繁的变化。
 - 变得更复杂。

- 企业内部的业务流程总数在不断增加。

.....

由此催生了 workflow 管理系统。

这个发展历程的具体时间列表如下：

- workflow 技术起源于 20 世纪 70 年代中期办公自动化领域的研究工作。其中的 SCOOP 和 OfficeTalk 系统，不但标志着 workflow 技术的开始，而且也是最早的 OA 系统。
- 20 世纪 80 年代初期， workflow 技术伴随着 OA 系统走向商用，但是很少、范围很有限。
- 20 世纪 80 年代后期， OA 系统的研究逐渐走向非主流，取而代之的是群件（Groupware）和 workflow 管理系统。
- 20 世纪 90 年代以后，相关的技术条件逐渐成熟， workflow 管理系统的开发与研究进入了一个热潮。但 20 世纪 90 年代 workflow 管理系统的迅速发展也带来了很多问题，例如术语的泛滥、过多的概念、体系结构上的五花八门、没有统一的可交互接口等。
- 1993 年 8 月， workflow 技术标准化的工业组织—— workflow 管理联盟（Workflow Management Coalition, WfMC）成立。
- 1994 年， workflow 管理联盟发布了用于规范和指导 workflow 管理系统架构的“ workflow 参考模型”，并相继制定了一系列工业标准。
- 2001 年， BPMI（Business Process Management Initiative）标准组织成立，于同年 11 月 13 日发布 BPML 1.0 业务流程语言规范。
- 2002 年 8 月 9 日， BEA 公司、微软公司和 IBM 公司共同发布了一个新的业务流程语言规范 BPEL4WS（Business Process Execution Language for Web Services），并提交到了 OASIS 组织。

以下将重点介绍 workflow 技术发展历程中的两个重要的里程碑—— workflow 管理系统参考模型和 BPM。

1.2.1 workflow 管理系统参考模型

许多软件开发商都有 workflow 产品，并且不断有新的 workflow 产品走入市场。市场上可选择的产品范围很大，因此每个开发商只关注产品的特殊功能，而用户可以采用不同的产品来满足不同的需求。

然而，由于各个厂商不兼容的流程控制方式，导致没有统一的规范使得不同的工作流产品协同工作。对于这个问题，业界一直认为，所有的工作流产品都有一些相同的特性，只要其各种功能遵循公共的标准，就可以实现不同工作流产品间的协同工作。

由此 WfMC 应运而生，WfMC 的全称是 Workflow Management Coalition——工作流管理联盟，它是由一些公司联合在一起成立的组织，从事工作流问题的研究和指导。

图 1-5 所示就是 WfMC 提出的工作流管理系统参考模型（Reference Model of the Workflow Management Coalition）。作为工作流技术标准化的工业组织，WfMC 的这个参考模型无疑为各家工作流管理软件提供者的系统规划设计给出了权威的参考，乃至标准。

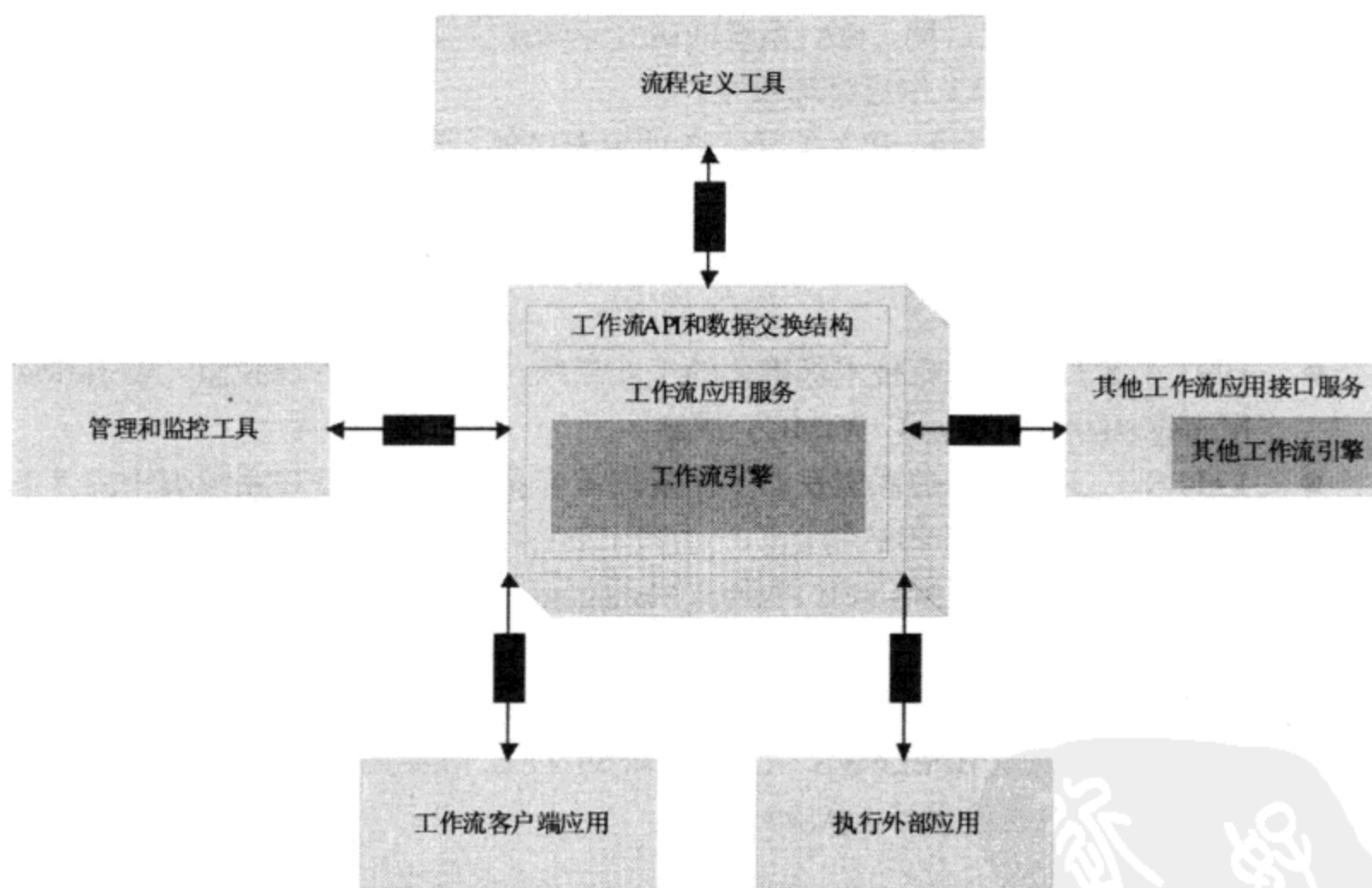


图 1-5 工作流管理系统参考模型

首先，最重要的部分就是中间的工作流引擎，可以说它就是整个工作流管理系统的核心，因为所有的工作流管理系统都要使用工作流引擎：

- 1) 为执行的流程实例解释流程定义——这些流程定义一般都是由接口 1 获得的。
- 2) 组织调度流程的实例，推进工作流程的前进，这包括条件流转、分支聚合、

父子流程……

- 3) 处理工作任务的分配、接受、提交等行为——无论是人工干预或自动执行的任务，都需要经过 workflow 引擎计算和持久化（如果需要的话）。
- 4) 管理调用其他的 4 个接口——这可能包括执行 workflow 定义中的一些外部脚本。

workflow 引擎做的工作就像心脏把血液不断地送到身体的各个部分一样。关于 workflow 引擎应该如何架构和设计，在本书后面章节会涉及。

那么，接下来说 workflow 管理系统“身体”的 5 个组成部分吧，也就是图 1-5 所示的 5 个接口。

- 接口 1——流程定义工具。

- 前面提到过我们使用它来设计业务流程定义供 workflow 引擎来实例化运行。所谓的“业务流程定义”一般来说就是一段 XML，它一般遵循 XPDL（XML Process Define Language）标准、BPEL（Business Process Execution Language）标准或其他厂商自定义的标准（例如 jBPM 的流程定义语言就是 jPDL）。事实上可以把流程定义工具理解为一个产生 XML 的图形化设计建模软件。这种软件各个厂商的技术实现可谓五花八门，仅基于 Web 的就有很多种技术实现，例如 Java Swing，Flash，ActiveX；当然，很多开源项目采用的还是基于客户端的实现，例如 jBPM 使用的是基于 Eclipse 图形化插件的实现，Shark Workflow 使用的则是 JAWES（一种基于 Java 技术实现的 XPDL 建模工具）。当然，它们的最终目的都是统一的——产生 XML 格式的流程定义。

- 接口 2——workflow 客户端应用。

- 这很有意思，当业务流程设计好了、运行起来了，那么我们——“人类”如何与 workflow 引擎交互呢？这时候，workflow 引擎就通过接口 2，为我们提供各种各样的工作任务列表、工作表单、流程列表以及一些查询功能。我们通过这些接口应用，就可以填写表单、处理任务……从而实现人与 workflow 引擎的沟通。

- 接口 3——执行外部应用。

- workflow 引擎通过这个接口去执行一些外部的或面向专门职能领域的应用程序，例如财务系统、报表系统等，让第三方系统参与进来，从而完成定义的工作流程。这看起来就像 EAI（Enterprise Application Integration，企业应用集成）的特性，而事实上它也可以说就是 Workflow EAI。同时

我们也可以发现接口 2 和接口 3 之间的界定有些模糊，难道接口 2 提到的“工作任务列表”不能算是外部的应用程序吗？没错！这个问题确实存在，这也就是为什么荷兰 workflow 大师 Aalst 在其著作《workflow 管理——模型、方法和系统》中写道“建议每个应用程序都由此‘应用程序执行服务’打开”的原因，他是在建议统一这两个接口吗？总之，接口 3 在标准化方面众口不一。

- 接口 4——其他 workflow 应用接口服务。

- 用来处理若干自治 workflow 管理系统之间的工作交换，例如实例转移、工作任务外包等。事实上，WfMC 组织的初衷是想通过这个接口来连接各个不同的 workflow 引擎和系统，使它们在一个统一的标准下工作和交流。想法是非常不错的，但是，由于种种原因吧，作者认为是商业利益的因素以及 WfMC 还没有强大到能“号令江湖，莫敢不从”的地步，所以到目前为止，接口 4 基本不被支持，也就是说，各大厂商的 workflow 产品并不能用同一种语言对话。但是，随着 jBPM4 推出的 PVM——流程虚拟机技术（这在本书的后面会涉及）的发展，实现接口 4 的障碍也许能被打破。您可以拭目以待。

- 接口 5——管理和监控工具。

- 虽然很多 workflow 管理系统，特别是开源 workflow 管理系统实现的最简单部分就是这个接口，但作者认为最能体现 workflow 管理系统在企业管理方面价值的就是这个部分，它主要被用来搜集管理信息，这包括诸如 workflow 系统功能管理工具、流程实时监视和控制工具，以及工作效率分析和流程覆盖面分析等各种商业智能工具，这为提升企业的管理能力、优化重组企业的业务流程、分析企业内部的工作效率瓶颈等提供了重要的量化数据支持。我们说“工业化解放人类的体力，信息化解放人类的智力”，这个接口提供的功能不正是解放了流程企业领导和决策者们的智力吗，而这正是企业信息化的初衷、workflow 管理的最终价值所在。传统的 workflow 管理系统在这个接口上的“短板”，正为下一节要说的 BPM (Business Process Management) 这个概念的支持者提供了攻击 workflow 技术的口实，BPM 概念在这个接口上的强化成了很多人认为“Workflow 系统”不等同于或弱于“BPM 系统”的重要原因。事实上，这都不过是些概念而已，实现 workflow 管理系统、解决业务流程改进方面的问题才是我们所要做的。

总结一下，workflow 管理系统参考模型的 5 大接口各自强调了什么？

接口 1——提供流程定义。

接口 2——提供工作任务列表等客户端应用程序，实现使用者与 workflow 引擎的沟通。

接口 3——支持外部应用程序参与 workflow。

接口 4——支持不同 workflow 引擎系统间的连接。

接口 5——提供监控工具，搜集管理信息。

还有一些问题供读者思考：

接口 3 和接口 5 标准化工作进展较为缓慢，为什么？读者可以参考上文的说明思考得出。

接口 3 和接口 4 需要完善的地方很多，例如，流程和工作任务的事务、回滚（包括被动退回和主动取回的任务）问题，在这两个问题上如何处理、怎么处理好、如何保持原子性，如何进行“补偿”，乃至如何支持“中国特色”的业务，都是很有思考空间的。

1.2.2 BPM

BPM 即业务流程管理（Business Process Management）。其注重点是通过建模、自动化、管理和优化流程，来优化公司业务的效率和效果。BPM 打破了跨部门、跨系统和跨用户，强调端对端的业务流程。BPM 系统运行在公司的内部和外部，不仅员工，客户、合作伙伴和提供商都能够进入该系统。同时，在公司内部 BPM 的应用系统还包含了提升业务的可视水平和可预见水平的功能。

BPM 通常以 Internet 方式实现信息传递、数据同步、业务监控和企业业务流程的持续升级优化。从这方面来说，BPM 不但涵盖了“传统工作流”的流程传递、流程监控的范畴，而且突破了“传统工作流”技术应用范围的瓶颈。

BPM 同样需要流程定义语言来描述流程。流程定义语言可以将企业中的各种业务流程表示成一种格式化（甚至可视化）的模型。

BPM 的相关技术标准可以用来定义业务流程和 Web Service 的集成与部署，以达成企业业务目标。也就是说，BPM 语言不仅拥有 XML 表示的流程定义，还延伸到了 SOAP, WSDL, UDDI 等多项技术规格。

除了 WfMC 的 XPD L (XML-based Process Definition Language) 语言外, 当前的 BPM 相关标准和语言为数还不少, 较为著名的有:

- BPMI 的 BPML (Business Process Modeling Language)。
- ebXML 的 BPSS (Business Process Specification Schema)。
- BEA, Microsoft, IBM 联合制定的 BPEL4WS (Business Process Execution Language for Web Services, BPEL)。
- Sun Microsystems, SAP, Oracle, Italo 等公司共同制定的 WSCI (Web Service Choreography Interface)。

事实上, 这些标准都利用活动作为组成流程定义的基本元素, 运行过程中, 每一个活动伴随一组流程实例相关数据 (Instant-Relevant Data), 这个相关数据最重要的作用就是作为流程传递逻辑 (Routing Logic) 的评估条件, 相关数据在 BPML 中称为 Property, 在 XPD L 中称为 Workflow-Relevant Data, 在 BPEL 中称为 Container。

在着重点方面, XPD L 标准着重在工作任务分配的相关处理, 例如如何指定活动运行所需的资源与应用程序。BPML 标准着重在定义 Web Service 的相关处理, 例如支持事务与异常处理、定义特定消息交换与事件处置的活动模型。BPEL 标准的着重点与 BPML 相类似。WSCI 标准着重在 Web 服务界面的行为。BPSS 以 ebXML 建议的模型化方法论为基础, 以支持在企业间以各种事务行为节点组合成所谓的企业协同应用为着重点。

对于最新推出的 jBPM4 来说, 由于其开放、可扩展的特性, 对于以上标准的着重点都有所支持或涵盖。因此, 对于 jBPM4 的开发者而言, 在工作中尽情发挥就好了, 不必担心被这些概念上的东西所束缚。

1.3 开源工作流选型

Optaros (www.optaros.com) 是著名的开源软件研究及解决方案咨询公司, 表 1-1 是其 2009 年开放源码目录中对于开源工作流管理系统的点评和介绍, 值得参考 (截止到 2009 年 3 月)。

表 1-1 Optaros 2009 年开放源码目录_基础设施解决方案_业务流程和 workflow 管理

产品	版本	许可证	支持	功能	社区支持	成熟度	ER-Rating	趋势
Bonita	3.1	LGPL	社区版	★★★	★★★	★★	★	→
	描述：拥有基于“活动预测模型”的工作流引擎，符合 WfMC 规范。只适用于 Jonas 应用服务器和 JBoss 应用服务器。 网址： http://bonita.objectweb.org							
Enhydra Shark	2.2	LGPL	专业/社区版	★★★	★★★	★★★	★★	↑
	描述：拥有基于 Java 技术、可扩展的工作流引擎，实现了 WfMC 规范，即使用 XPD 语言来定义流程。该项目提供了图形化流程设计器。 网址： http://shark.objectweb.org							
Intalio BPMS	5.1.1	Apache Eclipse Public License Custom	专业/社区版	★★★	★★	★★	★★	↑
	描述：是一个业务流程管理平台，提供了复杂的工具和底层技术用来支持流程的运行，包括流程设计器（基于 Eclipse）、流程引擎（ODE）和一些运行时组件。 网址： http://bpms.intalio.com							
IX Workflow	1.5	LGPL	专业/社区版	★	★	★	★	↑
	描述：基于 Java 体系的工作流系统，负责持久化以及处理业务流程，能很好地支持与 Domino, JBoss, Sun Glassfish 应用服务器的集成。流程设计器是基于 Eclipse 的插件。 网址： http://www.imixs.org							

(续表)

产品	版本	许可证	支持	功能	社区支持	成熟度	ER-Rating	趋势
JBoss jBPM	3.2.3	LGPL	专业/ 社区版	★★★★	★★★	★★★	★★★	↑
	描述：灵活且可扩展的工作流管理系统。使用管理者和开发者都可以理解的语言（如 jPDL 或 BPEL）来定义流程。其图形化流程设计器为 Eclipse 插件。 网址： http://www.jboss.com/products/jbpm							
ODE (Apache)	1.2	Apache License 2.0	社区版	★★	★★★	★★★	★	→
	描述：基于 Java 组件的工作流引擎，遵循 BPEL4WS 规范。ODE 工作流引擎早于 PXE 工作流引擎面世。 网址： http://ode.apache.org							

根据这份报告可以很明显地看出，在众多的开源工作流管理系统中，jBPM 这个项目在各项评比中都居于第一位，其许可证为 LGPL，可以在合法的范围内被作为商业应用。jBPM 不仅有着开源社区的支持，同时作为 RedHat/JBoss 的子项目，也具有一定的商业支持服务保证。尽管它还有一些不足，例如流程设计器功能过于简化、对企业应用集成的支持有待完善等，但毫无疑问，它是众多项目型公司低成本工作流应用解决方案的不二之选。

Shark 在这份报告中应该居于第二位，其严格遵循 WfMC 规范的 XPD L 流程定义语言无疑是个亮点，这比 jBPM 主要采用自己的 jPDL 语言似乎更标准、更通用一些，不过 jBPM4 对 BPEL 的支持和 PVM 的设计理念让 Shark 的这个优势显得并不突出。

其他的几个项目或多或少地存在着明显的短板，有的甚至已经停滞不前了，因此，在国内的应用并不多见。

值得注意的是，在这份报告中，参加评比的 jBPM 版本还是 3.X，而今，jBPM4 已经逐渐成为主流，而 jBPM4 比之 jBPM3 有着“飞跃”性的提升，读者可以在后面的章节中逐渐体会到。

1.4 jBPM

jBPM, 全称是 java Business Process Management, 是一种基于 JavaEE 的轻量级 workflow 管理软件包, 由于 jBPM 架构的开放性, 它更像是一个支持面向流程编程的框架 (Framework)。jBPM 是开放源代码 (Open Source) 项目, 使用 jBPM 要遵循 LGPL 开放源代码协议。以下的介绍将使您对这个著名的项目有一个概念性的认识。

1.4.1 jBPM 前世今生

jBPM 项目于 2002 年 3 月由 Tom Baeyens 发起, 2003 年 12 月发布 1.0 版本。jBPM 在 2004 年 10 月 18 日, 发布了 2.0 版本, 并在同一天加入了 JBoss 组织, 成为了 JBoss 企业中间件平台的一个组成部分, 它的名称也改成 JBoss jBPM。随着 jBPM 加入 JBoss 组织, 以及 JBoss 被 RedHat 公司收购, jBPM 也进入一个全新的发展时代, 它获得了大量的社区和商业支持, 因此发展前景十分光明。

jBPM3 主要采用了 UML Activity Diagram (活动图) 的模型, 借鉴了 Petri Net 的 token 机制, 使用 “无限状态机” 模型控制流程实例的变迁。因此在理论模型基础上, jBPM 无疑是先进 workflow 产品。jBPM4 则引入了 PVM (流程虚拟机) 的设计理念, 为 jBPM4 的 “无限” 扩展和集成提供了有力的底层功能支持。

经过 8 年多的发展, JBoss jBPM 已经成为一流的开源 workflow 产品:

- 每月超过 20 000 次的下载量。
- 极度活跃的用户论坛和开发者论坛。
- 频繁更新 Web 站点和 Wiki。

本书以 2010 年第一季度最新发布的 jBPM4.3 版本为主要参照, 来介绍 jBPM。当然, 本书介绍的很多方法和思想不是与版本号 “绑定” 的, 即适用于所有 jBPM4 版本, 甚至所有 workflow 系统的研究和开发。

1.4.2 关于 jBPM4 您要知道的

JBoss jBPM 是一个可扩展、灵活的能够实现 workflow/业务流程管理的 enterprise 级开发框架, 提供了流程定义、流程部署、流程执行、流程管理等功能。

jBPM 是 JBoss 旗下的子项目, JBoss 下还包括有 Seam (JavaEE 开发框架)、Drools

(规则引擎)、Hibernate (ORM 持久化框架) 等众多领域的优秀开源项目。由于同属一个产品家族, 它们能与 jBPM 完美地结合, 互相都留有支持的接口, 方便开发者业务的扩展, 为 jBPM 提供延伸的价值。

同时 jBPM 作为 JBoss SOA 平台的一个重要组件, 与 JBoss Drools 规则引擎和 JBoss ESB 企业服务总线配合在一起为用户提供全面、完整的 SOA 解决方案。

JBoss jBPM 是一个支持“嵌入式”的业务流程管理产品, 理论上可以运行在任何 JavaEE 应用服务器之上, 也可以运行在桌面应用中。JBoss jBPM4 在流程虚拟机(PVM)技术的基础上, 能够同时支持多种流程定义语言, 目前已经支持的流程语言有:

- jPDL
- BPEL
- Seam PageFlow

根据 PVM 的设计理念, 未来的 JBoss jBPM 还会支持更多的流程定义语言。同时, 用户也能够根据需求定制自己个性化的流程模型和建模语言。

jBPM4 的结构特点如下。

1. 嵌入式的工作流引擎

jBPM4 是完全支持嵌入式应用的业务流程开发框架, 可以在事务处理、数据持久化等各个方面与业务应用程序进行灵活的集成。区别于传统的工作流平台, 它不需要依赖特定的中间件或服务器, 减少了硬件和软件的绑定, 同时降低了应用部署的网络复杂度, 使应用更加容易实现集群。软件开发人员可以把 jBPM4 框架作为业务流程管理的基础, 在此基础上开发自己独特的业务流程管理模块和功能。在部署时, 只需要把 jBPM4 作为 Java 依赖库发布就可以了。

2. 可插拔的体系架构

jBPM4 采用了模块化的架构设计, 采用了 IOC (依赖注入) 的设计理念, 各模块之间可以比较方便地解除耦合或替换不同的实现, 例如持久化、事务处理、身份认证、日志服务等, 都由可选模块实现。jBPM 的可插拔体系架构, 为软件开发人员灵活选择 jBPM 的功能、自定义已有功能和拓展新功能提供了“无限可能”。

3. 易扩展的流程语言

jBPM 框架内置的流程定义活动, 包括 start, task, fork, join 和 decision 等, 是构建完整业务流程所必需的组成部分, 它们提供了可以将业务逻辑 Java 代码和业务流程

编排无缝衔接的绑定机制。而除了这些内置的流程定义活动和流程结构之外，软件开发者还可以通过定制新的活动类型或者完全重新设计一种新的流程定义语言来描述特定领域的业务流程，满足独特环境下的需求。

从技术角度分析 jBPM4 的特点，简单罗列几点读者必须要了解的：

- jBPM4 的模型仍然基于 UML Activity Diagram，以便于需求人员和开发人员都理解业务流程。
- jBPM4 提供了可定制的 Event – Listener 观察者模式来处理事件触发，以辅助活动扩展的处理。
- jBPM4 提供了灵活的 EL 条件表达式机制，来辅助条件解析、简单业务逻辑脚本计算的处理。
- jBPM4 提供了可扩展的 Task 及任务分配机制，来满足复杂人工活动的处理。
- 借助 Hibernate ORM 的优势，jBPM4 能够支持在几乎所有的数据库系统之上运行。

jBPM4 作为一款开源的工作流框架，其更多的是关注“如何辅助开发者更容易地让流程运行完成”，而不是关注“记录流程运行的历史和轨迹”。这一点可能是东西方文化的差异性所在，因为国内的流程应用，比较关注“运行轨迹”。

同时，jBPM 项目从一开始就是不直接支持自由“回退”、“跳转”等操作的，这也是因为东西方文化的差异所在。西方人认为“往回流转的情况肯定也是一种业务规则所定义的，那么肯定可以通过分支或条件流转来解决”，而东方人则把回退作为一个“人性化管理和处理的潜在特点”来看待。当然，这正是本书所要解决的问题之一，在**第 20 章 中国特色工作流的 jBPM 实现**中会给出一些参考解决方案。

具体到 jBPM4 的当前发行版，您需要知道的有：

- 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。
- 如何获取 jBPM4
 - 可以从 SourceForge.net 上获取发布包：<http://sourceforge.net/projects/jbpm/files/>。
- 如何获取 jBPM4 的所有源代码

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

- 如何从 jBPM 3 升级到 jBPM 4

- 很遗憾，没办法实现直接从 jBPM3 升级到 jBPM4。但本书提供一套建议的方法供读者实践，参见第 11 章 升级 jBPM3 到 jBPM4。

- 如何报告问题

- 在开发过程中遇到问题无法解决可以在 jBPM 开发者社区 <http://community.jboss.org/en/jbpm> 寻求帮助，但发布问题需要遵循如下模板。

```
=== Environment =====
- jBPM Version : 您使用的是哪个版本的 jBPM?
- Database : 使用什么数据库以及数据库的版本?
- JDK : 使用哪个版本的 JDK? 如果不知道, 可以使用 java -version 命令查看版本信息
- Container : 使用什么容器? (例如 JBoss, Tomcat)
- Configuration : 您的 jbpm.cfg.xml 中是导入了 jbpm.jar 中的默认配置, 还是使用了自定义的配置?
- Libraries : 您使用了 jbpm 发布包中完全相同的依赖库的版本, 还是您修改了其中一些依赖库?

=== Process =====
这里填写 jPDL 流程定义

=== API =====
这里填写您调用 jBPM API 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。例如: API 没有如期望中那样工作, 或者 ExecutionService.SignalExecutionById 抛出了异常。
```

发布问题最好使用英文，因为这个社区论坛是英文的。同时，聪明的读者可能已经注意到了模板上的这些项已经指向了可能导致 jBPM 问题的几点原因，特别是对依赖库和配置文件的调整是最容易导致问题的操作。所以，开始在本书介绍的知识范围之外修改配置文件之前，一定要谨慎；同样在使用其他版本的依赖库替换默认的依赖库之前，也一定要谨慎。

1.5 小结

通过本章的介绍，相信读者已经对“工作流”和“jBPM”有了概念上的认识，已经能够阅读一些充满专业术语的 jBPM 应用开发相关的技术文档了。那么，我们言归正题，进入 jBPM4 应用，开始您的面向流程之旅吧！

在后面的章节中，本书将不再花费大量的精力去解释工作流、jBPM 相关的专业术语。如果您读到仍然感到迷惑的术语，可以尝试到附录 A jBPM 术语中寻求解释。



jBPM 需要安装？不是说它是一个框架（Framework）、一堆开放的源代码（Open Source），而非一套“应用程序（Application）”吗，据我所知一般只有应用程序才需要安装呀？

是的，在您深入了解 jBPM 后，可以把它的流程引擎看做一个 Java 工程——若干 Java 类、依赖库和配置文件；jBPM 的流程定义和运行时的上下文需要被存储在关系型数据库中——可以是基于调试目的的内存数据库 HSQLDB（hsqldb.org），也可以是真正的持久化数据库，诸如 MySQL（www.mysql.com），Oracle（www.oracle.com），PostgreSQL（www.postgresql.org），Sybase（www.sybase.com）；jBPM 的流程设计一般需要一个基于客户端的图形化流程设计器软件，当然在 jBPM4 版本以后也可以在 Web 上做流程设计（这要归功于 Signavio 项目，www.signavio.com）；最终，基于 jBPM 开发出的企业流程应用一般会被部署在一台应用服务器（Application Server）上，以便服务于来自 Web 的访问、监控和管理等。

但是，以上所说的这一切，如果是一名初学者，能很好地掌控吗？

在早期的版本中，要使用 jBPM，很多准备工作都需要自己来做，而且以上每一个步骤和细节都需要自己去关注，例如安装数据库、建表、安装应用服务器、安装图形化流程设计器插件等，是不是会让人感觉到上来就碰了个大钉子？

幸运的是，在我们要介绍的 jBPM4.3 版本中，几乎这一切工作 jBPM 软件包的发布者都帮助您做好了！只要您使用过 JavaEE，Eclipse 和 Ant，就可以通过 Ant build 脚本“一键获取”所需要的 jBPM4 整套开发、运行和管理环境。这就是传说中的“安装”。

2.1 jBPM4 安装先决条件

首先，我们要获取 jBPM4 的软件包，可以在 SourceForge.net 上找到它：<http://sourceforge.net/projects/jbpm/>。在本书开始写作的时候，jBPM 的最新发布版本为

作为一个“生机盎然”的开源项目，jBPM 的版本更迭比较快，约半年左右就会发布一个新的版本，而世事难料（在这里我指的是组织变更或商业并购），如果有一天你发现在上面所提到的 SourceForge.net 上找不到 jBPM 或其最新版本的时候，使用 Google 搜索“jBPM download”是一个比较保险的办法。

事实上，jBPM 在每一次大版本号变迁时的改动才是革命性的，例如 jBPM3 到 jBPM4；而 jBPM 小版本号的变迁则是相对有限的改变，几乎不会需要您重新学习什么，例如 jBPM4.1 到 jBPM4.3，改动不大，只要关注新版本的“**What's new**”即可，也就是说：掌握了 jBPM4.3，jBPM4.X 对您就不在话下啦！

把 jBPM4.3 (jbpm-4.3.zip，不区分操作系统) 下载下来之后，解压到硬盘上的任何一个目录中，这个目录就是您的 jBPM “工作目录”了。

注意：这个“工作目录”的绝对路径最好不要包含非英文字符，例如中文；最好也不要含有空格。如果您不遵守这两条规则，也许您在 jBPM 开发过程中会遇到些莫名其妙的麻烦。

下面以 `${jbpm.home}` 指代这个“工作目录”。

这个目录中包含如下子目录及文件。

- **doc**: 包括用户指南、Javadoc、Schemadoc 以及开发指南。
- **examples**: 包括用户指南中使用到的示例流程。
- **install**: 包括适用于不同环境的安装脚本。
- **lib**: 包括 jBPM 依赖的第三方库和一些特定的归档包。
- **src**: 全部 jBPM 源代码。
- **jbpm.jar**: jBPM 源代码归档包文件。
- **migration**: jBPM 升级功能解决方案包（这在本书第二篇的 11.2 流程定义转换工具中会提及）。

在开始使用 jBPM 前，还需要准备如下环境：

- 1) **JDK**（标准 Java 开发包）5 或更高版本。可以在 <http://java.sun.com/javase/downloads/> 获取到最新版本的 JDK。关于如何安装和设置 JDK 到您的操作系统请参考相关资料，本书不做说明。

- 2) 安装 jBPM 需要执行 Ant 脚本, 所以需要 Apache Ant 1.7.0 或更高版本。可以在 <http://ant.apache.org/bindownload.cgi> 获取到最新版本的 Ant。

2.2 快速开始吧

如果作为一名初学者, 拿到 jBPM 后, 您最想做什么? 没错, 快速地安装好, 开始运行。

下面的范例将以最简单的方式帮助您快速开始使用 jBPM。

提示: 如果您有下载过 `apache-tomcat-6.0.20.zip` 或 `jboss-5.0.0.GA.zip` (Tomcat 还是 JBoss? 视您想要运行 jBPM 的应用服务器类型而定, 二者选其一即可), 可以把它放到 `${jbp.home}/install/downloads` 目录下。这样可以避免安装脚本从网络上下载这些 jBPM 所需的软件。同理适用于 `eclipse-jee-galileo-win32.zip`, 或在 Linux 平台下的 `eclipse-jee-galileo-linux-gtk(-x86_64).tar.gz`, 或在 Mac OS X 平台下的 `eclipse-jee-galileo-macosx-carbon.tar.gz`。

按步骤来:

- 1) 打开命令控制台 (即 Windows 下的 `cmd`), 进入目录 `${jbp.home}/install`。
- 2) 运行脚本 (当然需要配置好 Ant 命令的路径) `ant demo.setup.tomcat` 或者 `ant demo.setup.jboss`。

没了, 就这么简单的两步。

实际上这两步帮您做了如下工作:

- 1) 把 Tomcat 安装到 `${jbp.home}/apache-tomcat-6.0.20` 目录下。
- 2) 把 jBPM 安装到 Tomcat 中。
- 3) 安装 HSQLDB, 并在后台启动。
- 4) 创建数据库表结构。
- 5) 在后台启动 Tomcat。
- 6) 根据示例 (来自 `examples` 目录) 创建一个 `examples.bar` 业务流程归档, 并把它发布到 jBPM 数据库中。
- 7) 从 `${jbp.home}/install/src/demo/example.identities.sql` 初始化用户和组。

- 8) 安装 Eclipse 到 `${jbpn.home}/eclipse`。
- 9) 启动 Eclipse。
- 10) 安装 jBPM Web 控制台。
- 11) 安装 Signavio Web 设计器。

当这些都完成后, Tomcat (或 JBoss, 由您之前运行的 `demo.setup` 脚本决定) 会在后台启动。

一旦 Eclipse 启动成功, 您可以在其上安装 GPD (图形化流程设计器), 使用这个基于 Eclipse 的客户端软件去进行流程建模, 如何安装请参考 2.9 安装图形化流程设计器 (GPD)。

或者您可以通过 Signavio web 设计器进行流程建模: <http://localhost:8080/jbpmeditor/p/explorer>。

这时候, 您也可以使用 jBPM 控制台: <http://localhost:8080/jbpm-console/>, 利用表 2-1 中所列用户之一进行登录。

表 2-1 jBPM 控制台用户

用户名	密码
alex	password
mike	password
peter	password
mary	password

注意: jBPM4.3 控制台目前存在一个问题——对于一些比较慢的机器, 在初始化流程报表时, 控制台的失效时间太短了, 所以当您第一次请求流程报表时, 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。同时, 这个问题已经提交到了官方 JIRA - JBPM-2508。

2.3 安装脚本详解

在上一节中我们使用了 jBPM 安装脚本中名为 `demo.setup.tomcat` 的目标任务 (target), 如您所知, 这是一个 Ant build target, 接下来我们将完整解读 jBPM4.3 的安

装脚本。

jBPM 软件包目录中包含了一个 `install` 目录，此目录中有一个 Ant 的 `build.xml` 文件，您可以使用它来把 jBPM 按照需要安装到应用环境中。

注意：最好严格按照这组安装脚本进行安装和发布 jBPM 配置文件。当然，我们可以根据需要自定义 jBPM 的安装和配置，但这需要您的经验和谨慎。

如何调用安装脚本？打开命令行，进入 `${jbpm.home}/install` 目录。使用 `ant-p` 命令您可以看到这个安装脚本里所有可以使用的目标任务。其参数都设置了默认值，以便快速执行，下面给出了可用目标任务的概况。

- **demo.setup.jboss**: 依次执行——（下载）安装 JBoss，把 jBPM 安装到 JBoss 中，启动 JBoss，创建 jBPM 数据库表结构，部署示例，加载示例身份认证信息，（下载）安装并启动 Eclipse。
- **demo.setup.tomcat**: 依次执行——（下载）安装 Tomcat，把 JBoss 安装到 Tomcat 中，启动 Tomcat，创建 jBPM 数据库表结构，部署示例，加载示例身份认证信息，（下载）安装并启动 Eclipse。
- **clean.cfg.dir**: 清除配置——删除 `${jbpm.home}/install/generated/cfg` 目录。
- **create.cfg**: 基于当前的参数在 `${jbpm.home}/install/generated/cfg` 目录中创建一组配置。
- **create.jbpm.schema**: 在目标数据库中创建 jBPM 表结构。
- **create.user.webapp**: 根据您当前的配置，在 `${jbpm.home}/install/generated/user-webapp` 目录中创建一个最简的 jBPM JavaEE Web 应用程序。这个应用程序包括完整的 `WEB-INF` 目录，所有的 jBPM 依赖库都在其 `lib` 目录中。您可以将这个目标任务理解为创建一个“jBPM Blank Project”。
- **delete.jboss**: 删除已安装的 JBoss 应用服务器。
- **delete.tomcat**: 删除已安装的 Tomcat 应用服务器。
- **demo.teardown.jboss**: 删除 jBPM 数据库中的表并停止 JBoss 服务。
- **demo.teardown.tomcat**: 停止 Tomcat 服务，以及 HSQLDB 服务（如果 HSQLDB 服务正在运行的话）。
- **drop.jbpm.schema**: 从数据库中删除 jBPM 的表结构。
- **get.eclipse**: 下载 Eclipse，如果在 `${jbpm.home}/install/downloads` 目录中不存在的话。

- **get.jboss:** 下载与当前版本 jBPM 匹配的 JBoss AS (JBoss Application Server 即 JBoss 应用服务器), 如果 downloads 目录中不存在的话。
- **get.tomcat:** 下载与当前版本 jBPM 匹配的 Tomcat, 如果 downloads 目录中不存在的话。
- **hsqldb.databasesmanager:** 启动 HSQLDB 数据库管理控制台。
- **install.eclipse:** 安装 Eclipse, 执行下载 (如果 downloads 目录中不存在的话) 解压操作。
- **install.jboss:** 安装 JBoss 应用服务器, 执行下载 (如果 downloads 目录中不存在的话) 解压操作。
- **install.jbpm.into.jboss:** 安装 jBPM 到 JBoss 中。
- **install.tomcat:** 安装 Tomcat 应用服务器到 `${tomcat.distro.dir}` (在 build.xml 中配置), 执行下载 (如果 downloads 目录中不存在的话) 解压操作。
- **install.jbpm.into.tomcat:** 把 jBPM 安装到 Tomcat 中。
- **install.examples.into.tomcat:** 部署所有的示例流程到 Tomcat。
- **install.signavio.into.jboss:** 把 Signavio 流程设计器应用安装到 JBoss。
- **install.signavio.into.tomcat:** 把 Signavio 流程设计器应用安装到 Tomcat 中。
- **load.example.identities:** 装载示例用户和用户组数据到数据库中。
- **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。然后把 jBPM 安装到 JBoss 中。
- **reinstall.tomcat:** 删除之前安装的 Tomcat, 并重新安装 Tomcat。
- **reinstall.tomcat.and.jbpm:** 删除之前安装的 Tomcat, 并重新安装 Tomcat。然后把 jBPM 安装到 Tomcat 中。
- **start.eclipse:** 启动默认位置上的 Eclipse IDE。
- **start.jboss:** 启动 JBoss 服务, 并等待 JBoss 启动完成, JBoss 服务作为后台进程运行。
- **start.tomcat:** 启动 Tomcat 服务, 并等待 Tomcat 启动完成, Tomcat 服务作为后台进程运行。
- **stop.jboss:** 通知 JBoss 服务停止, 但并不等待停止完成。
- **stop.tomcat:** 通知 Tomcat 服务停止, 但并不等待停止完成。
- **upgrade.jbpm.schema:** 更新数据库中的 jBPM 表结构到当前版本。

以下将从配置文件和依赖库这两个角度, 向您介绍一些使用安装脚本应该注意的事项。

2.3.1 关于配置文件

通过上述目标任务，我们了解到安装脚本能为我们下载安装 Eclipse, JBoss, Tomcat……简直无所不能，但细心的读者会发现：数据库软件的安装呢？

没错，脚本没有能力完成所有的事，如果要成功运行 jBPM，您必须要修改一些配置文件，例如，数据库对应的配置文件在目录 `${jbp.home}/install/jdbc` 中，这个目录中列出了 jBPM 官方支持数据库类型的相应配置，根据您对数据库的选择配置相应的 `.properties` 文件吧，例如默认的 `mysql.properties` 内容如下：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/jbpmdb
jdbc.username=jbp
jdbc.password=jbp
```

假设您的 MySQL 服务位于 192.168.0.8 的 3308 端口，库名为 `jbpmdb`，用户名为 `root`，密码为 `root`，则您需要修改配置为：

```
jdbc.driver=com.mysql.jdbc.Driver
# 数据库访问地址
jdbc.url=jdbc:mysql://192.168.0.8:3308/jbpmdb
# 数据库用户名和密码
jdbc.username=root
jdbc.password=root
```

下面的安装脚本参数也可根据需要自定义。

- `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参数值，可以在运行 Ant 安装脚本时使用 `-D` 指令，例如：

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

在 `install` 目录的 `build.xml` 文件中，您可以发现所有的安装参数：

```
<!-- DEVELOPER SPECIFIC CONFIGURATIONS -->
看这里，没错！您可以建立一个 build.properties 文件，在其中统一设定自定义的参数值
<property file="${user.home}/.jbpm4/build.properties" />
<!-- USER CUSTOMIZABLE PROPERTIES -->
就是在这里。database 参数可以指定安装的目标数据库类型，对应相应的 .properties 文件
<property name="database" value="hsqldb" /> <!-- {hsqldb | mysql | oracle
| postgresql} -->
<echo message="database..... ${database}" />
```

```

<property name="tx" value="standalone" /> <!-- {standalone | jta | spring}
-->
<echo message="tx..... ${tx}" />
<property name="mail.smtp.host" value="localhost" />
<echo message="mail.smtp.host... ${mail.smtp.host}" />
<!-- INTERNAL PROPERTY DEFAULTS -->
<property name="jbpm.version" value="4.3" />
<property name="jbpm.parent.dir" value="../../" />
<property name="jbpm.home" value="${jbpm.parent.dir}/jbpm-${jbpm.version}" />
<property name="hibernate.connection.type" value="jdbc" /> <!-- jdbc |
datasource -->
<property name="logging" value="jdk" /> <!-- jdk | none -->
<property name="cfg.dest.dir" value="${jbpm.home}/install/generated/cfg" />
<property name="install.src.dir" value="${jbpm.home}/install/src" />
<property name="mail.cfg" value="default" />
<property name="jdbc.properties.dir" value="${jbpm.home}/install/jdbc" />
<property file="${jdbc.properties.dir}/${database}.properties" />
<property name="examples.file" value="${jbpm.home}/examples/target/
examples.jar"/>
<property name="tomcat.version" value="6.0.20" />
.....

```

其中“INTERNAL PROPERTY DEFAULTS”部分设定了jBPM安装属性的一些默认值，您可以根据需要变更之，例如可以将最后一行的Tomcat版本改为“6.0.24”，当然，这些修改需要您的知识和经验作为保证。

2.3.2 关于依赖库

通过以上介绍，我们了解到jBPM提供了自动安装的Ant脚本。这些脚本会将正确的依赖库和正确的配置文件安装到正确的位置。但如果想在应用中定制属于自己的jBPM，那么不可避免地需要了解jBPM依赖库的细节和如何集成其他第三方库和框架……关于这方面的知识，您可以在第9章jBPM4扩展研发先决条件中了解到。

2.4 安装到JBoss

执行安装脚本中的install.jbpm.into.jboss目标任务会把jBPM安装到JBoss应用服务器中。这会把jBPM安装成为一个JBoss的服务，因此这台JBoss上所有应用程序都可以使用这个安装的jBPM流程引擎。

如同前文所说，您可以在运行脚本任务时通过如下参数来指定 JBoss 的安装路径：

```
-Djboss.home=PathToYourJBossInstallation
```

安装成功后，在 JBoss 环境中，应用程序可以通过 JNDI 查询服务获取 ProcessEngine（流程引擎）对象：

```
new InitialContext().lookup("java:/ProcessEngine")
```

同样，也可以通过 Configuration.getProcessEngine() 获取 ProcessEngine 对象。

2.5 安装到 Tomcat

安装脚本的 install.jbpm.into.tomcat 目标任务会把 jBPM 安装到 Tomcat 应用服务器中。这个任务对 Tomcat 主要做了如下“手脚”：

- 1) 复制了若干 jar 包文件到\${tomcat.home}/lib 目录中，这包括 jBPM 本身的库及其依赖的第三方库，当然还有相应的数据库驱动程序包。
- 2) 分别将 Signavio 流程设计器、jBPM 控制台、GWT 控制台服务这 3 个应用程序安装到\${tomcat.home}/webapps 目录，对应的 war 包文件是 jbpmeditor.war, jbpm-console.war, gwt-console-server.war。
- 3) 安装用于 jBPM 控制台流程分析报表的 BIRT（开源报表项目）模板及其依赖文件到\${tomcat.home}/birt 目录。
- 4) 最后，在\${tomcat.home}/conf/server.xml 文件中配置用于 jBPM 控制台用户身份认证的数据源。

.....

```
<Engine name="Catalina" defaultHost="localhost">
  <Host name="localhost" appBase="webapps" unpackWARs="true"
    autoDeploy="true" xmlValidation="false" xmlNamespaceAware="false">
    <!--
      Current workaround for using the JBPM identity tables on Tomcat.
      Setting the Realm on the context with path /jbpm-console did not seem
      to work. Note that this workaround means that the complete localhost
      domain will use this realm !
      这里就是 jBPM 控制台为了用户身份认证而在 Tomcat 上设置的数据源
    -->
    <Realm className="org.jbpm.integration.tomcat6.JbpmConsoleRealm"
      driverName="com.mysql.jdbc.Driver"
      connectionUrl="jdbc:mysql://localhost:3306/jbpmdb"
      connectionName="root"
```

```
        connectionPassword="root" />
    </Host>
</Engine>
.....
```

2.6 基于 Web 的 Signavio 流程设计器

从 jBPM3 开始, 这个著名的开源项目就因为没有基于浏览器的图形化流程设计器而饱受用户诟病, 没错, 进入了 Web 时代, 流程设计器没有理由总是停留在 CS (Client Server) 阶段。有很多公司认识到了这一点, 因此市场上的 jBPM Web 流程设计器并不少见, 但它们或不开放源代码、或基于特定的业务、或存在支持升级问题。总之, 没有 RedHat JBoss jBPM 的官方认证和支持, 您用着总会不放心, 不是吗?

现在您不用担心了。从 Version 4.1 开始, jBPM 官方发布包绑定了一个完全开源的基于 Web 的 BPM 流程设计器, 代号为 Signavio。

2.6.1 jBPM Web 流程设计器简介

Signavio Web 流程建模工具是和 JBoss jBPM 团队、德国的 Signavio 公司和 Hasso Plattner Institute (HPI 软件工程研究所) 紧密协作的成果。Signavio 项目基于 Web 建模工具 Oryx, Oryx 是由 HPI 主持的开源项目。HPI 和 Signavio 公司都会持续地在 Oryx 项目和 Signavio 项目中投入人员和资金的支持。关于这两个项目, 您可以在 Google Code 上找到: <http://code.google.com/p/signavio-oryx-initiative/>。

图 2-1 所示便是 Signavio 流程设计器的使用界面, 它能很好地支持 IE 和 Firefox 浏览器。

使用 Signavio 可以让业务流程分析人员通过浏览器建立业务流程模型。Signavio 输出的流程文件格式正是 jPDL。这意味着 Signavio 设计出的流程定义文件可以直接导入到 Eclipse GPD, 反之亦然。流程定义文件会保存在硬盘上, 位于 \$jbpm_home/signavio-repository 中, 这个参数在安装脚本中有默认值。

注意: Signavio 是基于 web 的业务流程建模工具, 绑定在 jBPM 中, 是 100% 开源的 (基于 MIT 开放源代码许可证)。同时, Signavio 公司也提供商业版的同名工具, 毫无疑问, 商业版的 Signavio 有更多的功能。

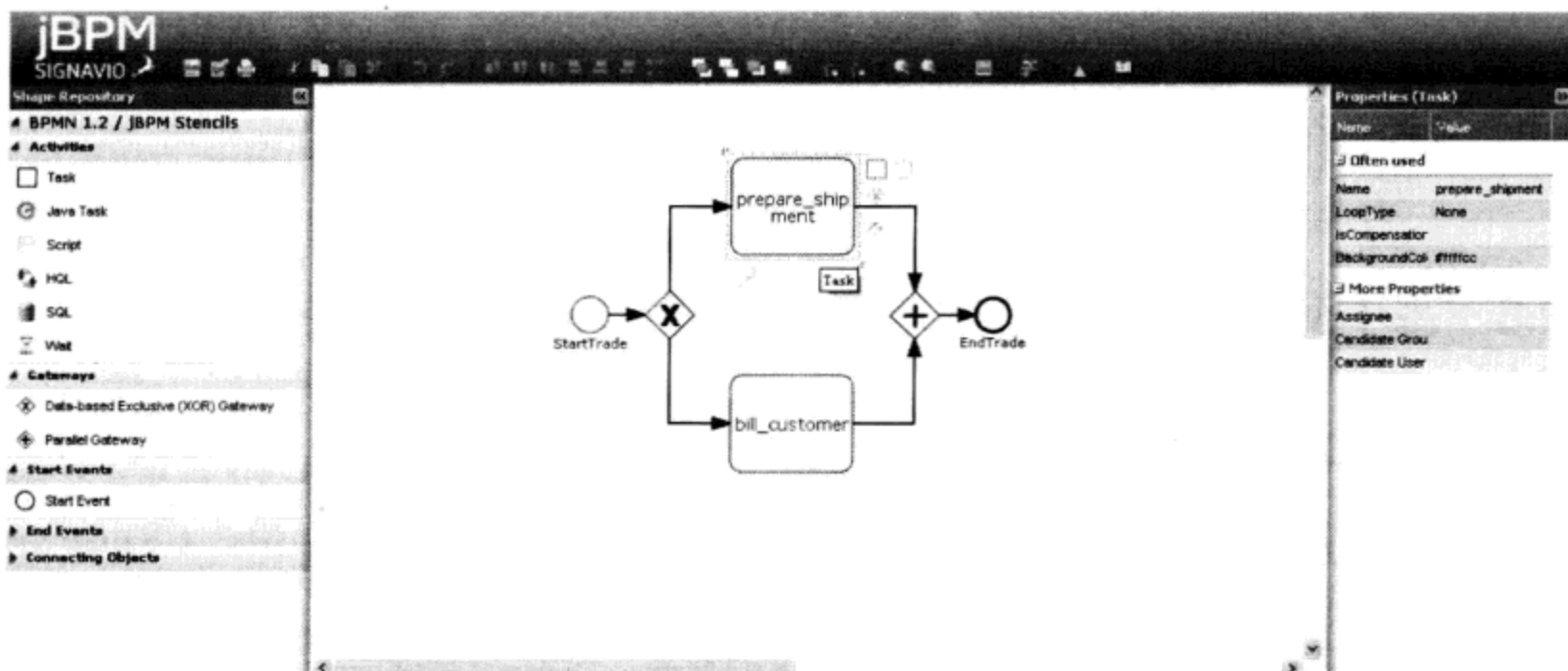


图 2-1 Signavio——基于 Web 的流程设计器

2.6.2 独立安装 Signavio

如本章开始所提及的，您可以使用 `$jbpm_home/install` 中的 `demo.setup.tomcat`（或 `jboss`）脚本任务把 Signavio 安装到您的 Web 容器（Tomcat 或 JBoss）中去。

这里还有几种把 Signavio 独立安装到 Web 容器中的方式：

- 使用 `$jbpm_home/install` 中的 `install.signavio.into.tomcat(or jboss)` 脚本任务，安装到默认的 Web 容器（Tomcat or JBoss）中。
- 复制 `$jbpm_home/install/src/signavio/jbpmeditor.war` 到您的目标 Web 容器中。

注意：不同的 Web 容器可能会存在些兼容性的小问题，例如，Signavio 安装到默认的 Tomcat6 后，在流程设计中输入中文字符保存后，再打开，可能会造成无法加载的问题。

2.6.3 配置 Signavio

Signavio 配置很简单，大多数参数在 `web.xml` 中修改即可，您可以在 `jbpmeditor.war/WEB-INF/` 目录中找到。其中最重要的是 `fileSystemRootDirectory` 参数。这个参数的值必须为一个物理上存在的本地目录，它指定了流程定义文件（即

*.jpd.xml 文件) 存储的位置, 以下是此参数的 web.xml 片段:

```
.....
<context-param>
  <description>Filesystem directory that is used to store models</description>
  <param-name>fileSystemRootDirectory</param-name>

<param-value>C:/opensource/jboss/jbpm-4.3/signavio-repository</param-value>
</context-param>
.....
```

如果使用\$jbpm_home/install 中提供的安装脚本, fileSystemRootDirectory 会被默认地设置在\$jbpm_home/signavio-repository 目录中。

2.7 用户自定义 jBPM Web 应用程序

如果您希望创建一个基于 jBPM 的 Web 应用程序供您自由发挥, 可以使用 create.user.webapp 这个脚本任务。正如 2.3.1 关于配置文件中说明的, 这将在\${jbpm.home}/install/generated/user-webapp 目录中创建一个最简的 jBPM JavaEE Web 应用程序……可以理解为创建一个“jBPM Blank Project”。

提示: 如果您在 JBoss AS 或其他包含 jta.jar 的应用服务器上部署了您的自定义 jBPM Web 应用程序, 需要把文件 \${jbpm.home}/install/generated/user-webapp/WEB-INF/lib/jta.jar 删除。

2.8 安装 jBPM 数据库

安装脚本包含了安装数据库的必要操作, 例如创建表, 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 目录的.properties 文件中指定连接参数。

下面以 MySQL 数据库系统为例进行说明。

2.8.1 新数据库安装

在一张白纸上写字很容易！同样，安装一个全新的 jBPM 数据库也不难。

- 执行 `${jbpn.home}/install` 目录下的脚本任务 `create.jbpm.schema` 单独创建 jBPM 数据库表，当然，先要在 DBMS 上建立好“库”。作为创建表和约束的一部分，JBPM4_PROPERTY 表会被初始化一些种子数据，例如当前的 jBPM 引擎版本（`key db.version`）和 ID 生成器版本（`key next.dbid`）。
- 执行 `drop.jbpm.schema` 脚本任务删除表结构。注意这个操作会删除 jBPM 数据库表中的所有数据。

2.8.2 升级旧的数据库

升级数据库，兼容旧版本会麻烦点。需要注意的是，本书使用的是 jBPM4.3 版本，升级只在 jBPM4 系列中进行。至于从 jBPM3 升级到 jBPM4 的数据库，最好根据您的业务单独做个 ETL（Extract, Transform, Load，数据的萃取转换装载，数据挖掘的前置工作）。

执行 `${jbpn.home}/install` 目录下的 `upgrade.jbpm.schema` 脚本任务开始升级。

升级包含两步操作：

- 1) 添加新版本中增加的表、列以及约束。
- 2) 插入种子数据。

jBPM4.3 版本前的一些数据库升级要点：

- 从 4.0 到 4.1 版本，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和 Hibernate 类型映射。这个列是可以为 null 的，因为这个功能在 4.0 中没有支持，所以支持没有初始值。
- 从 4.1 到 4.2 版本，jBPM 表现出了强大的兼容性潜力，这包括 JBPM4_PROPERTY 表的横空出世。
 - 一个新的表 JBPM4_PROPERTY 被用来保存流程引擎环境相关的数据。
 - jBPM 版本保存在 JBPM4_PROPERTY 表中，`key db.version` 字段被用来在未来的发布版本中作为区分标识。
 - ID 生成策略是完全跨数据库系统的。下一个有效的 ID 是通过搜索所有包含主键的表计算出的，它被保存在 JBPM4_PROPERTY 表中的 `key`

next.dbid 字段。

- 流程语言被设置为 jpdL-4.0，作用于所有已经存在的流程定义，对应 JBPM4_DEPLOYPROP 表中的 key langid 字段。jPDL 解析器根据 langid 属性来读取流程定义，以此支持向后兼容。

2.9 安装图形化流程设计器 (GPD)

所谓图形化流程设计器，即 Graph Process Designer（以下简称 GPD），使用户能够通过图形拖曳、属性设置等可视化的方式进行业务流程设计，建立并展现业务流程模型。这个模型在 jBPM4 中一般为 .jpdL.xml 文件，遵循 jPDL 规范，此文件即“流程定义”文件，在运行时由 workflow 引擎解释执行，生成“流程实例”……

jBPM 的经典 GPD 一直使用 Eclipse（Eclipse 是著名的跨平台自由集成开发环境，即 Integrated Development Environment，IDE。您可以在 www.eclipse.org 上了解关于 Eclipse 的更多信息）作为其客户端支撑平台，本节将介绍如何获取和安装 Eclipse，并将 GPD 插件安装到 Eclipse IDE 之上。

2.9.1 获取 Eclipse

使用 jBPM4.3 GPD，需要 Eclipse IDE for Java EE Developers 3.5.0 版本（jBPM4.0 依赖于 Eclipse 3.4 版本）。

可以选择之前介绍的脚本安装或手工下载安装这个 Eclipse 版本，大小约 163 MB。

注意，Eclipse 传统版本无法满足 GPD 的使用要求，因为它没有 XML 编辑器，但是 Eclipse IDE for Java Developers 3.5.0 可以使用 GPD。

2.9.2 在 Eclipse 中安装 GPD 插件

使用 Eclipse 软件升级（Software Update）功能安装 GPD 是非常简单的。文件 install/src/gpd/jbpm-gpd-site.zip 就是 GPD 的站点更新存档（archived update site）。

在 Eclipse 里添加 GPD 站点更新存档的步骤：

- 1) 选择 Help→Install New Software 命令。
- 2) 单击 Add 按钮。

- 3) 在 Add Site 对话框中, 单击 Archive 按钮。
- 4) 找到 install/src/gpd/jbpm-gpd-site.zip 文件, 并单击 Open 按钮。
- 5) 在 Add Site 对话框中单击 OK 按钮, 返回到 Install 对话框。
- 6) 选中出现的 jPDL 4 GPD Update Site 选项。
- 7) 单击 Next 按钮, 然后单击 Finish 按钮。
- 8) 接受软件许可证协议。
- 9) 询问时重启 Eclipse。

图 2-2 所示是主要操作步骤的示意图。

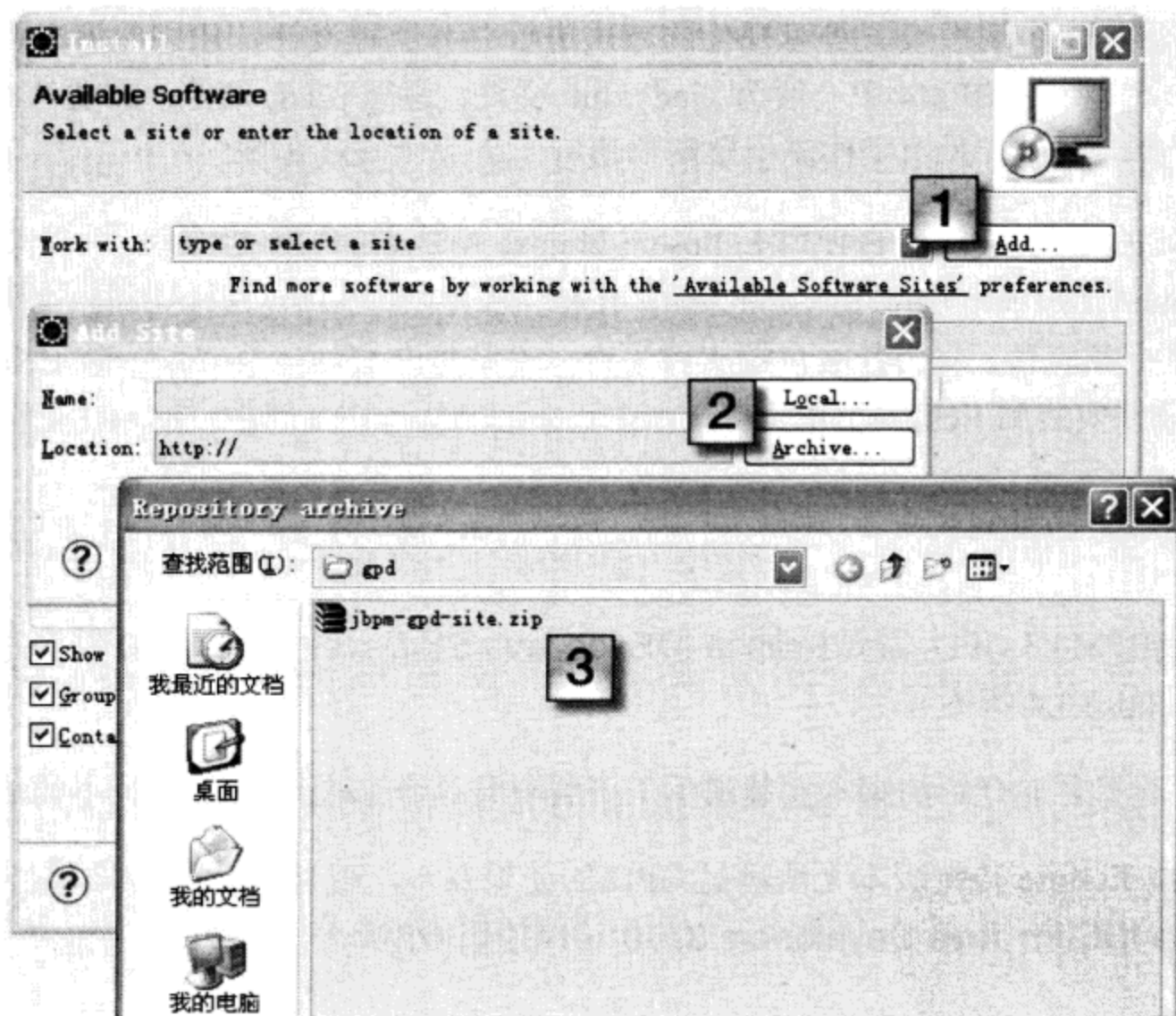


图 2-2 在 Eclipse 里添加 GPD 站点更新存档

2.9.3 配置 jBPM 运行环境

为了方便地基于 jBPM 开发流程应用, 需要在 Eclipse 中配置其运行环境, 步骤如下:

- 1) 选择 Window→Preferences 命令。
- 2) 选择 JBoss jBPM→jBPM 4→Runtime Locations 选项。
- 3) 单击 Add 按钮。
- 4) 在 Edit Location 对话框中输入一个名称，例如 jbp43，然后单击 Search 按钮。
- 5) 在 Browse For Folder 对话框中，选择 jBPM 安装目录，然后单击 OK 按钮。
- 6) 在 Edit Location 对话框中单击 OK 按钮。

图 2-3 所示是主要操作步骤的示意图。

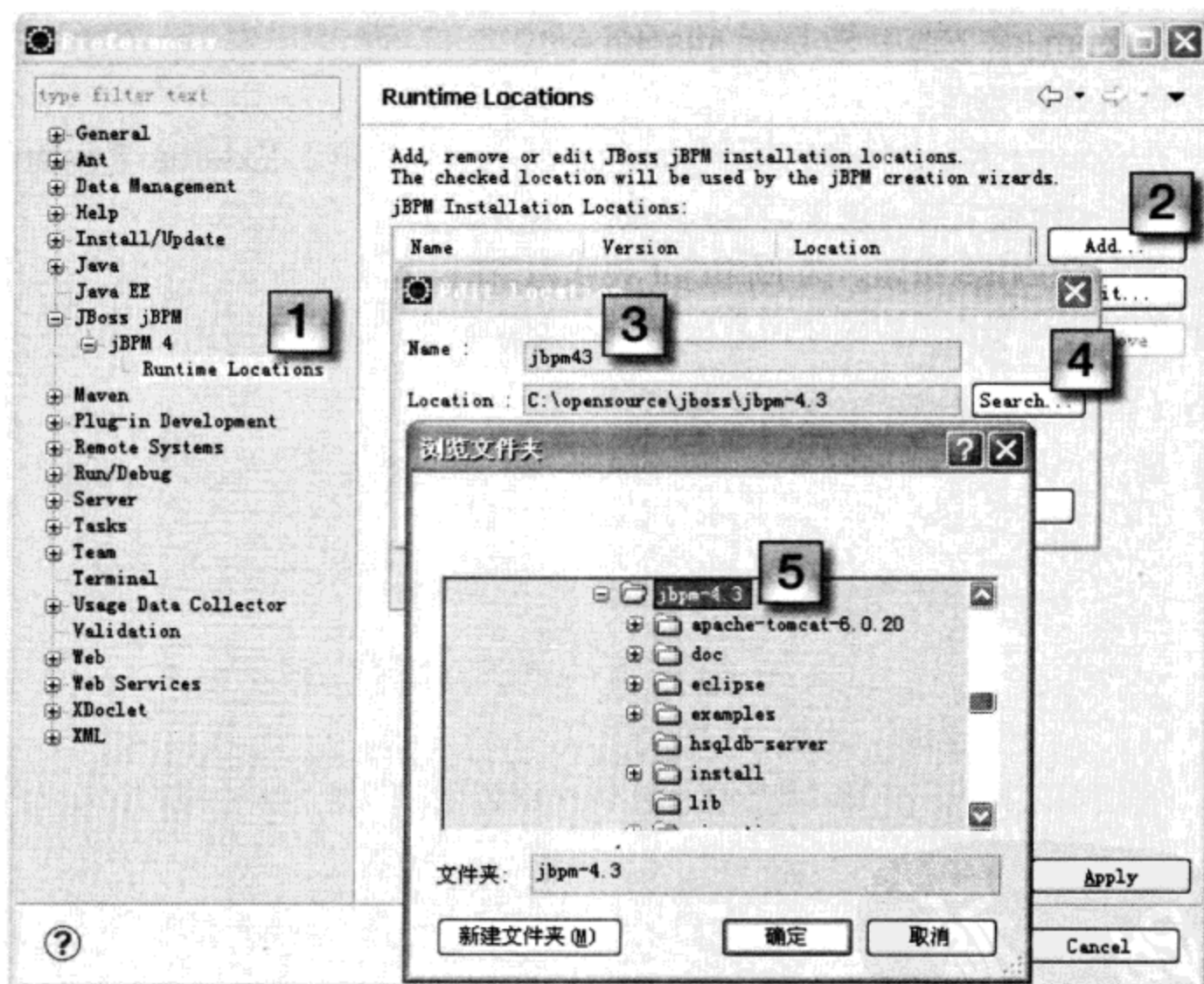


图 2-3 在 Eclipse 中配置 jBPM 运行环境

接下来为您的工作空间定义一个 jBPM 用户库 (User Libraries) 吧，它可以被用来引用 jBPM 的所有依赖库文件。如果您新建一个 jBPM 工程，只需将这个 jBPM 用户库添加到 build path 下即可。

- 1) 选择 Window→Preferences 命令。
- 2) 选择 Java→Build Path→User Libraries 选项。

- 3) 单击 **New** 按钮。
- 4) 输入名称 **jBPM Libraries**。
- 5) 单击 **Add JARs** 按钮。
- 6) 找到 jBPM 安装目录下的 lib 目录。
- 7) 选择 lib 目录下的所有 jar 文件，并单击 **Open** 按钮。
- 8) 选中刚才新建的 **jBPM Libraries**。
- 9) 重新单击 **Add JARs** 按钮。
- 10) 在 jBPM 的安装目录下选择 **jbpm.jar** 文件。
- 11) 单击 **Open** 按钮。
- 12) 在 **jbpm.jar** 下选中 **Source attachment**。
- 13) 单击 **Edit** 按钮。
- 14) 在 **Source Attachment Configuration** 对话框中，单击 **External Folder** 按钮。
- 15) 找到 jBPM 安装目录下的 **src** 目录。
- 16) 单击 **Choose** 按钮，为 **jbpm.jar** 关联源代码。
- 17) 单击两次 **OK** 按钮关闭所有对话框，完成。

图 2-4 所示是主要操作步骤的示意图。

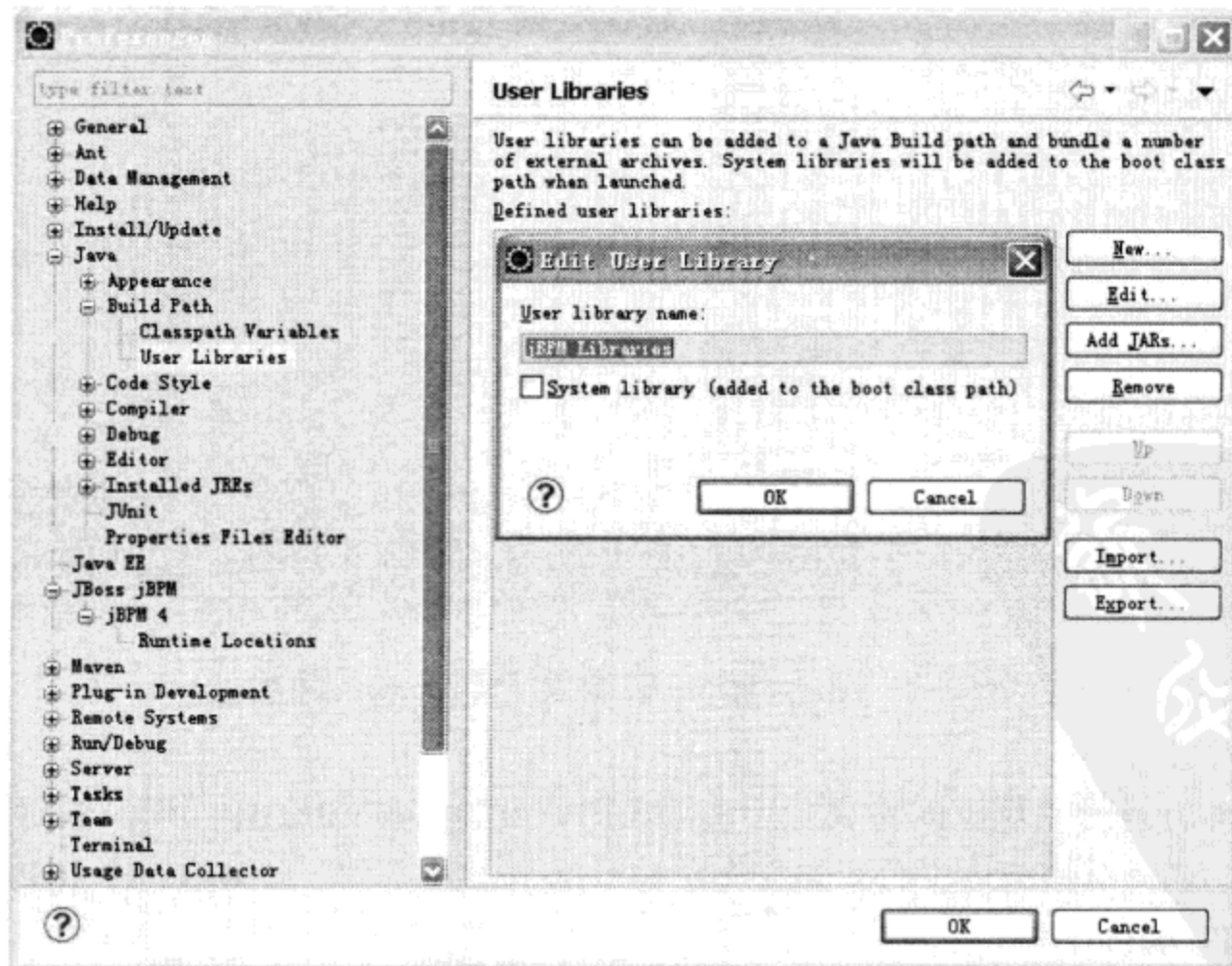


图 2-4 为工作空间定义 jBPM 用户库

2.9.4 添加 jPDL4 Schema 校验

正如之前我们所了解的，jPDL 是 jBPM 独有的流程定义语言，它以 XML 文件的形式描述业务流程。

在 jBPM3 即 jPDL3 中，要描述一个业务流程，系统需要产生两个文件：

- `gpd.xml`——这个文件记录业务流程的图形化表述，即一个单纯的图形坐标指示文件。
- `processdefinition.xml`——这才是真正的 jPDL 文件，描述业务流程的定义。

现在，在 jBPM4 即 jPDL4 体系设计中，已经将上述两个文件合二为一，即在 jPDL 中支持图形坐标位置的描述，开发者再也不用同时维护两个流程定义描述文件了，这是一个有效的简化。

由于 jBPM 官方提供的图形化流程设计器功能并不是特别全面，很多设计并不能全在图形界面下完成。因此，在很多情况下，我们需要直接编辑 jPDL 的 XML 源代码，所以，最好为 jPDL XML 指定 Schema，这样在编辑流程定义源代码的时候，可以通过编辑器快捷键“Alt+/”快速呼出语法提示，同时这个 Schema 关联还可以帮助您校验出 jPDL 的语法错误，帮助您更为高效地编写流程定义 XML 源代码。

在 Eclipse 中配置此 Schema 的过程如下：

- 1) 选择 Window→Preferences 命令。
- 2) 选择 XML→XML Catalog 选项。
- 3) 单击 Add 按钮。
- 4) 将添加 XML Catalog Entry 的窗口打开。
- 5) 单击 File System 按钮。
- 6) 在打开的对话框中，选择 jBPM4 安装目录下 src 文件夹中 `jpdl.xsd` 文件。
- 7) 单击打开按钮。
- 8) 关闭所有的对话框，配置完成。

图 2-5 所示是主要操作步骤的示意图。

2.9.5 导入和使用范例

掌握一门新技术，最高效的手段就是在理解概念的基础上，自己亲自动手实验，

不是吗？

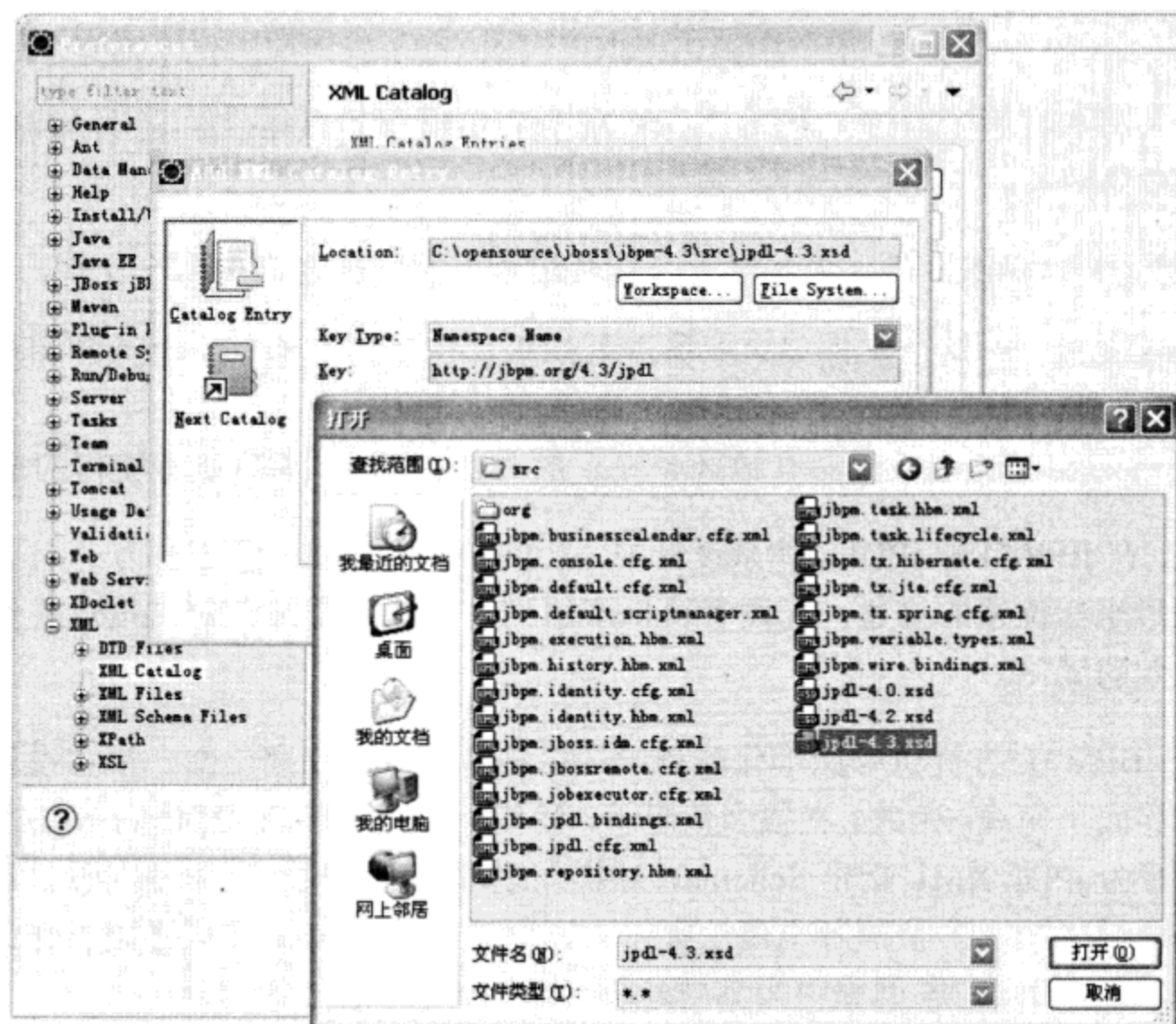


图 2-5 在 Eclipse IDE 中添加 jPDL4 Schema 校验

在 jBPM4 的软件包中，含有丰富的范例流程和测试代码，下面介绍如何将这些范例导入您的 Eclipse IDE，成为一个 examples 工程，供您学习和研究。

- 1) 选择 File→Import 命令。
- 2) 选择 General→Existing Projects into Workspace 选项。
- 3) 单击 Next 按钮。
- 4) 单击 Browse 按钮去选择 jBPM4 安装目录下的 examples 子目录。
- 5) 单击 OK 按钮。
- 6) 范例工程 examples 会被识别并准备导入。
- 7) 单击 Finish 按钮，完成。

在配置了 jBPM4 用户依赖库并且导入了范例工程后，范例中所有的单元测试类（继承自 org.jbpm.test.JbpmTestCase）都可以作为 JUnit Test 运行了，在测试类或方法上单击鼠标右键，选择 Run As→JUnit Test 命令即可。

设置完成了，还等什么呢？运行几个范例试试看吧！

提示：您可以使用 Eclipse + Ant 来处理范例流程的发布。首先，选择 Window → Show View → Other → Ant → Ant 命令，打开 Ant 视图；然后，将范例工程中的 Ant 构建文件 build.xml，从包视图拖曳到 Ant 视图，即可使用其中的 Ant 构建任务(target)来发布范例流程到目标服务器上。关于部署流程的细节，可参见第 4 章 把流程部署到服务器上去。

2.10 例程：jBPM HelloWorld

在本例程中，我们将设计一个最简单的流程定义——“HelloWorld”。

步骤如下：

- 1) 打开已经安装 GPD 的 Eclipse，新建一个“jBPM4 Process Definition”(jBPM4 流程定义文件)，命名为 process.jpdl.xml，如图 2-6 所示。

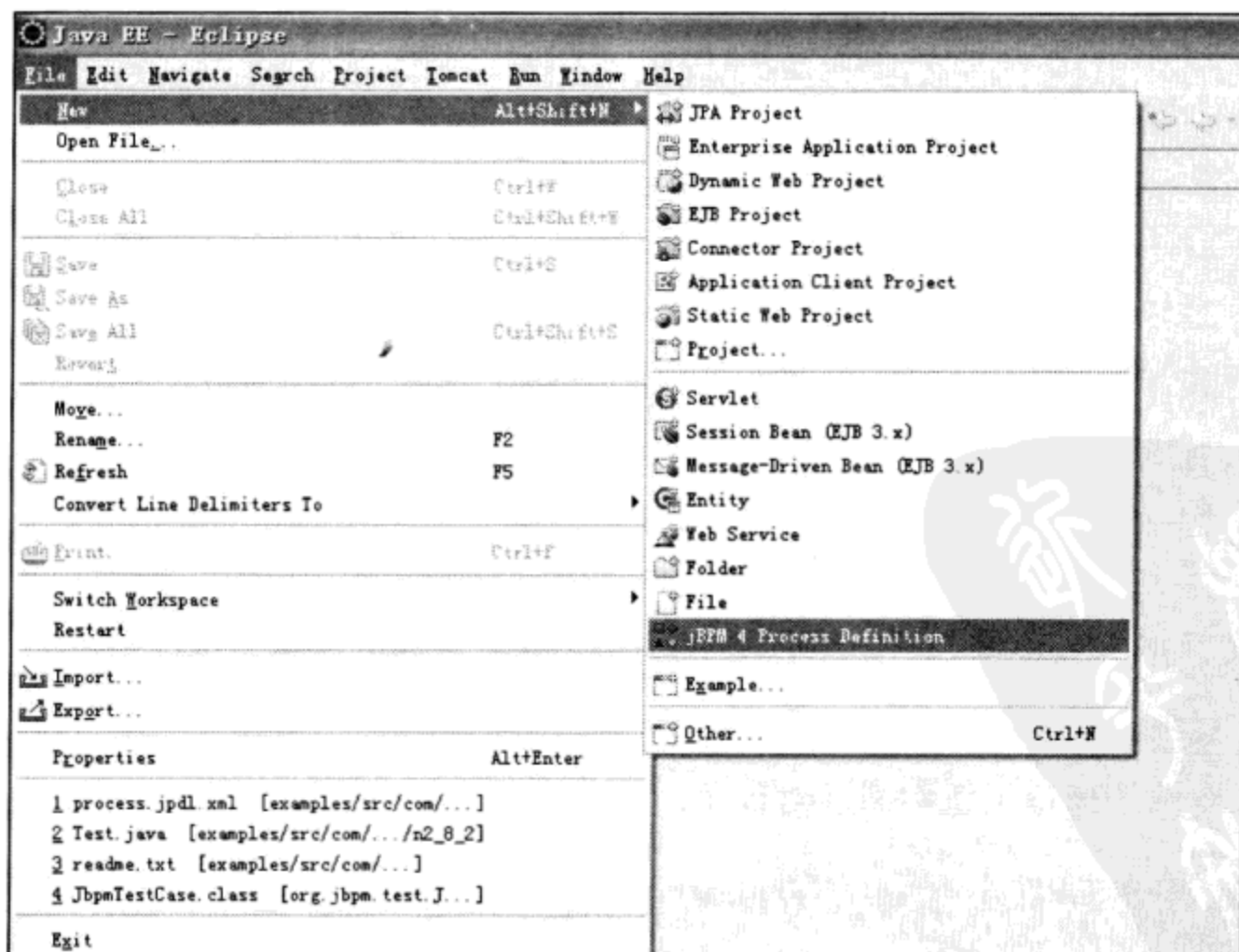


图 2-6 新建 jBPM4 流程定义文件

- 2) 进入流程定义设计界面，单击空白的流程图，在属性窗口中设置流程名称为“HelloWorld”。
- 3) 从左侧的组件工具栏（Components）中拖曳 start 活动、state 活动、end 活动至流程图。
- 4) 单击这 3 个活动，在其属性窗口中分别命名为“start”、“state”、“end”。
- 5) 从组件工具栏中选择 transition 转移线连接 start 活动至 state 活动、state 活动至 end 活动。完成设计后，如图 2-7 所示。

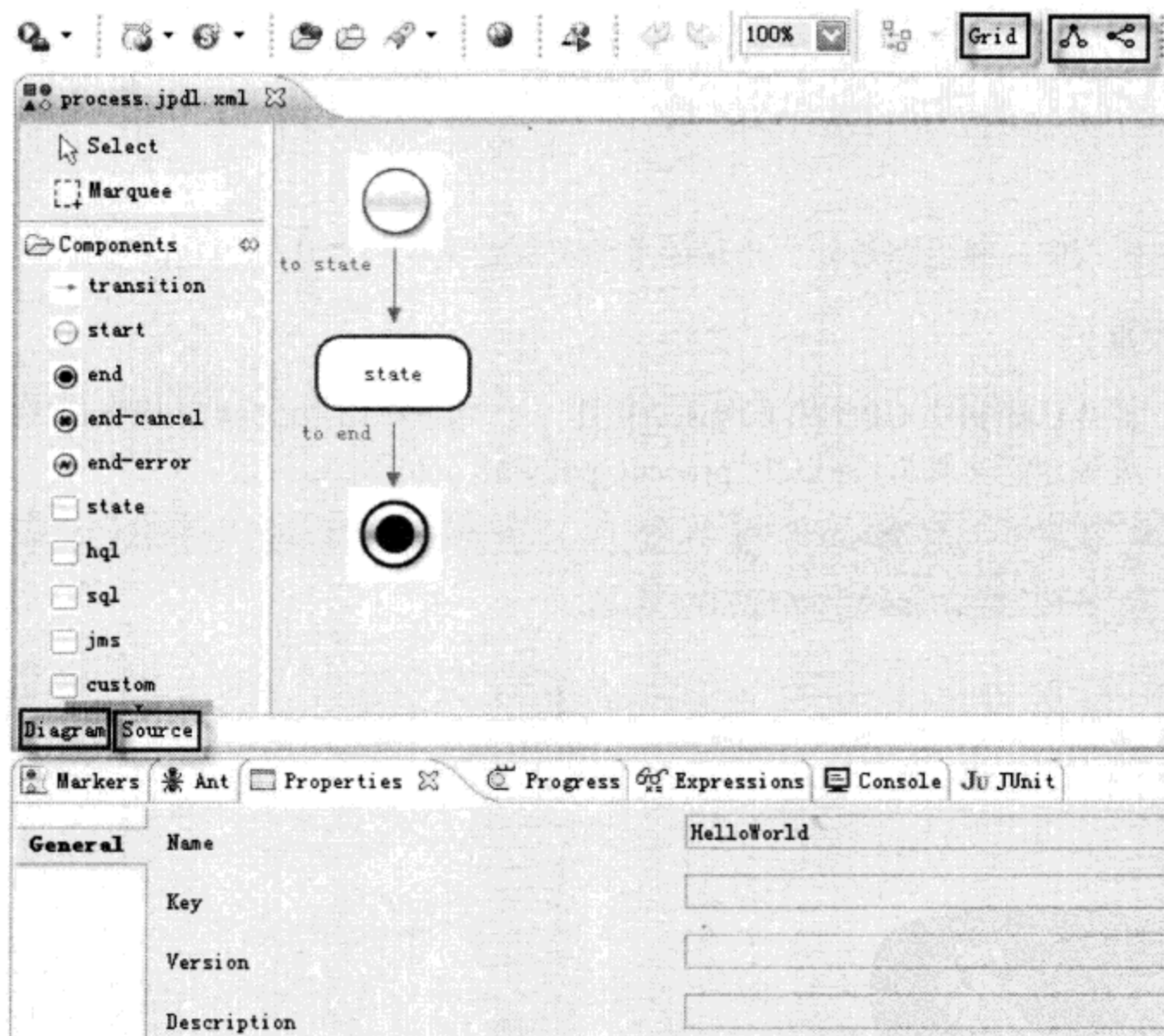



图 2-7 HelloWorld 流程定义设计示意图

- 6) 使用 GPD 顶部的 Grid 按钮为流程图显示网格线，用于对齐；还可以使用  两个按钮使流程图自动布局。

最终，单击 GPD 左下侧的“Diagram”页签，可以查看“HelloWorld”流程定义的图形化表达，如图 2-8 所示。

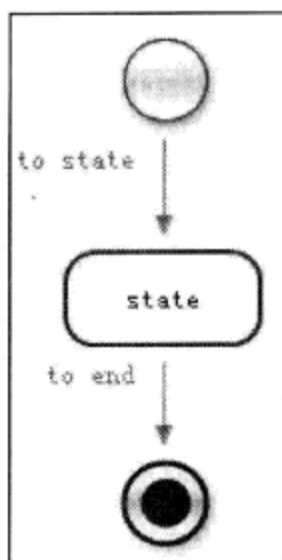


图 2-8 HelloWorld 流程定义图形

单击 GPD 左下侧的“Source”页签，可以查看流程图对应的 jPDL 源代码：

```
<process name="HelloWorld" xmlns="http://jbpm.org/4.3/jpdl">
  <start name="start">
    <transition name="to state" to="state"/>
  </start>
  <state name="state">
    <transition name="to end" to="end"/>
  </state>
  <end name="end"/>
</process>
```

2.11 小结

通过本章的内容，您学会了如何安装 jBPM4 的开发运行环境到目标操作系统，并且可以根据自己的安装需求做出一些个性化配置，最后通过一个简单的 Hello World 例程向世界宣布——jBPM，我来了！

下一章将介绍使用 jBPM 开发应用程序的第一个步骤——使用 jBPM 图形化流程设计器（GPD）来设计业务流程。

使用 jBPM 图形化流程设计器设计流程

本章将介绍如何使用 GPD (Graph Process Designer), 即 jBPM 图形化流程设计器。通过上一章的介绍, 我们安装好 GPD 和导入范例工程之后, 在 Eclipse IDE 中会看到 jPDL 流程定义文件都有一个对应的特殊图标^{▲◆}。在左侧的包视图的下面双击此类图标文件, 就会在 GPD 中打开 jPDL 流程定义文件, 如图 3-1 所示。

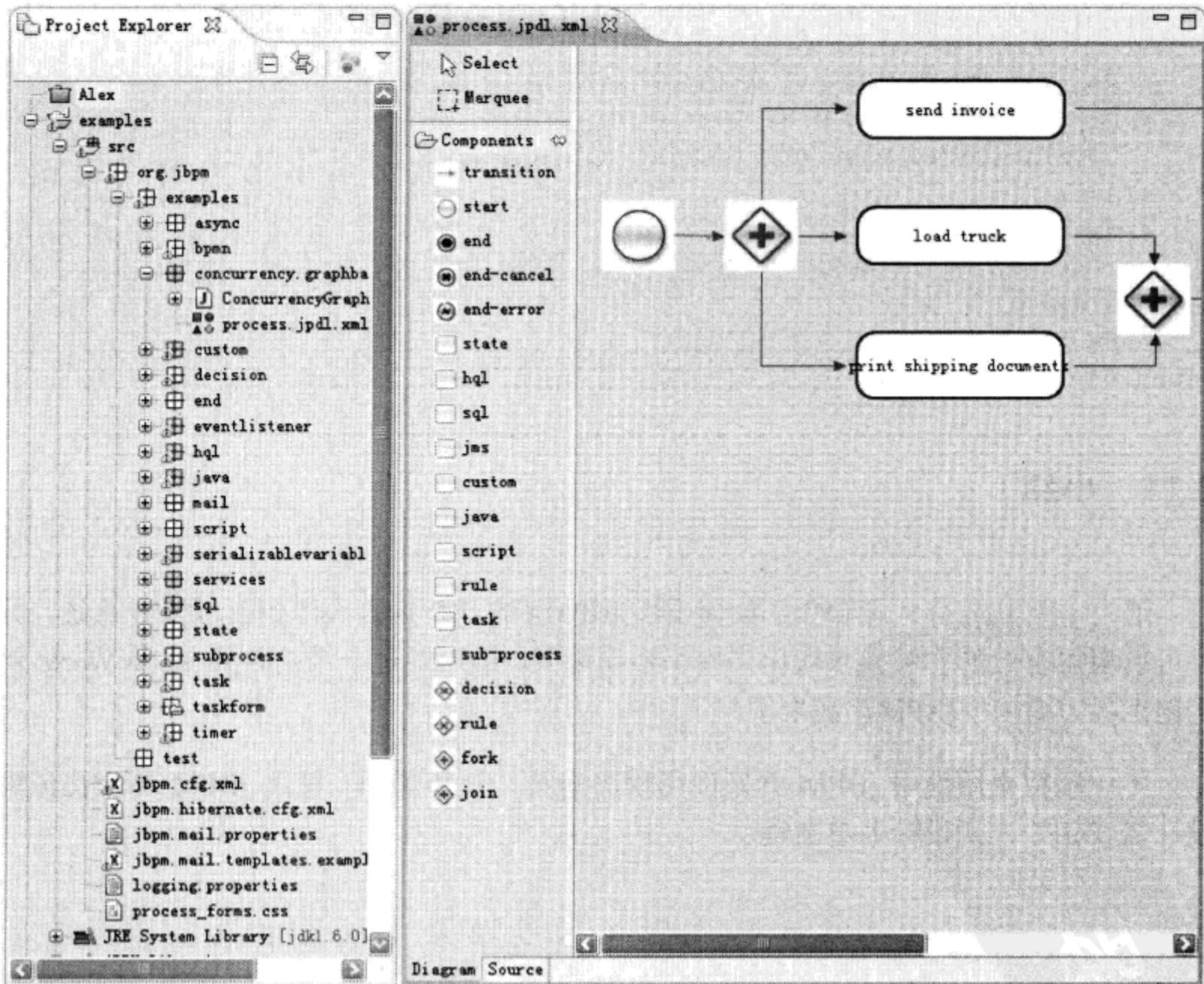


图 3-1 jBPM4 GPD 效果图

需要知道的是：

- 与 jBPM3 不同，在 jBPM4 中，图形描述文件和流程定义文件合二为一，即在同一个 `jpdl.xml` 文件中描述图形和流程，这在前面的章节中详细说明过。
- 目前，jBPM4.3 版本的 GPD 只支持对 jPDL 语言描述的流程进行可视化设计。对于 jBPM4 workflow 引擎内核同样支持的以 BPEL/BPMN 语言描述的流程定义，目前的 GPD 则不能支持其可视化的设计过程。

3.1 创建一个新流程

使用 GPD 创建一个新的流程定义的过程如下：

- 1) 使用快捷键 `Ctrl+N` 打开向导选择器。这个操作等同于使用菜单命令 `File→New→Other`。
- 2) 在向导选择器中选择 `JBoss jBPM→jBPM 4 Process Definition` 选项，如图 3-2 所示。

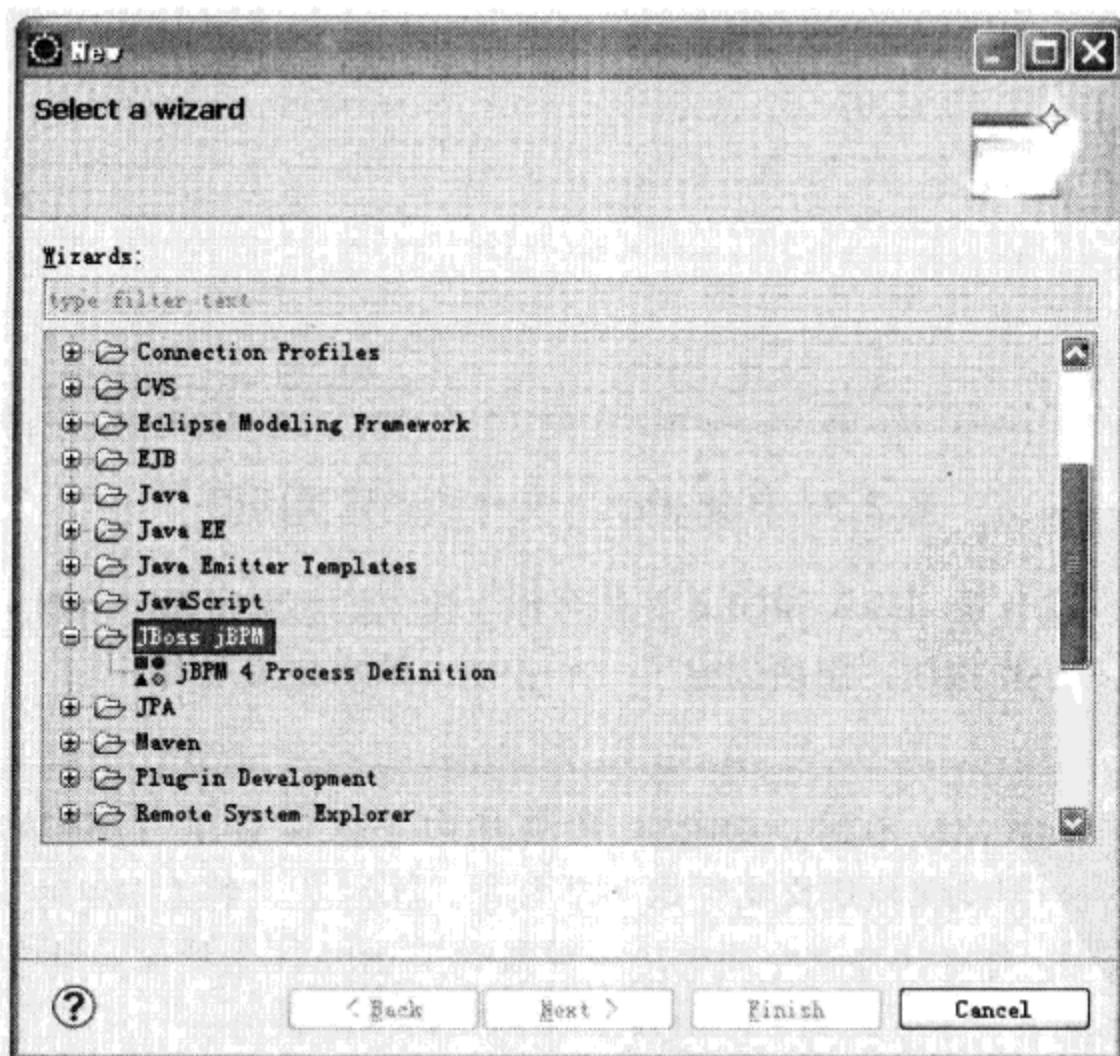


图 3-2 向导选择器

3) 单击 **Next** 按钮，就会进入创建新 jPDL4 文件的向导，如图 3-3 所示。

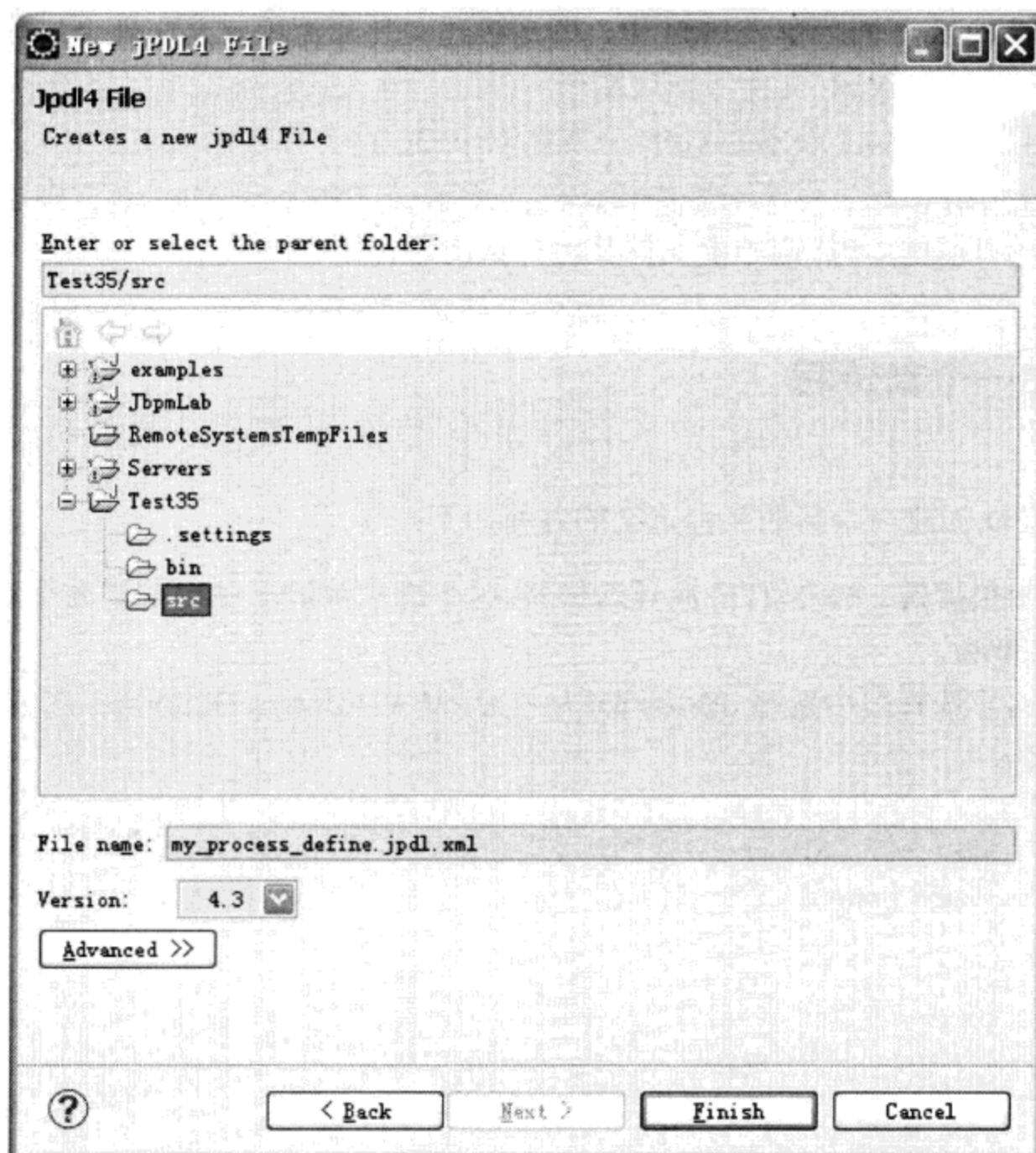


图 3-3 创建 jPDL4 文件的向导对话框

- 4) 选定创建目录，输入 jPDL4 文件名称（建议扩展名遵循 *.jpdl.xml 规范）。
- 5) 单击 **Finish** 按钮，您便创建了一个 jPDL4 流程定义文件。

提示：与 jBPM3 不同，jBPM4 的创建向导仅保留了“jBPM 4 Process Definition”——创建流程定义文件这一个向导，取消了“创建 jBPM 工程（jBPM Project）”的向导。作者认为这个简化进一步强调了 jBPM4 支持嵌入所有 Java 工程的特性，同时体现了“不要让用户在选择中迷惑”的体验强化。

3.2 编辑流程定义源

如同前面提到的，GPD 只能帮助开发者完成有限的可视化流程定义设计，很多 jPDL 语言的高级特性仍然需要依靠开发者手工编辑 jPDL 定义文件的 XML 源代码实现。

当打开一个 jPDL 流程定义的时候，在主界面的左下角有一个标签 **Source**，单击这个标签即可切换到直接编辑 jPDL XML 内容的界面，如图 3-4 所示。

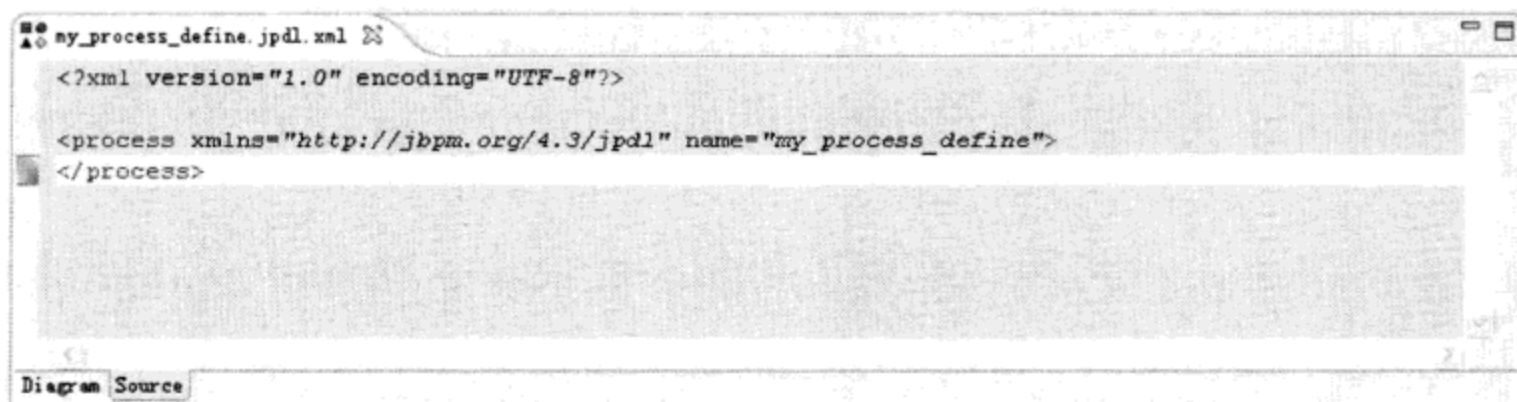


图 3-4 在 Source 视图下编辑 jPDL XML

在 Source 视图下编辑结束后，可以单击 **Diagram** 标签切换回图形化设计界面，您刚才的修改会忠实地反映在图形化设计界面上，如果相应的图形元素被 GPD 支持的话。

3.3 例程：设计一个“复杂的”业务流程

掌握了 GPD 的使用后，我们可以尝试用它画出一个稍微复杂点的业务流程图。这不仅是流程开发者需要具有的技能，同样，使用 GPD 也是业务流程分析者要做的事。

假设某制造型企业有“订单-生产”流程，步骤如下：

- 1) 订单输入。
- 2) 订单审核。
- 3) 如果订单审核不通过，则该笔业务结束。
- 4) 如果订单审核通过，则开始进入“生产-交付”过程。
- 5) 产品的生产、装运过程与财务收支同步进行。
- 6) 当产品交付和财务结算都完成后，则该笔业务结束。

先思考一下该如何设计这个流程……

打开 GPD。我们使用任务活动 `order apply` 和 `order check` 定义步骤 1) 订单输入和步骤 2) 订单审核，因为这两个步骤需要人工办理；使用判断活动 `order auditing` 处理步骤 3) 订单审核的意见，如果审核不通过则经过 `cancel` 转移结束流程。如果审核通过则经过 `approve` 转移流向分支活动 `fork`，`fork` 活动产生两个并行的流程分支，一支定义产品的生产、装运过程，即 `shipping` 活动和 `receiving` 活动；另一支定义财务收支的办理，即 `financing` 活动；当两个流程分支同时完成，才能通过聚合活动 `join` 结束整个流程。

最终，设计出的“订单-生产”流程定义图如图 3-5 所示。

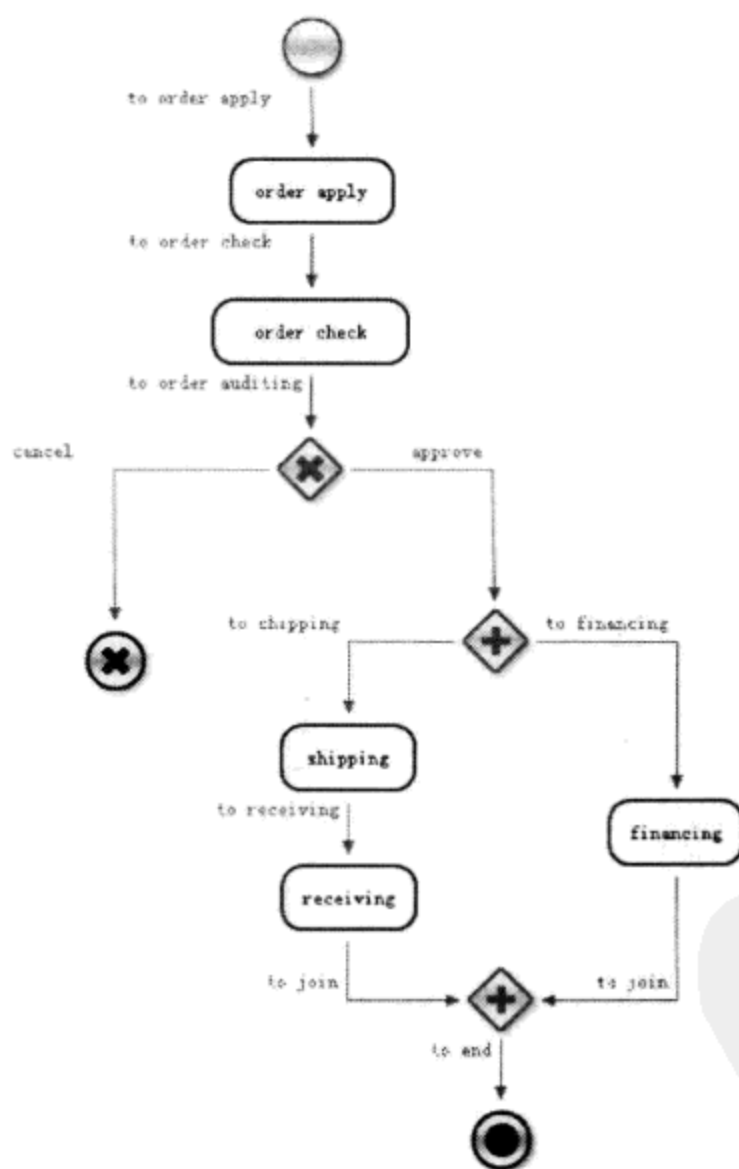


图 3-5 “订单-生产”流程图

当然，在实际的应用中，将流程图“画”完还远远未完成流程定义。作为开发者需要在流程图的基础上，填入相应的属性值和根据需求编写用户代码并引用之，甚至在必要的时候手工修改流程图的 jPDL 源文件来完善流程定义。例如：

- 选中流程活动，定义其属性，如图 3-6 所示。

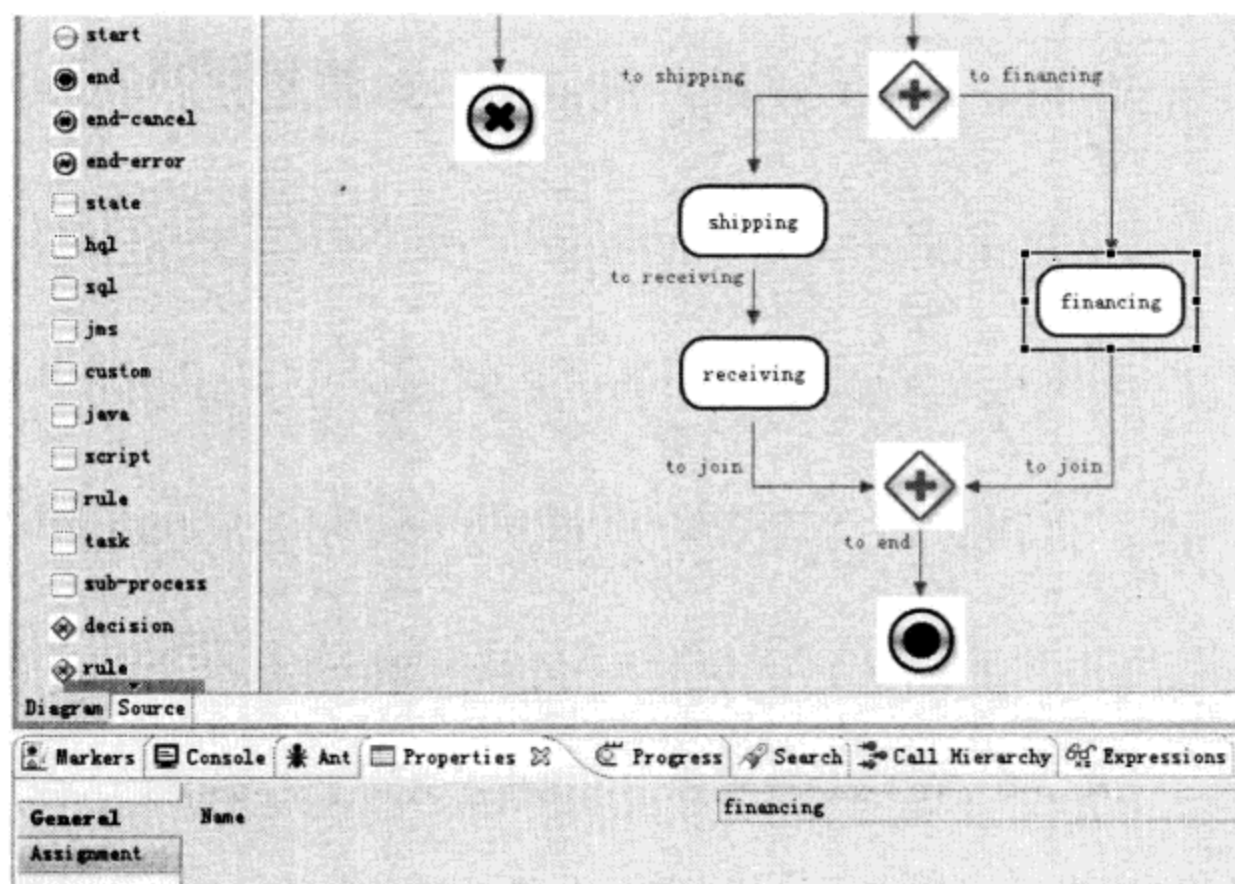


图 3-6 通过 Properties 选项卡编辑 financing 活动的属性

- 选中流程活动，为之定义“事件-监听器”。事件-监听器是用户代码，在后面的章节中会介绍，如图 3-7 所示。

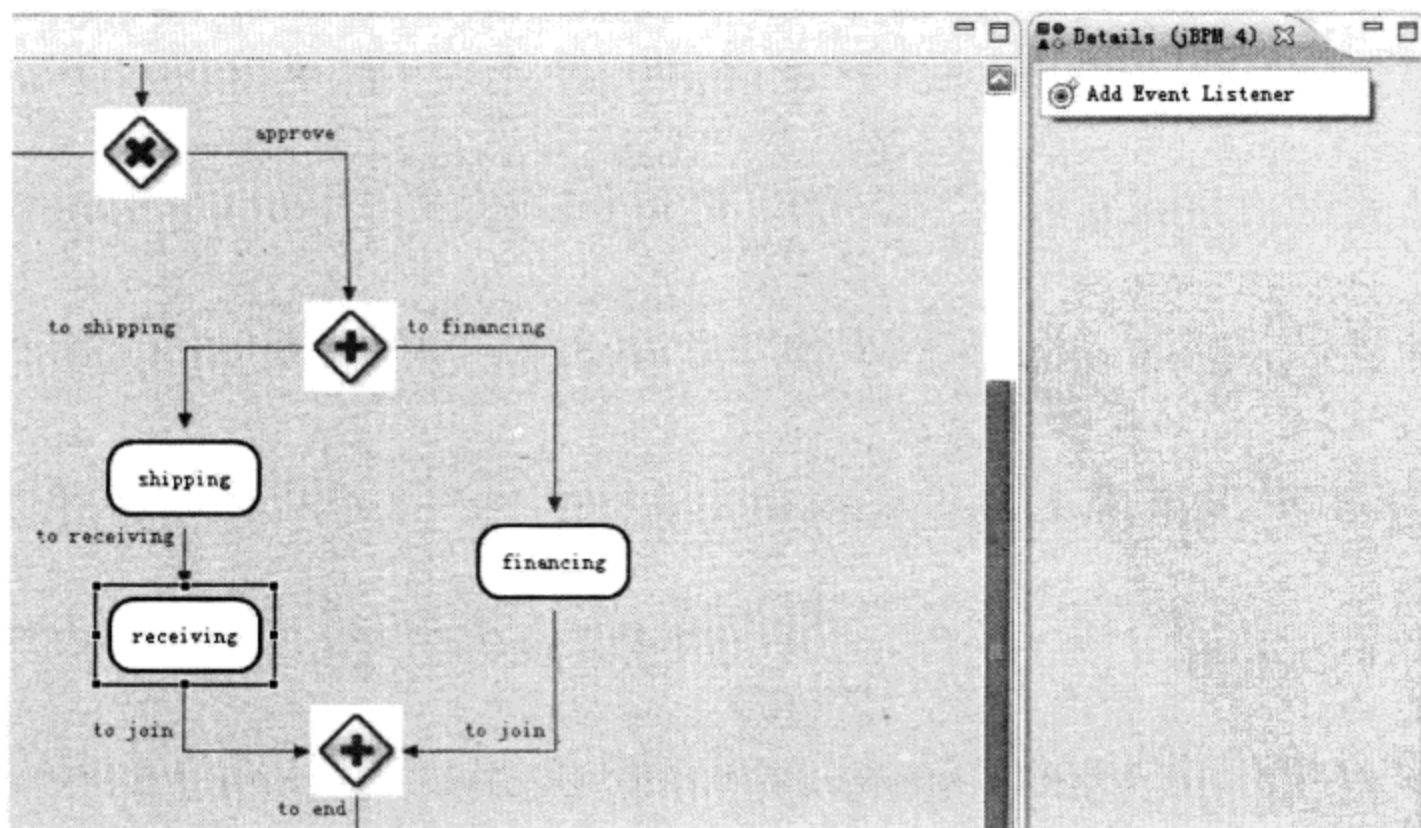


图 3-7 在右侧 Details 面板中为 receiving 活动添加“事件-监听器”元素

- 单击流程图空白处，可以定义全局的流程元素，例如泳道、事件-监听器、定时器，如图 3-8 所示。

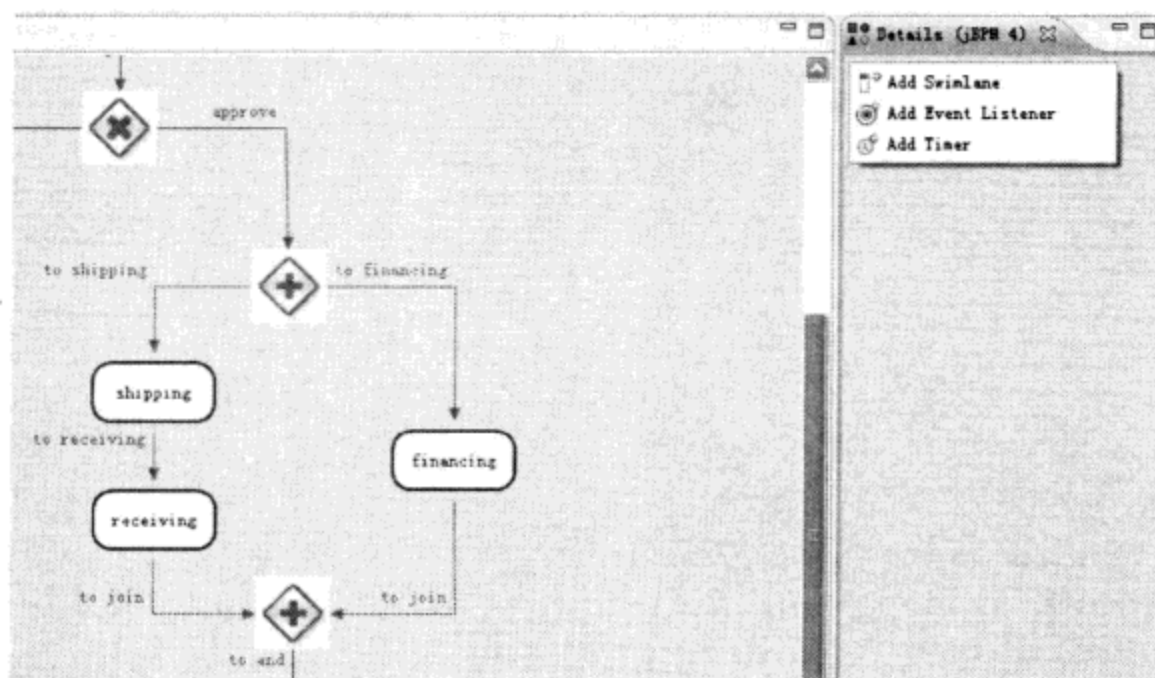


图 3-8 在 Details 面板中为流程定义添加全局元素

- 当某些流程定义需求无法使用图形化设计器直接满足时，则可单击 Source 选项卡直接编辑 jPDL 源代码实现，如图 3-9 所示。

```
<process name="OrderProcess" xmlns="http://jbpm.org/4.3/jpdl">
  <start g="216,17,48,52" name="start">
    <transition g="-107,-17" name="to order apply" to="order apply" />
  </start>
  <task g="186,101,109,52" name="order apply">
    <transition g="-107,-17" name="to order check" to="order check" />
  </task>
  <task g="176,185,128,52" name="order check">
    <transition g="-107,-17" name="to order auditing" to="order auditing" />
  </task>
  <decision expr="#{var}" g="216,269,48,48" name="order auditing">
    <transition g="125,293:-59,-17" name="cancel" to="cancel" />
    <transition g="346,293:-47,-17" name="approve" to="fork" />
  </decision>
  <end-cancel g="103,384,48,52" name="cancel" />
  <fork g="325,372,48,52" name="fork">
    <transition g="455,396:-77,-17" name="to financing" to="financing" />
    <transition g="263,396:-71,-17" name="to shipping" to="shipping" />
  </fork>
  <java expr="#{ShippingClass}" g="217,439,92,52" method="shipping"
    name="shipping">
    <transition g="-77,-17" name="to receiving" to="receiving" />
  </java>
  <state g="218,522,92,52" name="receiving">
    <transition g="263,609:-47,-17" name="to join" to="join" />
  </state>
  <task g="409,483,92,52" name="financing">
    <transition g="456,609:-47,-17" name="to join" to="join" />
  </task>
  <join g="329,585,48,52" name="join">
    <transition g="-41,-17" name="to end" to="end" />
  </join>
  <end g="330,669,48,52" name="end" />
</process>
```

图 3-9 “订单-生产” 流程定义的 jPDL 源代码

3.4 小结

通过本章的介绍，您了解了如何操作基于 Eclipse IDE 的图形化流程设计器——GPD，去创造和设计流程定义。事实上，最关键的难点在于如何将客户的业务流程需求正确地转化为 jPDL 流程定义，即选择正确的活动“组装出”符合需求的业务流程。如何选择正确的活动？我认为首先需要对每个活动的功能、所适用的业务场景了如指掌，相信**第 6 章 掌握 jBPM 流程定义语言**能帮助您解决绝大部分这方面的问题。

在下面一章中将介绍如何把设计完成的流程定义部署到 jBPM4 运行环境中去。



把流程部署到服务器上去

当我们的业务流程被设计开发完毕后，会有许多种相关的文件“散落”在工程中，这包括：

- 定义流程的 jPDL 文件。
- 根据图形化流程定义同步生成的流程图片文件（PNG 格式）。
- 业务流程中用于人机交互的表单页面文件。
- 事件监听器等用户自定义代码的 Java 类文件。
- 其他流程资源文件，例如小图标、CSS 样式表、脚本文件、属性文件……

这么多“乱七八糟”的文件，如果一个个地手工往服务器上部署，岂不累煞我等？所以，jBPM4 支持将流程定义及其相关资源打包成一个 JAR（Java 归档）格式的文件，部署到服务器上（其实是服务所连接的 jBPM 数据库中），然后流程定义就可以被执行了。这个归档文件就是“**业务流程归档**”。

4.1 部署流程定义和资源文件

如何将流程定义和流程相关资源部署到 jBPM 数据库中？

jBPM4 workflow 引擎提供了一个基于 Ant 任务的 API 来部署业务流程归档——`org.jbpm.pvm.internal.ant.JbpmDeployTask`。

`JbpmDeployTask` 不仅可以部署单个业务流程归档，也可以部署一组业务流程归档到服务器上。`JbpmDeployTask` 通过读取 `jbpm.cfg.xml` 中的 JDBC 数据连接信息直接将业务流程归档部署到数据库中。因此，在使用 `JbpmDeployTask` 部署您的流程定义之前，请确保部署的目标数据库正在运行。

创建和部署业务流程归档的例子位于 jBPM4 发布包的 `examples` 目录下的 `Ant build.xml` 中，任务名称是 `create.and.deploy.examples`。这个 Ant 任务解决问题的步骤如下：

第一步, 需要声明一个 path 任务来指定包含 JbpmDeployTask 的 jbpm.jar 及其所有依赖库。

```
<target name="jbpm.libs.path">
  <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>
</target>
```

注意: 使用的数据库系统的 JDBC 驱动程序 jar 包应该被包含在这个 path 中。尽管 MySQL, PostgreSQL 和 HSQLDB 等开源数据库系统的驱动程序默认都包含在 jBPM4 发布包中。但是诸如 SQL Server, Oracle 之类商业数据库系统的驱动程序必须自己获取并包含进去。

第二步, 需要创建一个业务流程归档, 可以使用 Ant 的 jar 任务, 下面我们创建一个名为 examples.jar 的业务流程归档, 上面我们定义的 jbpm.libs.path 在这个任务中被用到了。

```
<target name="create.and.deploy.examples" depends="jbpm.libs.path,
examples.jar">
  <mkdir dir="${jbpm.home}/examples/target" />
  <copy file="${jbpm.home}/install/src/cfg/hibernate/jdbc/${database}.
hibernate.cfg.xml"

  tofile="${jbpm.home}/examples/target/classes/jbpm.hibernate.cfg.xml"
  overwrite="true">
    <filterset
filtersfile="${jbpm.home}/install/jdbc/${database}.properties" />
  </copy>
  <jar destfile="${jbpm.home}/examples/target/examples.bar">
    <fileset dir="${jbpm.home}/examples/src">
      <exclude name="jbpm.cfg.xml" />
      <exclude name="jbpm.hibernate.cfg.xml" />
      <exclude name="jbpm.mail.properties" />
      <exclude name="jbpm.mail.templates.examples.xml" />
      <exclude name="logging.properties" />
      ...
    </fileset>
  </jar>
```

```
...  
</target>
```

在这里有必要解释一下 jBPM workflow 引擎将如何处理这个业务流程归档： workflow 引擎扫描业务流程归档中所有以 `.jpdL.xml` 结尾的文件，所有这些文件的内容都会被当做 jPDL 流程定义解析，然后可以被用来发起流程实例。业务流程归档中所有其他资源也会在部署过程中被持久化到数据库中。

所有这些资源被统一编号保存在数据库表 `jbpm4_lob` 中。因此，我们可以很方便地通过 jBPM4 提供的 `RepositoryService.getResourceAsStream` API 随时访问这些资源。

第三步，就是部署。我们需要先在 Ant 中将 `JbpmDeployTask` 声明成一个名为 `jbpm-deploy` 的自定义任务：

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

然后就可以调用这个自定义的 Ant 任务：

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

通过 `JbpmDeployTask` 预留的 `file` 接口参数，指定需要部署的业务流程归档文件。

查看 `org.jbpm.pvm.internal.ant.JbpmDeployTask` 类的源代码，可以发现 `JbpmDeployTask` 预留了 4 个接口参数：

```
String jbpmCfg = null;  
//这个 file 参数就是我们上面用到的，用于指定需要部署的业务流程归档文件。  
File file = null;  
List fileSets = new ArrayList();  
boolean failOnError = true;
```

因此，相应的 `jbpm-deploy` 这个我们自定义的 Ant 任务，有如表 4-1 和表 4-2 所示的可配置属性和元素。

表 4-1 jbpm-deploy 任务支持的属性

属性	类型	默认值	是否必填	描述
file	文件	无	可选	指定需要被部署的业务流程定义文件。以 .xml 结尾的文件会被作为流程定义文件直接部署；以 “*ar” 结尾的文件，比如 .bar 或 .jar 文件，则会作为业务流程归档部署

属性	类型	默认值	是否必填	描述
cfg	文件	jbpm.cfg.xml	可选	指定 jBPM 配置文件。默认寻找 classpath 根目录下的 jbpm.cfg.xml。如果需要自定义, 这个路径应该位于 jbpm-deploy 任务定义的 classpath 范围内

表 4-2 jbpm-deploy 任务支持的元素

元素	关联关系	描述
fileset	0..*	指定需要被部署的业务流程定义文件集合, 以一个简单的 Ant 的 fileset 表示。以 .xml 结尾的文件会被作为流程定义文件直接部署; 以 “*ar” 结尾的文件, 比如 .bar 或 .jar 文件, 则会作为业务流程归档部署

4.2 部署流程 Java 类的 3 个方法

从 jBPM 4.2 版本开始, jBPM4 就拥有了一颗像 jBPM3 一样的流程专用类加载器, 用来加载被部署在数据库中的流程 Java 类。

这个 jBPM4 类加载器与 jBPM3 的不同之处在于, 您可以理解为它会在业务流程归档的根目录中搜索 .class 类资源。例如, 当类 com.mytrade.Order 在流程执行中被引用时, jBPM4 类加载器就会在相应的业务流程归档中查询此路径: /com/mytrade/Order.class。Order 类会被作为“键-值”缓存, 其 key 值是结合了业务流程归档发布信息 and 上下文类加载器得出的。因此 jBPM4 类加载器应该比 jBPM3 类加载器的工作效率更高。

当然, 您也可以不把 Java 类打包在业务流程归档中部署。也许通过以下两种方式去部署流程 Java 类对您来说更有效率:

- 把 Java 类部署到应用服务器的类库中。例如在 Tomcat 和 JBoss 服务器的 lib 目录中部署流程 Java 类的 jar 包。这种方式加载类的优先级为最高。
- 把流程 Java 类部署到 Web 应用或企业应用相应目录中, 例如 WEB-INF/classes 目录或 WEB-INF/lib 目录。Tomcat 或 JBoss 服务器在运行时会在此 jBPM Web 应用或企业应用中, 找到流程所需的 Java 类并调用之。这种方式加载类的优

优先级为次高。

关于第一种方式,您可以参考 jBPM4 安装脚本中的 `install.examples.into.tomcat` 和 `install.examples.into.jboss` 任务。

在未来的 jBPM4 发布版本中,很可能在 jPDL 流程定义文件中支持包含 Java 类。

4.3 例程: 部署业务流程定义

以下例程将介绍如何将已经设计好的流程定义部署到数据库中。

首先,需要配置好 jBPM 运行环境,即在 `classpath` 根目录下的 `jbpm.hibernate.cfg.xml` 中设置好数据库连接,并确保数据库服务正常。

本例程将介绍两种部署流程定义的方式。

- 1) 通过执行 Ant 脚本的方式进行部署。这需要将要部署的流程定义及其相关资源按照 `classpath` 的层次结构打包归档。假设业务流程归档为 `process.jar`,则部署 `process.jar` 的 Ant 脚本如下:

```
<project name="jbpm4examples.deployment">
  <!-- 使用 jbpm.home 表示 jBPM4 的安装目录。 -->
  <property name="jbpm.home" value="C:/opensource/jboss/jbpm-4.3" />
  <!-- 这个 Ant 任务专门用来定义 jBPM 的运行库、依赖库以及配置文件的 classpath 路径。 -->

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

  <target name="process.jar" depends="jbpm.libs.path">
  </target>

  <!-- 使用 jBPM4  workflow 引擎提供的 Ant API—— JbpmDeployTask 来执行部署。 -->
  <target name="create.and.deploy.examples" depends="jbpm.libs.path,process.jar">
    <taskdef name="jbpm-deploy" classname="org.jbpm.pvm.internal.ant.
```

```

JbpmDeployTask"
    classpathref="jbpm.libs.incl.dependencies" />
    <!-- 在这里指定流程定义打包 -->
    <jbpm-deploy file="process.jar" />
</target>
</project>

```

最后，执行 Ant 任务 `create.and.deploy.examples` 即可部署流程定义打包 `process.jar` 至 jBPM 数据库中。

- 2) 通过编写 Java 代码直接调用 jBPM workflow 引擎提供的部署服务 API 完成流程定义部署。在下面的例程中，将使用基于 JUnit 的单元测试架构来演示这一调用。完整的单元测试代码如下：

```

// JbpmTestCase 继承了 JUnit 的 TestCase 类，是 jBPM4 对 JUnit 框架的扩展
public class Test extends JbpmTestCase {
    //这个成员域为单元测试保存流程定义的部署 ID
    String deploymentId;
    //一般在单元测试的初始化方法（setUp）中，执行流程定义的部署工作
    //这是 jBPM4 单元测试的约定。在后面的单元测试代码中，都将默认执行此约定
    @Override
    protected void setUp() throws Exception {
        super.setUp();
        //使用 RepositoryService 提供的 API 方法从 classpath 中部署流程定义
        deploymentId = repositoryService.createDeployment()

            .addResourceFromClasspath("com/examples/jbpm4/n2_3_4/process.jpdl.xml")
            .deploy();
        //当然，在这里可以多次调用 addResourceFromClasspath 方法，将流程定义的其他
        资源都部署到数据库中
    }
    //一般在单元测试的结束方法（tearDown）中，执行删除流程定义部署的工作
    //这也是 jBPM4 单元测试的约定。在后面的单元测试代码中，都将默认执行此约定
    @Override
    protected void tearDown() throws Exception {
        //调用 RepositoryService.deleteDeploymentCascade 方法，将物理清除
        deploymentId 对应的流程定义及其所有相关资源，并关联清除基于此流程定义的流程实例、活动
        实例、任务、历史流程实例等所有运行时及历史的流程实体记录
        repositoryService.deleteDeploymentCascade(deploymentId);
        super.tearDown();
    }
    //这里是单元测试方法
}

```



```

public void test() {
    // “真正的”单元测试代码从这里开始
    ...
}
}

```

解释一下。上面单元测试中流程引擎服务（如 RepositoryService）的初始化工作是由 JbpmTestCase.setUp 方法完成的。JbpmTestCase 作为 jBPM4 框架的单元测试超类，为我们做了如下 6 个流程引擎服务（在第 5 章 使用 jBPM4 Service API 控制流程中有详细说明）的初始化工作：

```

protected static ProcessEngine processEngine = null;
//资源库服务
protected static RepositoryService repositoryService;
//执行服务
protected static ExecutionService executionService;
//管理服务
protected static ManagementService managementService;
//任务服务
protected static TaskService taskService;
//历史服务
protected static HistoryService historyService;
//身份认证服务
protected static IdentityService identityService;

```

JbpmTestCase 在其 initialize 方法中使用如下代码实现了上述流程引擎服务的初始化：

```

if (processEngine==null) {
//根据默认配置，生成 workflow 引擎对象
    processEngine = Configuration.getProcessEngine();
    //利用 workflow 引擎对象，获取 6 个流程引擎服务
    repositoryService = processEngine.get(RepositoryService.class);
    executionService = processEngine.getExecutionService();
    historyService = processEngine.getHistoryService();
    managementService = processEngine.getManagementService();
    taskService = processEngine.getTaskService();
    identityService = processEngine.getIdentityService();
}

```

在某些客户端应用的实现中，开发者也许要参考上述代码自行编码获取某些流程引擎服务。

4.4 小结

通过本章的介绍,您了解到了如何将设计好的流程定义通过 Ant 脚本和调用 jBPM 部署服务 API 等方式发布到 jBPM 的数据库持久化环境中,使得流程定义得到保存,为后续的流程实例化运行提供“模板”基础。

在下一章中将介绍如何使用 jBPM4 的 Service API 基于已部署的流程定义执行:发起流程实例、控制流程运行以及查询流程数据等操作。



使用 jBPM4 Service API 控制流程

当我们定义完流程之后，流程定义在运行时会被实例化，因此我们要创建流程实例；当流程实例在执行中时，我们要控制和监视流程，以确保业务流程执行在监控之中；当流程实例执行完毕，jBPM4 会将其归档到“历史流程”中去，从而提高运行中流程实例的执行效率，而我们需要从历史流程中进行数据分析以优化和重组业务……

以上这些功能的开发，都需要依赖 jBPM4 提供的 Service API 去帮助我们实现。这包括：

- 1) 管理流程部署。
- 2) 管理流程实例。
- 3) 管理流程任务。
- 4) 管理流程历史。
- 5) 以及管理流程的一切……

5.1 流程定义、流程实例和执行的概

首先，通过一个示例流程使您明白在 jBPM4 中流程定义、流程实例以及执行（executions）的概念。

流程定义是对业务过程步骤的描述，在 jBPM4 中它表现为若干“活动”节点通过“转移”线条串联。例如，一个金融公司有一个信贷流程定义，描述公司如何处理信贷请求步骤，如图 5-1 所示。

流程实例是表示流程定义在运行时特有的执行例程。打个比方：你可以把流程定义理解为 Java Class 定义，而流程实例则可以理解为由 Java Class 定义实例化生成的 Java Object 对象。例如，上周五某人提出贷款买车申请，即代表一个信贷流程定义被实例化了——一个信贷流程的实例产生了。

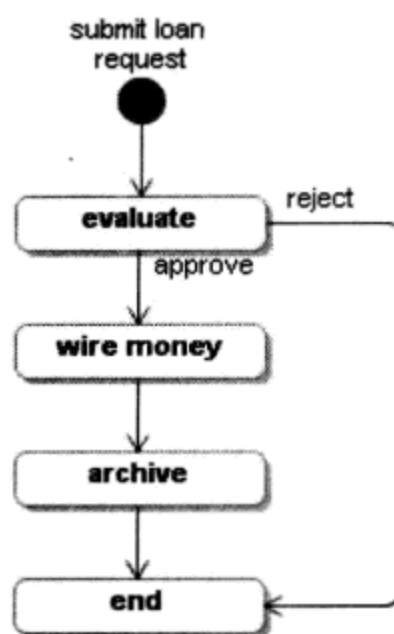


图 5-1 信贷流程定义示例

一个流程实例在其生命周期中，最典型的特征就是具有指向当前执行活动的指针——“执行”。如图 5-2 所示的流程实例正执行到“archive”活动。

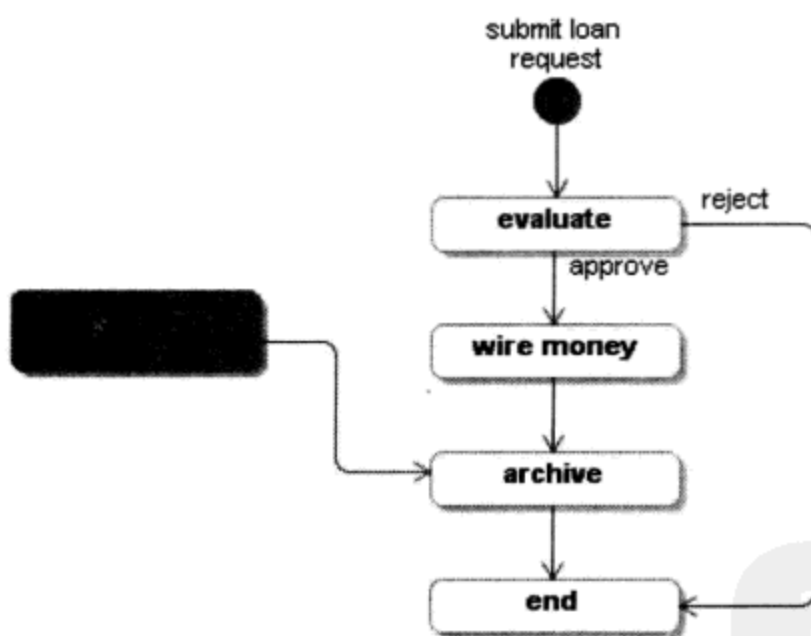


图 5-2 信贷流程实例的执行示例

提示：“执行”这个概念在 jBPM3 中被称为“token”，到了 jBPM4 中则变成了“executions”。这个变化也会在本书关于 jBPM3 迁移到 jBPM4 的章节中详细介绍。

因为流程实例支持“并行”执行，所以在同一个流程实例中的执行数量并非绝对

唯一。修改一下上面的流程定义，汇款“wire money”和存档“archive”被定义为并行执行，那么主流程实例就包含了两个用来跟踪状态的子执行了，如图 5-3 所示。

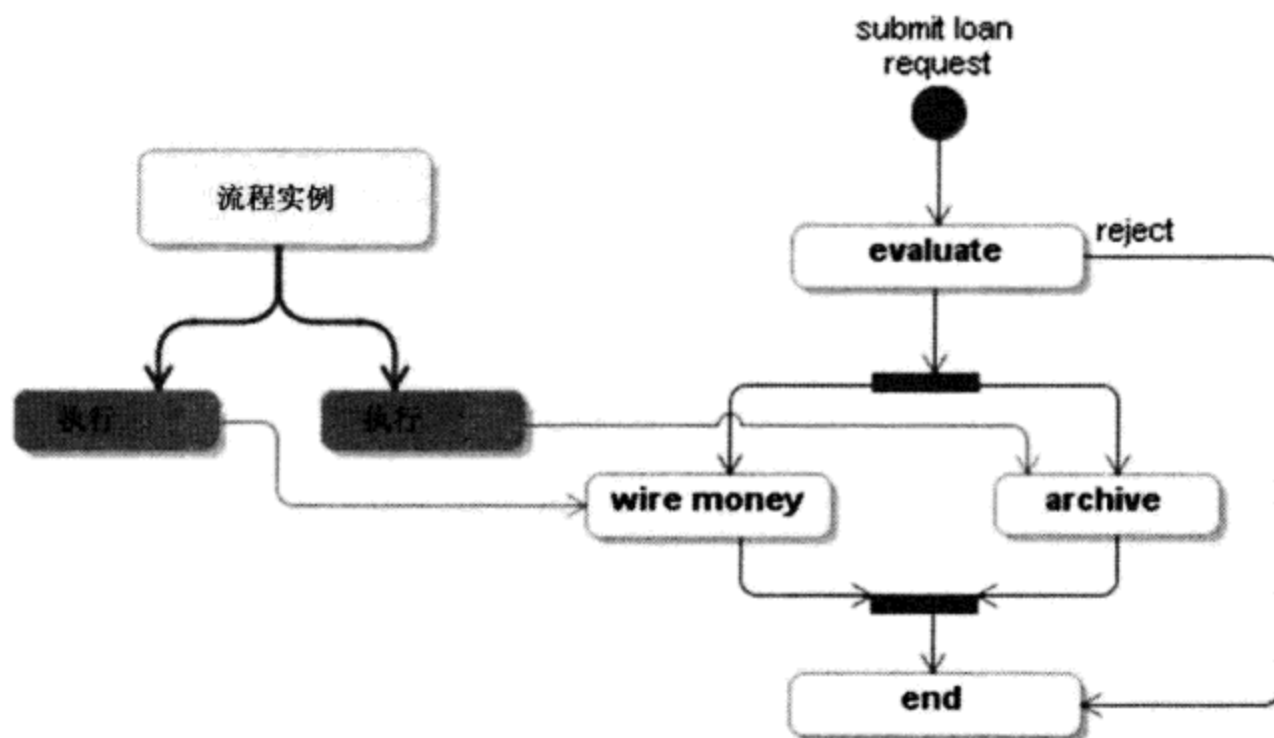


图 5-3 信贷流程实例并行执行的示例

一般情况下，一个流程实例可以理解为一棵“执行树”。当一个流程实例启动时，最初的执行处于这棵执行树的“根”节点位置，之后可以根据定义的需要产生子执行，即“树枝”。

jBPM 使用树状执行结构的原因在于：这一概念实际上只有一条执行路径，所以子执行终将归于（join）根执行。这样，流程执行模型的实现和使用就更简单、更易理解了，不是吗？

业务 API 不需要了解流程实例和执行之间的功能区别，jBPM Service API 里只需要一个执行类型来引用流程实例。

5.2 流程引擎 API

流程引擎对象——org.jbpm.api.ProcessEngine 是 jBPM4 所有 Service API 之源。

在 jBPM4 中各种服务相互依存。但所有的 Service API（服务接口）都从 ProcessEngine 中获得。ProcessEngine 是由 Configuration 类构建的，即 workflow 引擎根据

配置产生。

`ProcessEngine` 是线程安全的，因此它可以保存在静态变量中，甚至 JNDI 命名服务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 `ProcessEngine` 对象，以下代码告诉您如何获得 `ProcessEngine`：

```
ProcessEngine processEngine = Configuration.getProcessEngine();
```

提示：本节中涉及的代码，可以从 jBPM4.3 发布包中的 `org.jbpm.examples.services.ServicesTest` 示例中获得。

上面代码中的 `Configuration` 使用了 `classpath` 根目录下的默认配置文件 `jbpm.cfg.xml` 创建一个 `ProcessEngine`。如果您要指定其他位置的 jBPM 配置文件，请使用 `Configuration.setResource` 方法：

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

当然，还有其他 `setXxxx()` 方法可以获得 jBPM 配置内容，例如从 `InputStream` 中、从 XML 字符串中、从 `InputSource` 中、从 URL 中、从文件中等，这里就不再示范了。

我们可以根据 `ProcessEngine` 得到如下的服务——在 jBPM4 中这些服务对外统一封装为 6 个 Service API，可通过如下代码分别获取：

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

这 6 个 Service API 也可以根据流程引擎对象提供的 Java 类接口方法 `processEngine.get(Class<T>)` 或者根据名称接口方法 `processEngine.get(String)` 来获取。

这 6 个 Service API 都位于 `org.jbpm.api` 包中：

- **RepositoryService**——流程资源服务的接口。提供对流程定义的部署、查询、删除等操作。
- **ExecutionService**——流程执行服务的接口。提供启动流程实例、“执行”推进、设置流程变量等操作。

- **ManagementService**——流程管理控制服务的接口。在 jBPM4.3 中只提供异步工作 (Job) 相关的执行和查询操作。
- **TaskService**——人工任务服务的接口。提供对任务 (Task) 的创建、提交、查询、保存、删除等操作。
- **HistoryService**——流程历史服务的接口。提供对流程历史库 (即已完成的流程实例归档) 中历史流程实例、历史活动实例等记录的查询操作。还提供诸如某个流程定义中所有活动的平均持续时间、某个流程定义中某转移的经过次数等数据分析服务。
- **IdentityService**——身份认证服务的接口。提供对流程用户、用户组以及组成员关系的相关服务。

另外, 阅读 jBPM4 的源代码时可能会发现 **CommandService**, 这是一个比较特殊的 Service API, 它是怎么使用的呢? 拿上面 6 个 Service API 之一的 **ManagementService** 的实现 **ManagementServiceImpl** 来说, 您可以发现 **ManagementServiceImpl** 继承了 **AbstractServiceImpl** 类, 事实上 6 个 Service API 都继承了 **AbstractServiceImpl** 类:

```
public class ManagementServiceImpl extends AbstractServiceImpl
implements ManagementService
```

而 **AbstractServiceImpl** 依赖 **CommandService**, 如下源码:

```
public class AbstractServiceImpl
{
    ...
    public CommandService getCommandService()
    {
        return commandService;
    }
    public void setCommandService(CommandService commandService)
    {
        this.commandService = commandService;
    }
    protected CommandService commandService;
}
```

看到此, 相信聪明的您已经有所明白, 所谓 **CommandService** 实际上就是“Command 模式”的服务接口, 就是将客户端的请求全部封装在一个调用接口中, 然后由这个接口去调用 `org.jbpm.api.cmd.Command` 接口的众多实现, 例如 `StartExecutionCmd`, `SignalCmd`, `SetVariablesCmd`, `GetTimersCmd`, `DeployCmd`, `NewTaskCmd`, `SubmitTask`,

ExecuteJobCmd……具体可参见 org.jbpm.pvm.internal.cmd 包下 org.jbpm.api.cmd.Command 接口的实现类。这是典型的 Command 设计模式的应用。

提示：jBPM4 Service API 的实现广泛地采用了 Command 设计模式，何谓 Command 模式？jBPM4 为什么要采用 Command 模式？

抽象出待执行的动作以参数化某对象，您可用面向过程语言中的回调（callback）函数表达这种参数化机制。所谓回调函数是指函数先在某处注册，而它将在稍后某个需要的时候被调用。可以说 Command 模式是回调机制的一个面向对象的替代品。

Command 模式的目的是在不同的时刻指定、排列和执行请求。一个 Command 对象可以有一个与初始请求无关的生存期。如果一个请求的接收者可用一种与地址空间无关的方式表达，那么就可将负责该请求的命令对象传送给另一个不同的进程并在那儿实现该请求。

Command 模式的优势在于：

支持取消操作。Command 的 Execute 操作可在实施操作前将状态存储起来，在取消操作时这个状态用来消除该操作的影响。Command 接口必须添加一个 Unexecute 操作，该操作取消上一次 Execute 调用的效果。执行的命令被存储在一个历史列表中。可通过向后和向前遍历这一列表并分别调用 Unexecute 和 Execute 来实现次数不限的“取消”和“重做”。

支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。在 Command 接口中添加装载操作和存储操作，可以用来动态保持一个一致的修改日志。从崩溃中恢复的过程包括从磁盘中重新读入记录下来的命令并用 Execute 操作重新执行它们。

用构建在原语操作上的高层操作构造一个系统。这样一种结构在支持事务（transaction）的信息系统中很常见。一个事务封装了对数据的一组变动。Command 模式提供了对事务进行建模的方法。Command 有一个公共的接口，使得用户可以用同一种方式调用所有的事务。同时使用该模式也易于添加新事务以扩展系统。

在第 13 章 jBPM4 的设计思想中您可以详细地了解 jBPM4 的 Command 设计模式应用。

5.3 利用 API 部署流程

RepositoryService 提供了管理 jBPM4 发布资源的所有接口。以下代码使用

RepositoryService 部署 classpath 中的一个流程定义资源：

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

通过类似上面的 addResourceFromXxxx 系列方法，流程定义 XML 的内容可以从文件、Web URL、字符串、输入流或 zip 流中获取。

每次部署的资源的内容都是字节数组的形式。jPDL 流程定义文件以扩展名 .jpdl.xml 被识别。其他资源包括任务表单、Java 类、脚本等。如果不仅要部署 .jpdl.xml 流程定义文件，而且要部署一系列流程定义资源，则可以以流程定义归档的方式部署，流程引擎会自动识别归档中扩展名为.jpdl.xml 的文件为流程定义文件。

在部署过程中，流程引擎会把一个 ID 分配给流程定义。这个 ID 的格式为 {key}-{version}，即流程键和流程版本之间通过连字符连接。

如果流程定义没有指定 key（键），key 则会在流程名称的基础上自动生成。生成的 key 会把所有不是字母和数字的字符替换成下划线，例如空格。

同一个流程名称只能关联到一个 key，反之亦然。

如果没有为流程定义文件指定版本号，流程引擎也会自动为之分配一个版本号。部署 key 已存在的流程定义，其版本号自动递增。新部署流程定义的版本号会自动分配为 1。

例如在下面这个流程定义里，只提供了流程的名字，没有提供其他信息：

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

假设这个流程定义是第一次部署，表 5-1 就是它部署后的属性。

表 5-1 没有指定 key 值属性的流程定义

属性名称	属性值	来源
name	Insurance claim	流程定义文件
key	Insurance_claim	系统生成
version	1	系统生成
id	Insurance_claim-1	系统生成

在这个例子中我们通过指定流程定义的 key 来获得更短的 id:

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

同样假设这个流程定义是第一次部署，它的属性如表 5-2 所示。

表 5-2 指定 key 值属性的流程定义

属性名称	属性值	来源
name	Insurance claim	流程定义文件
key	ICL	流程定义文件
version	1	系统生成
id	ICL-1	系统生成

5.4 通过 API 删除已部署的流程

可以通过 RepositoryService 提供的 API 删除一个已经部署的流程定义，请注意这会是物理删除，即在数据库中彻底销毁这条流程定义的记录：

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

但是如果删除的流程定义还存在未完成流程实例的话，执行 deleteDeployment 方法就会抛出异常。

因此，如果希望级联删除一个已经发布的流程定义及其所有产生的流程实例的话，可以使用 RepositoryService 的 deleteDeploymentCascade 方法——“连根拔起”。

5.5 使用 API 发起新的流程实例

本节将介绍如何利用 ExecutionService 提供的 API，根据已部署的流程定义，通过多种不同的途径去发起新的流程实例。

5.5.1 发起流程实例的常规方法

下面是为流程定义“ICL”发起一个新流程实例的最简方法：

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

上面的 `startProcessInstanceByKey` 方法会去查找 key 为 ICL 的最新版本流程定义，然后根据最新版本的 ICL 流程定义启动流程实例。当流程定义部署了一个新版本后，`startProcessInstanceByKey` 方法会自动切换到最新部署的流程定义版本。

如果想根据特定的流程定义版本发起流程实例，则可以使用流程定义的 ID 启动流程实例。通过如下 API 调用：

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

5.5.2 指定业务键发起流程实例

上面我们使用 `ExecutionService` 的 `startProcessInstanceByKey` 方法根据一个流程定义键（`processDefinitionKey`）去发起流程实例。但是这个流程定义键对特定的“业务实例”来说是没有任何用处的。如果我们需要将每个流程实例和一个独特的业务实例关联起来，例如一个保险流程的实例必然需要和一个保险单号关联，以便将来业务上的查询和索引，那么我们可以通过为新启动的流程实例分配一个业务键（`processInstanceKey`）来做到。

这个业务键是用户执行流程的时候根据业务定义的。一个业务键必须在此流程定义所有版本的流程实例范围内是唯一的。不要担心在业务流程领域找不到这种业务键，想想吧：一个订单号、一个保险单号、一笔交易流水号……

使用如下 API 指定业务键发起流程实例：

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

在上面的例子中，CL92837 即是业务键，它也许是笔交易的流水号或是什么，总之只和特定的业务有关系。

业务键可以被用来创建流程实例的 ID，格式为 {`processDefinitionKey`}. {`processInstanceKey`}。所以上面的代码会创建一个 ID 为“ICL.CL92837”的流程实例。

如果没有提供用户定义的业务键，数据库就会把流程定义主键作为 Key。这样可以使用如下代码发起流程实例并获取流程实例 ID：

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

提示：最好指定一个业务键发起流程实例。一般情况下，在业务应用中，找到这样的业务键并不困难。提供一个业务键的好处是可以根据业务来执行流程实例搜索（这也是最常见的需求），而不是根据流程变量来执行流程实例的搜索，后者十分消耗执行效率。

这是一个最佳实践。

5.5.3 指定变量发起流程实例

如果新的流程实例需要一些输入参数启动。那么，可以将这些参数放在流程变量里，然后在发起流程时传入流程变量对象。流程变量对象一般是一个 Map<String,Object>。代码如下：

```
//创建并填充流程变量 Map
Map<String,Object> variables = new HashMap<String,Object>();
variables.put("customerName", "Alex Miller");
variables.put("type", "Accident");
variables.put("amount", new Float(763.74));
//传入 Map，带着流程变量发起流程实例
ProcessInstance processInstance = executionService.startProcessInstanceByKey("ICL", variables);
```

5.6 唤醒一个等待状态的执行

当流程执行进入一个 state 活动时，执行（也可以说流程实例）会在到达 state 活动的时候进入等待状态——Wait State，这是 jBPM 的一个重要概念，task 等活动也会陷入等待状态，直到一个 signal（可理解为一个“外部触发信号”）出现，才能进入下一个步骤的活动。ExecutionService 的 signalExecution*方法可以用来发出 signal——这个方法需要传入“执行”对象。

在多数情况下，到达 state 活动的执行会是流程实例本身。但在定时器异步和并发的情况下，流程实例会停留在根执行上。所以必须先确认您的 signal 作用在了正确的

流程执行上。

获得正确的执行比较好的实践是给 state 活动分配一个事件监听器，定义如下：

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

在事件监听器类 **StartExternalWork** 中，可以执行那些需要在此 state 活动中完成的工作。在这个事件监听器里，也可以通过 `execution.getId` 方法获得正确的执行 ID。有了这个执行 ID，在 state 活动的工作完成之后，可以用它来发出 signal，离开该活动，如以下代码所示：

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

这里还有一个不是太推荐的方式来获得执行 ID。当流程执行到达 state 活动的时候，如果您知道该活动名称的话，则可以这么做：

```
//以下代码假设我们知道当前活动的名称
//发起或获取流程实例
ProcessInstance processInstance =
executionService.startProcessInstanceById (processDefinitionId);
//或
//ProcessInstance processInstance =
    executionService.signalProcessInstanceById(executionId);
//如上述假设，我们知道当前流程实例在名为“external work”的活动中等待
Execution execution = processInstance.findActiveExecutionIn("external work");
//获取执行 ID
String executionId = execution.getId();
```

注意：因为上面这种解决方式使得流程客户端实现和业务逻辑绑定得比较紧，如果不是特殊的业务场景需要，不建议采用。

5.7 任务服务 API

TaskService 的主要目的是提供对任务列表的访问操作，这里的任务是指 jBPM4 task 活动产生的人机交互业务。

下面的代码展示了如何获得 ID 为 alexmiller 的用户的任务列表：

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

一般来说，任务会有一个表单，显示在一些用户界面接口中（例如 JSP）。这个表单需要读/写与任务相关的流程数据，一般是通过任务变量的方式，如下代码所示：

```
long taskId = task.getId();
Set<String> variableNames = taskService.getVariableNames(taskId);
//读取任务变量
HashMap<String, Object> variables = taskService.getVariables(taskId,
variableNames);
//或自行创建 variables = new HashMap<String, Object>();
//设置“键-值”形式的任务变量
variables.put("category", "small");
variables.put("lires", 923874893);
//将变量存入任务
taskService.setVariables(taskId, variables);
```

TaskService 也用来完成任务，通过以下 4 种方式：

```
//根据指定的任务 ID 完成任务
taskService.completeTask(taskId);
//根据指定的任务 ID，同时设入变量，完成任务
taskService.completeTask(taskId, variables);
//指定 outcome，即下一步的转移路径，完成任务
taskService.completeTask(taskId, outcome);
//指定下一步的转移路径，同时设入变量，完成任务
taskService.completeTask(taskId, outcome, variables);
```

我们可以看出这些 API 允许设入变量 Map。这些变量在任务完成前将作为流程变量同步到流程实例中。

这些 API 还提供一个参数“outcome”，这个参数可以用来决定任务完成后流程流向哪个流出“转移”。完成任务后，流程将“何去何从”遵循如下的规则。

- 1) 如果任务拥有一个没有名称的流出转移：
 - a) taskService.getOutcomes(taskId) 返回包含一个 null 值的集合。
 - b) taskService.completeTask(taskId) 会经过这个流出转移。
 - c) taskService.completeTask(taskId, null) 会经过这个流出转移。
 - d) taskService.completeTask(taskId, "anyvalue") 会抛出一个异常。
- 2) 如果任务拥有一个已命名为 myName 的流出转移：

- a) `taskService.getOutcomes(taskId)` 返回包含这个流出转移的名称集合。
 - b) `taskService.completeTask(taskId)` 会经过这个流出转移。
 - c) `taskService.completeTask(taskId, null)` 会抛出一个异常。这是因为此任务没有无名称的流出转移。
 - d) `taskService.completeTask(taskId, "anyvalue")` 会抛出一个异常。
 - e) `taskService.completeTask(taskId, "myName")` 会经过这个流出转移。
- 3) 如果任务拥有多个流出转移，而其中一个没有名称，其他都有名称：
- a) `taskService.getOutcomes(taskId)` 返回包含一个 `null` 值和其他流出转移名称的集合。
 - b) `taskService.completeTask(taskId)` 会经过没有名称的流出转移。
 - c) `taskService.completeTask(taskId, null)` 会经过没有名称的流出转移。
 - d) `taskService.completeTask(taskId, "anyvalue")` 会抛出一个异常。
 - e) `taskService.completeTask(taskId, "myName")` 会经过名称为 `myName` 的流出转移（我们假设 `myName` 存在有名称的流出转移中）。
- 4) 如果任务拥有多个流出转移，且每个流出转移都拥有唯一的名称：
- a) `taskService.getOutcomes(taskId)` 返回包含所有流出转移名称的集合。
 - b) `taskService.completeTask(taskId)` 会抛出一个异常。因为没有无名称的流出转移。
 - c) `taskService.completeTask(taskId, null)` 会抛出一个异常。因为没有无名称的流出转移。
 - d) `taskService.completeTask(taskId, "anyvalue")` 会抛出一个异常。
 - e) `taskService.completeTask(taskId, "myName")` 会经过名称为 `myName` 的流出转移（我们假设 `myName` 存在流出转移中）。

任务可以拥有多个候选人，候选人可以是单个用户也可以是用户组。用户可以接收候选人是自己的任务，接收任务的意思是用户会被流程引擎设置为任务的分配者，接收任务是个“排他”操作，因此在任务被“接收 - 分配”之后，其他的用户就不能接收并办理此任务了。

一般情况下，除非用户被分配到这个任务上，否则不能办理此任务。当然，如果实现了“代理人”机制后除外，关于代理人会在中国特色工作流的实现中提到。

用户接收任务后，一般需要客户端应用程序界面显示任务表单，并引导用户完成任务。对于有候选人，但是还没有被分配的任务，唯一应该暴露给用户的操作就是“接

收任务”。

关于“任务”更多的说明请参见 6.2.6 task (人工任务活动)。

5.8 历史服务 API

在流程实例执行的过程中，会不断触发事件，通过这些事件，已完成流程实例的历史信息会被收集到流程历史数据表中。而 HistoryService API 提供了对这些历史信息的访问服务。

如果想查找某一特定流程定义的所有历史流程实例，可以通过如下 API 调用：

```
List<HistoryProcessInstance> historyProcessInstances = historyService
    .createHistoryProcessInstanceQuery()
    //查询 ID 为“ICL-1”的流程定义
    .processDefinitionId("ICL-1")
    //返回的结果集按开始时间正序排列
    .orderAsc(HistoryProcessInstanceQuery.PROPERTY_STARTTIME)
    .list();
```

历史的活动实例被作为 HistoryActivityInstance 保存到历史活动数据表中，通过如下 API 查询：

```
List<HistoryActivityInstance> histActInsts = historyService
    .createHistoryActivityInstanceQuery()
    //查询 ID 为“ICL-1”的流程定义
    .processDefinitionId("ICL-1")
    //名称为“a”的活动实例
    .activityName("a")
    .list();
```

查询历史任务等服务的调用示例在此不再列出。同时，HistoryService 也提供如下方法来支持对流程历史进行分析的需求：

- avgDurationPerActivity——获取指定流程定义中每个活动的平均执行时间。
- choiceDistribution——获取指定活动定义每个转移的经过次数。

建议通过 jBPM4 的 javadocs 了解关于 HistoryService API 的更多信息。

5.9 管理服务 API

ManagementService 即管理服务,通常用来管理 Job(异步工作)。ManagementService 的功能在诸如 jBPM4 Web 控制台等客户端应用上被调用。

ManagementService 的接口定义很简单,仅包括如下两个方法:

```
//执行指定 ID 的 Job
void executeJob(String jobId);
//获取 Job 查询接口
JobQuery createJobQuery();
```

第一个 API 很简单,就是执行 Job;第二个 API 负责无条件生成 JobQuery 接口对象,提供 Job 的查询接口。JobQuery 接口的功能还是比较“丰富”的,它提供的方法如下面的源代码所示:

```
public interface JobQuery {
    ...
    /** 查询所有的消息型 Job */
    JobQuery messages();
    /** 查询所有的定时器 Job */
    JobQuery timers();
    /** 查询属于指定流程实例的 Job */
    JobQuery processInstanceId(String processInstanceId);
    /** 查询由于异常回滚而产生的 Job */
    JobQuery exception(boolean hasException);
    /** 将查询结果根据指定属性正序排列 */
    JobQuery orderAsc(String property);
    /** 将查询结果根据指定属性逆序排列 */
    JobQuery orderDesc(String property);
    /** 用于支持查询结果分页 */
    JobQuery page(int firstResult, int maxResults);
    /** 执行查询,返回 Job 列表 */
    List<Job> list();
    /** 执行查询,返回单个 Job */
    Job uniqueResult();
    /** 执行查询,返回结果集数量。相当于执行 SQL 的 “count(*)” 操作 */
    long count();
}
```

5.10 查询服务 API

从 jBPM4 开始，流程查询系统由一组新的 API 来支持，这组 API 可以覆盖到大多数您能想到的查询。当然，如果您需要编写基于特定业务的查询，也可以直接使用 Hibenrate 的 HSQL 查询数据库，但是对于大多数场景来说，查询服务 API 是够用的。

查询服务 API 一般是基于在主要的 jBPM 概念实体上创建查询对象来实现的，这些概念实体包括流程实例、任务、流程历史……

例如下面这段对于流程实例的查询：

```
List<ProcessInstance> results = executionService
    //获取流程实例查询对象
    .createProcessInstanceQuery()
    //指定流程定义 ID
    .processDefinitionId("my_process_definition")
    //设定“未挂起”为过滤条件
    .notSuspended()
    //分页
    .page(0, 50)
    //查询执行，获得结果集列表
    .list();
```

上面这个例子会返回指定流程定义的所有未挂起的流程实例，结果集支持分页，获取前 50 条记录。

对于任务的查询也可以使用类似的查询对象实现：

```
List<Task> myTasks = taskService
    //获取任务查询对象
    .createTaskQuery()
    //指定流程实例 ID
    .processInstanceId(piId)
    //分配给 Alex 的任务
    .assignee("Alex")
    //分页
    .page(100, 120)
    //根据日期逆向排序
    .orderDesc(TaskQuery.PROPERTY_DUEDATE)
    //查询执行，获得结果集列表
    .list();
```

上面这个查询的目的是根据指定流程实例，获取分配给 Alex 的所有任务，使用分

页获取自第 100 条记录开始的 120 条记录，根据 DUEDATE 属性逆向排序。

几乎每个服务都拥有创建这些统一查询对象的功能，例如，查询 Job 需要通过 ManagementService 创建 JobQuery 对象；查询完成的流程实例需要通过 HistoryService 创建相应的查询对象……

5.11 例程：利用 jBPM Service API 完成流程实例

本例程将演示如何使用 jBPM Service API 基于一个简单的流程定义，发起、执行、完成整个流程实例并查询该流程实例的历史记录。

首先，流程定义如图 5-4 所示。

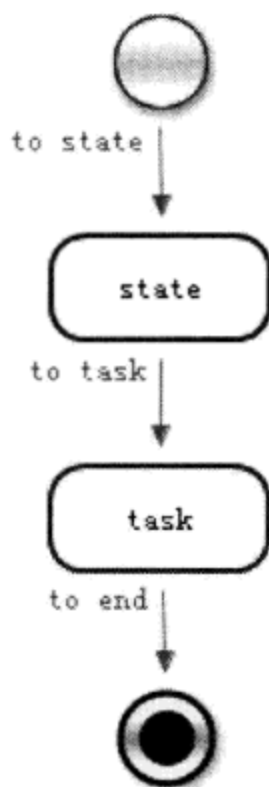


图 5-4 示例流程定义

对应的 jPDL:

```
<process name="process" xmlns="http://jbpm.org/4.3/jpdl">
  <start name="start">
    <transition name="to state" to="state"/>
  </start>
  <!-- state 活动是等待活动。需要收到一个外部的执行信号才能流转通过。 -->
  <state name="state">
    <transition name="to task" to="task"/>
  </state>
</process>
```



```

    </state>
    <!-- task 活动是等待活动。在这里被分配给用户 Alex 办理，Alex 办理完成提交任务后才能流转通过。 -->
    <task assignee="Alex" name="task">
        <transition name="to end" to="end"/>
    </task>
    <end name="end"/>
</process>

```

以下代码是基于 JbpmTestCase 的单元测试，使用 jBPM Service API 发起并完成上面名为 process 的流程定义。单元测试中执行流程定义部署的 setUp 方法和删除流程定义部署的 tearDown 方法将不再列出，后面的所有章节皆同。代码如下：

```

//使用执行服务：根据已部署流程定义的名称 process，发起流程实例
ProcessInstance processInstance = executionService.startProcessInstanceByKey(
    "process");
//获取流程实例 ID
String pid = processInstance.getId();
//获取当前活动的执行对象
Execution executionInState = processInstance.findActiveExecutionIn(
    "state");
//assertXxxx 系列方法是 JbpmTestCase 为单元测试专门提供的断言类方法
//断言当前活动即为 state
assertNotNull(executionInState);
//使用执行服务：发出执行信号结束当前活动，继续流程的执行
executionService.signalExecutionById(executionInState.getId());
//使用执行服务：从持久化层中获取“最新”的流程实例对象
processInstance = executionService.findProcessInstanceById(pid);
//获取当前活动的执行对象
Execution executionInTask = processInstance.findActiveExecutionIn(
    "task");
//断言当前活动即为 task
assertNotNull(executionInTask);
//使用任务服务：获取用户 Alex 的任务，即 task 活动产生的任务
Task task = taskService.findPersonalTasks("Alex").get(0);
//使用任务服务：完成任务
taskService.completeTask(task.getId());
//使用历史服务：创建历史任务查询
HistoryTask historyTask = historyService.createHistoryTaskQuery()
    .taskId(task.getId()).uniqueResult();
//断言上一步完成的任务已成为历史，即可通过历史任务查询获取之
assertNotNull(historyTask);
//断言该流程实例已经结束
assertProcessInstanceEnded(pid);
//使用历史服务：创建历史流程实例查询

```



```
HistoryProcessInstance historyProcInst = historyService
    .createHistoryProcessInstanceQuery().processInstanceId(pid)
    .uniqueResult();
//断言该流程实例已经成为历史, 即可通过历史流程实例查询获取它
assertNotNull(historyProcInst)
//单元测试至此执行完毕
```

5.12 小结

通过本章的介绍, 读者应该能够对 jBPM4 workflow 引擎提供的 Service API 有了初步的了解, 并且能够调用相应的 Service API 完成部署流程定义、发起流程实例、推进流程执行乃至查询流程历史等操作, 即利用 Service API 使一个流程实例走完它的整个生命周期过程。熟悉所有 Service API 的功能及其适用场景是开发 jBPM 客户端应用程序的基础。

在下一章中将详细介绍 jPDL 流程定义语言的各个活动及重要功能元素, 包括它们的特性、使用方法、适用的业务场景, 并提供大量详尽的应用例程。因此下一章是入门和掌握 jBPM 框架的必读内容, 也是本书最重要的章节。



可以这么说, jPDL (jBPM Process Define Language, jBPM 流程定义语言) 是 jBPM4 独有的、最重要的“资产”。jPDL 的设计目标是尽量地精简和尽可能地对开发者友好, 即提供所有您期望从业务流程定义语言中得到的功能的同时, 也可以很“精练”地描述业务流程的定义和图形结构, 最终使得业务分析师和流程开发者能使用“同一种语言说话”、极大地减少了他们之间的交流障碍。同时 jPDL 明晰的语义和良好的扩展性使得无论是人类还是工作流引擎都能很“舒服”地读懂它。

本章将会详细介绍 jPDL 流程定义语言各个活动及重要功能元素, 是全书最重要、最长的部分。本章的内容既是初学者入门的必经之路, 也是有经验开发者快速获取帮助的指南。

jPDL 的 XML Schema 定义文件 (例如位于安装目录中的 src/jpdl-4.3.xsd) 包含了全部的 jPDL 属性和元素。而本章介绍的则是 jPDL (4.3 版本) 中被 jBPM 官方宣布稳定支持的部分。

下面说明一个典型 jPDL 流程定义文件的内容:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 声明流程定义的名称属性, 指定遵循的 XML Schema 名称空间属性 -->
<process name="Purchase order" xmlns="http://jbpm.org/4.3/jpdl">
<!-- 定义“开始”活动 (必须), 并指定下一步活动的“转移”元素 -->
<start>
<transition to="Verify supplier" />
</start>
<!-- 定义一个“状态”活动, 并指定两个“转移”元素 -->
<state name="Verify supplier">
<transition name="Supplier ok" to="Check supplier data" />
<transition name="Supplier not ok" to="Error" />
</state>
<!-- 定义一个“判断”活动, 也指定两个“转移”元素 -->
<decision name="Check supplier data">
<transition name="nok" to="Error" />
<transition name="ok" to="Completed" />
</decision>
<!-- 定义“结束”活动 (必须) -->
```

```

<end name="Completed" />
<!-- 定义一个出错“结束”的活动，这是根据业务的需要来选择定义的 -->
<end name="Error" />
</process>

```

注意：上面这段 jPDL 流程定义并未描述图形元素的坐标。这是被允许的，并不影响流程定义被 workflow 引擎解析和执行。

同时，我们还从上面的典型流程定义中发现，几乎所有的活动都会包含以下属性和元素。这些通用的活动属性如表 6-1 所示，元素如表 6-2 所示。

表 6-1 通用的活动属性

属性	类型	默认值	是否必需	描述
name	字符串	无	必需	活动的名称

表 6-2 通用的活动元素

元素	个数	描述
transition	0..*	定义活动的流出转移

以上的 name 属性和 transition 元素是所有 jBPM4 jPDL 活动都具有的属性或元素，因此，在下面具体的活动介绍和 XML Schema 片段中，除非它们被赋予特殊意义，否则将不再赘述。

以下将逐个介绍 jPDL 中的各个活动，为了方便读者在实践中的使用，对活动及其属性、元素等的称谓都以英文原名表示。

6.1 process (流程)

在 jPDL 中 process (流程) 元素是每个流程定义的顶级元素，即任何流程定义都必须以如下形式开始和结束：

```

<process>
...
</process>

```

process 元素具有如表 6-3 所示的属性。

表 6-3 process 元素的属性

属性	类型	默认值	是否必需	作用描述
name	名称文本	无	必需	在面对最终用户展示和交互时，作为流程显示的标签
key	字母、数字、下画线的组合	如果省略，key 会根据 name 生成，name 中非字母以及非数字的字符会被替换为下画线	可选	用来标识不同的流程定义。拥有同一个 key 的流程定义允许有不同的 version。对于所有已发布的同一流程定义各个版本来说，“key:name”组合必须是完全一样的
version	整型	同一流程定义（即二者的“key:name”相同），后部署的会比先部署的 version 加 1 如果是第一次部署，version 从 1 开始	可选	标识同一流程定义的不同版本

process 元素具有如表 6-4 所示的元素。

表 6-4 process 元素的子元素

元素	个数	描述
description	0..1	描述流程的文案
activities	1..*	活动。流程定义中会有很多不同的活动，例如 start, state, decision 等，但至少要有 1 个是 start 活动

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 process 元素的定义片段，已标出支持的元素和属性供参考。

```
<element name="process">
  <annotation>...</annotation>    <complexType>
```

```

<!-- 以下是 process 元素的子元素定义。 -->
<sequence minOccurs="0" maxOccurs="unbounded">
  <element name="description" minOccurs="0" type="string" />
  <element ref="tns:swimlane" minOccurs="0" maxOccurs="unbounded" />
  <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
  <element ref="tns:timer" minOccurs="0" maxOccurs="unbounded"/>
  <group ref="tns:activityGroup" minOccurs="0" maxOccurs="unbounded" />
  <element ref="tns:migrate-instances" minOccurs="0" maxOccurs="1" />
</sequence>

<!-- 以下是 process 元素的属性定义。 -->
<attribute name="name" use="required" type="string">
  <annotation>...</annotation>
</attribute>
<attribute name="key" type="string">
  <annotation>...</annotation>
</attribute>
<attribute name="version" type="int">
  <annotation>...</annotation>
</attribute>
<anyAttribute processContents="skip">
  <annotation>...</annotation>
</anyAttribute>

</complexType>
</element>

```

其中，activityGroup 元素代表所有类型的活动，skip 属性是为扩展所保留的。

6.2 流转控制活动

以下介绍的是组成一个业务流程定义最基本、最常用的活动。利用这些活动，您完全可以组装出一条完整的流程定义，实现各种基本的流程流转控制场景，例如并行、串行、分支-聚合、条件判断流转等。这些活动有：

- start——开始活动。
- state——状态活动。
- decision——判断活动。
- fork – join——分支/聚合活动。
- end——结束活动。

- task——人工任务活动。
- sub-process——子流程活动。
- custom——自定义活动。

6.2.1 start (开始活动)

start 活动的意义在于指明一个流程的实例应该从哪里开始发起，即流程的入口。在一个流程定义里必须拥有一个 start 活动。start 活动必须有一个流出转移(transition)，这个转移会在流程通过 start 活动的时候执行。

注意：一个流程定义有且只能有一个 start 活动（在 group 中的 start 活动除外）。

表 6-5 start 活动的属性

属性	类型	默认值	是否必需	描述
name	文本	无	可选	start 活动的名称，在无流入转移指向 start 活动的情况下，name 属性可以不指定

表 6-6 start 活动的元素

元素	个数	描述
transition	1	流出转移，指向流程的下一个步骤

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 start 元素的定义片段，已标出支持的元素和属性供参考：

```
<element name="start">
  <annotation>...</annotation>
  <complexType>

    <!-- 以下是 start 活动的元素定义。 -->
    <sequence>
      <element name="description" minOccurs="0" type="string" />
      <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="tns:transition" minOccurs="0" maxOccurs="unbounded" />
    </sequence>
  </complexType>
</element>
```

```

        <!-- 以下是 start 活动的属性定义。 -->
        <!-- activityAttributes 集中定义了一些普通活动的公用属性，例如名称 (name)
等，在这里被引用。后面将不再赘述。 -->
        <attributeGroup ref="tns:activityAttributes" />
        <attribute name="form" type="string">
            <annotation>...</annotation>
        </attribute>

    </complexType>
</element>

```

6.2.2 state (状态活动)

当需要使业务流程受到某些特定的外部干预后再继续运行，而在这之前流程“陷入”一个中断等待的状态时，您需要的就是这么一个 state 活动。当流程运行到 state 活动时，会自动陷入等待状态 (waiting state)，也就是说流程引擎在收到外部触发信号之前会一直使流程实例在此 state 活动等待。从这个方面来说，task 活动可以说是一种特殊化的 state 活动，这在后面关于 task 活动的章节会谈到。

state 活动除了最基本的 name 属性和 transition 等元素外，没有独特的属性或元素。

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 state 元素的定义片段，已标出支持的元素和属性供参考：

```

<element name="state">
    <annotation>...</annotation>
    <complexType>

        <!-- 以下是 state 活动的元素定义。 -->
        <sequence>
            <element name="description" minOccurs="0" type="string" />
            <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
            <element name="transition" minOccurs="0" maxOccurs="unbounded">
                <complexType>
                    <complexContent>
                        <extension base="tns:transitionType">
                            <sequence>
                                <element ref="tns:timer" minOccurs="0" />
                            </sequence>
                        </extension>
                    </complexContent>
                </complexType>
            </element>

```

```

</sequence>

<!-- 以下是 state 活动的属性定义。 -->
<attributeGroup ref="tns:activityAttributes" />

</complexType>
</element>

```

以下是 state 活动最基本的用法——串行序列 state 的转移, 流程示例如图 6-1 所示。

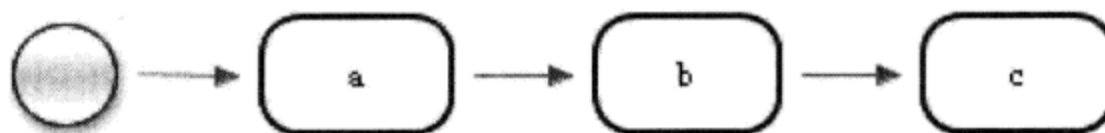


图 6-1 序列 state 转移

对应的 jPDL 如下:

```

<process name="StateSequence" xmlns="http://jbpm.org/4.3/jpdl">
  <!-- 流程开始后转移到 a state -->
  <start>
    <transition to="a"/>
  </start>
  <!-- a state 转移到 b state -->
  <state name="a">
    <transition to="b"/>
  </state>
  <!-- b state 转移到 c state -->
  <state name="b">
    <transition to="c"/>
  </state>
  <!-- 最终停留在 c state -->
  <state name="c"/>
</process>

```

下面我们编写代码使此流程运行完成。首先根据此流程定义发起实例:

```

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

```

在没有任何外部触发的情况下, 此流程实例会在 a state 一直等待, 调用 ExecutionService 的 signalExecution 方法则会发出一个外部信号, 触发流程走向下一步:

```

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

```

```

//断言流程实例在 a state 等待
assertNotNull(executionInA);
//发出触发执行的外部信号
processInstance = executionService.signalExecutionById(executionInA.getId());
Execution executionInB = processInstance.findActiveExecutionIn("b");
//断言流程实例走向了下一步 b state
assertNotNull(executionInB);
//继续触发
processInstance
executionService.signalExecutionById(executionInB.getId());
Execution executionInC = processInstance.findActiveExecutionIn("c");
//可以断言流程实例到达了 c state
assertNotNull(executionInC);

```

在 state 活动里可以定义多个 transition 元素。我们通过信号触发指定转移路径的名称，就可以选择其中的一个 transition 通过。如图 6-2 所示，“wait for response” 活动有两个 transition。

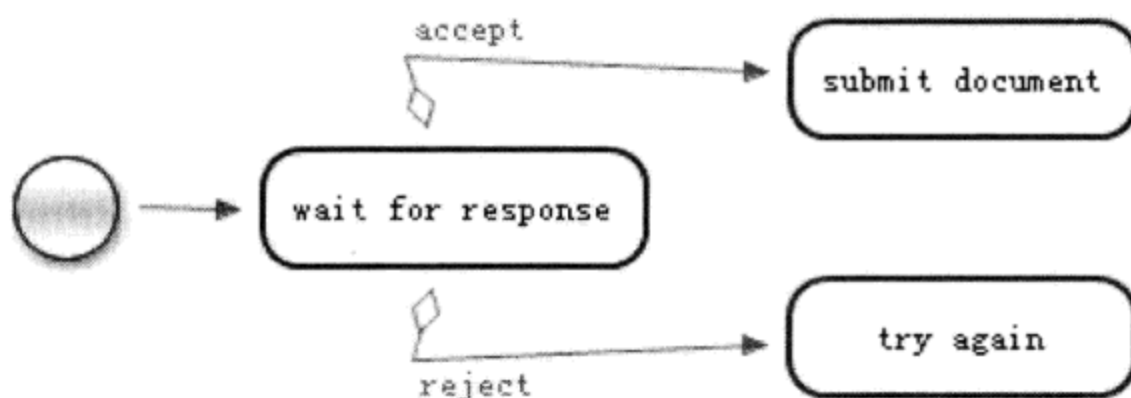


图 6-2 选择 state 转移

此流程对应的 jPDL 如下：

```

<process name="StateChoice" xmlns="http://jbpm.org/4.3/jpdl">
  <!-- 流程开始后转移到 wait for response state -->
  <start>
    <transition to="wait for response" />
  </start>
  <!-- 这里有两个 transition 供选择，二者选其一 -->
  <state name="wait for response">
    <transition name="accept" to="submit document" />
    <transition name="reject" to="try again" />
  </state>
  <!-- 此活动被上面名为 accept 的 transition 指向 -->
  <state name="submit document" />

```



```

    <!-- 此活动被上面名为 reject 的 transition 指向 -->
    <state name="try again" />
</process>

```

下面我们编写代码使此流程运行起来。首先根据此流程定义发起实例：

```

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

```

假设现在流程实例到达了 `wait for response state` 活动，则实例会一直等待外部触发的出现。此 `state` 活动拥有两个流出转移，外部触发调用通过提供不同的流出转移信号名称（Signal Name）来实现选择流转，下面使用 `accept` 作为信号名称：

```

//获取流程实例的 ID
String executionId = processInstance.findActiveExecutionIn("wait for
response").getId();
//触发 accept 信号
processInstance = executionService.signalExecutionById(executionId,
    "accept");
//断言流程实例流向了预期的活动
assertTrue(processInstance.isActive("submit document"));

```

可以预期，当使用 `reject` 作为信号名称参数执行 `signalExecutionXxxx` 方法时，流程会流向 `try again state` 活动。

6.2.3 decision（判断活动）

根据条件在多个流转路径中选择其一通过，也就是做一个决定性的判断，这时候使用 `decision` 活动是个正确的选择。

`decision` 活动可以拥有多个流出转移，当流程实例到达 `decision` 活动时，会根据最先匹配成功的一个条件自动地通过相应的流出转移。

以下是 XML Schema 文件 `jpd1-4.3.xsd` 中对于 `decision` 元素的定义片段，已标出支持的元素和属性供参考：

```

<element name="decision">
    <annotation>...</annotation>
    <complexType>

        <!-- 以下是 decision 活动的元素定义。 -->
        <sequence>
            <element name="description" minOccurs="0" type="string" />

```



```

<element name="handler" minOccurs="0" type="tns:wireObjectType" />
<element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
<element name="transition" minOccurs="0" maxOccurs="unbounded">
  <complexType>
    <complexContent>
      <extension base="tns:transitionType">
        <sequence>
          <element name="condition" minOccurs="0" maxOccurs="
"unbounded">
            <complexType>
              <sequence>
                <element name="handler" minOccurs="0" type=
"tns:wireObjectType" />
              </sequence>
              <attribute name="expr" type="string">
                <annotation>...</annotation>
              </attribute>
              <attribute name="lang" type="string">
                <annotation>...</annotation>
              </attribute>
            </complexType>
          </element>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</element>
</sequence>

<!-- 以下是 decision 活动的属性定义。 -->
<attributeGroup ref="tns:activityAttributes" />
<attribute name="expr" type="string">
  <annotation>...</annotation>
</attribute>
<attribute name="lang" type="string">
  <annotation>...</annotation>
</attribute>

</complexType>
</element>

```

使用 decision 活动判断流向哪个转移，可以选择如下 3 种方式实现。

1. 使用 decision 活动的 condition 元素

decision 活动中会运行并判断其每一个 transition 元素里的流转条件——流转条件由 condition 元素表示。当遇到一个 transition 的 condition 值为 true 或者一个没有设置 condition 的 transition，那么流程就立刻流向这个 transition。

表 6-7 是 condition 元素支持的属性。

表 6-7 decision 活动的 condition 元素属性

属性	类型	默认值	是否必需	描述
expr	表达式	无	必需	描述转移条件的表达式
lang	表达式语言名称	脚本引擎配置文件 (jbpm.default.scriptmanager.xml) 里定义的默认表达式语言名称, 一般为 jUEL, 即 EL 表达式语言	可选	指定转移条件表达式的语言类型

如图 6-3 所示是一个应用 decision condition 方式判断转移路径的流程定义：

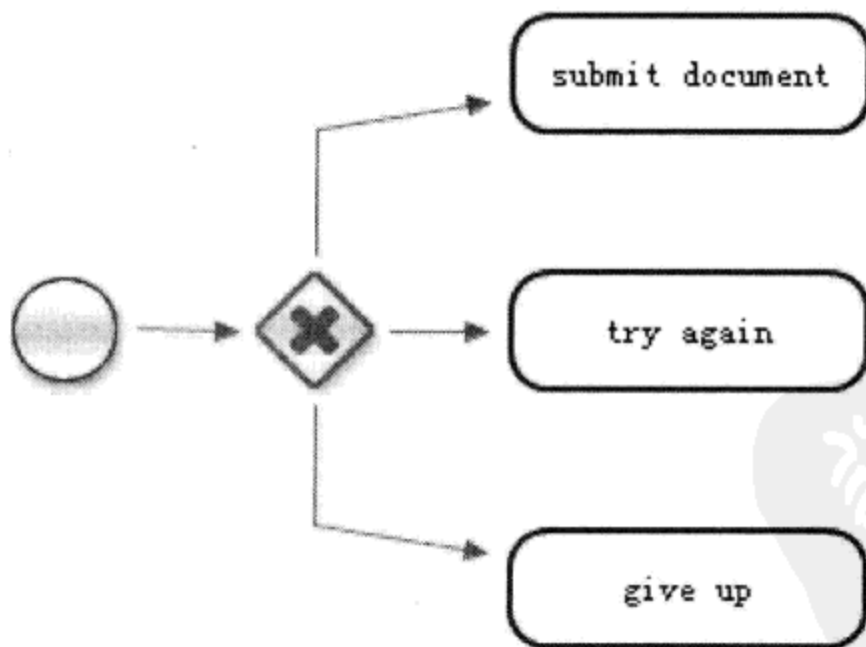


图 6-3 应用 decision condition 方式判断转移路径的流程定义

对应的 jPDL:

```
<process name="DecisionConditions" xmlns="http://jbpm.org/4.3/jpdl">
```

```

<start>
  <transition to="evaluate document" />
</start>
<decision name="evaluate document">
  <!-- 以下是两个流转条件表达式: -->
  <transition to="submit document">
    <!-- 变量 content 等于 good -->
    <condition expr="#{content=='good'}" />
  </transition>
  <transition to="try again">
    <!-- 变量 content 等于 bad -->
    <condition expr="#{content=='bad'}" />
  </transition>
  <!-- 无条件转移 -->
  <transition to="give up" />
</decision>
<state name="submit document" />
<state name="try again" />
<state name="give up" />
</process>

```

以下是运行上面流程定义的单元测试代码。首先，设置流程变量 `content` 为 `good`：

```

Map<String, Object> variables = new HashMap<String, Object>();
variables.put("content", "good");
//发起流程实例并传入流程变量
ProcessInstance processInstance =
executionService.startProcessInstanceByKey("DecisionConditions",
variables);

```

那么，根据定义，就可以断言流程实例流向了 `submit document` 活动：

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

2. 使用 decision 活动的 `expr` 属性

您可以利用 `decision` 活动本身具有的 `expr`（表达式）属性来判断流程的转向。`decision` 活动的 `expr` 属性值可直接返回字符串类型的流出转移名称，指定需要流转的路径。

表 6-8 是 `decision` 活动支持的属性列表。

表 6-8 decision 活动的属性

属性	类型	默认值	是否必需	描述
expr	表达式	无	必需	描述转移名称的表达式
lang	表达式语言名称	脚本引擎配置文件 (jbpm.default.scriptmanager.xml) 里定义的默认表达式语言名称, 一般为 jUEL, 即 EL 表达式语言	可选	指定转移条件表达式的语言类型

如图 6-4 所示是一个应用 decision expr 方式判断流转路径的流程定义。

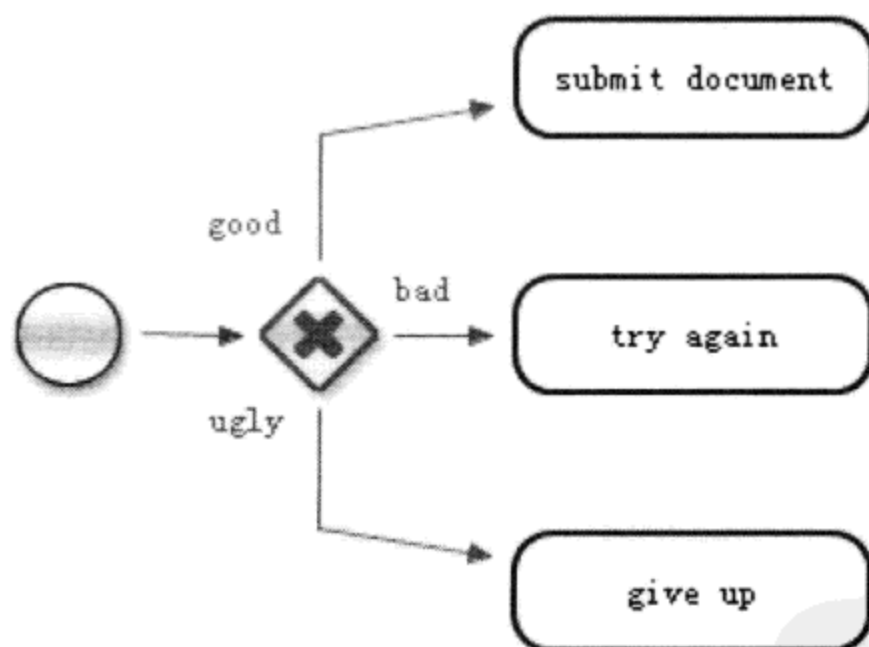


图 6-4 应用 decision expr 方式判断的流程定义

对应的 jPDL:

```

<process name="DecisionExpression" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="evaluate document" />
  </start>
  <!-- "#{content}" 即为判断表达式。在这里会返回变量 content 的字符串值 -->
  <decision expr="#{content}" name="evaluate document">

```



```

    <!-- content 值为 good 时的转移 -->
    <transition name="good" to="submit document" />
    <!-- content 值为 bad 时的转移 -->
    <transition name="bad" to="try again" />
    <!-- content 值为 ugly 时的转移 -->
    <transition name="ugly" to="give up" />
</decision>
<state name="submit document" />
<state name="try again" />
<state name="give up" />
</process>

```

在单元测试中，设置流程变量 `content` 的值为 `good`：

```

Map<String, Object> variables = new HashMap<String, Object>();
variables.put("content", "good");
//发起流程实例
ProcessInstance processInstance =
executionService.startProcessInstanceByKey("DecisionExpression",
variables);

```

那么，可以断言流程实例流向了 `submit document` 活动：

```

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

```

3. 使用 decision 活动的 handler 元素

也许以上两种判断流转的方式对您来说还是不够灵活，也许您需要在判断流转时计算大量、复杂的业务逻辑，那么，自己实现判断处理接口，即通过 `decision handler` 的方式，将给您无限自由的空间。

首先，必须实现 `DecisionHandler` 接口，将流转判断的决定权委派给这个实现类。这个接口的定义如下：

```

public interface DecisionHandler {
    //这个接口就一个方法，提供流程实例的执行上下文(execution)作为参数，需要返回字符串型的转移名称
    String decide(OpenExecution execution);
}

```

这个 `handler` 需要作为 `decision` 活动的子元素被配置。如图 6-5 所示是一个应用 `decision handler` 方式判断流转路径的流程定义。

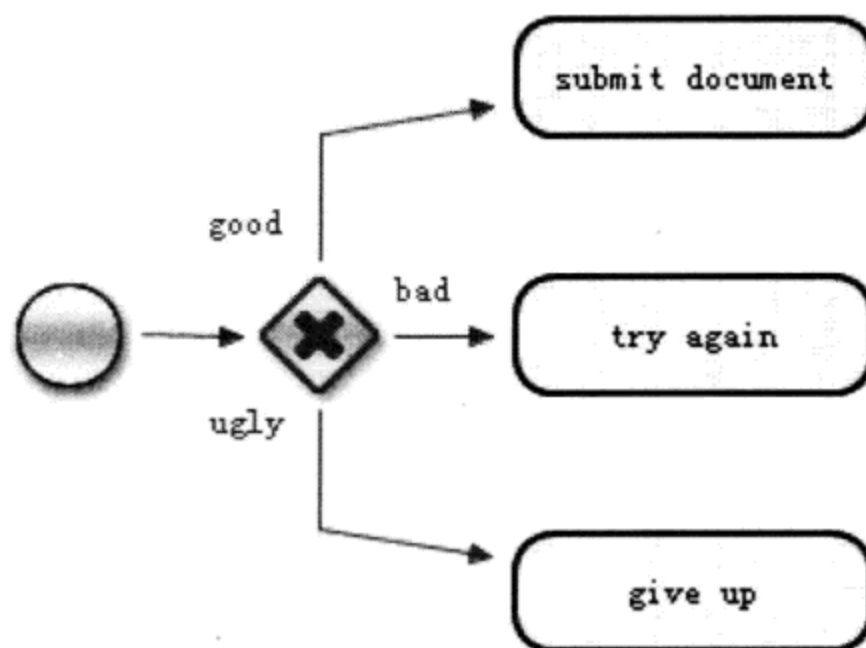


图 6-5 应用 decision handler 方式判断的流程定义

对应的 jPDL:

```

<process name="DecisionHandler" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="evaluate document" />
  </start>
  <decision name="evaluate document">
    <!-- 所有的流转判断逻辑都被委派给了这个 ContentEvaluation handler 统一处理 -->
    <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>

```

下面是判断处理器 ContentEvaluation 类的实现:

```

public class ContentEvaluation implements DecisionHandler {
  public String decide(OpenExecution execution) {
    //获取流程变量 content
    String content = (String) execution.getVariable("content");
  }
}

```

```

//如果 content 值为 "you're great", 则流向 good 转移
    if (content.equals("you're great")) {
        return "good";
    }
//如果 content 值为 "you gotta improve", 则流向 bad 转移
    if (content.equals("you gotta improve")) {
        return "bad";
    }
//如果都没有满足, 则流向 ugly 转移
    return "ugly";
}
}

```

运行流程实例的单元测试代码如下:

```

Map<String, Object> variables = new HashMap<String, Object>();
//设置流程变量 content 值为 "you're great"
variables.put("content", "you're great");
//发起流程实例, 并设置流程变量
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("DecisionHandler", variables);
//断言流向了预期的活动
assertTrue(processInstance.isActive("submit document"));

```

以上代码验证了: 当您发起流程实例, 并为变量 content 设置值为 you're great 时, 您实现的 decision handler 类 ContentEvaluation 会经过计算返回转移名称字符串 good, 流程实例便会根据 good 转移的定义流向 Submit document 活动。

思考: 似乎 decision 活动和 state 活动都能实现条件流转?

没错! 如您在本书中所见。

那么, 这两种方式有何区别, 在实际业务应用中该选择何种方式呢?

二者的区别如下:

如果 decision 活动定义的流转条件没有任何一个得到满足, 那么流程实例将无法进行下去, 会抛出异常。

如果 state 活动有多个流出转移, 且同样没有任何一个得到满足, 那么流程实例将流向 state 活动定义的第一条流出转移, 从而进行下去。

结论: decision 活动具有更加严格的条件判断特性, 如不定义默认路径, 则无条件满足, 即报错!

6.2.4 fork – join (分支/聚合活动)

当我们需要流程并发 (concurrency) 执行的时候, 就需要使用到 fork – join 活动的组合, fork 活动可以使流程在一条主干上出现并行的分支, join 活动则可以使流程的并行分支聚合成一条主干。

fork 活动仅具有 jBPM 活动的最基本特征, 即具有 1 个 name 属性和 n 个流出转移元素。

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 fork 元素的定义片段, 已标出支持的元素和属性供参考:

```
<element name="fork">
  <annotation>...</annotation>
  <complexType>

    <!-- 以下是 fork 活动的元素定义。 -->
    <sequence>
      <element name="description" minOccurs="0" type="string" />
      <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="tns:transition" minOccurs="0" maxOccurs="unbounded" />
    </sequence>

    <!-- 以下是 fork 活动的属性定义。 -->
    <attributeGroup ref="tns:activityAttributes" />

  </complexType>
</element>
```

join 活动则具有如表 6-9 所示的独特属性。需要注意的是 multiplicity 和 lockmode 这两个属性在 jBPM3 版本中是没有的, 是 jBPM4 根据广大流程开发者的呼声而“顺应民意”支持的。

表 6-9 join 活动的独特属性

属性	类型	默认值	是否必需	描述
multiplicity	integer	流入转移数	可选	流程执行中，当指定的流入转移数量 (multiplicity) 到达 join 活动后，流程即会聚合，沿着 join 活动的唯一流出转移继续执行流转。其他未到达的流入转移则被忽略，从而实现按流入转移数量聚合的场景，因此，multiplicity 属性不应该大于 join 活动定义的流入转移数量
lockmode	字符串枚举： {none, read, upgrade, upgrade_nowait, write}	upgrade	可选	指定 Hibernate 的数据锁模式。因为 join 活动支持并发自动活动的事务，因此需要在 join 活动上防止两个还没有聚合的同步事务活动互相锁定对方的事务资源，从而导致死锁

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 join 元素的定义片段，已标出支持的元素和属性供参考：

```
<element name="join">
  <annotation>...</annotation>
  <complexType>
```

```
<!-- 以下是 join 活动的元素定义。 -->
```



```

<sequence>
  <element name="description" minOccurs="0" type="string" />
  <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
  <element ref="tns:transition" minOccurs="0" maxOccurs="unbounded" />
</sequence>

<!-- 以下是 join 活动的属性定义。 -->
<attributeGroup ref="tns:activityAttributes" />
<attribute name="multiplicity" type="int" />
<attribute name="lockmode" default="upgrade">
  <simpleType>
    <restriction base="string">
      <enumeration value="none"/>
      <enumeration value="read"/>
      <enumeration value="upgrade"/>
      <enumeration value="upgrade_nowait"/>
      <enumeration value="write"/>
    </restriction>
  </simpleType>
</attribute>

</complexType>
</element>

```

如图 6-6 所示是一个 fork-join 流程定义的示例场景。

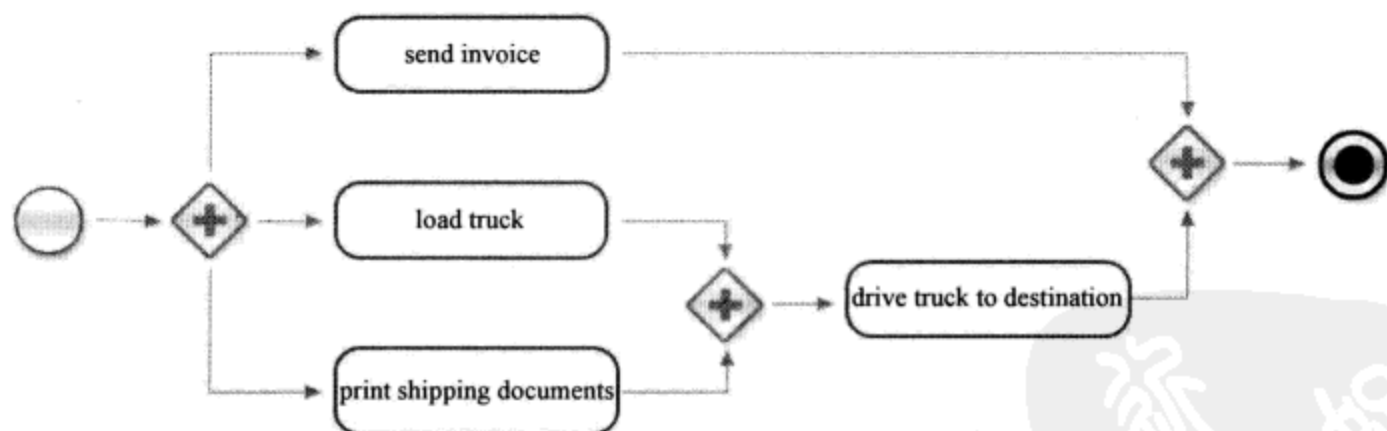


图 6-6 fork-join 流程定义的示例场景

对应的 jPDL:

```

<process name="ConcurrencyGraphBased" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="fork"/>
  </start>

```



```

<!-- 流程在此产生 3 个并行的分支 -->
<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>
<!-- 分支活动 load truck 和 print shipping documents 在此聚合 -->
<join name="shipping join" >
    <transition to="drive truck to destination" />
</join>
<state name="drive truck to destination" >
    <transition to="final join" />
</state>
<!-- 先前聚合的 drive truck to destination 活动和最初的 3 个分支之一的 send
invoice 活动在此完成最终的聚合 -->
<join name="final join" >
    <transition to="end"/>
</join>
<end name="end" />
</process>

```

编写单元测试代码执行上面的流程定义：

```

//发起 ConcurrencyGraphBased 流程实例
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("ConcurrencyGraphBased");
String pid = processInstance.getId();
//构造一个活动名称集合以验证分支。设置 3 个分支活动的名称
Set<String> expectedActivityNames = new HashSet<String>();
expectedActivityNames.add("send invoice");
expectedActivityNames.add("load truck");
expectedActivityNames.add("print shipping documents");
//断言当前活动即为产生的 3 个分支
assertEquals(expectedActivityNames,
processInstance.findActiveActivityNames());
//发出执行信号通过“send invoice”活动，这时候，流程会在最后的聚合活动“final

```

join”上等待其他分支的到来……

```
String sendInvoiceExecutionId = processInstance.findActiveExecutionIn
("send invoice").getId();
processInstance = executionService.signalExecutionById
(sendInvoiceExecutionId);
//在活动名称集合中排除“send invoice”活动
expectedActivityNames.remove("send invoice");
//此时, 仍然可以断言另外 2 个分支还在等待
assertNotNull(processInstance.findActiveExecutionIn("load
truck"));
assertNotNull(processInstance.findActiveExecutionIn("print shipping
documents"));
//发出执行信号通过剩下的第 1 个分支——load truck 活动
String loadTruckExecutionId = processInstance.findActiveExecutionIn
("load truck").getId();
processInstance = executionService.signalExecutionById(loadTruckExecutionId);
//在活动名称集合中排除“load truck”活动
expectedActivityNames.remove("load truck");
//发出执行信号通过剩下的第 2 个分支——print shipping documents 活动
String printShippingDocumentsId = processInstance
.findActiveExecutionIn("print shipping documents").getId();
processInstance = executionService.signalExecutionById
(printShippingDocumentsId);
//在活动名称集合中排除“print shipping documents”活动
expectedActivityNames.remove("print shipping documents");
//断言通过了第一个聚合活动 shipping join, 到达了“drive truck to
destination”活动
expectedActivityNames.add("drive truck to destination");
assertEquals(expectedActivityNames,
processInstance.findActiveActivityNames());
assertNotNull(processInstance.findActiveExecutionIn("drive truck
to destination"));
//发出执行信号通过“drive truck to destination”活动
String driveTruckExecutionId = processInstance.findActiveExecutionIn(
"drive truck to destination").getId();
processInstance = executionService.signalExecutionById
(driveTruckExecutionId);
//最终的聚合活动“final join”等到了它的最后一个流入转移后, 流向了 end 活动,
所以流程实例结束
//因此可以断言此流程实例已经不存在了
assertNull("execution " + pid + " should not exist", executionService.
findExecutionById(pid));
```

6.2.5 end (结束活动)

end 活动会终结流程。

默认情况下，当流程实例运行到 end 活动会结束。但是在到达 end 活动的流程实例中仍然活跃的流程活动(这可能是 fork-join 并发流转引起的)将会被保留继续执行。

如图 6-7 所示是一个最简单的 end 活动流程定义示例。

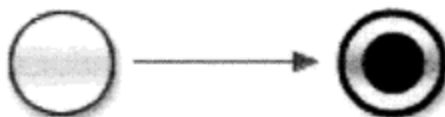


图 6-7 end 活动流程定义示例

对应的 jPDL:

```
<process name="EndProcessInstance" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="end" />
  </start>
  <end name="end" />
</process>
```

根据这个定义，新的流程实例一创建便会直接结束。单元测试代码如下：

```
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("EndProcessInstance");
//创建流程实例后，直接断言其结束
assertTrue(processInstance.isEnded());
```

下面来看一些复杂点的 end 活动。

在 jBPM4 中，流程定义允许有多个 end 活动，如果您需要：执行到特定的 end 活动时，流程实例会被完全结束，即其他仍然活跃的并发活动也需要被结束，那么，可以设置 end 活动的属性 “ends = "execution"” 来实现这种需求。表 6-10 是 ends 属性的详细解释。

表 6-10 end 活动的 ends 属性

属性	类型	默认值	是否必需	描述
ends	字符串枚举： {processinstance execution}	processinstance	可选	执行到达 end 活 动时整个流程实 例（包括其所有 活动）会被完全 结束

一个流程定义可以有多个 end 活动，以便通过事件机制触发不同的结束方式。如图 6-8 流程定义所示。

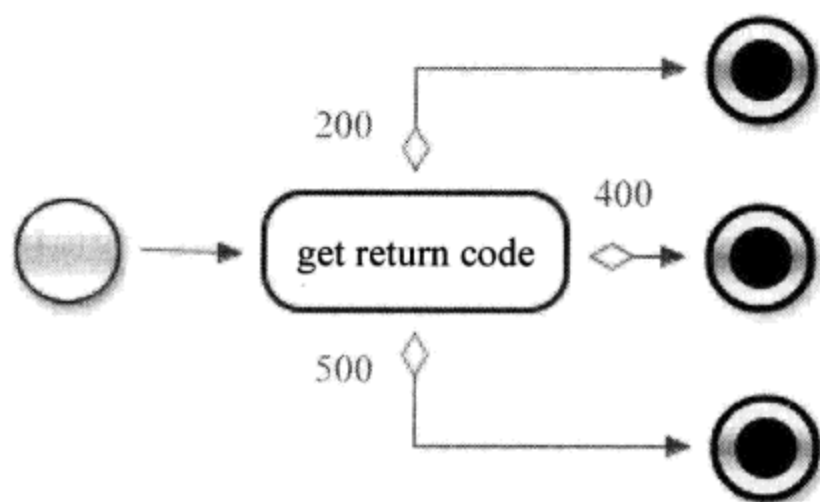


图 6-8 多个 end 活动的流程定义

对应的 jPDL:

```
<process name="EndMultiple" xmlns="http://jbpm.org/4/jpdl">
  <start>
    <transition to="get return code" />
  </start>
  <!-- 此活动有 3 条流出转移供选择，分别指向 3 个不同的 end 活动 -->
  <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>
```

下面的单元测试代码发起流程实例，并在“get return code”活动发出流转信号的时候指示流程通过名称为 400 的流出转移，那么，流程便会在名称为“bad request”的 end 活动上结束。

```
ProcessInstance processInstance = executionService.  
startProcessInstanceByKey("EndMultiple");  
String executionId = processInstance.getId();  
//在这里指定转移名称: 400  
processInstance = executionService.signalExecutionById(executionId,  
"400");  
//断言流程实例结束  
assertTrue(processInstance.isEnded());
```

同理，指定转移名称为 200 或 500 就会让流程实例在名称为 ok 或 internal server error 的 end 活动上结束。

在实际应用中，我们经常需要知道流程实例是以何种“状态”结束的。为了表明流程实例的结束状态，可以利用 end 活动的 state 属性标识，或者直接利用 jBPM4 提供的特殊 end 活动：end-cancel 活动和 end-error 活动。表 6-11 是 state 属性的详细解释。

表 6-11 end 活动的 state 属性

属性	类型	默认值	是否必需	描述
state	String		可选	用来自定义流程实例的状态。可以通过此 API 取出：processInstance.getState()

下面列出 jBPM4 目前支持的 3 种 end 活动的元素和属性的 XML Schema 源定义，这 3 种 end 活动分别是：end，end-cancel 和 end-error。

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 end 元素的定义片段，已标出支持的元素和属性供参考：

```
<element name="end">  
  <annotation>...</annotation>  
  <complexType>  
  
    <!-- 以下是 end 活动的元素定义。 -->  
    <sequence>  
      <element name="description" minOccurs="0" type="string" />  
      <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>  
    </sequence>
```



```

<!-- 以下是 end 活动的属性定义。 -->
<attributeGroup ref="tns:activityAttributes" />
<attribute name="ends" default="process-instance">
  <simpleType>
    <restriction base="string">
      <enumeration value="execution"/>
      <enumeration value="process-instance"/>
    </restriction>
  </simpleType>
</attribute>
<attribute name="state" default="ended" type="string">
  <annotation>...</annotation>
</attribute>

</complexType>
</element>

```

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 end-cancel 元素的定义片段。注意 end-cancel 活动比 end 活动少了 state 属性，因为 end-cancel 活动的 state 属性已经被约定为“cancel”。下面已标出 end-cancel 活动支持的元素和属性供参考：

```

<element name="end-cancel">
  <annotation>...</annotation>
  <complexType>

    <!-- 以下是 end-cancel 活动的元素定义。 -->
    <sequence>
      <element name="description" minOccurs="0" type="string" />
      <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>

    <!-- 以下是 end-cancel 活动的属性定义。 -->
    <attributeGroup ref="tns:activityAttributes" />
    <attribute name="ends" default="process-instance">
      <simpleType>
        <restriction base="string">
          <enumeration value="execution"/>
          <enumeration value="process-instance"/>
        </restriction>
      </simpleType>
    </attribute>

  </complexType>
</element>

```

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 end-error 元素的定义片段。注意 end-error 活动比 end 活动少了 state 属性，因为 end-error 活动的 state 属性已经被约定为 “error”。下面已标出 end-error 活动支持的元素和属性供参考：

```
<element name="end-error">
  <annotation>...</annotation>
  <complexType>

    <!-- 以下是 end-error 活动的元素定义。 -->
    <sequence>
      <element name="description" minOccurs="0" type="string" />
      <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>

    <!-- 以下是 end-error 活动的属性定义。 -->
    <attributeGroup ref="tns:activityAttributes" />
    <attribute name="ends" default="process-instance">
      <simpleType>
        <restriction base="string">
          <enumeration value="execution"/>
          <enumeration value="process-instance"/>
        </restriction>
      </simpleType>
    </attribute>

  </complexType>
</element>
```

如图 6-9 所示是一个拥有不同 end 状态的流程定义。

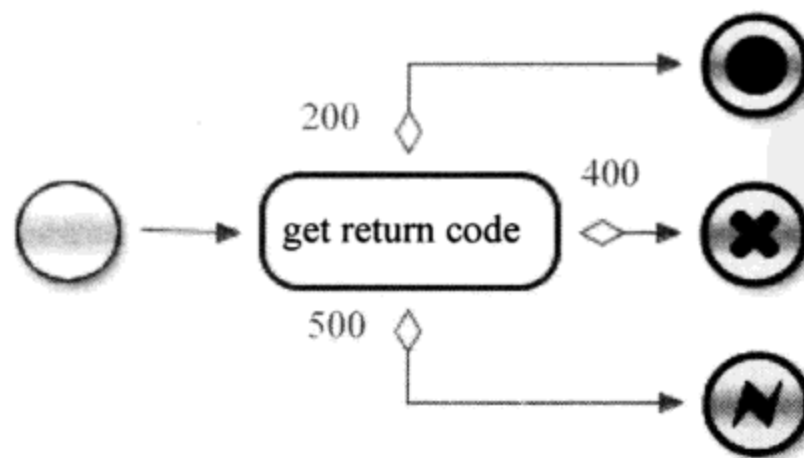


图 6-9 拥有不同 end 状态的流程定义

对应的 jPDL:

```

<process name="EndState" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="get return code"/>
  </start>
  <!-- 3 条流出转移指向具有不同状态的 end 活动 -->
  <state name="get return code">
    <transition name="200" to="ok"/>
    <transition name="400" to="bad request" />
    <transition name="500" to="internal server error"/>
  </state>
  <!-- 此 end 活动设置流程的状态为 completed -->
  <end name="ok" state="completed"/>
  <!-- 此 end 活动设置流程的状态为 cancel -->
  <end-cancel name="bad request"/>
  <!-- 此 end 活动设置流程的状态为 error -->
  <end-error name="internal server error"/>
</process>

```

这时候，我们在单元测试中指定名称为“400”的转移，则可断言流程实例结束于 cancel 状态：

```

ProcessInstance processInstance = executionService.startProcessInstanceByKey(
    "EndState");
String pid = processInstance.getId();
processInstance = executionService.signalExecutionById(pid, "400");
//断言流程实例的状态与定义的预期相符合
assertEquals("cancel", processInstance.getState());
assertTrue(processInstance.isEnded());

```

同样的道理作用于名称为“500”的转移。

指定名称为“200”的转移，则流程实例的状态取 end 活动的 state 属性值：

```

ProcessInstance processInstance = executionService.startProcessInstanceByKey(
    "EndState");
String pid = processInstance.getId();
processInstance = executionService.signalExecutionById(pid, "200");
//断言流程实例的状态与定义的预期相符合，state 属性值为 completed
assertEquals("completed", processInstance.getState());
assertTrue(processInstance.isEnded());

```

6.2.6 task (人工任务活动)

在 jBPM 中，task 活动一般用来处理涉及人机交互的活动。task 活动的功能在 jBPM

乃至整个工作流的应用中都具有极其重要的意义，因为处理人工任务、电子表单是工作流应用中最“烦琐”和细致的工作，所以本节也会比较详细地介绍。

以下是 XML Schema 文件 `jpdl-4.3.xsd` 中对于 `task` 元素的定义片段，已标出支持的元素和属性供参考：

```
<element name="task">
  <annotation>...</annotation>
  <complexType>

    <!-- 以下是 task 活动的元素定义。 -->
    <sequence>
      <element name="description" minOccurs="0" type="string" />
      <element name="assignment-handler" minOccurs="0"
type="tns:wireObjectType" />
      <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
      <element name="notification" minOccurs="0">
        <complexType>
          <attribute name="continue" type="tns:continueType" default="sync" />
          <attribute name="template" type="tns:templateType" use="optional" />
        </complexType>
      </element>
      <element name="reminder" minOccurs="0">
        <complexType>
          <attribute name="duedate" type="string" />
          <attribute name="repeat" type="string" />
          <attribute name="continue" type="tns:continueType" default="sync" />
          <attribute name="template" type="tns:templateType" use="optional"/>
        </complexType>
      </element>
      <element ref="tns:timer" minOccurs="0" maxOccurs="unbounded"/>
      <element name="transition" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <complexContent>
            <extension base="tns:transitionType">
              <sequence>
                <element ref="tns:timer" minOccurs="0" />
              </sequence>
            </extension>
          </complexContent>
        </complexType>
      </element>
    </sequence>
```

```

<!-- 以下是 task 活动的属性定义。 -->
<attributeGroup ref="tns:activityAttributes" />
<attributeGroup ref="tns:assignmentAttributes"/>
<attribute name="swimlane" type="string" />
<attribute name="form" type="string">
  <annotation>...</annotation>
</attribute>
<attribute name="duedate" type="string" />
<attribute name="on-transition" default="cancel">
  <simpleType>
    <restriction base="string">
      <enumeration value="keep"/>
      <enumeration value="cancel"/>
    </restriction>
  </simpleType>
</attribute>
<attribute name="completion" type="string" default="complete" />

</complexType>
</element>

```

可以看到 task 活动支持的元素和属性比较多，许多元素还拥有子元素。这也说明了 task 活动的功能较为丰富，以下将分别详细介绍。

1. 关于任务的分配者

我们可以使用 task 活动的 assignee 属性（分配者属性）简单地将一个任务分配给指定的用户。此属性定义如表 6-12 所示。

表 6-12 task 活动的分配者属性

属性	类型	默认值	是否必需	描述
assignee	表达式	无	可选	被分配到任务的用户 ID

如图 6-10 所示的流程定义使用到了 assignee 属性。

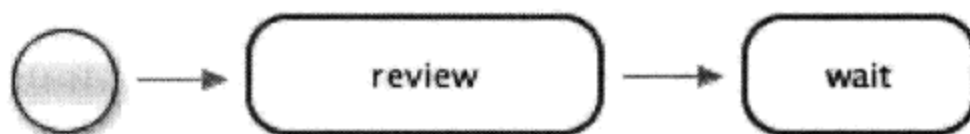


图 6-10 任务分配者示例流程

对应的 jPDL:

```
<process name="TaskAssignee" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="review" />
  </start>
  <!-- EL 表达式 "#{order.owner}" 的值在这里表示分配者 ID -->
  <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:

- 1) assignee 属性引用了一个用户, 即负责完成任务的人。
- 2) assignee 属性默认会被作为 EL 表达式来执行。此例中任务被分配给 `#{order.owner}`。这意味着流程引擎将首先使用 `order` 这个名称在任务对应的流程变量里查找一个对象, 然后通过 `order` 对象的 `getOwner` 方法获得用户 ID, 从而将任务分配给该用户。

order 对象的类定义:

```
public class Order implements Serializable {
  //owner 成员域保存用户的 ID
  String owner;
  public Order(String owner) {
    this.owner = owner;
  }
  public String getOwner() {
    return owner;
  }
  public void setOwner(String owner) {
    this.owner = owner;
  }
}
```

基于此定义发起流程实例, 并构造一个 order 对象传入流程实例:

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

用户 alexmiller 的任务列表可以通过如下 API 获得：

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

注意：assignee 属性可以解释纯文本，通过 assignee="alexmiller"，同样会把任务分配给 alexmiller。

2. 关于任务的候选者

jBPM 支持将任务分配给一组候选用户，组中的一个用户可以接受这个任务并完成之，这就是任务的候选者机制，其相关属性如表 6-13 所示。

表 6-13 task 活动的候选者属性

属性	类型	默认值	是否必需	描述
candidate-groups	表达式	无	可选	使用逗号分隔的用户组 ID 列表。所有组的用户将会成为任务的候选者
candidate-users	表达式	无	可选	使用逗号分隔的用户 ID 列表。所有列表中的用户将会成为任务的候选者

图 6-11 所示的流程定义使用到了任务的候选者属性。

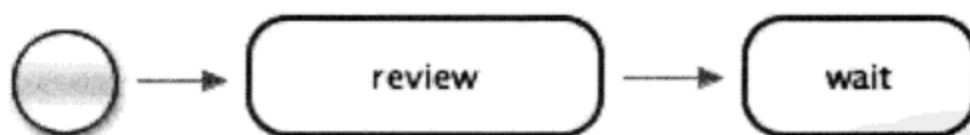


图 6-11 任务候选者示例流程

对应的 jPDL:

```
<process name="TaskCandidates">
  <start>
    <transition to="review" />
  </start>
  <!-- 在这里用字符串引用了一个用户组: sales-dept -->
  <task name="review" candidate-groups="sales-dept">
    <transition to="wait" />
  </task>
</process>
```

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

流程实例发起后，任务 **review** 会被创建。这个任务不会显示在任何人的个人任务列表中，因为还没有创建 **sales-dept** 组。因此，下面获取的个人任务列表将是空的：

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

但是此任务会显示在 **sales-dept** 组成员的分组任务列表中，可以通过 **Service API** `taskService.findGroupTasks` 获取。

那么，接下来，我们可以尝试将 **alexmilller** 和 **joesmoe** 这两个用户创建并加入 **sales-dept** 组，这需要使用到 **IdentityService**（身份认证服务）：

```
//首先要创建 sales-dept 组
identityService.createGroup("sales-dept");
//创建用户 alexmilller
identityService.createUser("alexmilller", "alexmilller", "Alex", "Miller");
//将 alexmilller 加入 sales-dept 组
identityService.createMembership("alexmilller", "sales-dept");
//创建用户 joesmoe
identityService.createUser("joesmoe", "joesmoe", "Joe", "Smoe");
//将 joesmoe 加入 sales-dept 组
identityService.createMembership("joesmoe", "sales-dept");
```

如此，在流程实例发起后，**review** 任务就会出现在用户 **alexmilller** 和 **joesmoe** 的分组任务列表中，即以下代码的执行结果非空：

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

即此任务有了 2 个候选者——**alexmilller** 和 **joesmoe**，候选者在处理任务之前，必须先“接受”任务。这会表现为两个候选者同时看到任务，并“竞争”之。在用户通过分组任务列表获取任务后，使用如下 **API** 对任务执行“接受”操作：

```
// alexmilller 接受了任务
taskService.takeTask(task.getId(), "alexmilller");
```

当用户 **alexmilller** 接受了任务后，**alexmilller** 就会由任务的候选者变为任务的分配者。同时，此任务会从所有候选者的任务列表中消失，它会出现 **alexmilller** 的已分配任务列表中。

顺便提一下，在客户端应用设计中，用户应当只允许在他们的个人任务列表上

工作。

与 `candidate-groups` 属性类似的，`candidate-users` 属性可以用来处理用逗号分隔的一系列用户 ID，`candidate-users` 属性可以和其他任务分配属性结合使用。

jBPM4 的任务候选者机制可以看做 jBPM3 中的 Pooled Actor（任务办理者池）机制的升级版。

通过以上的两种方法，我们可以把任务直接分配到用户，或把任务分配到一个“候选者池”，让池中的用户自己决定是否接受任务。但是，如果我们需要在分配任务时加入一些复杂的业务逻辑计算呢？显然以上两种方式就表现得不够灵活了。在这种需求下，开发一个“任务分配处理器”是个不错的选择。

3. 关于任务分配处理器（AssignmentHandler）

我们可以通过开发 AssignmentHandler 的方式来编码计算任务的分配者和候选者。

任务分配处理器需要实现 AssignmentHandler 接口：

```
public interface AssignmentHandler extends Serializable {  
    /** 在这个提供的 assignable 对象中设置分配者和候选者 */  
    void assign(Assignable assignable, OpenExecution execution) throws Exception;  
}
```

注意：上面 assign 方法中的 Assignable 类型是任务和泳道（Swimlanes）的通用接口。所以，任务分配处理器既可以作为任务活动的元素，也可以作为泳道元素的子元素。关于泳道元素在本节后面的内容中会介绍。

任务分配处理器作为任务活动的一个子元素，名称为 `assignment-handler`。它指向用户代码实现的 AssignmentHandler 接口。关于 `assignment-handler` 的属性和元素的详细解释请查看 6.6 用户代码。

图 6-12 所示的流程定义使用到了任务分配处理器。

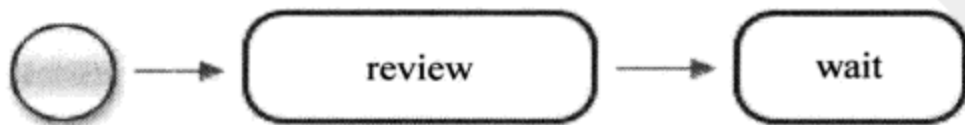


图 6-12 使用任务分配处理器的示例流程

对应的 jPDL:

```
<process name="TaskAssignmentHandler" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="review" />
  </start>
  <task name="review">
    <!-- 在这里的 org.jbpm.examples.task.assignmenthandler.AssignTask 即是任务
分配处理器的实现。 -->
    <assignment-handler
class="org.jbpm.examples.task.assignmenthandler.AssignTask">
      <!-- field 元素为任务分配处理器的 assignee 域注入值。这种注入方式对于 jBPM 中的用户代
码都是通用的。 -->
      <field name="assignee">
        <string value="alexmilller" />
      </field>
    </assignment-handler>
    <transition to="wait" />
  </task>
  <state name="wait" />
</process>
```

类 `org.jbpm.examples.task.assignmenthandler.AssignTask` 实现了一个最简单的任务分配处理器:

```
public class AssignTask implements AssignmentHandler {
  //记得吗, 这个域的值是在上面的 jPDL 定义中被注入的
  String assignee;
  public void assign(Assignable assignable, OpenExecution execution) {
    //设置任务的分配者
    assignable.setAssignee(assignee);
  }
}
```

以下单元测试代码运行并验证了上面的流程定义:

```
//发起流程实例
executionService.startProcessInstanceByKey("TaskAssignmentHandler");
// alexmilller 是通过 jPDL 定义注入的任务分配者
List<Task> taskList = taskService.findPersonalTasks("alexmilller");
//断言 alexmilller 有一个任务
assertEquals(1, taskList.size());
Task task = taskList.get(0);
//断言任务名称如定义的 "review"
assertEquals("review", task.getName());
//断言任务的分配者如预期
```



```
assertEquals("alexmilller", task.getAssignee());
```

以上代码发起流程实例后立刻运行到任务活动 review。当 review 任务被创建时，AssignTask 任务分配处理器被调用，这将设置 ID 为 alexmilller 的用户为此任务的分配者。所以 Alex Miller 将在他的个人任务列表中找到这个任务。

如您所见，AssignmentHandler 提供的 execution 对象可以获得流程上下文和变量，可以结合其他任何 API 来帮助您“计算”出任务的分配者和候选者、单个用户和用户组。

4. 关于任务泳道

在实际的业务应用中，经常会遇到这样一种场景：流程定义中的多个任务需要被分配或候选给同一个群用户。那么我们可以统一将这个“同一群用户”定义为“一个泳道”。泳道作为流程定义的直接子元素被整个流程定义所见，因此同一流程定义中的任何一个任务都可以引用泳道。属于同一个泳道的任务将会被分配或候选给这个泳道中的所有用户。

图 6-13 所示是一个应用泳道（Swimlanes）机制的流程示意图。

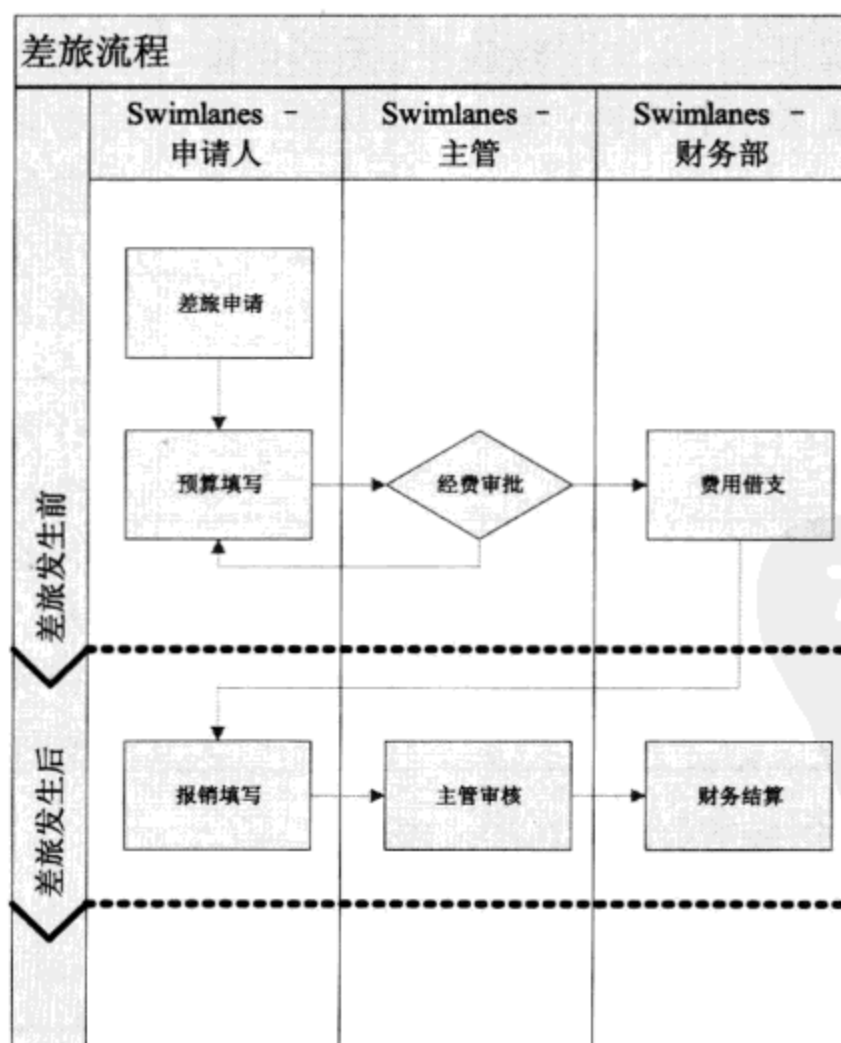


图 6-13 “泳道”在流程定义中应用的示意图

在上面这个简单的差旅流程中，有 3 个泳道——申请人、主管、财务部，他们分别需要处理 3，2，2 个任务。

泳道的概念也可以理解为流程定义的“全局用户组”。泳道也可以被当做一个流程规则。在一些情况下，泳道可能与后面提到的身份认证组件中的权限角色相似，但是实际上它们并不是同一个东西。

表 6-14 列出了任务活动的泳道属性。

表 6-14 任务活动的泳道属性

属性	类型	默认值	是否必需	描述
swimlane	泳道名称字符串	无	可选	引用一个在流程中定义的泳道

表 6-14 的 swimlane 属性是任务活动对泳道的引用，泳道本身是作为 process 流程定义的子元素被定义在整个流程范围内的。表 6-15 列出了泳道元素的属性。

表 6-15 泳道 (swimlane) 元素的属性

属性	类型	默认值	是否必需	描述
name	泳道名称字符串	无	必需	这个泳道名称将在任务的泳道属性中引用
assignee	表达式	无	可选	引用的单个用户 ID
candidate-groups	表达式	无	可选	使用逗号分隔的用户组 ID 列表。此组中的所有用户将作为引用此泳道任务的候选人
candidate-users	表达式	无	可选	使用逗号分隔的用户 ID 列表。此列表中的所有用户将作为引用此泳道任务的候选人

图 6-14 所示是使用泳道分配任务的流程定义。

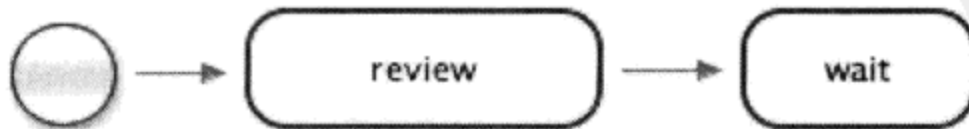


图 6-14 泳道示例流程

对应的 jPDL:

```
<process name="TaskSwimlane" xmlns="http://jbpm.org/4.3/jpdl">
  <!-- 在这里定义泳道，泳道是为流程定义的子元素 -->
  <swimlane name="sales representative" candidate-groups="sales-dept" />
  <start>
    <transition to="enter order data" />
  </start>
  <!-- 以下 2 个任务的分配工作，都交给上面定义的泳道完成 -->
  <task name="enter order data" swimlane="sales representative">
    <transition to="calculate quote"/>
  </task>
  <task name="calculate quote" swimlane="sales representative">
  </task>
</process>
```

上面定义的泳道“sales representative”引用了一个用户组 sales-dept。在流程运行前，这个用户组需要被创建出来，利用身份认证服务 IdentityService:

```
identityService.createGroup("sales-dept");
//创建用户 alexmiller 并使他加入 sales-dept 组
identityService.createUser("alexmiller", "alexmiller", "Alex", "Miller");
identityService.createMembership("alexmiller", "sales-dept");
```

在发起流程实例后，用户 alexmiller 将成为任务“enter order data”的唯一候选者（因为组里只有他一个用户）。还是像任务候选者的例子一样，alexmiller 需要先接受这个任务：

```
taskService.takeTask(taskId, "alexmiller");
```

接受这个任务将使 alexmiller 成为任务的分配者，同时泳道“sales representative”也会发生变化，alexmiller 在这个流程实例中会被固化为分配者

接下来，alexmiller 可以通过 completeTask API 完成任务：

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

完成此任务后流程实例将会流转到下一个任务“calculate quote”，这个任务也引用泳道“sales representative”。因此，任务会直接分配给 alexmiller。可以通过如下代码验证：

```
taskList = taskService.findPersonalTasks("alexmiller");
//断言 alexmiller 直接拿到了任务
assertEquals(1, taskList.size());
task = taskList.get(0);
//断言是否为预期的任务和分配者
```

```
assertEquals("calculate quote", task.getName());
assertEquals("alexmilller", task.getAssignee());
```

5. 关于任务变量

在之前的很多示例中，我们已经多次利用任务变量来为流程任务提供数据的输入输出了。

任务可以读取（即输入）、更新（即输出）流程变量。任务还可以定义任务自有的变量，即任务变量。一般来说，任务变量的主要作用是作为任务表单的数据容器——任务表单负责展现来自任务和流程的变量数据；同时用户通过任务表单录入的数据则会被设置为任务变量，任务变量根据需要也可能被输出成为流程变量。

关于流程变量的详细介绍可在第 7 章 **流程变量**中得到。

假设我们发起了如下流程定义：

```
<process name="TaskVariables" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="review" />
  </start>
  <!-- 任务定义在这里 -->
  <task name="review" assignee="alexmilller">
    <transition to="wait" />
  </task>
  <state name="wait" />
</process>
```

发起这个定义的流程实例后，获得任务变量可通过如下单元测试代码实现：

```
//首先拿到任务，这个任务被定义分配给了 alexmilller
List<Task> taskList = taskService.findPersonalTasks("alexmilller");
Task task = taskList.get(0);
long taskId = task.getId();
//根据此任务 ID 获取任务变量集
Set<String> variableNames = taskService.getVariableNames(taskId);
//拿到所有的任务变量。任务变量以 Map 对象的方式提供
Map<String, Object> variables = taskService.getVariables(taskId, variableNames);
```

有了任务变量 Map 对象，就可以这么操作它：

```
//下面添加（或更新，如果 Map 中存在同名键的话）具体的任务变量。任务变量可以是任何支持序列化的 Java Object
variables.put("category", "small");
variables.put("lires", 923874893);
//将任务变量设置到任务中。这是个必不可少的持久化操作
```

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

6. 关于任务提醒邮件

在很多企业中，电子邮件往往比流程任务列表更能引起人们的注意。幸运的是，jBPM4 支持使用电子邮件进行任务提醒。

我们可以为任务的分配者提供一个电子邮件提醒，这包括：

- 当一个任务出现在某人的任务列表中时立即提醒。
- 指定时间间隔进行反复提醒。

电子邮件的内容是根据一个模板生成出来的，此模板默认使用 jBPM 内置的，您也可以在 jBPM 配置文件中 `process-engine-context` 部分指定自定义的模板。关于电子邮件支持的详细介绍（这包括邮件服务器的配置等），您可以参见第 16 章 深入 jBPM4 电子邮件支持。

表 6-16 所示是任务活动支持的电子邮件相关元素。

表 6-16 任务活动的电子邮件相关元素

元素	数目	描述
notification	0..1	当一个任务被分配的时候立即发送一封提醒邮件。如果没有指定模板，邮件会使用默认的 <code>task-notification</code> 模板
reminder	0..1	根据指定的时间间隔发送提醒邮件。如果没有指定模板，邮件会使用默认的 <code>task-reminder</code> 模板

表 6-17 和表 6-18 所示分别是表 6-16 中两个元素的属性列表。

表 6-17 notification 元素的属性

属性	类型	默认值	是否必需	描述
continue	字符串枚举 {sync async exclusive}	sync	可选	以同步、异步还是独占模式发送 notification 提醒邮件

表 6-18 reminder 元素的属性

属性	类型	默认值	是否必需	描述
duedate	延迟时间(可包含表达式的字符串)	无	必需	reminder 提醒电子邮件在任务产生后延迟多少时间发送
repeat	间隔时间(可包含表达式的字符串)	无	可选	reminder 提醒电子邮件每间隔多少时间就再发送一次,直到任务被办理
continue	字符串枚举 {sync async exclusive}	sync	可选	以同步、异步还是独占模式发送 reminder 提醒邮件

以下是一段涉及任务邮件提醒的流程定义 jPDL, 应用到了表 6-16、表 6-17 和表 6-18 中的元素和属性:

```
<process name="TaskReminder" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="review" />
  </start>
  <task name="review" assignee="{order.owner}">
    <!-- 这表示在任务产生后, 立即以同步的方式发送提醒邮件 -->
    <notification />
    <!-- 这表示在任务产生 2 天后, 开始发送提醒邮件, 如果任务得不到处理, 每隔 1 天再提醒一次 -->
    <reminder duedate="2 days" repeat="1 day" />
    <transition to="wait" />
  </task>
  <state name="wait" />
</process>
```

7. 关于任务表单

任务表单是把任务展现给用户的最常用的客户端应用方式。任务表单的应用比较灵活、多样且有技巧性, 在 10.6 自定义 Web 任务表单中会介绍。

6.2.7 sub-process (子流程活动)

当我们的流程复杂到一定程度的时候, 就需要按照一定规则把业务拆分成若干个子流程, 这样业务模块之间才能明晰易于划分。有时候, 为了方便相对独立的流程之

间的拼装、重组，划分出子流程管理也是个明智的选择。

JBPM4 提供了 sub-process——子流程活动，这允许您在“主干流程”定义中调用其他的流程定义，从而“组装”您的流程定义。在运行到子流程活动时，工作流引擎将创建一个子流程实例，然后等待直到其完成，当子流程实例完成后，流程就会流向下一步。

注意：子流程活动在 JBPM3 版本中被称做 process-state，到了 JBPM4 中则为 sub-process，这更符合人们的习惯，因此，作者认为这个名称的变更是值得赞赏的。

表 6-19 和表 6-20 所示是 sub-process 活动的属性和元素列表。

表 6-19 sub-process 活动的属性

属性	类型	默认值	是否必需	描述
sub-process-id	字符串	无	sub-process-id 和 sub-process-key, 必选其一	流程定义的 ID 标识。正如我们前面所说的，可以通过一个流程的 ID 去引用此流程定义的指定版本
sub-process-key	字符串	无	sub-process-key 和 sub-process-id, 必选其一	流程的 Key 标识。通过 Key 去引用流程定义，意味着引用了该流程定义的最新版本。注意，该流程定义的最新版本会在每次活动实例执行时计算得出
outcome	表达式	无	当 sub-process 活动的 transition 元素具有 outcome-value 时必需	当子流程活动执行结束时执行的表达式。表达式值用来匹配流出转移中的 outcome-value 元素值（这个元素下面会提到），起到选择 sub-process 活动下一步流向的作用

表 6-20 sub-process 活动的元素

元素	聚合关系	描述
parameter-in	0..*	子流程输入参数。即声明一个变量，在创建子流程实例时传入
parameter-out	0..*	子流程输出参数。即声明一个变量，在子流程实例结束时，返回父流程实例

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 sub-process 元素的定义片段，已标出支持的元素和属性供参考：

```

<element name="sub-process">
  <annotation>...</annotation>
  <complexType>

    <!-- 以下是 sub-process 活动的元素定义。 -->
    <sequence minOccurs="0" maxOccurs="unbounded">
      <element name="description" minOccurs="0" type="string" />
      <element name="parameter-in"
type="tns:parameterType" minOccurs="0" maxOccurs="unbounded" />
      <element name="parameter-out"
type="tns:parameterType" minOccurs="0" maxOccurs="unbounded" />
      <element ref="tns:timer" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="tns:on" minOccurs="0" maxOccurs="unbounded">
        <annotation>...</annotation>
      </element>
      <element name="swimlane-mapping" minOccurs="0" maxOccurs="
"unbounded">
        <complexType>
          <attribute name="swimlane" type="string" use="required" />
          <attribute name="sub-swimlane" type="string" use="required" />
        </complexType>
      </element>
      <element name="transition" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <complexContent>
            <extension base="tns:transitionType">
              <sequence minOccurs="0" maxOccurs="unbounded">
                <element name="outcome-value">
                  <complexType>

```

```

        <group ref="tns:wireObjectGroup" />
        </complexType>
    </element>
</sequence>
</extension>
</complexContent>
</complexType>
</element>
</sequence>

<!-- 以下是 sub-process 活动的属性定义。 -->
<attribute name="sub-process-id" type="string">
    <annotation>...</annotation>
</attribute>
<attribute name="sub-process-key" type="string">
    <annotation>...</annotation>
</attribute>
<attribute name="outcome" type="string">
    <annotation>...</annotation>
</attribute>
<attributeGroup ref="tns:activityAttributes" />

</complexType>
</element>

```

具体解释一下 parameter-in 元素和 parameter-out 元素的属性，分别如表 6-21 和表 6-22 所示。

表 6-21 sub-process 的 parameter-in——子流程活动输入元素的属性

属性	类型	默认值	是否必需	描述
subvar	字符串	无	必需	被赋值的子流程变量的名称
var	字符串	无	var 和 expr 二者必须指定一个	从父流程环境中输入的变量名称
expr	字符串	无	var 和 expr 二者必须指定一个	此表达式在父流程环境中解析，结果值会被输入到对应的子流程变量中
lang	字符串	EL 表达式	可选	表达式使用的脚本语言

表 6-22 sub-process 的 parameter-out——子流程活动输出元素的属性

属性	类型	默认值	是否必需	描述
var	字符串	无	必需	输出的目标——父流程中的变量名称
subvar	字符串	无	subvar 和 expr 二者必须指定一个	子流程中需要被输出的变量名称
expr	字符串	无	subvar 和 expr 二者必须指定一个	此表达式在子流程环境中解析，结果值会被输入到对应的父流程变量中
lang	字符串	EL 表达式	可选	表达式使用的脚本语言

在上面我们提到了 sub-process 活动的 outcome 元素，outcome 元素必须有与之相对应的 outcome-value 元素，这个 outcome-value 元素被定义在子流程活动的流出转移（transition）中，表 6-23 是关于这个元素的说明。

表 6-23 sub-process transition 的 outcome-value 元素

元素	聚合关系	描述
outcome-value	0..1	outcome-value 是一个值表达式。子流程活动结束时，如果某转移的 outcome-value 值与子流程的 outcome 值匹配，那么，父流程的下一步将会通过此转移。注意：这个 outcome-value 值是由一个子元素定义的

下面是一段使用 outcome 属性 - outcome-value 元素的流程定义 jPDL：

```
<process name="SubProcessDocument" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="review" />
  </start>
  <!-- 在这里定义了一个 outcome 属性，它引用变量 result -->
  <sub-process name="review" sub-process-key="SubProcessReview"
outcome="#{result}" />
```

<!-- 在这个流出转移中，定义了与 outcome 属性呼应的 outcome-value 值。即如果 outcome 值等于 99.99，则子流程结束后，父流程会通过名称为 ok 的转移，继续执行，而 nok 和 reject 两个转移则会被略过不执行。 -->


```

    <transition name="ok" to="next step">
      <outcome-value>
        <double value="99.99" />
      </outcome-value>
    </transition>
    <transition name="nok" to="update" />
    <transition name="reject" to="close" />
  </sub-process>
  <state name="next step" />
  <state name="update" />
  <state name="close" />
</process>

```

通过上例，我们发现：流程变量是父子流程用来沟通的纽带。父流程在子流程启动时将自己的“父流程变量”输入子流程，反之，子流程在结束时可以将自己的“子流程变量”返回父流程，从而实现父子流程间的数据交换。

下例的一组流程定义（父流程定义和子流程定义）将展示父子流程工作的基本方式，这包括上面所说的父子流程之间数据的输入输出。

假设这么一种场景，父流程的一个步骤是审查文档（review），这个审查文档的过程调用一个子流程实现。

图 6-15 是父流程的示意图。

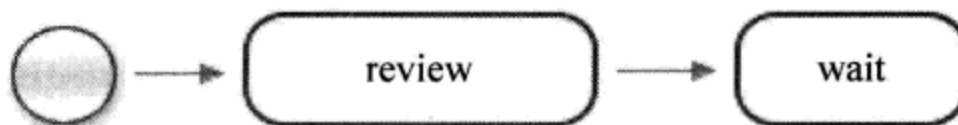


图 6-15 父流程定义的示例

对应的 jPDL:

```

<process name="SubProcessDocument" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="review" />
  </start>
  <!-- 引用标识为 SubProcessReview 的子流程定义 -->
  <sub-process name="review" sub-process-key="SubProcessReview">
    <!-- 父流程将变量 document 输入子流程，对应的子流程变量名称也为 document -->
    <parameter-in var="document" subvar="document" />
    <!-- 子流程将变量 result 返回父流程，对应的父流程变量名为 reviewResult -->
  </sub-process>
</process>

```

```

        <parameter-out var="reviewResult" subvar="result" />
        <transition to="wait" />
    </sub-process>
    <state name="wait" />
</process>

```

审查文档的流程定义如图 6-16 所示，它被上面的父流程引用为子流程。实际上，在业务中，有很多这样的可重用流程，它们可以被各种各样的其他流程所引用。

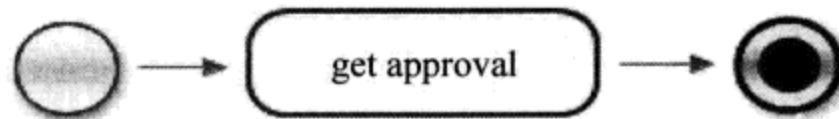


图 6-16 子流程定义（审查文档流程）的示例

对应的 jPDL:

```

<process name="SubProcessReview" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="get approval"/>
  </start>
  <!-- 子流程中的一个任务，分配给 alexmiller。alexmiller 从个人任务列表中拿到任务后，
  可以从任务中获取子流程变量。 -->
  <task name="get approval" assignee="alexmiller">
    <transition to="end"/>
  </task>
  <!-- 子流程在这里结束。将哪些子流程变量返回父流程由调用它的父流程决定。 -->
  <end name="end" />
</process>

```

我们可以编写以下单元测试代码验证这组父子流程定义。

首先，发起父流程，并赋予其一个名为 `document` 的变量：

```

//创建变量 Map，并设置 document 变量
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("document", "This document describes how we can make more
money...");
//带变量发起父流程实例
ProcessInstance processInstance = executionService
.startProcessInstanceByKey("SubProcessDocument", variables);

```

根据定义，父流程实例会立即到达子流程活动，这时一个 `SubProcessReview` 子流程实例会被创建并与父流程实例关联。`SubProcessReview` 子流程实例会立即到达 `get`

approval 任务，根据定义，这个任务被分配给了用户 alexmiller，那么我们可以：

```
//获取用户 alexmiller 的任务列表
List<Task> taskList = taskService.findPersonalTasks("alexmiller");
//获取这个 get approval 任务（假设 alexmiller 有且仅有此任务）
Task task = taskList.get(0);
```

我们在这个子流程任务中，可以拿到从父流程实例输入的 document 变量：

```
//获取任务变量
String document = (String) taskService.getVariable(task.getId(), "document");
//断言与父流程发起时设置得一致
assertEquals("This document describes how we can make more money...",
document);
```

然后我们在这个子流程任务上设置一个变量 result。在实际业务中这一般都是通过用户填写表单来完成的，这里我们使用编程方式设置：

```
Map<String, Object> variables = new HashMap<String, Object>();
//设置 result 值为 accept
variables.put("result", "accept");
//这个 result 任务变量会作为子流程变量返回父流程实例
taskService.setVariables(task.getId(), variables);
//完成这个任务。根据定义，这会结束子流程实例
taskService.completeTask(task.getId());
```

还记得父流程中的这段定义吗：`<parameter-out var="reviewResult" subvar="result" />`，当子流程实例结束后，根据这段定义，子流程中的 result 变量将被返回到父流程的 reviewResult 变量中，同时父流程实例会收到子流程结束返回的信号，然后离开 review（子流程）活动，继续下一步执行：

```
processInstance = executionService.findProcessInstanceId
(processInstance.getId());
//断言父流程通过了子流程活动，到达了下一步 wait 活动
assertNotNull(processInstance.findActiveExecutionIn("wait"));
//验证父流程的 reviewResult 变量值，即子流程的输出结果
String result = (String) executionService.getVariable(processInstance.
getId(), "reviewResult");
assertEquals("accept", result);
```

很多时候，我们仅仅想根据子流程的执行结果去影响父流程的转移，那么使用上述父子流程变量交换的方式是否显得太“重”了？所以，jBPM4 提供了更精简的两种方式控制父流程的流出转移。

1. 通过 sub-process 活动的 outcome 属性去影响父流程的流出转移

还记得本节一开始提到的 `outcome` 属性吗？在下面的例子中，我们通过设置 `outcome` 属性所引用的子流程变量值来选择父流程的流出转移。

父流程定义如图 6-17 所示。

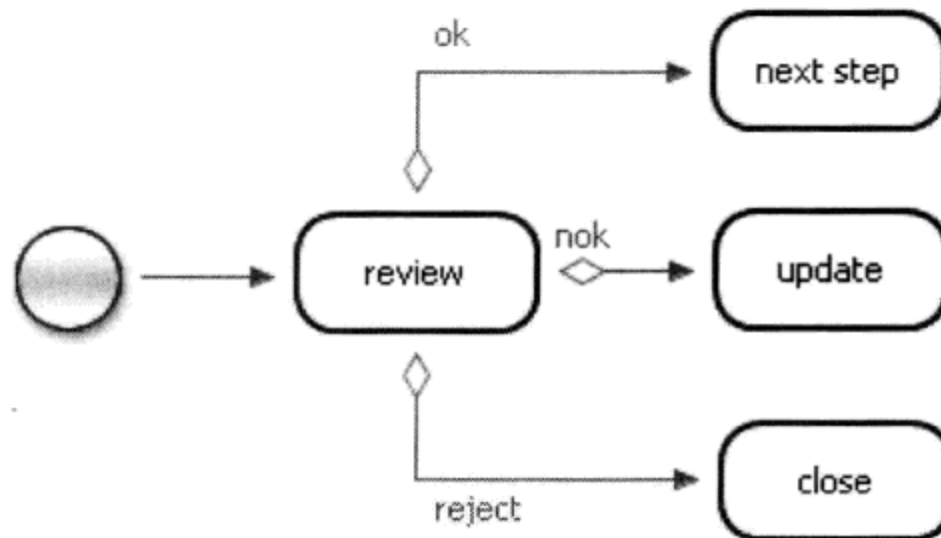


图 6-17 多个流出转移的父流程定义

对应的 jPDL:

```
<process name="SubProcessDocument">
  <start>
    <transition to="review" />
  </start>
  <!-- 子流程活动的 outcome 属性引用名称为 result 的子流程变量 -->
  <sub-process name="review" sub-process-key="SubProcessReview" outcome=
    "#{result}">
    <!-- 如果 result 值等于 ok, 则流向此转移 -->
    <transition name="ok" to="next step" />
    <!-- 如果 result 值等于 nok, 则流向此转移 -->
    <transition name="nok" to="update" />
    <!-- 如果 result 值等于 reject, 则流向此转移 -->
    <transition name="reject" to="close" />
  </sub-process>
  <state name="next step" />
  <state name="update" />
  <state name="close" />
</process>
```

引用的子流程 `SubProcessReview`（参见图 6-18）和上面介绍过的“审查文档流程”是相同的。

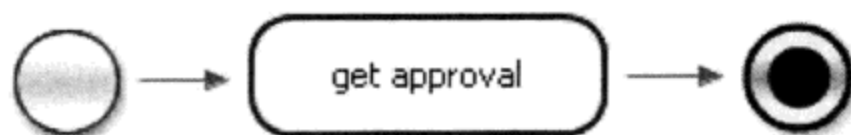


图 6-18 子流程定义（审查文档流程）的示例

对应的 jPDL:

```
<process name="SubProcessReview" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="get approval"/>
  </start>
  <task name="get approval" assignee="alexmilller">
    <transition to="end"/>
  </task>
  <end name="end" />
</process>
```

关键在编写单元测试程序模拟执行。先发起父流程实例:

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

进入子流程，任务已产生（过程不再赘述，看定义）。用户 alexmilller 拿到子流程中的任务:

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

在子流程中设置 result 变量值为 ok，这个 ok 值会被传递给 outcome 属性以决定父流程的走向。

```
Map<String, Object> variables = new HashMap<String, Object>();
//设置 result 为 ok，则根据定义 outcome 属性对应的值为 ok，即父流程将通过名称为 ok 的流
出转移
variables.put("result", "ok");
taskService.setVariables(task.getId(), variables);
//完成任务，即子流程结束返回
taskService.completeTask(task.getId());
processInstance = executionService.findProcessInstanceById
(processInstance.getId());
//回到了父流程，我们就可以断言流程通过 ok 转移到达了“next step”活动
assertNotNull(processInstance.findActiveExecutionIn("next step"));
```


2. 通过设置不同的子流程 end 活动名称自动关联父流程的流出转移

这种方法比使用 outcome 属性更容易理解：一个流程可以定义多个 end 活动，那么子流程可以定义多个不同名称的 end 活动，这些 end 活动的名称自动与父流程的流出转移名称做同名关联，名称相匹配则通过之。

看例子，父流程的定义如图 6-19 所示。

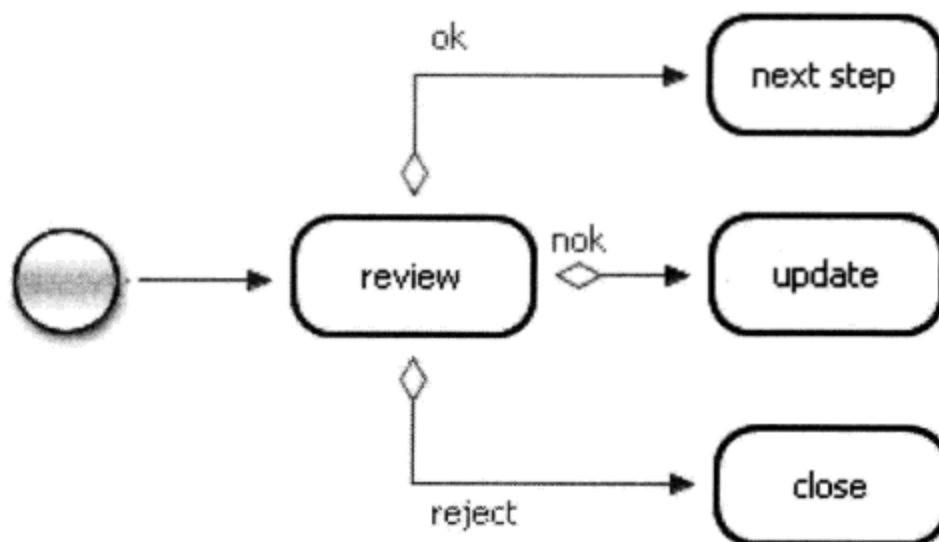


图 6-19 多个流出转移的父流程定义

对应的 jPDL:

```
<process name="SubProcessDocument">
  <start>
    <transition to="review" />
  </start>
  <!-- 在这里引用子流程，比上一例父流程定义仅少了 outcome 属性 -->
  <sub-process name="review" sub-process-key="SubProcessReview">
    <transition name="ok" to="next step" />
    <transition name="nok" to="update" />
  </sub-process>
  <transition name="reject" to="close" />
  <state name="next step" />
  <state name="update" />
  <state name="close" />
</process>
```

SubProcessReview 子流程定义了 3 个不同名称的 end 活动，分别为 ok, nok, reject, 如图 6-20 所示。

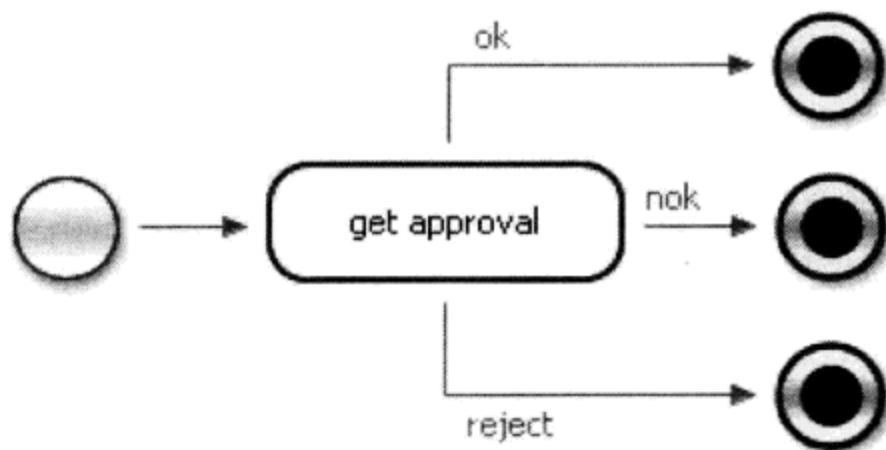


图 6-20 具有多个 end 活动的子流程定义

对应的 jPDL:

```

<process name="SubProcessReview" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="get approval"/>
  </start>
  <task name="get approval" assignee="alexmilller">
    <!-- 运行时，子流程的“get approval”任务负责在 3 条流出转移中选择其一 -->
    <transition name="ok" to="ok"/>
    <transition name="nok" to="nok"/>
    <transition name="reject" to="reject"/>
  </task>
  <!-- 以下 3 个结束活动表示：这个子流程有 3 种结束的可能 -->
  <end name="ok" />
  <end name="nok" />
  <end name="reject" />
</process>

```

编写单元测试代码验证之，先启动父流程实例：

```

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

```

用户 alexmilller 获取任务：

```

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

```

使 alexmilller 的任务流向 ok 转移，则根据定义，子流程也会在名称为“ok”的 end 活动上结束：

```

taskService.completeTask(task.getId(), "ok");

```

子流程实例结束于 ok 活动返回父流程实例，则根据名称关联机制，父流程也会自动地通过名称为 ok 的转移，进入“next step”活动等待：

```
processInstance = executionService.findProcessInstanceIdBy(
    processInstance.getId());
//在这里可以断言父流程实例到达了“next step”活动
assertNotNull(processInstance.findActiveExecutionIn("next step"));
```

6.2.8 自定义活动

在了解了上面众多活动的功能后，聪明的读者会意识到：在流程执行过程中，只要拿到了流程实例及其上下文对象，再通过某种机制获得流程定义的输入数据、发布输出数据（更灵活的活动数据输入输出完全可以交给流程变量去传递），那么自己实现上述的活动也不是什么困难的事了，因为这些活动独特的东西也就是它们的处理逻辑——深入一想，如果我有特殊而复杂的业务需求，与其生套 jBPM 本身提供的流转控制活动，不如自己实现一个自定义的活动来使用，岂不快哉？

jBPM4 提供了这样的功能！可以通过 custom 活动完全自定义一套活动行为，调用自己的代码，实现定制的活动逻辑。

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 custom 元素的定义片段，已标出支持的元素和属性供参考：

```
<element name="custom">
  <annotation>...</annotation>
  <complexType>
    <complexContent>
      <extension base="tns:wireObjectType">

        <!-- 以下是 custom 活动的元素定义。 -->
        <sequence>
          <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
          <element ref="tns:transition" minOccurs="0" maxOccurs="unbounded" />
        </sequence>

        <!-- 以下是 custom 活动的属性定义。 -->
        <attributeGroup ref="tns:activityAttributes" />

      </extension>
    </complexContent>
  </complexType>
</element>
```

您可以在 6.6 用户代码中找到 custom 活动支持的属性和元素的详细解释。在这里，我们先通过一个示例了解 jBPM4 custom 活动的应用方法。

图 6-21 所示是一个应用 custom 活动的流程定义。

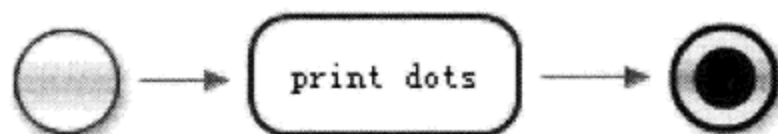


图 6-21 应用 custom 活动的流程定义

对应的 jPDL:

```
<process name="Custom" xmlns="http://jbpm.org/4.3/jpdl">
  <start >
    <transition to="print dots" />
  </start>
  <!-- 在这里定义了一个名为 "print dots" 的 custom 活动，它的逻辑实现在
  org.jbpm.examples.custom.PrintDots 类中，这个类决定了此活动的行为 -->
  <custom name="print dots" class="org.jbpm.examples.custom.PrintDots">
    <transition to="end" />
  </custom>
  <end name="end" />
</process>
```

这个 `org.jbpm.examples.custom.PrintDots` 类需要实现 `ExternalActivityBehaviour` 接口，而 `ExternalActivityBehaviour` 接口继承自 `ActivityBehaviour` 接口——`ActivityBehaviour` 接口是所有 jBPM4 内置活动都需要实现的接口。因此，我们自定义的 `PrintDots` 类共需要实现两个接口方法：

1) 来自 `ActivityBehaviour` 的 `void execute (ActivityExecution execution)`

- 这个没啥好说的了，所有的 jBPM4 活动都要实现之，在流程实例进入到此活动时执行此方法，完成主要的活动逻辑，提供 `ActivityExecution` 对象作为参数，通过它可以拿到流程实例、执行上下文等您能得到的一切流程运行时对象。

2) `void signal (ActivityExecution execution, String signalName, Map<String, ?> parameters)`

- 自定义活动需要实现的方法，在流程实例得到执行信号离开此活动时执行此方法，这个接口为您提供了更为全面的流程运行时对象，包括信号

的名称 `signalName`。在实现此方法时需要做的是：离开活动时的业务处理逻辑，以及根据 `signalName` 使流程实例通过正确的流出转移走向下一步。

下面是 `PrintDots` 类的实现代码，演示了自定义活动对流程的控制。注意到在两个方法的最后分别使用了：

- 1) `execution.waitForSignal`——等待一个执行信号使流程引擎进入 `signal` 方法的处理。
- 2) `execution.take (signalName)`——使流程继续执行，进入下一步活动。

```
public class PrintDots implements ExternalActivityBehaviour {
    public void execute(ActivityExecution execution) {
        //在这里执行您自定义的处理逻辑——“想干什么都行”
        ...
        //使流程陷入“等待”状态
        execution.waitForSignal();
        //当然您也可以调用 execution.take(signalName) 在这里自动发出执行信号，不等待而直接
        完成这个活动——一切都由您决定
    }
    public void signal(ActivityExecution execution, String signalName,
        Map<String, ?> parameters) {
        //活动收到执行信号后，进入到这里，您同样有足够的接口参数去“为所欲为”
        //当然，最后别忘了调用下面的方法使流程实例进入下一步
        execution.take(signalName);
    }
}
```

6.3 自动活动

通过上一节的介绍，我们已经了解到 `jBPM` 能很好地尽到经典工作流管理系统的传统“义务”——处理人与机器之间的交互活动。同样，`jBPM4` 也能很好支持处理多种自动活动，所谓自动活动就是在执行过程中完全无须人工干预地编排好地程序，`jBPM4` 在处理和执行这些自动活动时能把人工活动产生的数据通过流程变量等方式与之完美地结合。

`jBPM4` 默认支持的自动活动类型有：

- `java` - Java 程序活动。
- `script` - 脚本活动。

- hql - Hibernate 查询语言活动。
- sql - 结构化查询语言活动。
- mail - 邮件活动。

6.3.1 java (Java 程序活动)

java 活动可以指定一个 Java 类的方法 (Java Method)，当流程执行到此活动时，便会自动执行此 Java 方法。

如表 6-24 所示是 java 活动的属性列表，如表 6-25 所示是 java 活动支持的元素。

表 6-24 java 活动的属性

属性	类型	默认值	是否必需	描述
class	类名字符串	无	class 或 expr 二者 必选其一	带有包路径的完整 Java 类名。此 Java 类被“用户代码类加载器”加载到 JVM。此类的对象在活动执行时被“延迟创建”（注意该类需要提供无参构造方法），即不随 jBPM 工作流引擎启动而创建。当这些对象创建后，将作为流程定义的一部分被缓存
expr	表达式字符串	无	class 或 expr 二者 必选其一	此表达式返回一个 Java 类对象，含有下面指定的方法
method	方法名字符串	无	必需	调用的方法名称
var	流程变量名字 符串	无	可选	存储方法执行结果的流程变量名称

表 6-25 java 活动支持的元素

元素	个数	描述
field	0..*	在方法被调用之前给指定的类成员域注入指定的值
arg	0..*	给被调用的方法提供参数

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 java 活动的定义片段，已标出支持的元素和属性供参考：

```
<element name="java">
  <annotation>...</annotation>
  <complexType>
    <complexContent>
      <extension base="tns:javaType">

        <!-- 以下是 java 活动的元素定义。 -->
        <sequence>
          <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
          <element ref="tns:transition" minOccurs="0" maxOccurs="unbounded" />
        </sequence>

        <!-- 以下是 java 活动的属性定义。 -->
        <attributeGroup ref="tns:activityAttributes" />

      </extension>
    </complexContent>
  </complexType>
</element>
```

图 6-22 所示是应用 java 活动的流程定义示例。

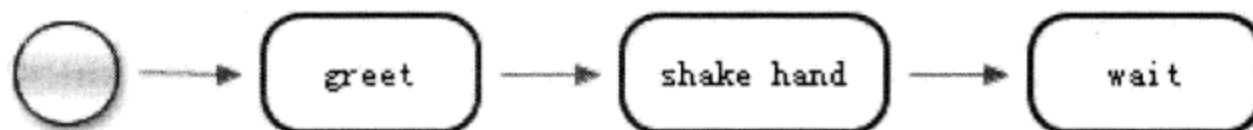


图 6-22 应用 java 活动的流程定义

对应的 jPDL:

```
<process name="Java" xmlns="http://jbpm.org/4.3/jpdl">
  <start >
    <transition to="greet" />
  </start>
  <!-- 在这里定义 java 活动，调用类 org.jbpm.examples.java.JohnDoe 对象的 hello 方法，并将执行结果存储到流程实例的 answer 变量中。 -->
  <java name="greet"
    class="org.jbpm.examples.java.JohnDoe"
    method="hello"
    var="answer">
    <!-- 为此对象的成员域 state 注入值 fine -->
```

```

<field name="state"><string value="fine"/></field>
<!-- 为调用的 hello 方法提供一个字符串参数 -->
    <arg><string value="Hi, how are you?"/></arg>
    <transition to="shake hand" />
</java>
<!-- 在这里通过表达式调用一个名称为 hand 的流程变量对象的 shake 方法, 结果保存在流程变量 hand 中 (没错, 保存在“自己中”)。 -->
    <java name="shake hand"
        expr="#{hand}"
        method="shake"
        var="hand">
<!-- 通过表达式引用流程变量, 为 shake 方法提供 2 个参数 (按先后顺序) -->
    <arg><object expr="#{joesmoe.handshakes.force}"/></arg>
    <arg><object expr="#{joesmoe.handshakes.duration}"/></arg>
    <transition to="wait" />
</java>
<state name="wait" />
</process>

```

对上面流程的解释如下: 流程定义在 greet 活动先调用了 Java 类 JohnDoe, 即在流程执行时, 类 org.jbpm.examples.java.JohnDoe 会先被实例化出一个对象, 接着这个对象的 hello 方法被调用, 并将调用的返回对象存入名为 answer 的流程变量中。

以下是类 JohnDoe 的代码:

```

public class JohnDoe {
    //此类具有 1 个成员域 state, 其在流程定义中被注入值 fine
    String state;
    // hello 方法在流程定义中被调用
    public String hello(String msg) {
        //流程定义中为此方法注入了参数 “Hi, how are you?”, 对应到 msg
        if (msg.indexOf("how are you?") != -1)
        {
            //根据流程定义, 返回的字符串应为 “I'm fine, thank you.”
            return "I'm "+state+", thank you.";
        }
        return null;
    }
}

```

流程定义在 “shake hand” 活动调用了第 2 个 Java 方法, 首先流程会处理 #{hand} 表达式, 根据此表达式获取名为 hand 的流程变量对象, 此对象具有一个名为 shake 的方法, 因此, 这个 hand 对象的类定义代码应该是这样的:

```

public class Hand implements Serializable {

```



```

private boolean isShaken;
//这就是流程定义指定要掉用的 shake 方法，它具有 2 个输入参数
public Hand shake(Integer force, Integer duration) {
    if (force>3 && duration>7) {
        isShaken = true;
    }
    return this;
}
public boolean isShaken() {
    return isShaken;
}
}

```

根据定义，在流程实例运行到“shake hand”活动前，这个 hand 对象需要被预先注入流程变量。

要调用的 shake 方法需要两个参数，这两个参数被定义为根据表达式 `#{joesmoe.handshakes.force}` 和 `#{joesmoe.handshakes.duration}` 计算得出，那么我们就需要在调用 shake 方法前设置 joesmoe 对象变量，joesmoe 对象可以具有一个名为 handshakes 的 Map 类型成员域，handshakes Map 里有 force 和 duration 键即可，所以 joesmoe 对象的类代码则可以是这样的：

```

public class JoeSmoe implements Serializable {
    static Map<String, Integer> handshakes = new HashMap<String, Integer>();
    //为成员 Map handshakes 初始化 2 个 键-值 对
    {
        handshakes.put("force", 5);
        handshakes.put("duration", 12);
    }
    //有了 getHandshakes 方法，即可在流程定义的 EL 表达式中这样引用
    #{joesmoe.handshakes}
    public Map<String, Integer> getHandshakes() {
        return handshakes;
    }
}

```

Hand 类 shake 方法的执行结果是一个 Hand 的对象，而根据 java 活动“shake hand”的属性定义 `var="hand"` 会将此结果覆盖活动执行的本体——流程变量对象 hand，这有点绕，但很有意思。

我们可以使用以下单元测试代码简单地验证此流程定义：

```

//首先，如同前面所说的，要预先设置好流程变量 hand 和 joesmoe

```

```

Map<String, Object> variables = new HashMap<String, Object>();
variables.put("hand", new Hand());
variables.put("joesmoe", new JoeSmoe());
//带着流程变量 hand 和 joesmoe 发起流程实例
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("Java", variables);
String pid = processInstance.getId();
//获取流程变量 answer
String answer = (String) executionService.getVariable(pid, "answer");
//断言定义的第 1 个 Java 活动 greet 的执行如预期
assertEquals("I'm fine, thank you.", answer);
//获取流程变量 hand
Hand hand = (Hand) executionService.getVariable(pid, "hand");
//断言定义的第 2 个 Java 活动 shake hand 的执行如预期
assertTrue(hand.isShaken());

```

6.3.2 script (脚本活动)

在上一节中我们了解到了如果需要在 jBPM 的流程编排中自动执行一段程序，可以通过调用 Java 方法的方式。但是，如果要自动实现一些简单的功能，您不觉得再编写一段 Java 代码显得太“兴师动众”了么？

script——脚本活动为我们解决了这个问题，您可以在 **script** 活动中定义一段 EL 表达式脚本，工作流引擎执行此活动时会解析这段脚本。实际上，不仅是 EL 表达式语言，任何一种符合 JSR-223 规范的脚本语言都可以在这里使用，当然您需要在脚本引擎配置文件（`jbpm.default.scriptmanager.xml`）里定义您要使用的脚本语言名称。

jBPM4 默认的脚本语言为 jUEL，即 EL 表达式语言。

以下是 XML Schema 文件 `jpd1-4.3.xsd` 中对于 **script** 元素的定义片段，已标出支持的元素和属性供参考：

```

<element name="script">
  <annotation>...</annotation>
  <complexType>
    <complexContent>
      <extension base="tns:scriptType">

        <!-- 以下是 script 活动的元素定义。 -->
        <sequence>
          <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>

```



```

        <element ref="tns:transition" minOccurs="0" maxOccurs="unbounded" />
    </sequence>

    <!-- 以下是 script 活动的属性定义。 -->
    <attributeGroup ref="tns:activityAttributes" />

    </extension>
</complexContent>
</complexType>
</element>

```

复杂类型 **scriptType** 的定义如下：

```

<complexType name="scriptType">
    <sequence>
        <element name="description" minOccurs="0" maxOccurs="unbounded"
type="string" />
        <element name="text" type="string" minOccurs="0">
            <annotation>...</annotation>
        </element>
    </sequence>
    <attribute name="expr" type="string">
        <annotation>...</annotation>
    </attribute>
    <attribute name="lang" type="string">
        <annotation>...</annotation>
    </attribute>
    <attribute name="var" type="string">
        <annotation>...</annotation>
    </attribute>
</complexType>

```

有两种方式定义脚本活动。

1. 通过脚本表达式 (script expression) 的方式

script 活动具有 **expr** 属性，可以通过 **expr** 属性定义短小简单的脚本，“短小”是相对下面将要提到的脚本文本 (script text) 方式而言的。如果没有 **lang** 属性，则会使用 jBPM4 默认的脚本表达式语言 (default-expression-language) ——jUEL。具体属性如表 6-26 所示。

表 6-26 script 活动的属性

属性	类型	默认值	是否必需	描述
expr	字符串	无	必需	需要执行的脚本表达式
lang	脚本名称字符串	jUEL	可选	指定的脚本语言
var	变量名称字符串	无	可选	脚本执行返回值存入的流程变量名

在图 6-23 所示的这个流程定义中，我们使用 script 活动执行脚本表达式，然后将脚本执行返回的结果存储在流程变量里。

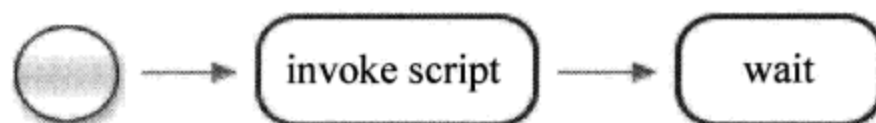


图 6-23 应用 script 活动脚本表达式的流程定义

对应的 jPDL:

```

<process name="ScriptExpression" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="invoke script" />
  </start>
  <!-- 很明显，该脚本表达式返回一个字符串值，这个值将被存储在名称为 text 的流程变量中 -->
  <script name="invoke script"
    expr="Send packet to #{order.address}"
    var="text">
    <transition to="wait" />
  </script>
  <state name="wait" />
</process>

```

上面的流程定义引用到了 Order 对象，此对象的 Java 类代码如下：

```

public class Order implements Serializable {
  // Order 类型具有 address 域，使得在脚本中使用表达式 #{order.address} 成为可能
  String address;
  public Order(String address) {
    this.address = address;
  }
}

```

```

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

```

以下单元测试代码将验证上面的流程定义：

```

Map<String, Object> variables = new HashMap<String, Object>();
//在流程变量中设置 Order 对象
variables.put("order", new Order("Berlin"));
//带着 Order 对象发起流程实例
Execution execution = executionService.startProcessInstanceByKey(
    "ScriptExpression", variables);
String executionId = execution.getId();
//获取流程变量 text。根据定义，这个变量接收了 script 活动的执行结果
String text = (String) executionService.getVariable(executionId,
"text");
//断言 text 变量的值如预期结果
assertTextPresent("Send packet to Berlin", text);

```

最终，流程变量 text 的值为“Send packet to Berlin”，与定义的脚本表达式“Send packet to #{order.address}”的执行结果一致。

2. 通过脚本文本（script text）的方式

对于脚本有多行的情况，在属性里面书写脚本表达式就不太方便了。因此，在这种情况下，我们需要采用第 2 种方式，在 script 活动的 text 元素里编写脚本。text 元素的属性如表 6-27 所示。

表 6-27 script 活动 text 元素的属性

属性	类型	默认值	是否必需	描述
lang	脚本类型名称字符串	jUEL	可选	指定脚本语言的类型
var	流程变量名称字符串	无	可选	脚本执行结果存入的流程变量名称

text 元素的内容就是脚本字符串，可以写多行，定义如表 6-28 所示。

表 6-28 text 元素的内容元素

元素	个数	描述
文本	1	脚本文本

应用脚本文本的示例流程如图 6-24 所示。

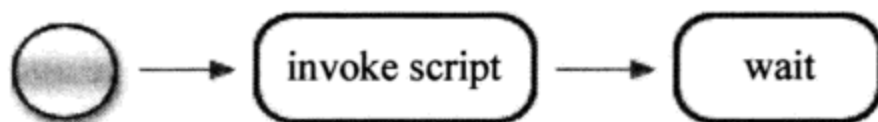


图 6-24 应用 script 活动脚本文本的流程定义

对应的 jPDL:

```

<process name="ScriptText" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="invoke script" />
  </start>
  <!-- 在这里指定脚本的计算结果存入流程变量 text -->
  <script name="invoke script" var="text">
    <!-- 在 text 元素里，可以编写多行的脚本，直接把脚本文本写入 -->
    <text>
      Send packet to #{person.address}
    </text>
    <transition to="wait" />
  </script>
  <state name="wait" />
</process>

```

细心的读者已经发现，此流程定义和上面的“脚本表达式”流程定义效果类似，只不过 Order 对象的名称被换成了 Person 而已。我们可以通过如下单元测试代码来验证上面的流程定义：

```

//为流程变量创建 Person 对象
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("person", new Person("Honolulu"));
//带着 Person 变量发起流程实例
Execution execution = executionService.startProcessInstanceByKey(
    "ScriptText", variables);
String executionId = execution.getId();
String text = (String) executionService.getVariable(executionId,
    "text");

```

```
//断言执行结果如流程定义的预期: <text>Send packet to #{person.address}
</text>
assertTextPresent("Send packet to Honolulu", text);
```

6.3.3 hql (Hibernate 查询语言活动)

在一些特殊的情况下,我们可能需要直接从持久化层读取流程数据,例如在 ETL、数据挖掘等需求中。因为 jBPM 的持久化层是基于 Hibernate 框架实现的,因此使用 hql 活动,直接面向数据库执行 HQL (Hibernate Query Language) 语言查询,并将返回的结果保存到流程变量中,是个不错的办法。

hql 是 jBPM4 支持的活动 (在 jBPM3 中没有),表 6-29 列出了它的属性。

表 6-29 hql 活动的属性

属性	类型	默认值	是否必需	描述
var	变量名称字符串	无	可选	存储 hql 执行结果的流程变量名
unique	{true,false}	false	可选	此属性值为 true 时,在查询结果上调用 uniqueResult() 方法取得 HQL 查询的唯一结果集 (Recorder Set); 此属性默认值为 false,即查询结果调用 list() 方法得到 HQL 查询的结果集列表 (Recorder Set List)

hql 活动支持如表 6-30 所示的元素。

表 6-30 hql 活动的元素

元素	个数	描述
query	1	HQL 查询语句
parameter	0..*	HQL 查询语句的外部参数

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 hql 元素的定义片段,已标出支持的元素和属性供参考:

```
<element name="hql">
```



```

<annotation>...</annotation>
<complexType>
  <complexContent>
    <extension base="tns:qlType">

      <!-- 以下是 hql 活动的元素定义。 -->
      <sequence>
        <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
        <element ref="tns:transition" minOccurs="0" maxOccurs="unbounded" />
      </sequence>

      <!-- 以下是 hql 活动的属性定义。 -->
      <attributeGroup ref="tns:activityAttributes" />

    </extension>
  </complexContent>
</complexType>
</element>

```

复杂类型 **qlType** 的定义如下:

```

<complexType name="qlType">
  <sequence>
    <element name="description" minOccurs="0" maxOccurs="unbounded" type="string" />
    <element name="query" type="string">
      <annotation>...</annotation>
    </element>
    <element name="parameters" minOccurs="0">
      <annotation>...</annotation>
      <complexType>
        <sequence>
          <group ref="tns:wireObjectGroup" maxOccurs="unbounded" />
        </sequence>
      </complexType>
    </element>
  </sequence>
  <attribute name="var" type="string">
    <annotation>...</annotation>
  </attribute>
  <attribute name="unique" type="string">
    <annotation>...</annotation>
  </attribute>
</complexType>

```

下面给出一条用到 hql 活动的流程定义，如图 6-25 所示。

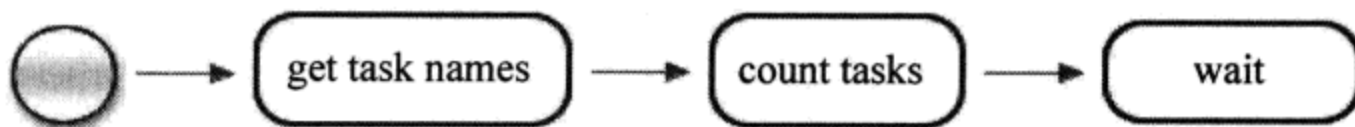


图 6-25 使用 hql 活动的流程定义

对应的 jPDL:

```
<process name="Hql" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="get task names" />
  </start>
  <!-- 此 hql 活动将查询获取名称中含有字母 "i" 的所有任务名称 -->
  <hql name="get task names" var="tasknames with i">
    <!-- 下面是一条 HQL 查询语句，得到任务名称的结果集列表 -->
    <query>
      select task.name
      from org.jbpm.pvm.internal.task.TaskImpl as task
      where task.name like :taskName
    </query>
    <parameters>
      <!-- 这里定义 HQL 语句的输入参数值为 "%i%", 这将替换 HQL 语句中的
      ":taskName" 部分 -->
      <string name="taskName" value="%i%" />
    </parameters>
    <transition to="count tasks" />
  </hql>
  <!-- 这个 hql 活动定义了 unique 属性为 true，以获得唯一结果集，即结果集列表中的第一
  笔记录 -->
  <hql name="count tasks" var="tasks" unique="true">
    <query>
      select count(*)
      from org.jbpm.pvm.internal.task.TaskImpl
    </query>
    <transition to="wait" />
  </hql>
  <state name="wait" />
</process>
```


假设当前的流程数据库中一共有 3 条任务 (task) 记录, 它们的名称 (task.name) 分别是 laundry, dishes, iron。我们可以编写如下单元测试代码来验证上述流程定义:

```
ProcessInstance processInstance = executionService.  
startProcessInstanceByKey("Hql");  
String processInstanceId = processInstance.getId();  
//很明显, 名称中含有 i 字母的任务有 dishes 和 iron, 我们在这里设定预期结果  
Set<String> expectedTaskNames = new HashSet<String>();  
expectedTaskNames.add("dishes");  
expectedTaskNames.add("iron");  
//获取第一个 hql 活动的执行结果——流程变量 "tasknames with i"  
Collection<String> taskNames = (Collection<String>)  
executionService  
    .getVariable(processInstanceId, "tasknames with i");  
taskNames = new HashSet<String>(taskNames);  
//在这里验证第一个 hql 活动的执行如预期  
assertEquals(expectedTaskNames, taskNames);  
//在下面将验证第二个 hql 活动的执行如预期  
Object activities = executionService.getVariable(processInstanceId,  
"tasks");  
//如上面所说的, 我们的流程数据库中一共有 3 条任务记录  
assertEquals("3", activities.toString());
```

6.3.4 sql (结构化查询语言活动)

与上面的 hql 活动相似, jBPM4 的 sql 活动能够支持使用 SQL——结构化查询语言直接从流程数据库中查询数据, 将结果返回到流程变量中。sql 活动与 hql 活动的属性和元素基本都相同, 唯一不同的就是 sql 活动中 query 元素的内容应该填写 SQL 语句而非 HQL 语句。

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 sql 元素的定义片段, 已标出支持的元素和属性供参考:

```
<element name="sql">  
  <annotation>...</annotation>  
  <complexType>  
    <complexContent>  
      <extension base="tns:qlType">  
  
        <!-- 以下是 sql 活动的元素定义。 -->  
        <sequence>
```

```

        <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
        <element ref="tns:transition" minOccurs="0" maxOccurs="unbounded" />
    </sequence>

    <!-- 以下是 sql 活动的属性定义。 -->
    <attributeGroup ref="tns:activityAttributes" />

</extension>
</complexContent>
</complexType>
</element>

```

qlType 的定义见上一节 hql 活动。

下面的示例流程定义应用了 sql 活动，主要在于说明以下两点：

- 1) 查询获得的数据库结果集 (RecorderSet) 以 java.lang.Object[] 对象数组的形式返回。
- 2) 查询获得的数据库结果集列表以 java.util.Collection<Object[]>接口的形式返回。

此两点不仅适用于 sql 活动，同样也适用于 hql 活动。

图 6-26 所示是使用 sql 活动的流程定义。



图 6-26 使用 sql 活动的流程定义

对应的 jPDL:

```

<process name="Sql" xmlns="http://jbpm.org/4.3/jpdl">
    <start>

```



```

        <transition to="get task objects"/>
    </start>
    <!-- 使用 SQL 语句查询，从 jBPM4 的任务表中获取所有记录集的 5 个属性，存入
taskObjects 流程变量 -->
    <sql name="get task objects" var="taskObjects">
        <query>
            select DBID_, NAME_, STATE_, PRIORITY_, DUEDATE_ from JBPM4_TASK
        </query>
        <transition name="to wait" to="wait" />
    </sql>
    <state name="wait"/>
</process>

```

同样，假设当前的流程数据库任务表中一共有 3 条任务（task）记录，它们的名称（task.name）分别是 laundry, dishes, iron。我们可以编写如下单元测试代码来验证上述流程定义：

```

    Execution execution = executionService.startProcessInstanceByKey(
("Sql");
    String executionId = execution.getId();
    //获取存储 SQL 查询结果的流程变量。正如我们前面所说的，查询得到的结果以“对象
数组列表”——Collection<Object[]>的形式返回
    Collection<Object[]> taskObjects =
        (Collection<Object[]>)
executionService.getVariable(executionId, "taskObjects");
    //断言一共有 3 条任务记录
    assertEquals(3, taskObjects.size());
    //断言每条记录含有 5 个属性
    assertEquals(5, taskObjects.iterator().next().length);

```

6.3.5 mail（邮件活动）

电子邮件是现代企业工作中不可缺少的通知和沟通方式。作为解决企业业务流程管理的利器，jBPM 必须对电子邮件活动有良好的支持。

通过 mail 活动，“流程”可以给一个或多个邮件地址发送电子邮件。电子邮件的内容可以在定义中指定，也可以根据一个“模板”生成。模板可以在 mail 活动的元素中定义，也可以在 jBPM 配置文件的 process-engine-context → mail-template 元素中定义。在模板中可以引用 jBPM 流程变量。

表 6-31 所列是 mail 活动独有的属性。

表 6-31 mail 活动的独有属性

属性	类型	默认值	是否必需	描述
template	字符串	无	否	配置文件中 mail-template 元素的名称，即引用的电子邮件模板。如果此名称没有找到，那么就使用 mail 活动的 text 或 html 元素值作为电子邮件的内容

表 6-32 所列是 mail 活动支持的元素。

表 6-32 mail 活动支持的元素

元素	个数	描述
from	0..1	发件人列表
to	1	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	电子邮件的标题
text	0..1	电子邮件的内容，纯文本形式
html	0..1	电子邮件的内容，HTML 形式
attachments	0..1	电子邮件的附件，可以引用 URL 资源、classpath 中的资源，甚至本地文件

上述这些元素也有各自的属性，这里不再详细描述。

以下是 XML Schema 文件 jpd1-4.3.xsd 中对于 mail 元素的定义片段，已标出支持的元素和属性供参考：

```

<element name="mail">
  <annotation>...</annotation>
  <complexType>
    <complexContent>
      <extension base="tns:mailType">

        <!-- 以下是 mail 活动的元素定义。 -->
        <sequence>
          <element ref="tns:on" minOccurs="0" maxOccurs="unbounded"/>
          <element ref="tns:transition" minOccurs="0" maxOccurs="

```

```

"unbounded" />
    </sequence>

    <!-- 以下是 mail 活动的属性定义。 -->
    <attributeGroup ref="tns:activityAttributes" />

    </extension>
</complexContent>
</complexType>
</element>

```

复杂类型 **mailType** 的定义如下:

```

<complexType name="mailType">
  <sequence>
    <element name="description" minOccurs="0" maxOccurs="unbounded"
type="string" />
    <element name="from" type="tns:mailRecipientType" minOccurs="0" />
    <element name="to" type="tns:mailRecipientType" minOccurs="0" />
    <element name="cc" type="tns:mailRecipientType" minOccurs="0" />
    <element name="bcc" type="tns:mailRecipientType" minOccurs="0" />
    <element name="subject" type="string" minOccurs="0" />
    <element name="text" type="string" minOccurs="0" />
    <element name="html" type="string" minOccurs="0" />
    <element name="attachments" minOccurs="0" >
      <complexType>
        <sequence>
          <element name="attachment" maxOccurs="unbounded">
            <complexType>
              <attribute name="url" type="string">
                <annotation>...</annotation>
              </attribute>
              <attribute name="resource" type="string">
                <annotation>...</annotation>
              </attribute>
              <attribute name="file" type="string">
                <annotation>...</annotation>
              </attribute>
            </complexType>
          </element>
        </sequence>
      </complexType>
    </element>
  </sequence>
  <attribute name="template" type="tns:templateType" />
</complexType>

```


图 6-27 所示是使用 mail 活动的流程定义。

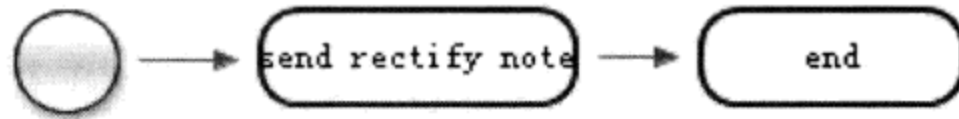


图 6-27 使用 mail 活动的流程定义

对应的 jPDL:

```
<process name="InlineMail" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="send rectify note" />
  </start>
  <mail name="send rectify note">
    <!-- 定义收件人 -->
    <to addresses="winston@minitrue" />
    <!-- 定义抄送收件人, 在这里指定了用户 bb 和整个用户组 innerparty -->
    <cc users="bb" groups="innerparty" />
    <!-- 定义密送收件人 -->
    <bcc groups="thinkpol" />
    <!-- 定义邮件标题 -->
    <subject>rectify ${newspaper}</subject>
    <!-- 以下是纯文本形式的电子邮件内容, 其中还引用了 2 个流程变量 -->
    <!-- text 元素和 html 元素在同一个 mail 活动中只能存在其一, 因此先将 text 元素注释掉
    <text>${newspaper} ${date} reporting bb dayorder doubleplusungood
    refs unpersons rewrite fullwise upsub antefiling</text>
    -->
    <!-- html 元素能为 mail 活动定义 HTML (超文本) 形式的电子邮件内容 -->
    <html>
      <table>
        <tr>
          <td>${newspaper}</td><td>${date}</td>
          <td>reporting bb dayorder doubleplusungood refs
unpersons rewrite fullwise upsub
antefiling</td>
        </tr>
      </table>
    </html>
    <!-- 为此邮件指定 3 个附件, 它们来自 Web URL、classpath、本地文件系统, 分别
    通过 url, resource, file 属性指定 -->
```



```

        <attachments>
            <attachment
url='http://www.google.com.hk/intl/zh-CN/images/logo_cn.gif' />
            <attachment resource='org/example/pic.jpg' />
            <attachment file='C:/face.zip' />
        </attachments>
        <transition to="end" />
    </mail>
    <state name="end" />
</process>

```

在 jBPM4 安装的默认配置 classpath 根目录下的 jbpm.mail.properties 文件中，可以指定 jBPM4 邮件服务器的如下属性：

- mail.smtp.host ——smtp 邮件服务器地址。
- mail.smtp.port ——smtp 邮件服务器端口。
- mail.from——默认的邮件发送人地址。

在第 16 章 深入 jBPM4 电子邮件支持中，本书将对 jBPM4 的电子邮件功能进行更进一步的介绍。

6.4 事件

jBPM 的事件监听机制“自古以来（即自 jBPM 早期的版本以来的意思）”就是其一大特色，这种设计使得我们可以很方便地在流程、活动、任务生命周期的各种阶段横向地插入定制的代码逻辑，这种基于事件-监听的设计模式具有一定的 AOP（面向切面编程，也译为面向方面编程）思想。作者认为，正是这种机制从根本上给予了 jBPM 这种面向流程编程的框架“无限”扩展的可能。在第 20 章 中国特色工作流的 jBPM 实现中，我们将利用此机制给予 jBPM4 许多具有中国业务特色的生命力，当然，此乃后话。

事件（event）用来定位在流程执行过程中特定的“点”，例如“流程实例开始”、“状态活动结束”等，可以在这些“点”注册一系列的监听器（listener）。当流程的执行通过这些被监听的点时，监听器中设定的逻辑就会被执行。

事件和监听器不会显示在流程定义的图形中，这是因为它们更注重灵活的内部逻辑而不适合用图形表示。正如前面所说的：事件来源于流程定义中的元素，例如流程定义、活动定义或转移定义，以及这些元素在执行过程中的生命周期“点”。

编写一个事件监听器需要实现 `EventListener` 接口：

```
public interface EventListener extends Serializable {  
    //接口方法提供了流程的执行对象 execution, 这足够拿到当前流程的任何信息  
    void notify(EventListenerExecution execution) throws Exception;  
}
```

既然事件监听器就是一段“自动的”流程处理逻辑，那么所有的自动活动都可以作为事件监听器被用来监听事件喽？是的！您可以使用 `java`, `script`, `hql` 等所有自动活动来处理监听到的事件，如果您不想自己实现 `EventListener` 接口的话。

那么，接下来的问题就是如何定义事件了？为了给流程或活动分配一系列的事件监听器，可以使用 `on` 元素来为事件监听器分组并指定事件，`on` 元素可以嵌入到 `process` 元素或 `process` 元素下的任何流程活动中。表 6-33 所列是 `on` 元素的属性。

表 6-33 `on` 元素的属性

属性	类型	默认值	是否必需	描述
<code>event</code>	<code>{start end}</code>	无	必需	事件的名称

`on` 元素的 `event` 属性目前只支持“`start` - 开始事件”和“`end` - 结束事件”的监听（在 `jBPM3` 中支持的事件类型要多得多，但是也比较混乱）。

对于转移的执行事件（`transition take`），只需直接在 `transition` 元素嵌入相应的事件监听器即可。

表 6-34 所列是 `on` 元素支持的子元素，事件监听器就在这里定义。

表 6-34 `on` 元素支持的子元素

元素	个数	描述
<code>event-listener</code>	<code>0..*</code>	指定自定义的事件监听器，实现 <code>EventListener</code> 接口
任何自动活动	<code>0..*</code>	使用自动活动（ <code>java</code> , <code>script</code> , <code>hql</code> ...）作为事件监听器

自定义的事件监听器 `event-listener` 元素指向一段用户编写的代码，所以它具有像 6.6 用户代码中所描述的用户代码活动的通用属性。而所有的事件监听器，包括自动活动作为事件监听器，都具有如表 6-35 所示得独特的属性。

表 6-35 事件监听器的独特属性

属性	类型	默认值	是否必需	描述
propagation	{enabled disabled true false on off}	disabled	可选	指定该事件监听器是否支持被传播的事件调用
continue	{sync async exclusive}	sync	可选	指定事件监听器是否不阻断流程而异步执行。可以参考 6.5 异步执行

6.4.1 事件监听

图 6-28 所示是一个使用事件监听器的流程定义。

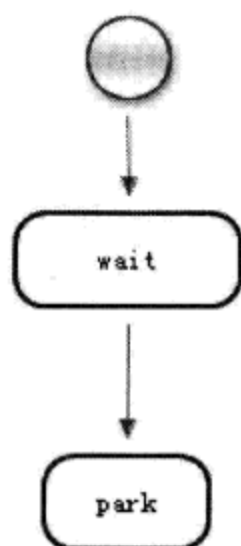


图 6-28 使用事件监听器的流程定义

对应的 jPDL:

```

<process name="EventListener" xmlns="http://jbpm.org/4.3/jpdl">
  <!-- 流程全局的事件监听。在这里捕获流程实例启动的事件 (event="start"), 由
  LogListener 监听器处理之 -->
  <on event="start">

```

```

<event-listener class="org.jbpm.examples.eventlistener.LogListener">
  <!-- 为此事件监听器注入域 msg 的值 -->
  <field name="msg">
<string value="start on process definition"/>
</field>
  </event-listener>
</on>
<start>
  <transition to="wait"/>
</start>
<state name="wait">
  <!-- 在这里捕获 wait 活动开始 (event="start") 的事件, 同样使用 LogListener 来处理 -->
  <on event="start">
    <event-listener class="org.jbpm.examples.eventlistener.LogListener">
      <field name="msg"><string value="start on activity wait"/></field>
    </event-listener>
  </on>
  <!-- 在这里捕获 wait 活动结束 (event="end") 的事件, 同样使用 LogListener 来处理 -->
  <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">
    <!-- 在这里捕获 wait 活动通过其流出转移的事件, 也使用 LogListener 来处理 -->
    <event-listener class="org.jbpm.examples.eventlistener.LogListener">
      <field name="msg"><string value="take transition"/></field>
    </event-listener>
  </transition>
</state>
<state name="park"/>
</process>

```

在这里我们将事件监听器 LogListener 的业务逻辑定义为维护流程运行的日志, 并保存在流程变量中。实现代码如下:

```

//事件监听器必须要实现 EventListener 接口
public class LogListener implements EventListener {
  //还记得吗, 在上面的流程定义中, 我们每次都为这个域注入值, 以描述日志内容
  String msg;
  public void notify(EventListenerExecution execution) {

```



```
//通过 execution 接口对象，我们可以获取所有流程变量
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);
}
```

编写单元测试程序验证此流程定义，首先我们需要发起流程实例：

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

然后流程实例将自动执行到 wait 活动并陷入等待状态，所以我们提供一个执行信号，使流程离开等待状态，完成执行。

```
Execution execution = processInstance.findActiveExecutionIn("wait");
//发出执行信号，离开 wait 活动
executionService.signalExecutionById(execution.getId());
```

那么，可以预期，在流程变量 logs 中的日志消息列表打印出来将会是这样的：

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

根据我们定义的事件监听器逻辑，上述日志消息都是在流程定义中注入 LogListener 的 msg 域的字符串。

6.4.2 事件传播

jBPM 的事件触发是支持从父元素传播到子元素的。

默认情况下，事件监听器（event-listener）只对其当前订阅的元素所触发的事件起作用，即 propagation="false"。但是，通过指定事件监听器的传播属性 propagation="enabled"（或 propagation="true"亦可），则该事件监听器可以对其监听元素的所有子元素起作用。

图 6-29 所示是一个应用事件传播机制的流程定义。

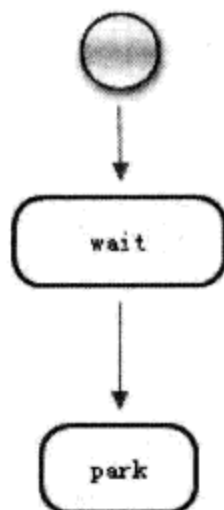


图 6-29 应用事件传播机制的流程定义

对应的 jPDL:

```

<process name="EventListenerPropagation" xmlns="http://jbpm.org/4.3/jpdl">
  <!-- 在这里定义一个流程全局的事件，监听 start 事件 -->
  <on event="start">
    <!-- 该事件监听器支持传播，因为定义在全局，也即所有活动的 start 事件都将被其捕获 -->
    <event-listener propagation="enabled"
      class="org.jbpm.examples.eventlistener.propagation.LogListener">
      <field name="msg">
        <string value="start on "/>
      </field>
    </event-listener>
  </on>
  <start>
    <transition to="wait"/>
  </start>
  <!-- 可以预期，wait 活动的 start 事件将被捕获 -->
  <state name="wait">
    <transition to="park"/>
  </state>
  <!-- 可以预期，park 活动的 start 事件将被捕获 -->
  <state name="park"/>
</process>

```

该事件监听器的实现代码如下：

```

public class LogListener implements EventListener {
  // 通过流程定义注入的 msg 域在这里定义

```

```

String msg;
public void notify(EventListenerExecution execution) {
    //获取流程变量 logs, 用来记录跟踪日志
    List<String> logs = (List<String>) execution.getVariable("logs");
    if (logs == null) {
        logs = new ArrayList<String>();
        execution.setVariable("logs", logs);
    }
    String actName = ((ExecutionImpl) execution).getActivityName();
    //注意这里: 结合流程定义, 该事件监听器被定义在流程全局, 因此它首先将监听到流程
    实例的 start 事件, 而在此事件中, 活动名称 (actName) 为 null, 所以我们在这里要做相应的
    处理——利用 process 替换
    logs.add(msg + ((actName == null) ? "process" : actName));
    execution.setVariable("logs", logs);
}
}

```

编写单元测试代码验证这个应用事件传播的流程定义:

```

//首先发起流程实例
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("EventListenerPropagation");
Execution execution =
processInstance.findActiveExecutionIn("wait");
//发出执行信号, 通过 wait 活动
executionService.signalExecutionById(execution.getId());
//根据事件传播定义, 在这里可以设置预期: 流程实例的执行先后触发了 process 流程
全局、wait 活动、partk 活动 3 个 start 事件
List<String> expectedLogs = new ArrayList<String>();
expectedLogs.add("start on process");
expectedLogs.add("start on wait");
expectedLogs.add("start on park");
//获取流程变量 logs, 日志信息都存在里面
List<String> logs = (List<String>) executionService.getVariable
(processInstance.getId(), "logs");
//断言是 3 条日志信息
assertEquals(3, logs.size());
//断言日志结果如预期
assertEquals(expectedLogs, logs);

```

6.4.3 处理异常事件

在流程的执行过程中, 不可避免地会产生业务或系统的异常情况, 该如何处理呢?

这要先从 jBPM3 版本说起, jBPM3 有一个内置的异常处理器定义, 能够被应用在

单一活动或者整个流程范围，可以这样定义：

```
<!-- 捕获所有的 java.lang.Exception, 然后交给 com.sample.handlers.  
BPMEExceptionHandler 处理 -->  
<exception-handler exception-class = "java.lang.Exception">  
<action class = "com.sample.handlers.BPMEExceptionHandler" />  
</exception-handler>
```

有一定经验的开发者或许曾经被诱惑在 jBPM3 中使用这样的异常处理机制来控制流程的执行，因为这很像 Java 语言的异常处理机制。

请不要这样做！jBPM 的异常并不完全与 Java 语言的异常处理相似，在 Java 中，一个被捕获的异常能够对程序控制流产生影响，例如中断整个程序的执行；而在 jBPM 中，业务控制流不会被 jBPM 的异常改变，异常可以被捕获也可以不捕获——不捕获的异常被扔给客户端（例如调用 `token.signal` 或 `signalExecutionXxxx` 方法的客户端处理程序），如果异常被 jBPM3 的异常处理器捕获，流程引擎则会当做没有异常发生而继续执行。

“幸运”的是，在 jBPM4 版本中，完全废除了 `exception-handler` 这种机制。无论在 jBPM3 或者 jBPM4 中处理异常的最佳实践是——使用事件监听器。对于 jBPM4 来说就是：监听那些您预期可能发生异常的事件（event），为它们设置 `event-listener`，在 `event-listener` 中您可以做设置变量标识异常、发送邮件报告异常、利用 JMS 消息发布异常等处理，然后或者继续让流程实例执行您期望的行为（对异常作出反应），或者为了使事务失败而重新抛出异常，并且做出一些“补偿”的操作使流程实例回到异常发生前的状态。

提示：在事件监听器中捕获业务异常并且设置一些流程变量来标识是一个很好的设计习惯，然后您可以在流程定义中设置若干 `decision` 活动（判断活动），用来选择一个处理异常的转移路径。

6.5 异步执行

jBPM 对于所有的流程执行操作默认都是同步的。也就是说，每当您调用 `ExecutionService.startProcessInstanceById` 或 `ExecutionService.signalProcessInstanceById` 之类的方法时，会让流程的执行在您发起的请求线程中“同步”完成，即上述方法会使得该流程实例一直处于执行状态，直到流程实例到达一个“等待状态”后才能返回

(中断执行)。

这种行为之所以为默认，是因为它有很多优点，以下列举两点：

- 用户业务系统的事务可以很容易地接入 jBPM，这样 jBPM 的数据库持久化操作就可以在用户业务系统的事务环境中完成。
- 当流程的执行过程中某些操作出现错误的时候，客户端可以立刻获得一个异常。

一般地，在流程执行的两个等待状态之间（即若干自动活动）需要完成的工作都是耗时比较短的，所以在大多数情况下，我们希望是在一个事务中执行所有这些工作。这也就解释了 jPDL 定义的默认行为： workflow 引擎会在客户端发起的线程中同步执行流程的所有工作，直到陷入等待状态或流程结束。

但是在另外一些情况下，如果在两个等待状态之间有长时间的自动活动组群或者其他您不想等待的自动活动情况下，您需要流程立即返回从而执行下一步，那么 jPDL4 提供了异步执行流程的定义方式满足您的需求。

jBPM4 workflow 引擎在事务边界上对异步执行进行了良好的控制，流程一旦进入异步执行方式，一个异步消息会被作为当前事务的一部分发送出去，然后当前事务会立即自动提交，服务方法调用（诸如 `ExecutionService.startProcessInstanceId` 或 `ExecutionService.signalProcessInstanceId`）会直接返回，同时 workflow 引擎会为此异步执行启动一个新事务。最后，jBPM workflow 引擎会使用“异步消息”来通知您（或客户端应用程序）完成了异步执行工作（Job）。

几乎所有的活动都支持异步属性，活动的异步属性如表 6-36 所示。

表 6-36 活动的异步属性

属性	类型	默认值	是否必需	描述
continue	{sync async exclusive}	sync	可选	指定活动的异步属性

continue 属性的值用来标识活动是否使用异步执行，它有 3 种情况：

- sync（默认值）——同步执行，使用当前事务。
- async——启动异步执行。当前事务被自动提交，活动在一个新事务中执行。 workflow 引擎使用“异步消息”来通知您完成异步执行工作（Job），此异步消息的产生被包含在这个新的事务中。

- **exclusive**——启动独占模式的异步执行。当前事务被自动提交，活动在一个新事务中执行。工作流引擎使用“异步消息”来通知您完成异步执行工作（Job）。与上面不同的是，此异步消息具有独占性，jBPM 工作流引擎会为每一个异步消息启动一个新的事务，以确保同一个流程实例中的每个“异步 Job”都是绝对排他的、非同时被执行，即使您在 jBPM 中配置了在不同系统中运行的多个异步消息执行器（例如 jobExecutor）。使用这个属性可以用来防止在异步多并发执行情况下的事务乐观锁失败，即防止有潜在冲突（例如资源互锁）可能的异步 Job 被安排在同一个事务中，当然，使用此属性，相应的系统开销也会加大。这个属性对于在服务器集群环境中使用异步执行机制至关重要。

6.5.1 异步活动

图 6-30 所示是一个简单的应用异步活动的流程定义示例。

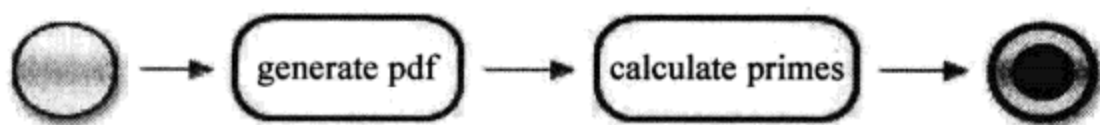


图 6-30 应用异步活动的流程定义

对应的 jPDL:

```
<process name="AsyncActivity" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="generate pdf"/>
  </start>
  <!-- 对于一些大型文档来说，即使是自动生成 PDF，也比较消耗时间，因此我们有把流程中诸如此类的自动活动定义为异步执行的需求：continue="async"。 -->
  <java name="generate pdf"
        continue="async"
        class="org.jbpm.examples.async.activity.Application"
        method="generatePdf" >
    <transition to="calculate primes"/>
  </java>
  <!-- 如果方法 org.jbpm.examples.async.activity.Application.calculatePrimes 也很消耗时间，为什么不把它也定义成异步的自动活动呢。 -->
  <java name="calculate primes"
        continue="async"
```

```

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

```

如果设置异步执行属性 `continue="sync"`（同步方式），此流程定义将是一个完全自动执行的过程，流程会在实例发起后从头执行到尾而无须任何干预。

而根据上面的流程定义，在设置了异步执行属性后，流程只会执行到“generate pdf”活动，然后一个异步消息会产生，发起流程实例的方法（线程）会立即返回……

根据定义，我们先模拟出“比较耗时”的 `generatePdf` 和 `calculatePrimes` 方法：

```

public class Application implements Serializable {
    public void generatePdf() throws InterruptedException {
        //假设此方法执行需要消耗较长的时间，在这里模拟为 1 秒钟
        Thread.sleep(1000);
    }
    public void calculatePrimes() throws InterruptedException {
        //假设此方法执行需要消耗较长的时间，在这里模拟为 2 秒钟
        Thread.sleep(2000);
    }
}

```

编写单元测试程序验证此流程定义。需要注意的是：在生产环境的配置中，`job-executor`（工作执行器，默认的在 `jbpm.jobexecutor.cfg.xml` 配置文件中指定）会自动获得异步消息并执行它。但在验证测试中，我们希望控制这些异步消息什么时候被执行，所以没有设置 `job-executor`，因此我们可以使用此方法 `ManagementService.executeJob` 来“手工”执行异步消息。

所有的代码如下：

```

//发起流程实例
ProcessInstance processInstance = executionService.
startProcessInstanceByKey("AsyncActivity");
String processInstanceId = processInstance.getId();
//根据定义，在这里断言流程实例处于异步执行状态
assertEquals(Execution.STATE_ASYNC, processInstance.getState());
//获取此流程实例异步消息队列中的第 1 条消息 (Job)
Job job = managementService.createJobQuery().processInstanceId(
    processInstanceId).uniqueResult();
//手工执行异步消息

```



```

managementService.executeJob(job.getId());
processInstance = executionService.findProcessInstanceById
(processInstanceId);
//由于还有一条异步消息，所以断言流程实例仍然处于异步执行状态
assertEquals(Execution.STATE_ASYNC, processInstance.getState());
//获取第2条消息(Job)并执行之
job = managementService.createJobQuery().processInstanceId
(processInstanceId).uniqueResult();
managementService.executeJob(job.getId());
//这时候，方可断言流程实例已经结束
assertNull(executionService.findProcessInstanceById(processInstanceId));

```

6.5.2 异步分支/聚合

以下是稍微复杂的异步流程应用。这里我们定义一个异步执行的分支/聚合流程，可以说在这个定义中，异步执行是并行进行的，如图 6-31 所示。

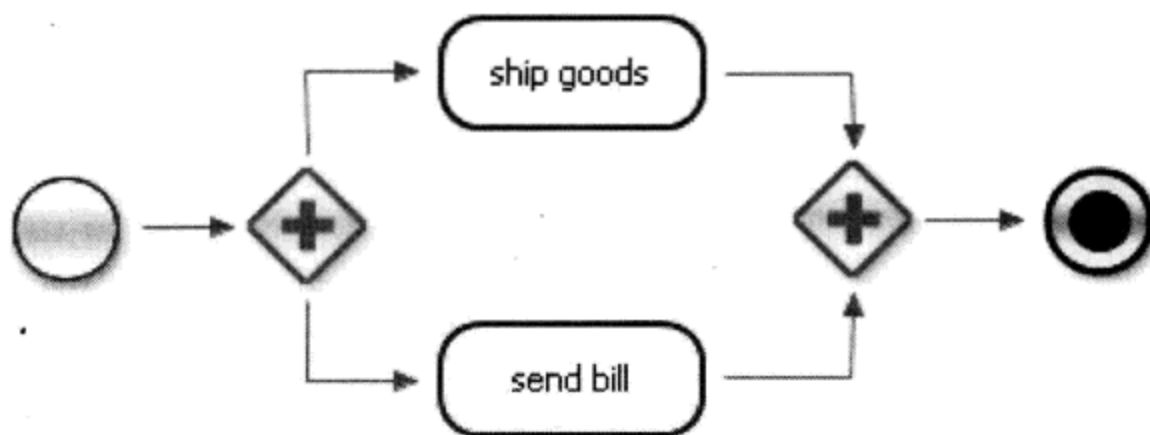


图 6-31 异步分支/聚合的流程定义

对应的 jPDL:

```

<process name="AsyncFork" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="fork" />
  </start>
  <fork name="fork">
    <!-- 在这里定义：并行的流程分支以独占方式 异步执行 -->
    <on event="end" continue="exclusive" />
    <transition to="ship goods" />
    <transition to="send bill" />
  </fork>
  <!-- 以下是两个典型的自动执行 Java 方法的活动： -->

```



```

<java class="org.jbpm.examples.async.fork.Application"
    method="shipGoods" name="ship goods">
    <transition to="join" />
</java>
<java class="org.jbpm.examples.async.fork.Application"
    method="sendBill" name="send bill">
    <transition to="join" />
</java>
<!-- 流程在这里聚合, 结束 -->
<join name="join">
    <transition to="end" />
</join>
<end name="end" />
</process>

```

两个 java 自动活动所依赖的代码:

```

public class Application implements Serializable {
    //假设 shipGoods 耗时 1 秒
    public void shipGoods() throws InterruptedException {
        Thread.sleep(1000);
    }
    //假设 sendBill 耗时 1 秒
    public void sendBill() throws InterruptedException {
        Thread.sleep(2000);
    }
}

```

根据流程定义, 通过在 fork 活动的 end 事件上设置异步调用 `<on event="end" continue="exclusive" />`, 将导致所有的两个分支转移指向的活动都会使用独占方式异步执行。

注意这里设置的 exclusive 属性值, 这将导致两个异步消息的 Job 对象分别进行持久化, 分别在各自的独立事务中执行“ship goods”和“send bill”活动, 然后都会聚合达到 join 活动。在 join 活动聚合时, 如果是非独占方式的异步执行, 两个事务有可能会同时操作同一个 jBPM4 执行, 即在数据库中同时读写同一个 execution 对象, 这可能导致一个潜在的乐观锁失败。

以下单元测试代码可以验证此流程定义:

```

ProcessInstance processInstance = executionService.
startProcessInstanceByKey("AsyncFork");
String processInstanceId = processInstance.getId();
//获取异步消息的列表

```

```

List<Job> jobs = managementService.createJobQuery()
    .processInstanceId(processInstanceId).list();
//有两个分支，所以可以断言有两条异步消息
assertEquals(2, jobs.size());
Job job = jobs.get(0);
//手工执行其一
managementService.executeJob(job.getId());
job = jobs.get(1);
//手工执行其二
managementService.executeJob(job.getId());
//从历史流程库中获取此流程实例的结束时间
Date endTime = historyService
    .createHistoryProcessInstanceQuery()
    .processInstanceId(processInstance.getId()).uniqueResult()
    .getEndTime();
//断言结束时间非空，即证明该流程实例已经结束
assertNotNull(endTime);

```

6.6 用户代码

事实上，关于用户代码（User Code），我们在前面的章节中已经多次用到了。所谓用户代码，就是 jBPM 提供接口，由用户根据自己的业务逻辑需要去实现，然后在 jPDL 中定义引用，工作流引擎在执行的时候调用。

在这里总结一下，我们介绍过的用户代码：

- **custom**——自定义活动，需要实现 `ExternalActivityBehaviour` 接口。
- **event-listener**——事件监听器，需要实现 `EventListener` 接口。
- **task** 人工任务活动中的 *assignment-handler* 元素——自定义任务分配处理器，需要实现 `AssignmentHandler` 接口。
- **decision** 判断活动中的 *handler* 元素——自定义判断逻辑处理器，需要实现 `DecisionHandler` 接口。

本节将总结所有这些用户代码定义的通用属性和元素，以及对这些用户代码进行初始化定义和配置注入的技巧。

6.6.1 用户代码的定义

用户代码的通用属性如表 6-37 所示。

表 6-37 用户代码的通用属性

属性	类型	默认值	是否必需	描述
class	全类名	无	{class expr} 必须二选其一	引用的全 Java 类名（包括包名）。对于每个流程定义初始化只会进行一次，即调用构造方法。初始化生成的对象会被作为流程定义的一部分在执行环境中缓存
expr	表达式	无	{class expr} 必须二选其一	表达式的值会被作为对象返回。表达式会在每次使用时被计算执行，即表达式的执行结果值不会被缓存

用户代码的通用元素如表 6-38 所示。

表 6-38 用户代码的通用元素

元素	数目	描述
field	0..*	描述一个 Java 成员域的值，在用户代码对象被使用之前直接注入（Autoware 机制）到该成员域中
property	0..*	描述一个指定名称的 Java 值，在用户代码对象被使用之前通过相应的 setter 方法注入

为了注入值，我们首先需要在 field 或 property 元素中指定成员域或 setter 方法目标的名称，这需要利用到如表 6-39 所示的属性。

表 6-39 field 和 property 元素的属性

属性	类型	默认值	是否必需	描述
name	名称字符串	无	必需	field 对应的成员域的名称，或 property 对应的 setter 方法目标的名称

指定了要注入的名称，那么接下来就是把要注入的值表示出来了。field 和 property 元素都可以通过设置如表 6-40 所示的子元素，来表示将被注入的值。

表 6-40 用来表示值的 field 和 property 元素的子元素

元素	数目	描述
string	0..1	表示一个 java.lang.String 对象
int	0..1	表示一个 java.lang.Integer 对象
long	0..1	表示一个 java.lang.Long 对象
float	0..1	表示一个 java.lang.Float 对象
double	0..1	表示一个 java.lang.Double 对象
true	0..1	表示一个 Boolean.TRUE 对象
false	0..1	表示一个 Boolean.FALSE 对象
object	0..1	表示任意对象，但此对象的类型需要支持类反射初始化

以上属于“基本表示类型”的元素有 string, int, long, float, double, 它们需要通过如表 6-41 所示的属性来指定具体的值。

表 6-41 string, int, long, float, double 元素的属性

属性	类型	默认值	是否必需	描述
value	文本	无	必需	在程序执行中，value 属性的文本值会被强制转化成对应的 Java 对象。因此在定义时要注意类型的匹配，例如不能将 long 元素的 value 属性设置为“Tom”

6.6.2 用户代码的类加载

我们把流程定义部署到数据库中，同样，作为流程定义的一部分，我们“定义”的用户代码也会被持久化在数据库中。

没错，jBPM 会把用户代码的 Java Class 文件直接存储。

当流程实例（Process Instance）被发起时，相应的流程定义（Process Define）会被缓存，默认情况下，如同流程定义一样，流程定义的所有用户代码对象也会被视为流程定义的一部分而被缓存。对于所有通过 Java 类名引用而生成的用户代码对象，都会在流程定义解析期间被初始化，这就意味着：这些用户代码对象不能保存具有状态的、可以“动态”改变的数据例如下面的事件监听器用户代码：


```
public class TestListener implements EventListener {
    //这个成员域的值不能保存，因为每次 TestListener 对象初始化的时候都会清空它！
    private String testField;
    ...
}
```

但是如果在流程定义中“静态”地为此事件监听器指定该成员域的值，则没有问题：

```
<process name="TestProcess" xmlns="http://jbpm.org/4.3/jpdl">
    ...
    <task assignee="Alex" name="leader audit">
        <on event="start">
            <event-listener class="com.examples.TestListener">
                <field name="testField">
                    <string value="data here" />
                </field>
            </event-listener>
        </on>
        <transition to="end" />
    </task>
    ...
</process>
```

从上例可以看到，用户代码对象一般是不能改变的。如果确实需要在用户代码中注入“动态”的数据，可以在定义中引用流程变量，就像这样：

```
<process name="TestProcess" xmlns="http://jbpm.org/4.3/jpdl">
    ...
    <task assignee="Alex" name="leader audit">
        <on event="start">
            <event-listener class="com.examples.TestListener">
                <field name="testField">
                    <string value="#{variable.name}" />
                </field>
            </event-listener>
        </on>
        <transition to="end" />
    </task>
    ...
</process>
```

流程变量通过 EL 表达式指定，而根据我们前面介绍的，表达式中引用的对象是支持动态计算得出的。

6.7 小结

本章介绍了 jPDL4.3 版本稳定支持的所有流程活动类型及用户代码元素，并逐一辅以实例详细说明。读者掌握本章的内容后，应该对 jBPM4 能解决何种业务问题成竹在胸了，并且能够独立设计和开发出一般的 jBPM4 应用程序。

下一章，将对流程运行中最重要的数据交换手段——流程变量的应用做详细的介绍。



通过前面的介绍，我们了解到如果需要在流程运行过程中与流程实例绑定或存储一些用户定义的、可以动态改变的数据，只能通过使用流程变量这种机制。

流程变量必然与流程实例（或属于流程实例的活动实例等）绑定，因此我们可以通过流程实例的执行服务 `ExecutionService`，使用如下方法操作流程变量：

- `ProcessInstance startProcessInstanceId (String processDefinitionId, Map<String, Object> variables)`
 - 在通过流程定义 ID 发起流程实例时，设置流程变量。
- `ProcessInstance startProcessInstanceId (String processDefinitionId, Map<String, Object> variables, String processInstanceKey)`
 - 在通过流程定义 ID 发起流程实例时，设置流程变量和流程实例业务键。
- `ProcessInstance startProcessInstanceByKey (String processDefinitionKey, Map<String, ?> variables)`
 - 在通过流程定义唯一键（可以指定流程定义的版本）发起流程实例时，设置流程变量。
- `ProcessInstance startProcessInstanceByKey (String processDefinitionKey, Map<String, ?> variables, String processInstanceKey)`
 - 在通过流程定义唯一键（可以指定流程定义的版本）发起流程实例时，设置流程变量和流程实例业务键。

`ExecutionService` 还提供如下方法操作具体的流程变量，在其他引擎服务中也存在类似的方法，例如 `TaskService` 也提供这些方法，只不过是操作了任务绑定的流程变量，相应的参数 `executionId` 就换为 `taskId`。

- `void setVariable (String executionId, String name, Object value)`
 - 为流程实例设置一个变量，指定变量名称和变量值。

- **void setVariables(String executionId, Map<String, ?> variables)**
 - 为流程实例设置一批变量，以变量 Map 的形式提供。
- **Object getVariable(String executionId, String variableName)**
 - 获取流程实例的一个指定名称的变量。
- **Set<String> getVariableNames(String executionId)**
 - 获取流程实例的所有变量，以 Set 数据集合的形式返回。
- **Map<String, Object> getVariables(String executionId, Set<String> variableNames)**
 - 提供一个变量名称集合，返回指定流程实例在此名称集合中的所有变量，以变量 Map 的形式返回。

流程运行中提供给用户代码的 **Execution** 接口对象，例如 **ActivityExecution** 或 **EventListenerExecution** 等，都有如下方法操作流程（及活动）变量：

- **Object getVariable(String key)**
 - 通过键（即名称）获取变量。
- **void setVariables(Map<String, ?> variables)**
 - 设置一批变量。
- **boolean hasVariable(String key)**
 - 该 **Execution** 对象是否具有此名称的变量。
- **boolean removeVariable(String key)**
 - 删除一个变量。
- **void removeVariables()**
 - 清除所有的变量。
- **boolean hasVariables()**
 - 该 **Execution** 对象是否具有变量。
- **Set<String> getVariableKeys()**
 - 获取该 **Execution** 对象所有变量名称的集合。
- **Map<String, Object> getVariables()**
 - 获取该 **Execution** 对象的所有变量。

- void createVariable(String key, Object value)
 - 为该 Execution 对象创建一个变量。
- void createVariable(String key, Object value, String typeName)
 - 为该 Execution 对象创建一个指定类型的变量。

注意：jBPM4 不能自动检测和持久化变量值的变化。如果从流程实例中获取了一个变量 Map，并修改了其中的变量元素，那么就需要调用相关持久化 API 把变化了的流程实例及其变量保存到数据库中。

7.1 变量作用域

正如我们前面所提到过的，变量是有可视范围的，即变量具有作用域（Scope）。

默认情况下，变量都被创建在顶级的流程实例作用域中，即被称做流程变量。作为顶级的流程变量意味着它们对整个流程实例中的所有 Execution 接口对象都是可见、可访问的。在 jBPM4 中，流程变量都是动态创建的，这意味着：**第一次访问此流程变量即创建之。**

每个 Execution 接口对象的范围都是一个变量作用域。在流程实例内包含的子 Execution 接口对象中，都可以“看到”该 Execution 自己的变量以及其上级 Execution 中的变量。正如我们所知，流程实例也是 Execution 对象，并且是顶级的，所以它的子 Execution 对象都能“看见”流程实例的变量。

jBPM4 在未来的发布版本中，可能会添加在 jPDL 流程定义语言中声明变量的功能。

提示：使用 Execution 接口对象的 createVariable 方法，可以为 ActivityExecution 或 EventListenerExecution 等子 Execution 对象创建在流程实例级别看不见的“局部变量”。例如活动变量、任务变量等。

7.2 变量类型

jBPM4 支持如表 7-1 所示 Java 类型的流程变量。

表 7-1 jBPM4 支持的变量类型

类型	描述
<code>java.lang.String</code>	字符串
<code>java.lang.Long</code>	长整型
<code>java.lang.Double</code>	双精度数字
<code>java.util.Date</code>	日期
<code>java.lang.Boolean</code>	布尔
<code>java.lang.Character</code>	字符
<code>java.lang.Byte</code>	字节
<code>java.lang.Short</code>	短整型
<code>java.lang.Integer</code>	整型
<code>java.lang.Float</code>	浮点
<code>byte[]</code> (byte array)	字节数组
<code>char[]</code> (char array)	字符数组
hibernate entity with a long id	具有长整型主键的 Hibernate 实体对象
hibernate entity with a string id	具有字符串型主键的 Hibernate 实体对象
serializable	所有可序列化的 Java 对象

事实上,您会发现,最后一个类型 `serializable` 可以包含上述所有类型。因此,jBPM4 为了高效的持久化变量,会按照上表的类型顺序对变量进行匹配,第一个匹配成功的类型,将决定变量如何保存。

注意:除非必要,尽量不要让 jBPM4 将变量保存为 `serializable` 类型,因为这将在 jBPM 数据库中的 JBPM4_LOB 表中增加一条二进制数据记录,我们知道,相比简单 Java 数据类型的数据库存取,这是非常消耗系统效率的。

7.3 变量的自动更新和序列化

我们前面不是说过 jBPM4 不能自动更新变量吗？现在情况出现了一点变化，在 jBPM4.3 以后的版本中， workflow 引擎在某些情况下能够自动保存那些已经序列化的了的流程变量，当然，第一次创建的变量仍然需要您调用 API “手工” 保存。

例如，在 `customs` 活动、事件监听器以及其他用户代码中，可以获取流程变量，当获取的流程变量已经作为持久化对象被保存时，可以直接修改此持久化变量，而不需要手工调用 API 进行保存。 workflow 引擎会管理这个经过持久化（即经过反序列化过程读取）的流程变量，如果检测到了改动就会自动保存之。例如图 7-1 所示的流程定义。

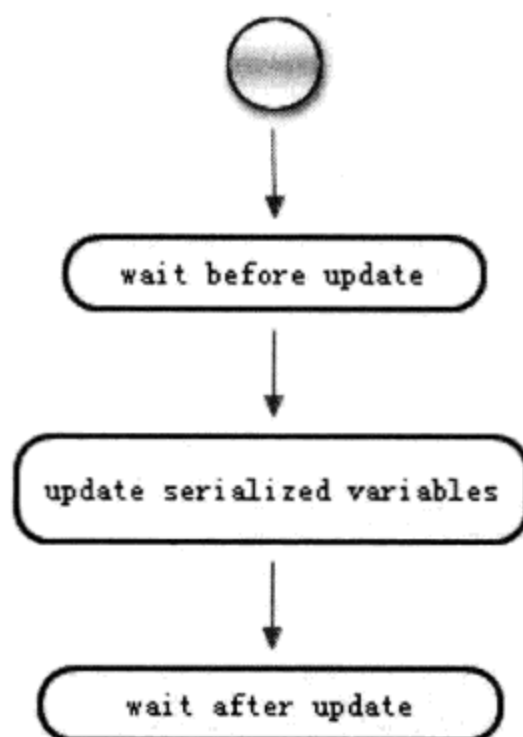


图 7-1 自动更新变量的流程定义

对应的 jPDL:

```
<process name="SerializableVariable" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="wait before update" />
  </start>
  <state name="wait before update">
    <transition to="update serialized variables" />
  </state>
  <!-- 在这里实验自动更新流程变量 -->
</process>
```

```

<custom
class="org.jbpm.examples.serializablevariable.UpdateSerializedVariables"
  name="update serialized variables">
  <transition to="wait after update" />
</custom>
<state name="wait after update" />
</process>

```

以下是自定义活动“update serialized variables”的实现类 UpdateSerializedVariables:

```

public class UpdateSerializedVariables implements ActivityBehaviour {
    public void execute(ActivityExecution execution) throws Exception {
        //在这里获取（假设）之前已经持久化过的流程变量 messages，修改之
        Set<String> messages = (Set<String>) execution.getVariable
("messages");
        messages.clear();
        messages.add("i");
        messages.add("was");
        messages.add("updated");
    }
}

```

编写单元测试代码验证上述流程:

```

//首先，创建流程变量 messages
Set<String> messages = new HashSet<String>();
messages.add("serialize");
messages.add("me");
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("messages", messages);
//在发起流程实例的时候，设入 messages 变量，即确保了 messages 在 custom 活动
更新前已经被 jBPM 持久化了
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("SerializableVariable",
variables);
String pid = processInstance.getId();
//发出执行信号通过“wait before update”活动，定义的 custom 活动“update
serialized variables”将自动执行
executionService.signalExecutionById(pid);
//验证：流程变量 messages 在 custom 活动中的修改，在事务被提交（wait after
update 是 state 活动，流程在此陷入等待状态，因此事务会被提交）时被自动持久化了
Set<String> expectedMessages = new HashSet<String>();
expectedMessages.add("i");

```



```

expectedMessages.add("was");
expectedMessages.add("updated");
messages = (Set<String>) executionService.getVariable(pid,
"messages");
assertEquals(expectedMessages, messages);

```

工作流引擎是如何感觉到流程变量变化的呢？当用户代码从持久化层（即数据库）中读取以序列化格式保存的流程变量时，工作流引擎在事务提交之前会监控该流程变量反序列化的过程，如果发现修改发生的话工作流引擎就会为持久化而保存流程变量的更新。

工作流引擎基于此规则检测流程变量的更新：如果一个流程变量对象被修改了，那么，其再次序列化后的字节数组，和从持久化层中一开始读出的该流程变量的序列化字节数组必然不同。

7.4 例程：用变量去控制一个流程的运行

在本例程中，将基于图 7-2 所示的流程定义发起流程实例运行，并使用流程变量控制流程转移的关键路径，以演示流程变量在流转控制中的关键作用。

流程定义如图 7-2 所示。

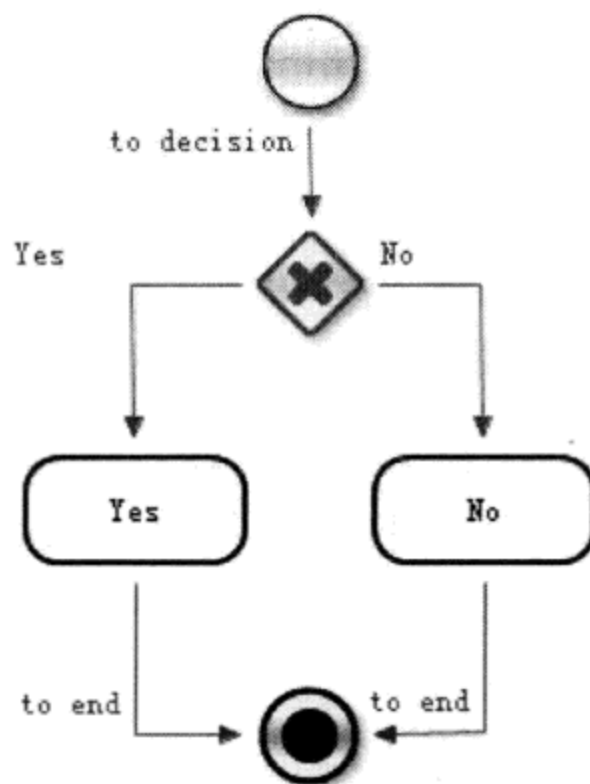


图 7-2 应用变量控制的流程定义

对应的 jPDL:

```
<process name="ProcessVariables" xmlns="http://jbpm.org/4.3/jpdl">
  <start name="start">
    <transition name="to decision" to="decision"/>
  </start>
  <!-- 在这里通过表达式引用流程变量 decision, 决定流程的下一步流转。 -->
  <decision expr="#{decision}" name="decision">
    <!-- 表达式的值与转移名称绑定, 即表达式值为 Yes 则流程流向 Yes 转移。 -->
    <transition name="Yes" to="Yes"/>
    <!-- 表达式值为 No 则流程流向 No 转移。 -->
    <transition name="No" to="No"/>
  </decision>
  <state name="Yes">
    <transition name="to end" to="end"/>
  </state>
  <state name="No">
    <transition name="to end" to="end"/>
  </state>
  <end name="end"/>
</process>
```

单元测试代码如下:

```
//创建流程变量的容器——Map 对象
Map<String, String> varMap = new HashMap<String, String>();
//设置流程变量 decision 的值为 Yes
varMap.put("decision", "Yes");
//发起流程实例, 同时设入流程变量
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("ProcessVariables", varMap);
String pid = processInstance.getId();
processInstance = executionService.findProcessInstanceById(pid);
Execution execution = processInstance.findActiveExecutionIn("Yes");
//断言当前活动名称为 Yes, 即流程根据变量的值选择了转移路径
assertNotNull(execution);
//发出执行信号通过 Yes 活动
executionService.signalExecutionById(execution.getId());
//断言流程实例结束
assertProcessInstanceEnded(pid);
```

7.5 小结

通过本章的介绍，我们对如何使用流程变量来为业务服务有了更加明确和全面的认识，在这里有个建议：**保持流程实例里变量的整洁、轻便。**

有一个 jBPM 应用的项目，它的糟糕之处在于：开发人员在流程实例上下文（在 jBPM3 版本中为 `executionContext`）中放置很多、很“重”的流程变量。这使得流程与业务系统的耦合显得额外紧密，维护不易，同时对系统的效率也有极大的影响。

我们明白，通过流程变量控制流程的流向是正确的做法，但是，不要被这种“方便”的机制诱惑而往流程实例里面放所有的东西，特别是与流转控制无关的业务数据！

假设您正在开发一个票务流程系统，需要存储一些持票人的附加信息，例如姓名、年龄、电子邮件……因此您在流程实例中保存了整个 Ticket 业务对象，即在您的流程实例上下文中混合了业务变量和流程变量。

正如前面所说，这样做是不正确的。

一个好的解决方案是：在您的业务系统中保存这些业务数据，同时在您的流程实例中仅保存一个对这些业务数据的主键引用，例如 `ticketId`。

图 7-3 所示是一个基于上述需求使用流程变量保存业务实体引用的示意流程。

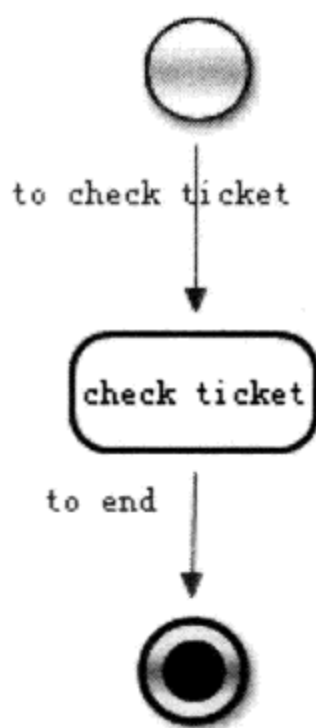


图 7-3 使用流程变量保存业务实体引用的流程定义

对应的 jPDL:

```
<process name="ProcessVariable" xmlns="http://jbpm.org/4.3/jpdl">
  <start name="start">
    <transition name="to check ticket" to="check ticket"/>
  </start>
  <state name="check ticket">
    <transition name="to end" to="end"/>
  </state>
  <end name="end"/>
</process>
```

假设需要引用的业务实体 Ticket 是个 POJO (简单 Java 对象):

```
public class Ticket implements Serializable {
    //这个是业务实体的主键, 将被流程实例引用
    private String ticketId;
    private String ticketName;
    private Integer price;
    private String level;
    private Date datetime;
    public String getTicketId() {
        return ticketId;
    }
    public void setTicketId(String ticketId) {
        this.ticketId = ticketId;
    }
    ...
}
```

在单元测试中发起流程实例, 开始模拟执行引用业务实体的流程:

```
//首先, 创建业务实体 Ticket 的对象
Ticket ticket = new Ticket();
ticket.setTicketName("Football");
ticket.setLevel("80");
ticket.setPrice(60);
ticket.setDatetime(new Date());
//调用业务系统的 API saveTicketToBizSys (略去业务系统的实现), 将业务实体存入业务系统, 返回业务主键, 即 ticketId
String ticketId = saveTicketToBizSys(ticket);
//将 ticketId 作为对业务实体引用的流程变量, 发起流程实例
Map<String, Object> variableMap = new HashMap<String, Object>();
```



```
variableMap.put("ticketId", ticketId);
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("ProcessVariable",
variableMap);
String pid = processInstance.getId();
```

在流程的运行过程中，随时可以通过 ticketId 从业务系统中获取 Ticket 实体：

```
String varTicketId = (String) executionService.getVariable(pid,
"ticketId");
//调用业务系统的 API loadTicketFromBizSys, 从业务系统中获取相应的 Ticket
实体
Ticket varTicket = loadTicketFromBizSys(varTicketId);
//断言是同一业务实体
assertEquals(varTicket, ticket);
assertActivityActive(pid, "check ticket");
```

正如所演示的，我们不需要在 jBPM 环境中保存整个业务实体 Ticket，因为它对流程的流转没有影响。随意在流程实例中添加变量，既不符合 jBPM 面向流程编程架构的初衷，也会在很大程度上降低系统运行的性能。

jBPM4 默认的流程表达式和脚本语言是 jUEL，即通常所说的“EL 表达式”语言。在 jBPM4.3 版本的 `jbpm.default.scriptmanager.xml` 配置文件中您可以发现 jBPM 除了 jUEL，还提供了其他脚本语言的支持：

```
<script-manager default-expression-language="juel"
  default-script-language="juel">
  <script-language name="juel"

factory="org.jbpm.pvm.internal.script.JuelScriptEngineFactory" />
  <script-language name="bsh"

factory="org.jbpm.pvm.internal.script.BshScriptEngineFactory" />
  <script-language name="groovy"

factory="org.jbpm.pvm.internal.script.GroovyScriptEngineFactory" />
</script-manager>
```

其中 bsh 是 BeanShell，Beanshell 是用 Java 语言写成的一个小型、免费、嵌入式的脚本解释器，它能支持对 Java 源代码的动态解释和执行；而 groovy 则代表 Groovy 语言，Groovy 语言是一个基于 Java 虚拟机的敏捷动态语言，它构建在 Java 语言之上，并具有 Python，Ruby 和 Smalltalk 等动态语言的诸多特征。

本书只介绍 jUEL。如果您对 JSP 2.0 的 EL 表达式比较了解且只需要书写基本的 jBPM4 脚本表达式，那么您可以先略过本章的介绍。

8.1 Java 统一表达式语言

jUEL，全称 Java Unified Expression Language，即 Java 统一表达式语言。以下简称 EL 表达式。

JSP 2.0 的 EL 表达式语言可以让开发者使用简单的表达式动态地从 JavaBean 对象

中读取数据。例如，下面 EL 表达式 if 标签的 test 属性被用来把 session 中 car 对象的 items 数量与 0 做大于比较，如果结果为真，则显示省略号表示的内容：

```
<c:if test="${sessionScope.cart.numberOfItems > 0}">
...
</c:if>
```

简而言之，EL 表达式语言能够让开发者使用简单的表达式执行下列任务：

- 动态地从 request 中的 JavaBean 对象里存取各种数据结构和内置对象。
- 动态地写入数据，例如将用户输入的表单数据写入到 JavaBean 对象。
- 执行对象上任意的 public 方法，包括 static public 方法。

8.1.1 语法特点

EL 表达式提供了一些标识符、存取器和运算符，用来检索和操作驻留在 JSP 容器（对于 jBPM 来说，则是工作流引擎）中的数据。EL 表达式在某种程度上以 EcmaScript（一种由 Ecma 国际，前身为欧洲计算机制造商协会，通过 ECMA-262 标准化的脚本程序设计语言。这种语言在互联网上应用广泛，它往往被称为 JavaScript 或 Jscript 语言）和 XPath 语言（XML Path Language，XML 路径语言）为基础，因此 HTML 设计人员和 Web 程序员都能很快熟悉 EL 表达式的语法。

EL 表达式擅长“寻找”对象的属性，然后对它们执行简单操作；EL 表达式不是编程语言，严格地说甚至不是脚本编制语言，但是，它与 JSTL 标记一起应用时，就能使用简单而又方便的符号来表示复杂的行为。jBPM 的 EL 表达式格式一般是这样的：用“#”号定界，内容包括在成对的花括号“{}”中，如下例所示：

```
... expr="#{user.name}" ...
```

此外，可以将多个表达式与静态文本组合在一起来构造动态的值，如下所示：

```
... expr="Hello, #{user.name}!" ...
```

独立的 EL 表达式由标识符、存取器、文字和运算符组成：

- 标识符用来引用存储在容器中的数据对象。
- EL 表达式有 11 个保留标识符，对应于 11 个隐式对象（见 8.1.3 隐式对象）。

- 存取器用来检索对象的属性或集合中的元素。
- 文字表示固定的值，例如数字、字符、字符串、布尔型或空值。
- 运算符则允许对数据和文字进行组合以及比较。

8.1.2 值和方法表达式

EL 表达式有两种类型：值表达式和方法表达式。值表达式可以产生或设置一个值。方法表达式可以被调用，用来返回一个值。

先说值表达式。

值表达式可以进一步分为“右值表达式”和“左值表达式”。右值表达式用来读取数据，但不能写入数据。左值表达式既能读取数据也能写入数据。

右值表达式需要在“\${}”分隔符中书写。用于延迟计算的表达式需要在“#{ }”分隔符中书写，这既可以作为右值表达式，也可以作为左值表达式。

如下面两个值表达式的例子：

```
<... value="${customer.name}" />
<... value="#{customer.name}" />
```

前者使用即时计算语法“\${}”，而后者使用的则是延迟计算语法“#{ }”。第 1 个表达式访问 customer 的 name 属性，获取其值，该值被立即计算出来，并加入 response 在页面上呈现（如果是 JSP 的话）。但第 2 个表达式对于 JSP 来说，在页面的生命周期内，如果需要的话，Web 容器可以推迟这个表达式的计算，即视为左值表达式。

使用值表达式引用 Java 对象时可支持以下的特性和属性：

- JavaBeans
- Collections
- Java 枚举类型
- JSP 隐式对象

关于 JSP 技术提供的隐式对象可参见 8.1.3 隐式对象。

下面的表达式引用一个名为 customer（客户）的 JavaBean：

```
${customer}
```


则 Web 容器会按照如下顺序去解释计算这个表达式：

- 1) 执行 `PageContext.findAttribute("customer")` 方法，即 Web 容器会按顺序在
 - a) page Scope
 - b) request Scope
 - c) session Scope
 - d) application Scope中寻找 `customer` 变量并返回其值。
- 2) 如果 `customer` 在 1) 中没有被找到，则 Web 容器会去匹配名为 `customer` 的隐式对象，并返回之。

对于 jBPM 来说，它的容器是工作流引擎，则容器首先会去对应的流程实例中寻找名称为 `customer` 的流程变量。

如果需要引用一个枚举（enum）类型的表达式，需要指明枚举字符串。例如如下枚举类：

```
public enum Suit {hearts, spades, diamonds, clubs}
```

对于这个枚举类型，如果您需要引用 `Suit.hearts` 枚举对象，则需要直接使用字符串“`hearts`”。在表达式中，这个“`hearts`”字符串会被“自动”转换为枚举对象。例如，在下面的场景中，我们假设“`mySuit`”是一个 `Suit` 类型的枚举对象，则在下面的表达式中，“`hearts`”字符串在执行“`==`”比较之前被自动转换成 `Suit.hearts` 枚举对象：

```
${mySuit == "hearts"}
```

如果需要使用 EL 表达式引用对象的属性、数组里的成员、枚举类型的实例对象，要用到“`.`”或“`[]`”标记。这遵循 ECMA 脚本规范。

例如，您需要引用 `customer` 对象的 `name` 属性，则可以使用如下两种表达方式：

```
${customer.name}
```

或

```
${customer["name"]}
```

这种方式支持对象属性的嵌套引用：

```
${customer.address["street"]}
```

枚举类型的属性同样支持这种引用方式，但是这有个前提条件，那就是枚举类型的属性必须具有相应的 `get<属性名称>` 方法。

例如，有如下的枚举类型 `Suit`，它的实例都具有 `desc` 属性，且存在相应的 `getDesc` 方法：

```
public enum Suit {
    hearts {
        @Override
        public String getDesc() {
            return this.desc;
        }
    },
    spades {
        @Override
        public String getDesc() {
            return this.desc;
        }
    },
    ...
    ;

    public String desc = "Suit Description";
    public abstract String getDesc();
}
```

因此，可以通过如下表达式去引用它：

```
${mySuit.desc}
```

引用数组里的成员（这包括 Java 数组和实现 `java.util.List` 接口的对象），可以使用“[下标]”的方式，如下的例子引用了 `customer` 对象中 `orders` 数组的第 2 个成员：

```
${customer.orders[1]}
```

如果 `orders` 是个 `Map` (`java.util.Map`) 对象，则可以直接通过 `key` 的方式去引用相对应的值，下面的例子引用了 `orders` `Map` 中 `key` 为 `socks` 的实体值：

```
${customer.orders["socks"]}
```

在 EL 表达式中，数字、字符串、布尔值，甚至 `Null` 值，都可以被文字表达式表示。字符串可以用单引号或双引号定界。布尔值则可以直接使用 `true` 或 `false` 表示。EL 表达式可以直接引用值，而非一定要是对象，在引用的同时还可以执行一些计算工作，

如下面的例子：

```
${"literal"}
```

返回一个字符串 `literal`。

```
${customer.age + 20}
```

如果 `customer.age` 是数字的话，则返回其加 20 计算操作的结果；否则返回拼接后的字符串。

```
${true}
```

返回布尔值 `true`。

```
${57}
```

返回数字 57。

现在介绍一下方法表达式。

EL 表达式语言的另一个特性就是支持方法表达式。方法表达式用于调用对象的任意公共方法，并返回一个结果。方法表达式可以用来支持很多场景，如同函数调用一样方便（题外话：对于 Java Server Faces 技术来说，方法表达式可以用来支持页面上 UI 组件的事件处理）。

因为方法可以在容器赋予的不同生命周期阶段被调用，所以，**方法表达式必须始终使用延迟计算的语法，即 “#{ }”**。

与左值表达式相似，方法表达式可以使用 “.” 和 “[]” 操作符去调用方法，例如 `#{object.method}` 与 `#{object["method"]}` 相同。字符串 `method` 被用来匹配同名的方法，匹配成功后，执行方法，同时返回结果数据。

如下的例子调用了名称为 `bean` 的 Java 对象的 `method` 方法，并将返回值存储在 `value` 变量中：

```
<... value="#{bean.method}"/>
```

8.1.3 隐式对象

表 8-1 列出了 11 个 EL 表达式的隐式对象。尽管它们都作用于 JSP，但请不要将

这些对象与 JSP 隐式对象（一共只有 9 个）混淆，其中只有一个对象是它们二者所共通的。

表 8-1 EL 表达式的隐式对象

类别	隐式对象标识符	描述
JSP	pageContext	当前页面的上下文
作用域	pageScope	在 JSP 页面作用域中的所有属性，以“名称-值”形式提供的 Map 类型
	requestScope	在 HTTP 请求作用域中的所有属性，以“名称-值”形式提供的 Map 类型
	sessionScope	在 HTTP 会话作用域中的所有属性，以“名称-值”形式提供的 Map 类型
	applicationScope	在 Web 应用程序范围内的所有属性，以“名称-值”形式提供的 Map 类型
请求参数	param	按“名称-值”形式提供 HTTP 请求参数 Map
	paramValues	HTTP 请求参数的所有值
请求头	header	按“名称-值”形式提供 HTTP 请求头 Map
	headerValues	HTTP 请求头的所有值
Cookie	cookie	按“名称-值”形式提供页面 Cookie Map
初始化参数	initParam	按“名称-值”形式提供的 Web 应用程序上下文初始化参数 Map

尽管 JSP 和 EL 表达式隐式对象中只有一个是公共的——pageContext，但通过 EL 表达式也可以访问其他 JSP 隐式对象。因为 pageContext 拥有访问所有其他 8 个 JSP 隐式对象的方法。实际上，这正是将 pageContext 包含在 EL 表达式隐式对象中的主要理由。

对于 jBPM 流程定义中的 EL 表达式来说，这些隐式对象都不存在。但是这些隐式对象仍然能在用于展现任务表单的 JSP 页面上大有“用武之地”。

8.1.4 运算符和保留字

EL 表达式可以通过使用标识符和存取器，遍历包含应用程序数据或关于环境信息（通过隐式对象）的对象层次结构。但是，只是访问这些数据，还不能满足我们对于

应用程序逻辑的需求。因此，我们需要 EL 表达式通过“运算符”计算一些简单的业务判断和逻辑规则。

表 8-2 显示了 EL 表达式运算符的优先级，按照从上至下、由左至右的顺序，运算的优先级从高到低。当然，您可以用成对的圆括号给表达式分组，以改变运算的优先级规则。

表 8-2 EL 表达式运算符

类别	运算符
算术运算符	[], .
	() (用于改变运算优先级)
	- (一元运算符), not, !, empty
	+, -, *, / (等同于 div), % (等同于 mod)
	+, - (位运算符)
关系运算符	< (等同于 lt), > (等同于 gt), <= (等同于 le), >= (等同于 ge), == (等同于 eq), != (等同于 ne)
逻辑运算符	&& (等同于 and), (等同于 or), ! (等同于 not)
验证运算符	empty (用于对象的前缀操作，确定对象是否为空串或 null 等情况)
条件运算符	A ? B : C (根据 A 的布尔结果，选择返回 B 或 C)

算术运算符用来计算数值的加法、减法、乘法和除法，另外还提供了一个求余运算符。需要了解的是：除法和求余运算符都有替代的、非符号的名称 `div` 和 `mod`，为的是兼容 XPath 语言的算术运算符。下面是一个应用算术运算符的 EL 表达式：

```
${item.price * (1 + taxRate[customer.address.zipcode])}
```

关系运算符用来比较数字或文本数据，比较的结果作为布尔值返回。逻辑运算符用来合并计算布尔值，返回新的布尔值。下面是应用关系运算符和逻辑运算符的 EL 表达式：

```
${(value >= min) && (value <= max)}
```

`empty` 也是一种运算符，它对于数据的非空验证特别简单有效。`empty` 运算符采用单个表达式作为其变量。例如 `${empty input}` 返回一个布尔值，该布尔值表示对表达式 `input` 求值的结果是否为“空”。这个“空”的解释有如下 3 种情况：

- 求值结果为 `null` 被视为空。
- 无元素的集合或数组被视为空。
- 长度为 0 的字符串，即空串视为空。

一旦满足上述 3 种中的任何一种情况，则返回布尔值 `true`。

所有这些运算符仅能在右值表达式 “`${}`” 中使用。

另外，使用 EL 表达式时要注意以下单词——作为 EL 表达式语言的保留字，不能被用于变量声明或标识符：

- `and` `or`
- `not` `eq` `ne`
- `lt` `gt` `le` `ge`
- `true` `false`
- `null` `empty`
- `instanceof`
- `div` `mod`

上面所列的一些单词目前也许并不在 EL 表达式语言的语法规范中，例如 `instanceof`，但将来可能会加入，所以必须小心，避免使用到它们。

8.1.5 一些经典 EL 表达式的例子

从一些最常用的例子中可以快速掌握编写 EL 表达式的技巧。这种方法是一些开发老手学习新技术“走捷径”的最好办法。

表 8-3 列出了一些经典的、常用的 EL 表达式，及其计算执行的结果：

表 8-3 经典 EL 表达式范例

EL 表达式	结果
<code>\${1 > (4/2)}</code>	<code>false</code>
<code>\${4.0 >= 3}</code>	<code>true</code>
<code>\${100.0 == 100}</code>	<code>true</code>
<code>\${(10*10) ne 100}</code>	<code>false</code>
<code>\${'a' < 'b'}</code>	<code>true</code>

(续表)

EL 表达式	结果
<code>\${'hip' gt 'hit'}</code>	false
<code>\${4 > 3}</code>	true
<code>\${1.2E4 + 1.4}</code>	12001.4
<code>\${3 div 4}</code>	0.75
<code>\${10 mod 4}</code>	2
<code>\${!empty param.Add}</code>	false ——如果 HTTP request 参数 Add 的值为“空”的话。反之则返回 true
<code>\${pageContext.request.contextPath}</code>	HTTP request 的上下文路径
<code>\${sessionScope.cart.numberOfItems}</code>	Http Session 中 cart 对象的 numberOfItems 属性的值
<code>\${param['mycom.productId']}</code>	名称为 mycom.productId 的 HTTP request 参数值
<code>\${header["host"]}</code>	Host 值, 位于 Http Header 中
<code>\${departments[deptName]}</code>	Map 对象 departments 中 key 为 deptName 的实体值
<code>\${requestScope['javax.servlet.forward.servlet_path']}</code>	HTTP request 中 名为 javax.servlet.forward.servlet_path 的属性的值
<code>#{customer.name}</code>	<ul style="list-style-type: none"> ● 在 HTTP request 初始化时, 获取 customer 对象的 name 属性值 ● 在 Http Post 方法回显时, 设置 customer 控件 name 属性的值
<code>#{customer.calcTotal}</code>	返回 customer 对象的 calcTotal 方法的调用结果

8.2 例程：用脚本去控制一个流程的运行

在本例程中，将基于以下流程定义发起流程实例运行，并使用脚本去影响流程的转移路径，以演示脚本在流转控制中的关键作用。

流程定义如图 8-1 所示。

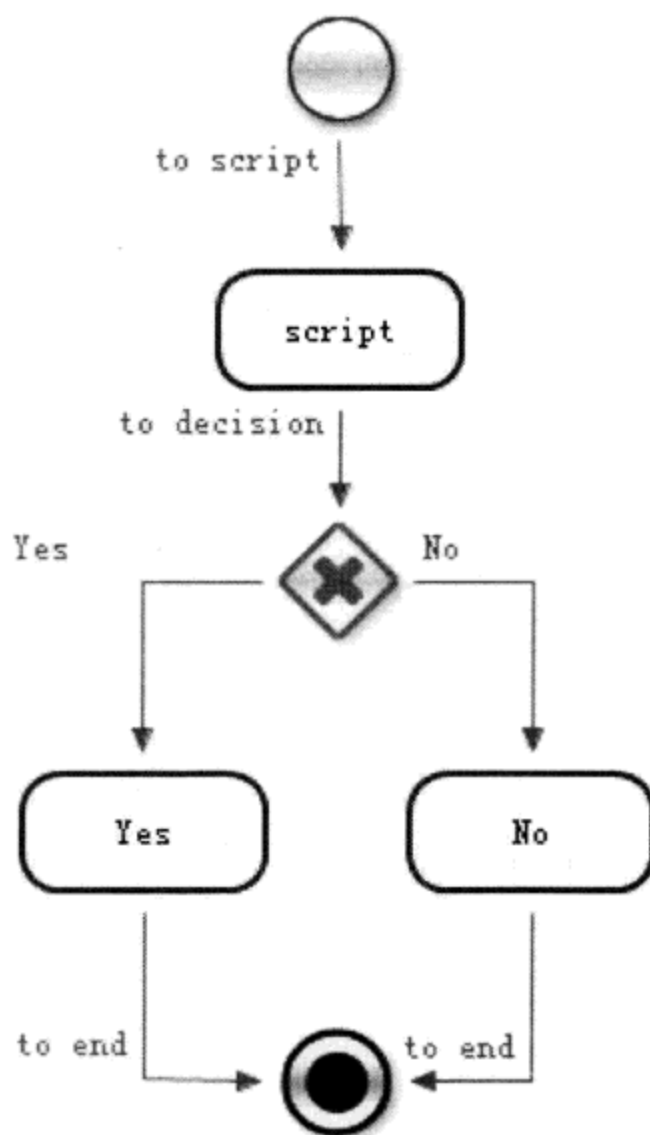


图 8-1 应用脚本控制的流程定义

对应的 jPDL:

```
<process name="ProcessScripts" xmlns="http://jbpm.org/4.3/jpdl">
  <start name="start">
    <transition name="to script" to="script"/>
  </start>
  <!-- 在这里定义脚本活动。表达式(expr)的值为字符串“Yes”，保存入流程变量 decision。 -->
```



```

<script name="script" expr="Yes" var="decision">
    <transition name="to decision" to="decision"/>
</script>
<!-- 这里的判断活动根据流程变量 decision 的值，将流程导向名称相匹配的转移路径。 -->
<decision expr="#{decision}" name="decision">
    <transition name="Yes" to="Yes"/>
    <transition name="No" to="No"/>
</decision>
<state name="Yes">
    <transition name="to end" to="end"/>
</state>
<state name="No">
    <transition name="to end" to="end"/>
</state>
<end name="end"/>
</process>

```

以下的单元测试代码，可验证上述流程定义：

```

//发起流程实例
ProcessInstance processInstance = executionService
    .startProcessInstanceByKey("ProcessScripts");
String pid = processInstance.getId();
//获取当前流程实例对象
processInstance = executionService.findProcessInstanceById(pid);
//根据流程定义，在 script 活动中，已经将流程变量 decision 赋值为“Yes”，因此
decision 活动将根据流程变量的值将流程导向相应 Yes 转移，而 Yes 转移被定义为流向 Yes 活动
//因此，在这里可以断言流程实例的当前活动名称为 Yes
Execution execution = processInstance.findActiveExecutionIn("Yes");
assertNotNull(execution);
//发出执行信号通过 Yes 活动
executionService.signalExecutionById(execution.getId());
//根据流程定义，在此可以断言该流程实例已经结束了
assertProcessInstanceEnded(pid);

```

最后，提出一个假设：如果在脚本活动中为流程变量 decision 赋予一个非“Yes”和“No”的值，流程实例的执行将会遭遇到何种情况？修改的流程定义部分如下：

```

...
<script name="script" expr="Yeah" var="decision">
    <transition name="to decision" to="decision"/>

```

```
</script>
```

```
...
```

读者可以思考一下，得出自己的答案后再修改流程定义运行，看结果是否如自己的预期。

8.3 小结

本章之所以使用了较长的篇幅介绍非 jBPM 及 workflow 技术本身的知识，是因为掌握 EL 表达式语言对设计 jBPM 流程定义至关重要。在流程定义中，对于流程变量、用户代码等 Java 对象的引用（默认的）都需要通过 EL 表达式来达成。

通过本章的介绍，读者应该能更加自如地设计出更加具有“个性”的业务流程，因为熟练使用流程脚本表达式能极大地扩展流程定义的语义，使流程定义能够承载更多的信息。



第二篇

定制属于自己的流程—— 深入 jBPM4 扩展研发

本篇主要内容能使读者能够成为一名合格的 jBPM 系统架构师。阅读本篇需要掌握第一篇的内容，或者具有一定 jBPM 业务流程应用开发的经验，因为只有真正感受过 jBPM 灵活的扩展性，才能很好地理解本篇的内容。

需要说明的是，本篇涉及的内容或来自 jBPM4 开发者的建议、或来自 jBPM4 开发者社区实践的结晶，但都非 jBPM 官方明确宣称支持的（但我确信很多内容在未来的 jBPM 版本中会被官方宣布稳定支持），因此，请结合实际情况，认真理解后应用。掌握本篇的内容必能使您在开发“自己的业务流程”过程中事半功倍。



jBPM4 扩展研发先决条件

在您决定开始扩展应用 jBPM4 框架和使用 jBPM4 的一些高级特性之前，了解本章的内容也许会使您的工作更富有效率，并避免一些失误。

9.1 深入应用 jBPM4 所需要知道的

虽然 jBPM 为开发者提供了全套的面向流程图编程的框架和思想，但 jBPM 并不是万能的上帝，对于复杂多变的企业流程应用，有如下的实践经验供您参考。

9.1.1 如果您的业务基于复杂的规则，在 jBPM 中加入 Drools 吧

首先需要明白：

- workflow 引擎系统，或者说面向图形编程的目标是制作一张表示执行过程的流程图。图中的流程活动可以陷入等待状态，直到外界干预（例如表单提交、客户端发出执行信号等）的到来流程才继续执行。
- 规则引擎的目标是制定一套规则，然后为这套规则指定一个“事实库”，并应用一套推理算法去将“规则”和“事实”匹配出结果。

Drools 是 JBoss 旗下的规则引擎，可以说是 jBPM 的同门师兄，它能起到什么作用？

我们知道：在 Java EE 的框架中组织应用程序的业务逻辑，如无意外，还是需要依靠硬编代码来解决。那么，随着系统复杂度的增加、业务规则的大量变化，会导致硬编码的应用程序不停地变更，这会加大系统稳定性降低和开发成本上升的风险。这种情况下，人们需要找到一套专门解决业务逻辑的架构，使得当业务规则不停变化时，无须修改任何应用程序的代码，也能保证应用系统稳定运行，并且立即适应特定的业务规则变化——Drools 就是这样一套应用在业务逻辑层的开源架构。

Drools 怎样同 jBPM 相结合呢？一种实现方式是：

- 1) 将 jBPM 流程定义中涉及到业务规则的逻辑都放到 Event-Listener (在 jBPM3 中是 Action-Handler) 中去。
- 2) 使用 Drools 来配置和管理 Event-Listener 中的全部或部分业务逻辑。

或者, 直接使用 jBPM4 提供的 “rule” 活动, rule 活动提供一个 “fact” 元素供您指定事实库去匹配 Drools 的规则。总而言之, 使用 Drools 规则引擎来驱动 jBPM 流程引擎的业务流程规则。

请注意, 这一实践并非适用于所有情况, 所以在决定启用 Drools 管理您的业务流程规则之前, 要问明白自己几件事:

- 您流程定义中的 Event-Listener 有多复杂?
 - 如果您仅仅只是需要从数据库中读取一些数据, 而不需要更多的业务逻辑判断, 在这种情况下使用 Drools, 就有点 “大炮打蚊子” 的感觉了。相反地, 在一个使用 Java 硬编码来处理大量业务逻辑的场合, 且当使用 jBPM 流程引擎来管理业务规则的时候, 使用 Drools 是值得考虑的, 这是因为大多数应用逻辑随着时间和业务的扩展会越来越复杂, 而 Drools 正会让您轻松地应付这些麻烦, 特别是在您应用程序的生命周期比较长的情况下。要明白, Drools 具有通过在一个或多个很容易变更的 XML 文件中配置业务规则的能力, 这可以帮助您轻松应对将来的业务规则变化。
- 您对于业务规则可读性的需求?
 - Drools 的一大优势是: Drools 可以 “指引” 开发人员正确地编写代码来 “做正确的事情”, 即可以通过制定 “规则” 驱动代码。同时, “规则” 比代码更容易阅读、更直观, 因此您在 Drools 中 “阅读” 业务时会更舒服。
- 您需要从规则中发现规律?
 - 在正确应用的情况下, Drools 能记住的不仅仅是业务规则信息, 而且还能很好地记录这些业务规则的运行/测试结果, 通过对各种业务规则的分析, 或许能帮助您发现业务逻辑优化的一条捷径。

最后, 如果您想获得更多关于 Drools 规则引擎的介绍, 请参考 Drools 的专门资料。

9.1.2 抉择, 是否使用 BPEL

jBPM4 工作流引擎在流程虚拟机架构的支持下, 能同时解释和执行 jPDL 和 BPEL

这两种流程定义语言。

BPEL (Business Process Execution Language, 业务流程执行语言) 是一种基于 XML 表述的语言, 用来描述长运行周期的 Web 交互服务流程。它的主要特点是被用来集中地编排消息交换, 因此在 SOA (Service Oriented Architecture, 面向服务的架构) 的实施中是一个关键技术。

那么 BPEL 与 jPDL 这两种语言有什么共同点呢?

它们的共同点如下:

- 这两种语言都可以描述流程。
- 都可以同外部代理应用交互。
- 都可以描述流程活动的调度。
- 都具有异常处理和错误恢复机制。

以上是 BPEL 与 jPDL 的一些共同点, 但组成这两种语言元素的具体表达不同导致了这两种语言有不同的受众, 它们对于流程的描述各有如下特点:

- jPDL 用更加简单的语义来描述和组织流程。
- BPEL 倾向于用复杂的语义来描述业务结构的组成。

BPEL 与 jPDL 和外部代理 (客户端) 应用的交互也被不同地实现:

- BPEL 是面向文档的, 因此可以在整个的组织业务范围中使用。
- jPDL 是面向对象的, 因此它适合作为串联组织业务组件的主干总线。
- BPEL 将人机交互的实现代理给 “Partner Service” 实现。
- jPDL 提供集成的任务管理机制来实现人机交互。

那么, 什么时候应该使用 BPEL 呢?

在您对于以下几点有强烈需求的时候可以考虑在 jBPM4 中使用 BPEL 代替 jPDL:

- **支持异构系统**——当您需要业务流程能很方便地伸展到 Java 平台之外, 即能很方便地、跨平台地不同编程语言系统间流转。BPEL 是一个通用的技术标准, BPEL 流程不仅能够在基于 Java 平台的服务器上被执行, 而且理论上能在任何其他的软件平台 (例如 .NET Framework, PHP) 上被执行。这一点在使用不同平台的合作系统之间进行业务交互的场景中是特别需要的。

- **无人交互，长运行周期的业务**——因为上面我们提到了 BPEL 本身并没有直接提供支持人机交互的语义。所以，在没有直接的人机交互（即无人工干预的业务流程，简单地说就是不需要人填写电子表单使流程继续运行）同时更需要长事务运行支持的业务流程的情况下，使用 BPEL 是合适的。
- **需要简单并且严格的事务补偿**——如果您需要以一个相对简单的方式取得事务补偿，即在已经完成的业务流程中获取补偿，消除部分已经造成的影响，或者撤销业务流程中某些已经走完的重要路径，那么也许 BPEL 能很好地帮助您。补偿的目标就是撤销：即消除业务流程实例中被要求放弃的“部分已完成活动”所造成的影响。

当上述需求不是您“特别”渴望的时候，我还是建议您使用 jPDL——这种 jBPM “原生态”的流程定义语言。

BPEL 在 jBPM4 中的具体应用方法本书将不做介绍。如果想详细了解 BPEL，请您参考 BPEL 的专门资料。

9.2 Maven 仓库和 Java 依赖库

如果您想成为一名优秀的 jBPM4 业务流程应用架构师，可能需要了解如下资源的情况：

1. jBPM4 源代码的 SVN 服务地址：<https://anonsvn.jboss.org/repos/jbpm/jbpm4>

jBPM4 的发布包中的 `jbpm.jar` 中包含了几几乎所有 jBPM 的模块，如 `jbpm-api`（jBPM 应用程序接口的定义）、`jbpm-log`（jBPM 系统日志模块）、`jbpm-test-base`（jBPM 单元测试基础类库）、`jbpm-pvm`（jBPM 流程虚拟机的实现）、`jbpm-jpdl`（jPDL 流程定义语言的解释模块）和 `jbpm-enterprise`（jBPM 企业应用模块）……如果您想自己实现 jBPM 的 API，并基于 PVM 打造一套定制的业务流程框架体系，那么您只需要使用 `jbpm-api` 和 `jbpm-pvm` 模块。在 jBPM4 的 Maven 仓库里，我们可以直接获取所需要的模块：<http://repository.jboss.com/maven2/org/jbpm/jbpm4>。

2. jBPM4 的依赖库

jBPM4 已经完全基于 Apache Maven 框架的规范描述源代码对于库的依赖了。也就是说 jBPM4 源代码的 jar 包依赖关系都是由 Maven POM 文件描述的，并且这些 jar 包

都能自动地从网络上的 Maven 仓库中获取。但是，为了方便开发者（也许是考虑到有的开发者访问网络有些麻烦，或使用 Maven 的学习成本较高），jBPM4 的发行版本仍然将这些依赖的 jar 包放在 lib 目录下提供出来了。这些放在 lib 目录下的依赖库，是经过 jBPM4 官方严格测试的，所以我们可以放心使用。

如果您想了解关于 jBPM4 源代码库依赖的明细，可以先获取相应模块在 SVN 上的源代码，其中包含有 *.pom 文件，然后在 pom 文件的路径下运行命令“mvn dependency:tree”查看即可。当然，在您的计算机上这需要先安装和配置好 Apache Maven。

9.3 小结

本章主要为想深入应用 jBPM4 的读者提供了一些指引，例如当业务规则足够复杂时，应当使用 Drools 规则引擎来取代 jBPM4 的用户代码，以“专业”的工具来管理复杂的逻辑；当流程更加趋向于面向异构系统的服务提供者时，应当使用 PVM 的另一种实现——BPEL。对于扩展 jBPM4 本身的功能，则介绍了一些我们必须知道的资源及其位置。

最后，大家需要知道的是，jBPM 的官方 WIKI 地址为：<http://community.jboss.org/wiki/jBPMWiki>，这不仅包括 jBPM4，还包括 jBPM3 的官方资源聚合页面。



深入 jPDL 和 jBPM Service API

本章可以视为对第 6 章 掌握 jBPM 流程定义语言的引深，将介绍一些更高级的 jPDL 的功能和活动。jBPM4 官方并不为这些活动和功能提供支持，所以，我建议要在确保自己真正可以“控制”这些技术后，再将其付诸于实际的生产环境中。

10.1 timer（定时器）能为您做什么

从 jBPM3 版本开始，jBPM 就支持定时器（timer）元素了，它可以为流程提供如下功能：

- timer 可以被定义在 transition(转移)元素中。在流程陷入 state, task, sub-process 等待活动时，可以在此等待活动的 transition 中定义一个 timer，当此 timer 的时间耗尽，转移将会被触发。
- timer 可以被定义在等待活动的用户代码中。负责事件监听的“on”元素支持“timeout”事件，当事件监听器 on 在监听 timeout 事件时，timer 元素应当是该 on 元素中的第一个子元素。当流程的执行进入此等待活动时，定时器被激活。如果此等待活动持续存在的时间超过定时器指定的时间（即 duedate，超时），事件（event）将会被触发；当流程的执行离开此等待活动后，timer 就会被取消。

timer 元素的属性只有两个，如表 10-1 所示。

表 10-1 timer 元素的属性

属性	类型	默认值	是否必需	描述
duedate	持续时间表达式（见下一节）	无	必需	指定多长时间后 timer 被触发。例如 30 分钟、2 个工作日……

属性	类型	默认值	是否必需	描述
repeat	持续时间表达式 (见下一节)	无	可选	timer 第一次被触发后, 指定每间隔多少时间再触发 timer, 这是一个持续的过程。例如 10 分钟、2 个自然日……

10.1.1 持续时间表达式

描述 timer 持续时间的表达式需要遵循如下语法规则:

```
quantity [business] {
second | seconds | minute | minutes | hour | hours |
day | days | week | weeks | month | months | year | years
}
```

quantity 是一个正整数, 描述时间的值。

business 参数是可选的, 如果带有 business 参数, 则表示按照“工作日历”计算时间; 如果没有 business 参数, 时间的计算将会遵循“自然日历”。自然日历就是我们平时使用的公历, 而工作日历则会按照预先配置的规则, 去除一些节假日的时间和休息时间, 使得计算时间时只考虑工作时间。

最后, 要加上时间单位, 例如秒 (second)、分 (minute)、时 (hour)、日 (day)……

关于如何配置工作日历请看下一节。

10.1.2 工作日历

在 jBPM4.3 版本中, 系统默认的工作日历配置位于 jbpm.businesscalendar.cfg.xml 文件中, 此配置文件存在于 classpath 根目录下, 其内容为:

```
<jbpm-configuration>
  <process-engine-context>
    <!-- 在 business-calendar 节点内配置工作时间 -->
    <business-calendar>
      <monday hours="9:00-12:00 and 12:30-17:00" />
      <tuesday hours="9:00-12:00 and 12:30-17:00" />
    
```

```

        <wednesday hours="9:00-12:00 and 12:30-17:00" />
        <thursday hours="9:00-12:00 and 12:30-17:00" />
        <friday hours="9:00-12:00 and 12:30-17:00" />
        <holiday period="01/07/2008 - 31/08/2008" />
    </business-calendar>
</process-engine-context>
</jbpm-configuration>

```

上面的工作日历配置表示的“工作时间”为：周一至周五的 9 点到 12 点和 12 点半到 17 点，另外指定 2008 年 7 月 1 日到 2008 年 8 月 31 日为假日时间。

上述配置是基于系统默认的工作日历实现类 `org.jbpm.pvm.internal.cal.BusinessCalendarImpl` 而生效的。如果此配置机制大致能满足您对工作日历的需求，您可以直接在 `business-calendar` 节点里调整工作时间，直到符合需求。

如果系统默认的工作日历实现 `BusinessCalendarImpl` 不能满足您的需求，即您的业务的工作时间算法完全“套”不上 `business-calendar` 节点的配置项，那么您可以通过实现 `org.jbpm.pvm.internal.cal.BusinessCalendar` 接口来实现定制的工作日历。举一个简单的例子：

```

public class CustomBusinessCalendar implements BusinessCalendar {
    //定制的工作日历必须实现接口方法 add, 用来计算从一个时间点 date, 依据工作日历算法经过
    //duration 后, 到达的时间点。这个到达的时间点被返回
    public Date add(Date date, String duration) {
        if ("birthday".equals(duration)) {
            //在这个实现中, duration 只简单地接受一种参数 birthday, 并返回固定的时间点:
            //12 月 22 日
            GregorianCalendar gregorianCalendar = new GregorianCalendar();
            gregorianCalendar.set(Calendar.MONTH, Calendar.DECEMBER);
            gregorianCalendar.set(Calendar.DAY_OF_MONTH, 22);
            return gregorianCalendar.getTime();
        }
        return null;
    }
}

```

`add` 方法的参数 `date` 表示需要计算的起始时间点，而参数 `duration` 表示从起始时间点开始要经历的时间段（或时间标识），方法返回经过工作日历算法处理后达到的时间点。`add` 方法要实现的内容正是所谓的“工作日历算法”。

注意：字符串参数 duration 可以通过 new Duration(String duration)方法被构造成为一个 org.jbpm.pvm.internal.cal.Duration 类型的对象，这个 Duration 对象表示的就是持续时间表达式。这可以为您表达“时间段”提供支持。您可以参考默认的实现 BusinessCalendarImpl 获得更多定制工作日历的“灵感”。

在 jbpm.cfg.xml 中添加如下配置即可使您定制的工作日历生效：

```
<process-engine-context>
<object class="yourpackage.CustomBusinessCalendar" />
</process-engine-context>
```

10.1.3 定时转移

下面的例子展示如何使一个 transition（转移）定时被触发，流程定义如图 10-1 所示。

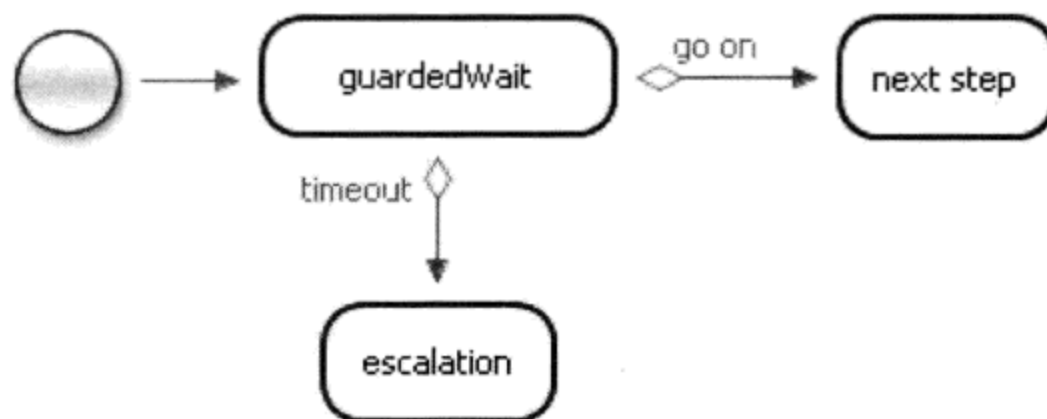


图 10-1 定时转移的流程定义

对应的 jPDL:

```
<process name="TimerTransition" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="guardedWait" />
  </start>
  <state name="guardedWait">
    <transition name="go on" to="next step" />
    <transition name="timeout" to="escalation">
      <!-- 如果流程在 guardedWait 活动停留 10 分钟，则 time out 转移将会发生，流向 escalation 活动 -->
    </transition>
  </state>
</process>
```



```

        <timer dueDate="10 minutes" />
    </transition>
</state>
<state name="next step" />
<state name="escalation" />
</process>

```

编写代码验证之。首先发起流程实例，它会在状态活动 `guardedWait` 等待。同时，一个定时器会被创建在 `timeout` 转移上，10 分钟后它将触发转移发生。

```

//发起流程实例
Execution processInstance = executionService.startProcessInstanceByKey(
    "TimerTransition");

```

定时器的产生会表现为一条 `Job` 记录被创建，我们可以通过 `ManagementService` 查询到关联此流程实例定时器的 `Job` 对象：

```

Job job = managementService.createJobQuery()
    .timers()
    .processInstanceId(processInstance.getId())
    .uniqueResult();

```

如果我们等待 10 分钟，那么 `jBPM` 引擎会使用 `JobExecutor` 来自动执行这个 `Job`。但是时间是宝贵的，我们没有必要为了测试而等待 10 分钟，因此可以在单元测试代码中使用 `ManagementService` 来模拟定时器被触发、执行：

```

managementService.executeJob(job.getId());

```

然后，流程实例就会经过 `timeout` 转移，进入状态活动 `escalation`：

```

processInstance =
    executionService.findExecutionById(processInstance.getId());
//断言流程进入 escalation 活动
assertEquals("escalation", processInstance.getActivityName());

```

如果在定时器被触发之前，即在 10 分钟内发出执行信号离开 `guardedWait` 活动，则该定时器产生的 `Job` 会被自动消除。

10.1.4 定时事件

下面的例子展示如何使自定义的事件被定时触发，流程定义如图 10-2 所示。



图 10-2 定时事件的流程定义

对应的 jPDL:

```

<process name="TimerEvent" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="guardedWait" />
  </start>
  <state name="guardedWait">
    <on event="timeout">
      <!-- timer 元素为 timeout 事件的必需元素。这里表示当流程在 guardedWait 活动时如
      果停留了 10 分钟，则触发 timeout (超时) 事件 -->
      <timer due="10 minutes" />
      <!-- 事件监听器 Escalate 被用来处理此超时事件 -->
      <event-listener class="org.jbpm.examples.timer.event.Escalate" />
    </on>
    <transition name="go on" to="next step" />
  </state>
  <state name="next step" />
</process>

```

我们可以在事件监听器 Escalate 中做如下标识:

```

public class Escalate implements EventListener {
    public void notify(EventListenerExecution execution) {
        //设置流程变量 escalation 的值为 true 来标识事件被触发
        execution.setVariable("escalation", Boolean.TRUE);
    }
}

```

编写单元测试代码验证上述流程定义:

```

//发起 TimerEvent 的流程实例
ProcessInstance processInstance = executionService.
startProcessInstanceByKey("TimerEvent");
//流程的执行一旦进入 guardedWait 活动，根据定义，timeout 事件监听器的 Job 会
立即产生，我们通过 ManagementService 获得之
Job job = managementService.createJobQuery().processInstanceId(

```

```

        processInstance.getId()).uniqueResult();
        //利用 ManagementService 立即执行此 Job，即人工模拟 timeout 事件的触发（根据定义，如果要正常触发 timeout 事件，需要等待 10 分钟）
        managementService.executeJob(job.getId());
        processInstance = executionService.findProcessInstanceById
(processInstance.getId());
        Set<String> expectedActivityNames = new HashSet<String>();
        expectedActivityNames.add("guardedWait");
        //因为我们在事件监听器 Escalate 中并没有发出执行信号使流程通过当前活动，因此
        断言流程实例仍然停留在 guardedWait 活动是成立的
        assertEquals(expectedActivityNames,
processInstance.findActiveActivityNames());
        //断言在事件监听器 Escalate 中设置流程变量标识成功，即证明事件被触发了
        assertEquals(Boolean.TRUE,
executionService.getVariable(processInstance.getId(), "escalation"));

```

我们可以想象：正常情况下，如果流程实例的 guardedWait 活动在 10 分钟内被完成，那么定时器就会被取消，Escalate 事件监听器也就不会被执行，相关的 Job 记录会在数据持久层被物理删除。

10.1.5 工作日历定时

下面的流程定义在 guardedWait 活动的流出转移中定义了一个定时器，这与 10.1.3 定时转移类似。但不同的是，timer 的计时采用了工作日历的标识——business。流程图如图 10-3 所示。

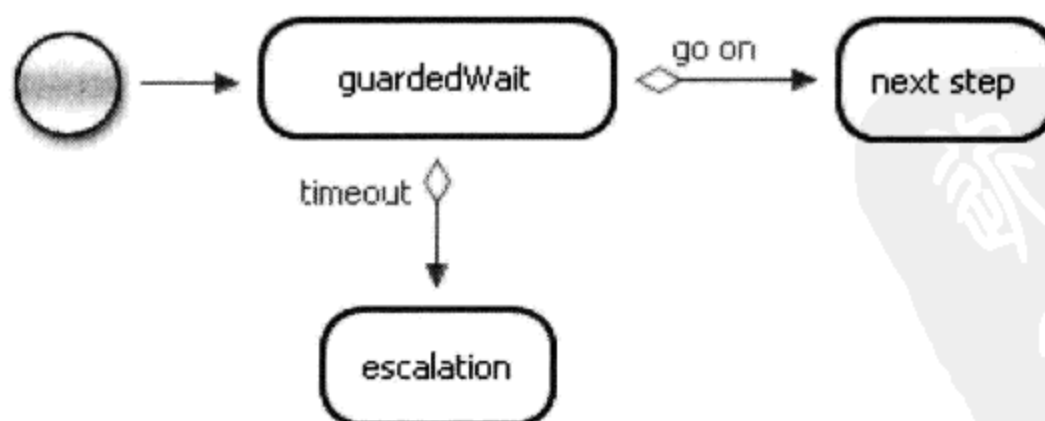


图 10-3 工作日历定时的流程定义

对应的 jPDL:

```

<process name="TimerBusinessTime" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="guardedWait" />
  </start>
  <state name="guardedWait" >
    <transition name="go on" to="next step" />
    <transition name="timeout" to="escalation" >
      <!-- 表示如果在当前活动停留 9 个“工作小时”后，将触发此 timeout 转移 -->
      <timer dueDate="9 business hours" />
    </transition>
  </state>
  <state name="next step" />
  <state name="escalation" />
</process>

```

还记得 jBPM4 默认的工作日历定义吗：

```

...
<business-calendar>
  <monday hours="9:00-12:00 and 12:30-17:00" />
  <tuesday hours="9:00-12:00 and 12:30-17:00" />
  ...

```

可以想象：在默认的工作日历配置下，假设 TimerBusinessTime 流程实例在周一的上午 11:30 被发起，然后一直没有被处理，那么，经过 9 个工作小时以后，即周二的下午 13:00，流程实例将会经过名为 timeout 的转移，离开 guardedWait 活动到达 escalation 活动。

10.1.6 定时重复

还记得 timer 的 repeat 属性吗？下面的流程定义展示了如何通过定时器让事件监听器反复地执行，即利用 timer 的 repeat 属性使事件不断地被重复触发。流程图如图 10-4 所示。

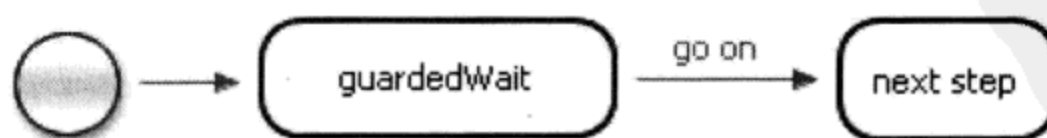


图 10-4 定时重复的流程定义

对应的 jPDL:

```
<process name="TimerRepeat" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="guardedWait" />
  </start>
  <state name="guardedWait">
    <on event="timeout">
      <!-- repeat 属性的值为 "10 seconds" 表示在 timeout 事件被触发后, 每隔 10 秒钟,
      事件监听器 Escalate 都会被执行一次 -->
      <timer dueDate="20 minutes" repeat="10 seconds" />
      <event-listener class="org.jbpm.examples.timer.repeat.Escalate" />
    </on>
    <transition name="go on" to="next step" />
  </state>
  <state name="next step" />
</process>
```

当发起 TimerRepeat 流程实例后, 如果一直没有任何外部操作, 那么 20 分钟后 (根据定义 dueDate="20 minutes"), timeout 事件将会被第一次触发, 即事件监听器 Escalate 会被第一次执行; 然后, 每隔 10 秒钟 (根据定义 repeat="10 seconds") 事件会被反复触发, 事件监听器 Escalate 会反复执行……就这样一直重复下去, 直到 guardedWait 活动被执行信号结束。当 guardedWait 活动的实例被结束后, 其产生的所有定时器都会被消除。

10.2 使用 group 活动编组流程

首先声明, 直到 jBPM4.3 版本, group 活动仍然没有被 jBPM4 的图形化流程设计器所支持, 也就是说, 虽然您能在流程定义中使用 group 活动, jBPM4 工作流引擎也能识别和执行 group 活动, 但是在 Eclipse 的图形化流程设计界面中, 含有 group 定义的流程是不能正常显示的。

jBPM4 的 group 活动对应的是 jBPM3 版本中的 super-state 节点。一对 group 标签可以将流程定义中的若干个连续活动包含起来, 从而实现将一群活动编组, 即可以按业务划分流程的一系列“阶段”。使用 group 活动不会像使用 sub-process (子流程) 活动那样产生新的流程定义。如下面的 jPDL 片段所示:

```

<process name="GroupSimple" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <!-- 进入 group 活动 evaluate document -->
    <transition to="evaluate document" />
  </start>
  <group name="evaluate document">
    <start>
      <transition to="distribute document" />
    </start>
    ...
    <end name="done" />
    <!-- group 活动的出口 -->
    <transition to="publish document" />
  </group>
  <state name="publish document" />
</process>

```

您可以对 group 活动的事件进行监听（event-listener），即一组活动可以被统一关联到一个事件监听中，这样做的一个重要意义就是：流程的执行能够被同时绑定在多个活动的事件上。例如，可以在流程的执行过程中很方便地检测到流程是否进入了某一个阶段。如图 10-5 所示是 jBPM3 流程定义图形。

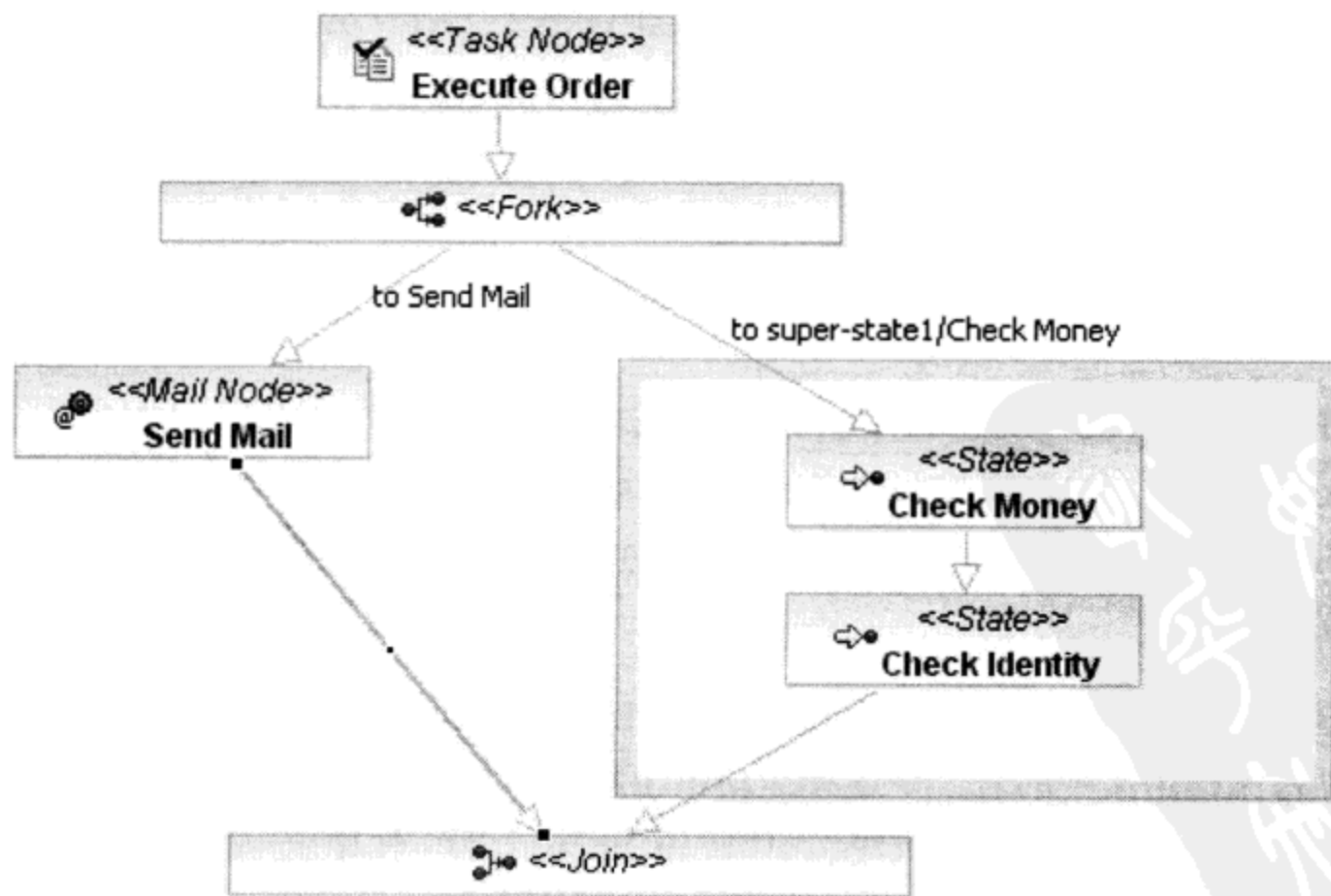


图 10-5 应用 super-state 节点（对应 jBPM4 的 group 活动）的 jBPM3 流程定义

可以看到“super-state1/Check Money”是由一组节点所组成的，在流程实例运行中，我们可以很方便地获知流程是否处于“super-state1/Check Money”这一阶段，同时，在定义过程中，也可以对“super-state1/Check Money”组中的节点统一设置事件监听。

因此，如果有一天 jBPM4 能很好地支持 group 活动（估计很快会实现），我们应该尽可能地使用它。将一条复杂的流程定义分解成各组 group 是一种很好的设计思想。同时，在一条“长达数屏的”复杂流程定义（根据经验，这在实际业务中很常见）中广泛使用 group 活动，能使流程的可读性大为增加，至少不会让业务分析人员或开发人员一眼看上去就会晕倒。

表 10-2 所列是 group 活动支持的元素。

表 10-2 Group 活动的元素

元素	数目	描述
any activity (任何活动)	0..*	group 组中包含的其他活动
transition	0..*	group 的流出转移，即 group 活动的出口

下面的流程定义展示了一个简单的 group 活动的应用。正如前面所提到的，group 活动在目前 jBPM4.3 版本的流程设计器中并不能被表达出来，所以我们只能列出该流程定义的 jPDL 源代码：

```
<process name="GroupSimple" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
    <transition to="evaluate document" />
  </start>
  <!-- 在这里开始对流程活动分组，分组名称为 evaluate document -->
  <group name="evaluate document">
    <!-- group 活动分组需要以一个 start 活动为开始 -->
    <start>
      <transition to="distribute document" />
    </start>
    <state name="distribute document">
      <transition to="collect feedback" />
    </state>
    <state name="collect feedback">
      <transition name="approved" to="done" />
      <transition name="rejected" to="update document" />
    </state>
  </group>
</process>
```



```

    </state>
    <state name="update document">
        <transition to="distribute document" />
    </state>
    <!-- group 活动分组可以以一个 end 活动来标识结束。不是必需。 -->
    <end name="done" />
    <!-- group 活动的出口 -->
    <transition to="publish document" />
</group>
<state name="publish document" />
</process>

```

以下单元测试程序验证上面的流程定义：

```

//发起流程实例 GroupSimple
ProcessInstance processInstance = executionService.
startProcessInstanceByKey("GroupSimple");
String pid = processInstance.getId();
//断言流程进入 group 组中的第一个活动 distribute document
assertTrue(processInstance.isActive("distribute document"));
//以下的流转都在 group 组内部进行
processInstance = executionService.signalExecutionById(pid);
assertTrue(processInstance.isActive("collect feedback"));
processInstance = executionService.signalExecutionById(pid, "rejected");
assertTrue(processInstance.isActive("update document"));
processInstance = executionService.signalExecutionById(pid);
assertTrue(processInstance.isActive("distribute document"));
processInstance = executionService.signalExecutionById(pid);
assertTrue(processInstance.isActive("collect feedback"));
//approved 转移指向 group 组的 end 活动，即在这里离开 group 活动 evaluate
document
processInstance = executionService.signalExecutionById(pid, "approved");
//断言流程实例处于 group 活动的下一步 "publish document"
assertTrue(processInstance.isActive("publish document"));

```

实际上，group 活动几乎具有所有通用的活动特性，而且它还有一些独特的功能。

- 可以在 group 活动上设置定时器 (timer)，以定时触发整组活动。
- group 活动可以具有多个 start 活动和多个 end 活动，既可有多个转移流入 group，也可有多个转移流出 group，例如下面的 jPDL 流程定义：

```

<process name="GroupMultipleEntries" xmlns="http://jbpm.org/4.3/jpdl">

```



```

...
<group name="evaluate project">
  <start name="play">
    <transition to="distribute document" />
  </start>
  <state name="distribute document" />
  <start name="plan">
    <transition to="make planning" />
  </start>
  <state name="make planning" />
</group>
...
</process>

```

- 我们知道了 group 可以有多个 start 活动，那么 group 内的流转可以由多个 start 活动并发启动，同步执行。例如下面的 jPDL 流程定义：

```

<process name="GroupConcurrency" xmlns="http://jbpm.org/4.3/jpdl">
  <start>
<!-- 指向 group 活动的转移 -->
    <transition to="evaluate project" />
  </start>
<!-- 当流程的执行进入 group 活动时，这两个 start 活动将同步并发开始执行 -->
  <group name="evaluate project">
    <start>
      <transition to="distribute document" />
    </start>
    ...
    <end name="document finished" />
    <start>
      <transition to="make planning" />
    </start>
    ...
    <end name="planning finished" />
    <transition to="public project announcement" />
  </group>
  <state name="public project announcement" />
</process>

```

- 尽管不建议这么做，但是我们需要知道：group 活动是支持外部活动直接“转移”到其内部活动的。同样，group 活动的内部活动也可以直接“转移”到外部活动。这给予了 group 活动流入流出转移极大的灵活性……

最后，需要再次重申，在本书写作的时候，可以说 jBPM4 对于 group 活动的支持仍然在“孵化”中。但是，也许本书面世的时候，group 活动已经正式获得支持了，即您在 jBPM4 的图形化流程设计器左侧工具栏上可以发现 group 活动的图标。

10.3 如何在活动中调用 EJB 方法

我们已经了解到 jBPM4 的 java 活动可以用来调用 Java 方法。但是，本节将向您介绍如何使用 java 活动来调用 EJB (Enterprise Java Bean, 企业级 Java Bean) 会话 Bean 的方法。

EJB 的一大特征就是将企业级 Java 应用组件封装成服务，然后通过应用级的 JNDI (Java Naming and Directory Interface, Java 命名和目录服务接口) 服务提供给外部使用。那么，如果要在 jBPM 中执行 EJB 方法，则必须要指定 JNDI 名称，这在 java 活动中对应的是 ejb-jndi-name 属性，这个属性的值指定了要调用的 EJB 的 JNDI 名称。

以实例说明。假设一个 EJB 会话 Bean 的代码如下：

```
//无状态的会话 Bean
@Stateless
public class CalculatorBean implements CalculatorRemote, CalculatorLocal {
    // CalculatorBean 提供两个简单的 EJB 方法，分别为整数的加法和减法
    public Integer add(Integer x, Integer y) {
        return x + y;
    }
    public Integer subtract(Integer x, Integer y) {
        return x - y;
    }
}
```

架设好 EJB 的环境（略）。设置上面的会话 Bean 的 JNDI 名称为“CalculatorBean/local”（这个名称一般被配置在整个应用服务器级别），那么，我们就可以使用图 10-6 所示的流程定义调用其 EJB 方法。

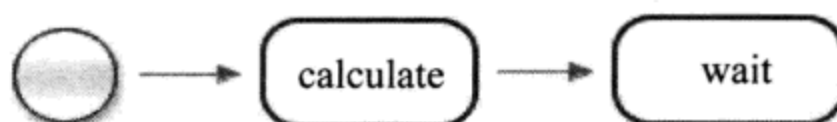


图 10-6 调用 EJB 方法流程定义

对应的 jPDL:

```
<process name="EJB">
  <start>
    <transition to="calculate" />
  </start>
  <!-- 在这个 java 活动中，分别指定 EJB 的 JNDI 名称——CalculatorBean/local、调用方法
  的名称——add，以及调用执行结果存入的流程变量名称——answer -->
  <java name="calculate" ejb-jndi-name="CalculatorBean/local" method="add"
  var="answer">
    <!-- 为 add 方法注入参数 -->
    <arg><int value="25"/></arg>
    <arg><int value="38"/></arg>
    <transition to="wait" />
  </java>
  <state name="wait" />
</process>
```

编写程序发起这个流程定义的实例，验证在此 java 活动中：

- 1) 根据 JNDI 名称获得 EJB 会话 Bean。
- 2) 执行会话 Bean 的 add 方法。
- 3) 执行结果被保存在流程变量 answer 中。

单元测试代码如下：

```
//发起流程实例
String executionId = executionService
    .startProcessInstanceByKey("EJB")
    .getProcessInstance().getId();
//java 活动 calculate 自动执行通过，流程实例将停留在活动 wait 上
//验证 add 方法的执行结果是否如预期
assertEquals(63, executionService.getVariable(executionId, "answer"));
```

10.4 使用 jms 活动

jBPM4 提供了 jms 活动以支持用户方便地发送 JMS (Java Message Service, Java 消息服务) 消息。目前 jms 活动仅支持发送三种格式的 JMS 消息：文本、Map 对象以及序列化的 Object 对象。

表 10-3 所列是 jms 活动支持的属性。

表 10-3 jms 活动的属性

属性	类型	默认	必需	备注
connection-factory	JNDI 名称	无	必需	JMS 连接工厂的 JNDI 名称
destination	JNDI 名称	无	必需	JMS Queue 或 Topic 的 JNDI 名称
transacted	布尔 {true, false}	true	可选	指定 JMS 消息的发送是否被包含在事务里。参考 JMS API 接口 javax.jms.QueueConnection 的方法: createQueueSession (boolean transactional, int acknowledgeMode)
acknowledge	{auto client dups-ok}	auto	可选	指定 JMS 消息的应答模式 (Acknowledge Mode)。参考 JMS API 接口 javax.jms.QueueConnection 的方法: createQueueSession (boolean transactional, int acknowledgeMode)

表 10-4 所列是 JMS 活动的元素，即用来定义上面提到的 3 种消息格式。

表 10-4 jms 活动的元素

元素	数目	备注
text	0..1	用来作为 JMS 消息发送的字符串
object	0..1	用来引用 JMS 消息发送的序列化 Java Object 对象
map	0..1	用来引用 JMS 消息发送的 Java Map 对象

jms 活动的消息必须使用 text, object 或 map 这 3 种元素之一表示。这 3 种元素分别会生成 TextMessage, ObjectMessage 或 MapMessage。

对于 jms 活动来说，connection-factory 和 destination 属性是必需的。connection-factory 属性指定 JMS 连接工厂的 JNDI 名称。destination 属性指定 JMS 消息队列或主题的 JNDI

名称, 我们知道, JMS 规范中只有两种消息类型: 队列 (Queue) 消息和主题 (Topic) 消息。connection-factory 指定了 JMS 服务的提供者, 而 destination 则指定了消息的目的地, 这二者是 JMS 规范中一条消息生命周期里不可缺少的两个要素, jBPM4 使用如下代码处理这两个属性:

```
InitialContext initialContext = new InitialContext();
//获取 JMS 消息的目的地
Destination destination = (Destination) initialContext.lookup
(destinationName);
//获取 JMS 消息服务的提供者
Object connectionFactory = initialContext.lookup(connectionFactoryName);
```

new InitialContext() 表明: jms 活动的 JNDI 查找会搜索整个应用服务器内的消息队列、消息主题以及 JMS 连接工厂。当您的流程是一个远程应用的客户端时, 应该使用系统属性指定 JNDI 环境。

为了方便测试, jBPM4 发布包中提供了 org.jbpm.test.JbpmTestCase.jmsCreateQueue 和 org.jbpm.test.JbpmTestCase.jmsCreateTopic 这两个方法供您在测试时模拟 JMS 消息队列和 JMS 消息主题。

注意: 在 jBPM4.3 版本中, jms 活动对于 XA (分布式事务处理规范) 事务的支持并没有实现。也就是说, 您需要配置 jms 活动的 transacted 属性为 false (transacted="false"), 这样应用是稳定的。我相信在 jBPM4 后续的版本中, jms 活动对于 XA 事务的支持会实现。

10.4.1 模拟 JMS 服务

当然, 您可以配置一个真正的 JMS 服务, 确保 jms 活动可以通过 JNDI 查找到它。但是, 为了“方便地”测试, 我们可以提供模拟的 JMS 服务来测试 jms 活动, 这在 jBPM4 中有很好的支持。

JbpmTestCase 基于原始 JMS API 实现了如下的 JMS 服务“模拟器”, 这些模拟器可以为您模拟一个等同于真实的 JMS 应用环境、帮助您快捷地测试 JMS 服务:

- JbpmTestCase.jmsConsumeMessageFromQueue (String connectionFactoryJndiName, String queueJndiName, long timeout, boolean transacted, int acknowledgeMode)

- 从队列中消费一条消息。
- `JbpmTestCase.jmsConsumeMessageFromQueue` (String connectionFactoryJndiName, String queueJndiName)
 - 从队列中消费一条消息。默认参数：timeout=1000 transacted=true acknowledgeMode= Session.AUTO_ACKNOWLEDGE。
- `JbpmTestCase.jmsConsumeMessageFromQueueXA` (String connectionFactoryJndiName, String queueJndiName, long timeout)
 - 在 XA 事务控制下从队列中消费一条消息。
- `JbpmTestCase.jmsAssertQueueEmpty` (String connectionFactoryJndiName, String queueJndiName, long timeout, boolean transacted, int acknowledgeMode)
 - 断言指定的消息队列为空。
- `JbpmTestCase.jmsAssertQueueEmptyXA` (String connectionFactoryJndiName, String queueJndiName, long timeout)
 - 在 XA 事务控制下断言指定的消息队列为空。
- `JbpmTestCase.jmsStartTopicListener` (String connectionFactoryJndiName, String topicJndiName, boolean transacted, int acknowledgeMode)
 - 开始监听消息主题，返回消息主题对象 `JmsTopicListener`。
- `JbpmTestCase.jmsStartTopicListenerXA` (String connectionFactoryJndiName, String topicJndiName)
 - 在 XA 事务控制下开始监听消息主题，返回消息主题对象 `JmsTopicListener`。
- `JmsTopicListener.getNextMessage` (long timeout)
 - 获取消息主题中的下一条消息。
- `JmsTopicListener.stop` ()
 - 停止对该消息主题的监听。

例如，在完成 jms 活动以后，可以像这样验证发出的 JMS 消息：

```
//提供 connection-factory 和 destination 的 JNDI 名称，获取消息对象
MapMessage mapMessage = (MapMessage)
jmsConsumeMessageFromQueue("java:/JmsXA", "queue/ProductQueue");
//断言消息内容如预期
assertEquals("shampoo", mapMessage.getString("product"));
```

同时，`JbpmTestCase` 基于 `Mockrunner` 项目包装了如下 API，使您能够在独立的运行环境中模拟 JMS 服务的提供者而无须启动任何 JMS 应用服务器，例如创建消息队列

和消息主题等：

- `void jmsCreateQueue (String connectionFactoryJndiName, String queueJndiName)`
 - 创建一个消息队列
- `void jmsRemoveQueue (String connectionFactoryJndiName, String queueJndiName)`
 - 删除一个消息队列
- `void jmsCreateTopic (String connectionFactoryJndiName, String topicJndiName)`
 - 创建一个消息主题
- `void jmsRemoveTopic (String connectionFactoryJndiName, String topicJndiName)`
 - 删除一个消息主题

提示：Mockrunner 项目是基于 Java EE 环境的单元测试框架。它支持对 Struts、Servlet、Filter、标签库等 Java EE 技术进行单元测试，还提供 JDBC 和 JMS 的测试框架。因此，Mockrunner 可以用于测试整套基于 EJB 的应用程序。

Mockrunner 项目的网址为 <http://mockrunner.sourceforge.net>。

例如，您可以在单元测试的 `setUp` 和 `tearDown` 方法中分别创建和删除一个 JMS 消息队列，模拟一个 JMS 应用服务器的环境，就像这样简单：

```
//在单元测试开始前，创建消息队列
protected void setUp() throws Exception {
    super.setUp();
    jmsCreateQueue("java:/JmsXA", "queue/ProductQueue");
}
//在单元测试结束后，删除消息队列，释放内存
protected void tearDown() throws Exception {
    jmsRemoveQueue("java:/JmsXA", "queue/ProductQueue");
    super.tearDown();
}
```

10.4.2 JMS 文本消息

图 10-7 所示是一条将一段文本消息发送到指定目的地的 JMS 流程定义。

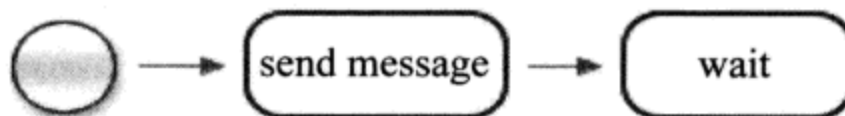


图 10-7 发送 JMS 文本消息的流程定义

对应的 jPDL:

```
<process name="JmsQueueText">
  <start>
    <transition to="send message"/>
  </start>
  <jms name="send message"
    connection-factory="java:JmsXA"
    destination="queue/jbpm-test-queue"
    transacted="false">
    <!-- text 元素指定文本格式的消息体 -->
    <text>This is the body</text>
    <transition to="wait"/>
  </jms>
  <state name="wait"/>
</process>
```

下面的单元测试程序根据上面的流程定义发起实例。这会发送一条消息到队列 queue/jbpm-test-queue，JMS 连接工厂的名称为 Java:JmsXA，消息体是文本“This is the body”:

```
//发起流程实例，产生队列消息
executionService.startProcessInstanceByKey("JmsQueueText");
//利用测试 API jmsConsumeMessageFromQueue 获取队列中的消息。
TextMessage textMessage =
    (TextMessage) jmsConsumeMessageFromQueue (
        "java:JmsXA",
        "queue/jbpm-test-queue",
        1000, false, Session.AUTO_ACKNOWLEDGE);
//断言消息文本如预期
assertEquals("This is the body", textMessage.getText());
```

注意：别忘了在运行上述代码前创建消息的目的地队列，调用测试 API “jmsCreateQueue("java:JmsXA", "queue/jbpm-test-queue");”，否则单元测试程序会报告“找不到 JMS 服务”异常。这一点在下面章节的介绍中将不再提醒。

10.4.3 JMS Object 消息

图 10-8 所示是一条将一个可序列化的 Java Object 作为消息发送到指定目的地的流

程定义。

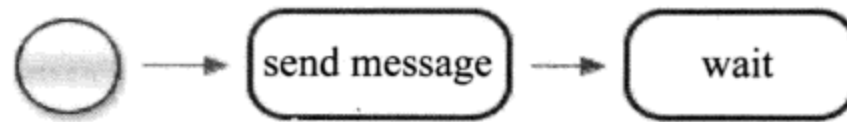


图 10-8 发送 JMS Object 消息的流程定义

对应的 jPDL:

```
<process name="JmsQueueObject">
  <start>
    <transition to="send message"/>
  </start>
  <jms name="send message"
    connection-factory="java:JmsXA"
    destination="queue/jbpm-test-queue"
    transacted="false">
    <!-- 在这里使用一个名为 object 的流程变量对象作为消息的内容发送 -->
    <object expr="${object}"/>
    <transition to="wait"/>
  </jms>
  <state name="wait"/>
</process>
```

发起该流程实例，则消息会被发送到名为 `queue/jbpm-test-queue` 的队列，消息体是序列化的 Java Object。在下面的单元测试代码中，先设置一个字符串到名称为 `object` 的流程变量，在流程定义中通过表达式引用提供给 `jms` 活动：

```
Map<String, Object> variables = new HashMap<String, Object>();
//设置 object 变量，以作为 JMS 消息发送
variables.put("object", "this is the object");
//带流程变量发起实例
executionService.startProcessInstanceByKey("JmsQueueObject", variables);
//获取 Object 格式的消息
ObjectMessage objectMessage = (ObjectMessage)jmsConsumeMessageFromQueue(
    "java:JmsXA",
    "queue/jbpm-test-queue",
    1000, false, Session.AUTO_ACKNOWLEDGE);
//断言消息如预期
assertEquals("this is the object", objectMessage.getObject());
```

10.4.4 JMS Map 消息

图 10-9 所示是一条将“键-值”结构的 Java Map 作为消息发送到指定目的地的流程定义：

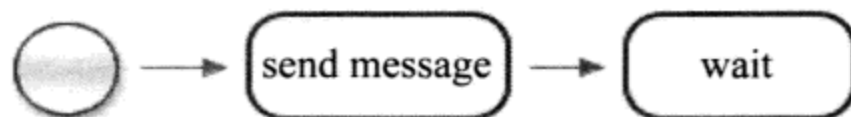


图 10-9 发送 JMS Map 消息的流程定义

对应的 jPDL:

```
<process name="JmsQueueMap">
  <start>
    <transition to="send message"/>
  </start>
  <jms name="send message"
    connection-factory="java:JmsXA"
    destination="queue/jbpm-test-queue"
    transacted="false">
    <!-- 在 map 元素中使用 entry 元素定义“键-值”结构的 Map 消息 -->
    <map>
      <entry>
        <key><string value="x"/></key>
        <value><string value="foo"/></value>
      </entry>
    </map>
    <transition to="wait"/>
  </jms>
  <state name="wait"/>
</process>
```

下列单元测试代码发起流程实例，并验证之：

```
//发起流程实例
executionService.startProcessInstanceByKey("JmsQueueMap");
//获取 Map 格式的消息
MapMessage mapMessage = (MapMessage)jmsConsumeMessageFromQueue(
    "java:JmsXA",
    "queue/jbpm-test-queue",
```

```

    1000, false, Session.AUTO_ACKNOWLEDGE);
//断言 Map 消息具有名称为 x 的键
assertTrue(mapMessage.itemExists("x"));
//断言 x 键对应的值为预期的 foo
assertEquals("foo", mapMessage.getObject("x"));

```

10.5 历史会话监听链

在流程实例的生命周期过程中会经历很多事件，例如流程实例的创建、任务的分配、活动的结束等，这一系列“历史事件”在发生时都会被工作流引擎默认的“历史会话”处理，记录发生时间等历史信息，然后写入持久层即数据库中。

现在，jBPM4 允许开发者自定义历史事件监听器，即定制自己的“历史会话”。当监听的历史事件被触发时，自定义的历史会话也会被调用。

在 jBPM4 配置文件的 `transaction-context` 节点中配置您自己的历史会话。自定义的历史会话监听之所以被称为“链”，是因为所有的历史事件都会被历史会话捕获，它们将按照配置顺序依次被调用，您可以在自己的历史会话中捕捉并处理需要的历史事件。这可以理解为监听器设计模式和职责链设计模式的配合使用。

以下 jBPM4 配置文件片段中的 `MyProcessStartListener` 和 `MyProcessEndListener` 就是自定义的历史会话：

```

...
<transaction-context>
  <history-sessions>
    <object class="org.jbpm.test.historysessionchain.
      MyProcessStartListener" />
    <object class="org.jbpm.test.historysessionchain.
      MyProcessEndListener" />
  </ history-sessions>
</transaction-context>
...

```

在 jBPM4 默认的配置文件 `jbpm.default.cfg.xml` 中，可以发现系统默认的历史会话配置：

```

<transaction-context>
  ...

```

```

        <history-sessions>
            <object
class="org.jbpm.pvm.internal.history.HistorySessionImpl" />
            </history-sessions>
        ...
    </transaction-context>

```

自定义历史会话需要在 classpath 中。历史会话必须实现 org.jbpm.pvm.internal.history.HistorySession 接口，如下例：

```

public class MyProcessStartListener implements HistorySession {
    //process 方法来自于 HistorySession 接口
    public void process(HistoryEvent historyEvent) {
        //在应用中，应该根据我们的实际需要来捕获相应的历史事件 (historyEvent)
        //在这里，我们只处理流程实例创建的事件——ProcessInstanceCreate
        if (historyEvent instanceof ProcessInstanceCreate) {
            ...
        }
    }
}

```

如上面的代码，我们可以根据 HistoryEvent 对象判断具体的历史事件类型。目前支持的历史事件类型有：

- ActivityEnd ——活动结束。
- ActivityStart ——活动开始。
- AutomaticEnd ——自动活动结束，即流程陷入等待状态。
- DecisionEnd ——判断活动结束。
- ProcessInstanceCreate ——流程实例创建。
- ProcessInstanceEnd ——流程实例结束。
- ProcessInstanceMigration ——流程实例合并。
- TaskActivityStart ——任务活动开始。
- TaskAssign ——任务分配。
- TaskComplete ——任务完成，继承自活动结束 (ActivityEnd) 事件。
- TaskCreated ——任务创建。
- TaskDelete ——任务被删除，注意区别于任务完成 (TaskComplete) 事件。
- TaskUpdated ——任务更新。
- VariableCreate ——变量创建。

- VariableUpdate ——变量更新。

它们都位于包 `org.jbpm.pvm.internal.history.events` 中,您可以查看这些历史事件的源代码获取它们支持的属性和操作信息。

10.6 自定义 Web 任务表单

我们说过, workflow 管理系统最重要的特点之一就是良好地支持人机交互的过程。而通过 Web, 使用表单来帮助 workflow 引擎获取用户的输入无疑是当前体验最好的方式之一了。

在 jBPM4 中使用 Web 任务表单, 可以通过 `form` 属性将表单绑定在 `task` 活动或 `start` 活动上。

如果您使用 jBPM Web Console (jBPM 官方发布的 Web 控制台) 作为 workflow 客户端时, 您配置的基于 FreeMarker 模板库的表单会自动被渲染并显示给用户, 本节的示例流程定义 `VacationRequest` 的 Web 表单就是基于 FreeMarker 技术、并通过 jBPM Web Console 进行展现的。

提示: FreeMarker 是开源的 Java 模板引擎, 是基于模板生成文本输出的通用工具, 使用纯 Java 语言编写。FreeMarker 一般被用来设计和生成 HTML Web 页面, 特别适用于基于 MVC 模式的 Java EE 应用程序。

FreeMarker 项目的官方网址是 <http://freemarker.sourceforge.net>。

10.6.1 基本思路

`start` 活动和 `task` 活动的定义支持 `form` 属性, `form` 属性的值即对表单的引用。无论您引用的是 `jsp` 表单还是 `ftl` (FreeMarker 模板库) 表单, jBPM4 workflow 引擎本身并不对这个属性做任何解释, 只是负责持久化它。图 10-10 所示是应用任务表单的流程定义 `VacationRequest`。



图 10-10 应用任务表单的流程定义

对应的 jPDL:

```

<process name="VacationRequest" xmlns="http://jbpm.org/4.3/jpdl">
  <start form="com/examples/jbpm4/taskform/request_vacation.ftl"
name="start">
    <transition to="verify_request" />
  </start>
  <task form="com/examples/jbpm4/taskform/verify_request.ftl"
    candidate-users="mike,peter" name="verify_request">
    <transition name="reject" to="vacation_rejected" />
    <transition name="accept" to="vacation_accepted" />
  </task>
  <end name="vacation_accepted" />
  <end name="vacation_rejected" />
</process>

```

workflow管理系统的客户端应用程序应该能从流程 API 中获取 form 属性的值,所以,对于任务表单的展现工作应该交给它。

10.6.2 表单格式

上面的流程定义 VacationRequest 中引用的任务表单是基于 FreeMarker 模板库的(扩展名为.ftl),jBPM4 Web Console 正好是这么一种能渲染和展现 ftl 任务模板的工作流客户端应用程序。

基于 jBPM4 Web Console 的 ftl 任务表单的结构是可以包含任意文本的 HTML 页面。但是,这个 HTML 页面中必须包含一个 form 元素,它需要遵循的规范如下。

- 需要以 ftl 为扩展名。

- 必须被部署到 jBPM4 持久化资源库中，这个操作可以使用如下 API 完成：

```
deployment.addResourceFromClasspath("com/examples/jbpm4/taskform/verify_request.ftl");
```

- form 元素的 action 属性必须是 `${form.action}`。
- form 元素的 enctype 属性需要是 `multipart/form-data`。
- form 元素中的每个输入控件会成为同名的流程变量。
- 所有的流程变量都会寻找 form 元素中同名的输入控件，注入值。
- form 元素中可以预设一个名称为 `outcome` 的控件，任务完成后这个控件的值将指示流程的下一步转移名称。

以下是示范的任务表单 `verify_request.ftl`：

```
<html>
<head>
...
</head>
<body>
  <!-- form (HTML 表单) 元素是必需的，如果用在 jBPM4 Web Console 上，其 action 属性也是固定的 -->
  <form action="${form.action}" method="POST" enctype="multipart/form-data">
    <!-- jBPM4 Web Console 会自动将下面这两个流程变量的值注入页面 -->
    <h3>Your employee, ${employee_name} would like to go on vacation</h3>
    Number of days: ${number_of_days}<br/>
    <hr/>
    In case you reject, please provide a reason :<br/>
    <!-- 下面这个 HTML 输入控件将会和流程变量 reason 绑定 -->
    <input type="text" name="reason"/><br/>
    <#list outcome.values as transition>
    <!-- 这个 submit 控件的名称为 outcome，则点此提交任务后，jBPM4 Web Console 将根据其值选择流程的转移 -->
    <input type="submit" name="outcome" value="${transition}"/>
    </#list>
  </form>
</body>
</html>
```

将流程定义和任务表单 `verify_request.ftl` 部署到 jBPM4 Web Console 上后发起流程实例，则任务表单 `verify_request.ftl` 将会被渲染成如图 10-11 所示的界面。

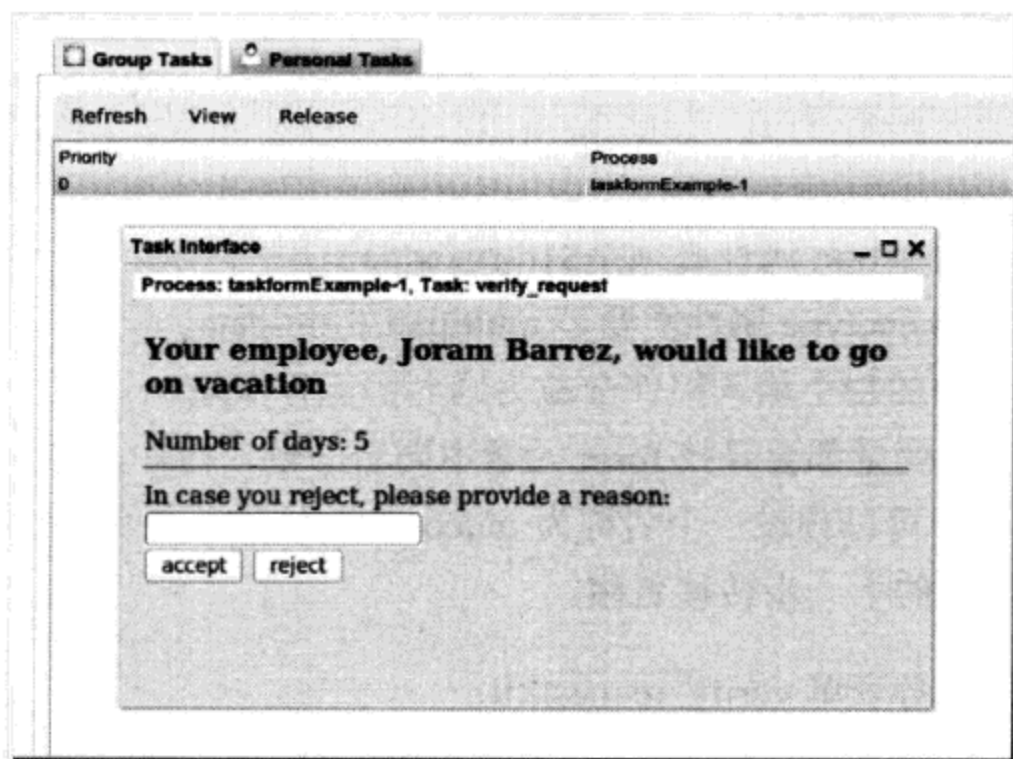


图 10-11 jBPM4 Web Console 渲染出的 ftl 任务表单

以上介绍的只是针对 jBPM4 Web Console 的 ftl 表单的处理思路,在大部分情况下,我们的工作流客户端应用展现可能是诸如 JSP 之类的技术。但正如本书所强调的, jBPM4 只是在持久化层中替您保存一个表单引用,即仅仅保存对客户端应用展现的一个“引用”而已。对于如何运用这个引用去渲染出界面、打通双向的数据交换(数据的界面展现和后台提交),还是需要您——业务流程应用的开发者去自行实现,当然,这也给了您在前端展现层的发挥空间。本节所介绍的例子对您使用 JSP 等技术去展现任务表单也有很好的借鉴作用。

10.7 流程实例的自动迁移

一般情况下, workflow 管理系统在流程定义版本升级后,会使用最新的流程定义版本发起流程实例。如果需要的话,也可以使用一个指定的旧流程定义版本来发起流程实例。但是,已经存在的流程实例则会一直绑定在发起它们的流程定义下运行,直到结束。这是 jBPM3 版本和世界上其他大多数 workflow 管理系统默认遵循的规则。

但是您的客户需求可能会对这个规则提出挑战,例如下面的两种情况。

- 当基于旧流程定义的实例不再重要时,要求:当新流程定义被发布后,这些旧的流程实例需要被立即无条件终止。

- 这是更为复杂的情况，我们需要对所有的或者一些特定的旧流程实例进行迁移，将这些基于旧流程定义的运行中的实例按照一定的规则切换到新版本的流程定义上去。

在 jBPM4 之前我们也许会对这些需求说“不”，或者说服客户接受一些变通的处理办法——例如人工迁移等。

现在 jPDL4 提供了一个工具——“migrate-instances”流程定义元素，可以比较好地自动支持上述两种需求。目前，jBPM4.3 版本已经稳定支持了 state 活动迁移的实现，但对于其他活动迁移的覆盖也许并不如您预期得那么好，当然，jBPM4 的开发者的承诺会在未来的版本中完全支持所有活动类型。

下面的 4 个子节将介绍 4 种流程实例迁移的处理场景。这 4 种场景下的流程定义都是基于如下原始流程定义的升级版本。原始的流程定义版本如图 10-12 所示。

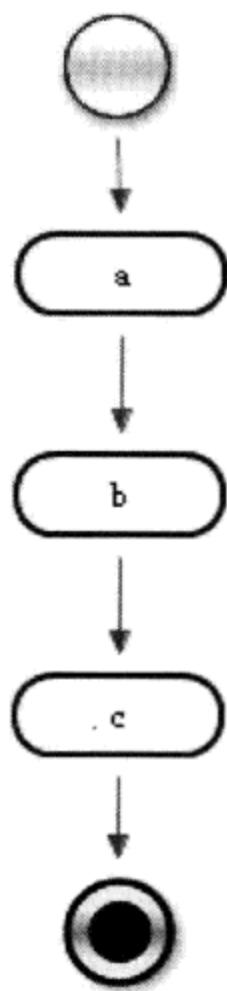


图 10-12 原始的流程定义

对应的 jPDL:

```
<process name="foobar" 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">
  <transition to="end" />
</state>
<end name="end" />
</process>

```

这是一个很简单的 3 状态变迁的线性流程定义，以下我们将基于它的 4 个升级版本进行流程实例迁移。

10.7.1 简单的流程实例迁移

如果新版本的流程定义相对旧版本的流程定义满足如下条件：

- 流程定义结构相同，覆盖旧流程定义的所有活动名称。
- 允许新流程定义有新活动添加，但是旧流程定义的所有活动都在新流程定义中存在。

那么，可以使用最简单的流程实例迁移方法来迫使旧的流程实例切换到新的定义上运行。这样做的意义之一在于可以把新的事件监听器定义“强加”在旧的流程实例上。

在旧流程定义基础上修改的新流程定义 jPDL 如下：

```

<process name="foobar" 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">
        <transition to="end"/>
    </state>
    <end name="end"/>
    <!-- 标识实例迁移的元素在此 -->
    <migrate-instances/>
</process>

```

实例迁移元素 `migrate-instances` 只有一个属性 `action`，默认值为 `migrate`，如上例所示。

假设上面的新 `foobar` 流程定义被部署，而旧的 `foobar` 流程定义已经有两个实例在运行了，这两个流程实例一个停留在 `state` 活动 `a` 上，另一个停留在 `state` 活动 `b` 上，那么这两个流程实例就会“被迁移”，如下单元测试代码可验证之：

```

//部署旧版本的 foobar 流程定义（即一开始介绍的“原始的流程定义”）
ProcessDefinition pd1 =

    deployProcessDefinition("com/examples/jbpm4/n3_migrate/process.jpdl.xml");

// startProcAndSignalTo（具体实现见后面的代码片段）这个方法将发起流程实例，并将流程实例执行到指定的活动。在这里将旧流程定义的实例执行到 a 活动
ProcessInstance pi1 = startProcAndSignalTo(pd1, "a");
//将旧流程定义的实例执行到 b 活动
ProcessInstance pi2 = startProcAndSignalTo(pd1, "b");
//部署新版本的 foobar 流程定义（即本节新定义的流程）
ProcessDefinition pd2 =

    deployProcessDefinition("com/examples/jbpm4/n3_migrate/simple/process.jpdl.xml");

//获取被新版本流程定义迁移后的两个实例：pi1 和 pi2
pi1 = executionService.findProcessInstanceById(pi1.getId());
pi2 = executionService.findProcessInstanceById(pi2.getId());
// pd2.getId() 获取新版本流程定义的 ID。在这里断言流程实例绑定的定义版本已经被切换到了新版本
assertEquals(pd2.getId(), pi1.getProcessDefinitionId());
assertEquals(pd2.getId(), pi2.getProcessDefinitionId());
//断言被迁移后流程实例的当前活动没有受到影响
assertEquals(pi1, pi1.findActiveExecutionIn("a"));
assertEquals(pi2, pi2.findActiveExecutionIn("b"));

```

上面代码中的自定义方法实现如下：

`deployProcessDefinition` 是部署指定路径（`procFilePath`）下流程定义的方法，为上面的单元测试代码所使用（在后面的 3 个小节中仍然会用到）：

```
protected ProcessDefinition deployProcessDefinition (String procFilePath)
{
    //部署流程定义
    String deploymentId = repositoryService.createDeployment()
        .addResourceFromClasspath(procFilePath).deploy();
    deploymentIds.add(deploymentId);
    //返回流程定义对象
    return
repositoryService.createProcessDefinitionQuery().deploymentId(
        deploymentId).uniqueResult();
}
```

`startProcAndSignalTo` 是根据流程定义（`procDef`）发起实例、并执行到指定活动（`actName`）的自定义方法。为上面的单元测试代码所使用（在后面的 3 个小节中仍然会被用到）：

```
protected ProcessInstance startProcAndSignalTo (
    ProcessDefinition procDef,
    String actName) {
    //发起流程实例
    ProcessInstance procInst = executionService.startProcessInstanceById
(procDef.getId());
    //不断向前推进流程实例，直到到达指定的活动
    while (!procInst.isActive(actName)) {
        procInst = executionService.signalExecutionById(procInst.getId());
    }
    //返回流程实例对象
    return procInst;
}
```

10.7.2 终止流程实例运行的迁移

凭经验来说，大多数业务会要求当新的流程定义产生后，依赖旧流程定义的实例自动结束。这个需求被 `migrate-instances` 元素的 `action="end"` 属性满足。实现此需求的新流程定义 jPDL 如下：


```

<process name="foobar" 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">
    <transition to="end" />
  </state>
  <end name="end"/>
  <!-- 以上的定义都与旧版本相同。不同的是，在下面设置了迁移的 action 属性为 end。 -->
  <migrate-instances action="end" />
</process>

```

同样地部署新旧两套流程定义，发起两条旧流程定义的实例（在部署新流程定义之前），一条到达 a 活动，一条到达 b 活动。不同的是这两条基于旧流程定义的实例将被终止，如下代码所示：

```

//部署旧版本的流程定义
ProcessDefinition pd1 =

deployProcessDefinition("com/examples/jbpm4/n3_migrate/process.jpdl.
xml");

//发起两条旧版本流程定义的实例，分别到达 a 活动和 b 活动
ProcessInstance pi1 = startProcAndSignalTo(pd1, "a");
ProcessInstance pi2 = startProcAndSignalTo(pd1, "b");
//部署新版本的流程定义

deployProcessDefinition("com/examples/jbpm4/n3_migrate/end/process.j
pdl.xml");

//断言流程实例已经终止
assertProcessInstanceEnded(pi1);
assertProcessInstanceEnded(pi2);

```

被 migrate-instances 元素终止的流程实例状态会被设置为 aborted，这个状态将被持久化在数据库中。

10.7.3 应用活动映射的迁移

在一些比较复杂的迁移业务情况下，我们需要映射旧流程定义的活动到新流程定义的活动。这种需求会被用在如下场景中：

- 1) 新版本的流程定义删除了某些旧流程定义的活动。
- 2) 旧版本流程定义的某些活动名称在新流程定义中被修改了。

为了支持这些业务需求，`migrate-instances` 元素提供了 `activity-mapping` 子元素。`activity-mapping` 元素拥有两个必需属性：

- `old-name`——映射的源，旧版本流程定义的活动名称。
- `new-name`——映射的目标，新版本流程定义的活动名称。

举例说明。应用活动映射的新流程定义 jPDL 如下：

```
<process name="foobar" 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">
    <transition to="d" />
  </state>
  <!-- 在这里相比旧流程定义增加了 d 活动 -->
  <state name="d">
    <transition to="end" />
  </state>
  <end name="end"/>
  <migrate-instances>
    <!-- 映射旧定义的 b 活动到新定义的 a 活动 -->
    <activity-mapping new-name="a" old-name="b" />
    <!-- 映射旧定义的 c 活动到新定义的 d 活动 -->
    <activity-mapping new-name="d" old-name="c" />
  </migrate-instances>
</process>
```

</process>

根据上述定义，则新旧两个版本流程定义的活动映射关系如图 10-13 所示。

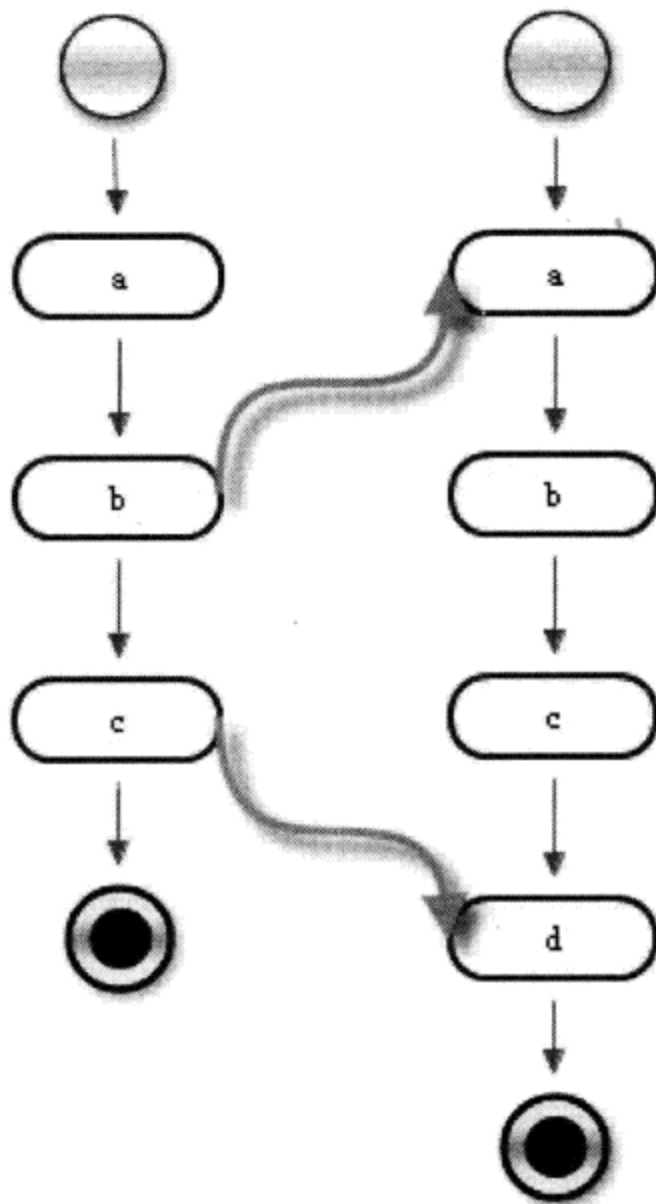


图 10-13 新旧版本流程定义的活动映射关系图

则当新流程定义发布后：旧流程定义的实例如果在 b 活动等待，则将被切换到新的 a 活动；旧流程定义的实例如果在 c 活动等待，则将被切换到新的 d 活动。如下单元测试代码验证了这种情况：

```
//部署旧流程定义
ProcessDefinition pd1 =

    deployProcessDefinition("com/examples/jbpm4/n3_migrate/process.jpdl.
xml");

//基于旧流程定义发起 3 个实例，分别执行到 a, b, c 活动
```

```

        ProcessInstance pi1 = startProcAndSignalTo(pd1, "a");
        ProcessInstance pi2 = startProcAndSignalTo(pd1, "b");
        ProcessInstance pi3 = startProcAndSignalTo(pd1, "c");
        //部署新流程定义
        ProcessDefinition pd2 =

        deployProcessDefinition("com/examples/jbpm4/n3_migrate/activitymapping/process.jpdl.xml");
        //获取经过迁移后的流程实例
        pi1 = executionService.findProcessInstanceId(pi1.getId());
        pi2 = executionService.findProcessInstanceId(pi2.getId());
        pi3 = executionService.findProcessInstanceId(pi3.getId());
        //断言流程定义已经切换到新版本
        assertEquals(pd2.getId(), pi1.getProcessDefinitionId());
        assertEquals(pd2.getId(), pi2.getProcessDefinitionId());
        assertEquals(pd2.getId(), pi3.getProcessDefinitionId());
        //根据迁移的规则, 经过迁移后, 断言 a 活动仍然在 a 活动
        assertEquals(pi1, pi1.findActiveExecutionIn("a"));
        //断言 b 活动被映射到了 a 活动
        assertEquals(pi2, pi2.findActiveExecutionIn("a"));
        //断言 c 活动被映射到了 d 活动
        assertEquals(pi3, pi3.findActiveExecutionIn("d"));
        //以下 3 个流程实例分别继续执行
        pi1 = executionService.signalExecutionById(pi1.getId());
        pi2 = executionService.signalExecutionById(pi2.getId());
        //注意: 这里流程实例 2 被执行了两次
        pi2 = executionService.signalExecutionById(pi2.getId());
        pi3 = executionService.signalExecutionById(pi3.getId());
        //断言: 根据新流程定义 a 活动执行到了 b 活动
        assertEquals(pi1, pi1.findActiveExecutionIn("b"));
        //断言: 由于流程实例 2 被执行了两次, 所以它由 a 活动到达了 c 活动
        assertEquals(pi2, pi2.findActiveExecutionIn("c"));
        //断言: 位于 d 活动的流程实例 3 经过一次执行, 就结束了
        assertTrue(pi3.isEnded());

```

10.7.4 自定义迁移处理器

如果以上的流程实例迁移方式仍然不能满足您的需求, 那么如您所知道的那样, jBPM 总会提供一种“用户代码”机制, 来让您“自由”地定制自己的迁移逻辑——这就是(流程实例)迁移处理器。

自定义迁移处理器需要实现 `org.jbpm.pvm.internal.migration.MigrationHandler` 接口，它只有一个方法，代码如下：

```
public interface MigrationHandler {
    //唯一的接口方法：migrateInstance，用来实现自定义的迁移逻辑。
    //3个参数分别为：新流程定义对象、当前流程实例对象和迁移描述对象。
    void migrateInstance(
        ProcessDefinition newProcessDefinition,
        ProcessInstance processInstance,
        MigrationDescriptor migrationDescriptor);
}
```

迁移处理器需要在流程定义的 jPDL 中作为 `migration-handler` 元素的属性来指定。一旦指定了自定义的迁移处理器，那么此流程定义所有需要迁移的实例都会在执行了系统默认的迁移处理逻辑后被自定义的迁移处理器处理；如果您设定了 `migrate-instances` 的 `action="end"`，则自定义的迁移处理器会在流程实例结束之前被调用。

我们通过迁移处理器可以在迁移流程实例时加入定制的逻辑；或设定特殊的流程实例结束状态；或根据 `migrationDescriptor` 参数，即迁移的上下文对象拿到 `migration-handler` 元素的全部定义信息，实现定制的活动映射逻辑等。以下是一个应用迁移处理器的新流程定义：

```
<process name="foobar" 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">
        <transition to="end" />
    </state>
    <end name="end" />
    <migrate-instances>
        <!-- 在这里，通过 class 属性指定迁移处理器 -->
        <migration-handler
```

```

        class="com.examples.jbpm4.n3_migrate.custom.MigrationHandlerImpl" />
    </migrate-instances>
</process>

```

以下是自定义迁移处理器的代码：

```

public class MigrationHandlerImpl implements MigrationHandler {
    public void migrateInstance(
        ProcessDefinition newProcessDefinition,
        ProcessInstance processInstance,
        MigrationDescriptor migrationDescriptor) {
        //获取流程实例执行服务
        ExecutionService executionService = EnvironmentImpl.getFromCurrent
        (ExecutionService.class);
        if (executionService == null) {
            return;
        }
        //这里是自定义的迁移逻辑：结束基于旧流程定义的实例，并设置其状态为“aborted by
        migration”（因为迁移而中止）
        executionService.endProcessInstance(processInstance.getId(),
        "aborted by migration");
    }
}

```

验证以上流程定义迁移的单元测试代码如下：

```

//部署旧流程定义
ProcessDefinition pd1 =

    deployProcessDefinition("com/examples/jbpm4/n3_migrate/process.jpdl.
    xml");
    //基于旧流程定义发起两个流程实例
    ProcessInstance pi1 = startProcAndSignalTo(pd1, "a");
    ProcessInstance pi2 = startProcAndSignalTo(pd1, "b");
    //断言流程实例未结束
    assertNotNull(pi1);
    assertNotNull(pi2);
    //部署新流程定义

    deployProcessDefinition("com/examples/jbpm4/n3_migrate/custom/proces
    s.jpdl.xml");
    //断言：根据自定义迁移处理器的逻辑，流程实例被结束

```

```
assertProcessInstanceEnded(pi1.getId());  
assertProcessInstanceEnded(pi2.getId());
```

10.8 小结

本章深入介绍了 jBPM4 一些高级功能的应用，这包括从这些功能的 jPDL 定义和 Service API 执行两个方面。尽管这些功能有些并没有被流程设计器等工具完美地支持或存在一定的局限性，但是这些功能必然可以为一些关键的业务需求所用，这包括：

- timer（定时器）和工作日历的应用。
- group 活动进行流程活动编组。
- 对 EJB 方法调用的支持。
- 使用 jms 活动“生产”和“消费”消息。
- 使用历史会话监听链，通过监听流程事件的方式记录流程实例的执行历程。
- 使用 jBPM4 的自定义任务表单机制开发客户端应用程序的展现层。
- 流程定义版本升级后，流程实例的迁移方法。

最后，您需要了解，根据 jBPM 官方的说法，随着版本的升级，这些高级功能会越来越完善，得到的工具支持会越来越好。



升级 jBPM3 到 jBPM4

本章内容可能是很多读者刚拿到本书时最先翻阅的。毕竟 jBPM3 经过这么多年的发展和应用已经深入人心。也许很多已经熟悉了 jBPM3 的开发者并不是太情愿接受 jBPM4 的推陈出新，因为在 jBPM3 的基础上解决各种复杂业务流程场景的实践已经十分丰富了。也许很多开发者还在怀疑 jBPM4 能否满足他们的需求，因为目前看来 jBPM4 的一些功能还有待完善……

但是，您阅读了本书之后，会发现 jBPM4 构建在一个比 jBPM3 更为先进的模型基础之上。jBPM4 从架构上解决了很多 jBPM3 不可能彻底根治的“顽疾”（例如历史流程归档、流程实例迁移等）。因为有了 PVM，jBPM4 的 API 更为灵活、更易扩展，更加将 jBPM 系列产品的开放精神发扬光大。相信不久之后，基于 jBPM4 的第三方工具（例如 Signavio Web 流程设计器）和业务解决方案将会层出不穷。所以，我们为什么不尽早把思维从 jBPM3 切换到 jBPM4 呢？

从引擎架构上来说，jBPM4 与它的前辈相比有如下特点：

- jBPM4 流程活动（在 jBPM3 中称为节点）类型的划分更清晰。
 - 将等待型活动、自动型活动、流转控制型活动明确地划清了界限。
- jBPM4 推出了基于“观察者设计模式”的事件-监听器（Event-Listener）模型。
 - jBPM4 中的活动实现（ActivityImpl）、转移实现（TransitionImpl）、流程定义实现（ProcessDefinitionImpl）等，都继承自 ObservableElementImpl 类型，因此组成流程定义的这 3 种主要元素“天生”就可以被用来“观察”。这些元素直接支持被注册各种事件，然后由相应的监听器来处理这些事件的触发，即在这些流程定义元素上您可以很方便地定制自己的“事件”。
- jBPM4 内核引擎采用了基于“命令设计模式”的执行模型。
 - jBPM4 采用了 Excution 代替 jBPM3 中的 Token。流程的推进依赖于在执行对象（ExcutionImpl）中调用各个原子操作（AtomicOperation，即独立的事务操作）的“命令（Command）”。
- jBPM4 将已完成的流程实例归档到流程历史库。

- 将不常使用的、理论上只读的流程历史数据与正在运行的流程数据进行分离，完美地解决了 jBPM3 饱受诟病的效率问题，即 jBPM 系统运行一段时间后，大量的历史数据和当前运行数据混合在一起，导致极大降低持久层存取效率的问题。
- jBPM4 首次推出了旧定义流程实例自动向新版本流程定义迁移的语义和机制。
 - 使得业务流程变更和重组的“无缝切换”成为可能。
- jBPM4 的架构清晰地区分了普通流程应用和高级流程应用之间的区别。
 - 相对而言，jBPM4 较为容易入门。但要开发出高级流程应用还是有一定门槛的，因为功能多了，扩展性更强了。
- 最后，根据 jBPM 官方的说法，在 jBPM3 时代，他们仅根据社区反馈来提升产品的稳定性和扩展性。到了 jBPM4 时代，他们在“JBoss OA 实验室”的持续集成环境中系统地构建和测试 jBPM，以保证 jBPM4 有更好的产品质量和更长的生命周期。

11.1 你所要知道的升级局限性

不要指望能完全“平滑”地、“自动化”地升级。

因为 jBPM4 为了在普通流程应用和高级流程应用之间做清晰的分隔，其 API 相对 jBPM3 已经完全重构了。jBPM4 和 jBPM3 中的包是严格分开的，也就是说二者的包路径是不存在重合的、也不会有任何冲突发生；另外，数据库表名的前缀由“JBPM_”改为了“JBPM4_”，这也保证了两个版本的数据库表完全不会重叠。上述设计的好处是保证了 jBPM3 和 jBPM4 的实例可以同时运行在一个应用中，这能够让您在两套 jBPM 系统的并行运行中完成升级。

遗憾的是，目前还没有可以把 jBPM3 数据库直接转换为 jBPM4 数据库的工具，因为业务（特别是敏感业务）数据升级带来的验证成本是巨大的。

另外，某些 jBPM3 中已经提供的功能在 jBPM4.3 版本中还没有实现。这些功能按照 jBPM4 在未来“可能”到“不可能”去实现的顺序如下：

- 用户自定义持久化和事务资源，例如使用 JDBC 代替 Hibernate。
- 异常处理器。

- 临时流程变量。
- 支持身份认证表达式，用来计算任务的分配人。
- 任务表单的“变量 - 参数转化器”。

11.2 流程定义转换工具

当您基于 jBPM3 的业务流程管理系统运行了很长时间后，必然会沉淀下许多使用 jPDL3 定义的流程，那么在升级到 jBPM4 的过程中您不可避免地要将其转换成 jPDL4 的格式。

幸运的是，jBPM4 发布包中提供了一个升级工具叫做 **migration**，它位于发布包的 **migration** 目录中。这里有一个命令行工具，可以把 jPDL3 流程定义文件转换成 jPDL4 流程定义文件，从而使您避免了完全使用手工方式升级 jPDL 流程定义文件。

转换后的流程定义可能没法运行，因为某些 jBPM4 的特性仍然可能会被忽略掉，或者转换器工具本身还没有实现某些功能。但是，至少这个工具能替我们预先解决掉大量乏味的格式转换工作。

这个工具的原理只是使用 **dom4j** 工具将两种格式的 XML 进行一下转换。因此，**migration** 工具的实现非常简单（这是故意的），其大多数转换逻辑都可以在 **org.jbpm.jpdl.internal.convert.Jpdl3Converter** 类中找到，这样做正是为了方便您根据自身业务的需要进行扩展。**Migration** 工具的源代码在同一路径下的 **src** 目录中。

11.2.1 命令行执行

执行 **migration** 目录中的 **jpdl-migration-4.X.jar**（这是一个可执行的 jar 文件，其入口类为 **org.jbpm.jpdl.internal.convert.JpdlConverterTool**），可以把一个 jPDL3 格式的文件作为输入（<processfile>），输出是 jPDL4 格式的文件（-o <outputfile>）。语法如下：

```
java -jar jpdl-migration-4.X.jar -v -o <outputfile> <processfile>
```

参数说明：

- -v (verbose)
 - 使用这个参数，会在标准输出中打印转换流程定义文件过程中的细节信息，也会在抛出转换异常时打印异常栈。

- -o (output) <outputfile>
 - 指定转换后的输出文件名。如果不使用此参数，则会基于输入的流程定义文件名称输出一个后缀为“converted.jpdl.xml”的 jPDL4 格式文件。

3 个使用示例如下：

```
java -jar jpdl-migration-4.X.jar processdefinition.xml
java -jar jpdl-migration-4.X.jar -v processdefinition.xml
java -jar jpdl-migration-4.X.jar -o C:/process.jpdl.xml processdefinition.xml
```

11.2.2 Java 编码执行

转换工具 `org.jbpm.jpdl.internal.convert.Jpdl3Converter` 可以很方便地在 Maven, Ant 或者 Java 代码中被调用。

下面是一条基于 jPDL3 的流程定义，文件名称为 `processdefinition.xml`：

```
<?xml version="1.0" encoding="UTF-8"?>
<process-definition xmlns="urn:jbpm.org:jpdl-3.2" name="jBPM3ProcessDefine">
  <start-state name="start-state">
    <transition to="state"></transition>
  </start-state>
  <state name="state">
    <transition to="node"></transition>
  </state>
  <node name="node">
    <action expression="{ActionHandler}"></action>
    <transition to="task-node"></transition>
  </node>
  <task-node name="task-node">
    <task name="MyTask">
      <assignment actor-id="Alex"></assignment>
    </task>
    <transition to="end-state"></transition>
  </task-node>
  <end-state name="end-state"></end-state>
</process-definition>
```

下面的 Java 代码展示了如何编程转换上面的流程定义到 jPDL4 格式：

```
//获取需要转换的流程定义文件的路径
```

```

URL url = new URL("File:/C:/opensource/jboss/jbpm-4.3/examples/src/"
    + "com/examples/jbpm4/n3_jpdl3converter/processdefinition.xml");
//创建转换器对象
Jpdl3Converter jpdlConverter = new Jpdl3Converter(url);
//调用转换方法 readAndConvert。转换成功，返回 dom4j 的 XML Document 对象
Document jpdl4Doc = jpdlConverter.readAndConvert();
//将转换成功的 jPDL4 XML 流程定义输出到文件系统中
Writer fileWriter = new FileWriter("C:/process.jpdl.xml");
OutputFormat format = OutputFormat.createPrettyPrint();
XMLWriter writer = new XMLWriter(fileWriter, format);
writer.write(jpdl4Doc);
writer.close();

```

最后，上面的 processdefinition.xml 会被成功转换，在 C 盘输出 process.jpdl.xml，内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://jbpm.org/4.3/jpdl" name="jBPM3ProcessDefine">
  <start name="start-state">
    <transition to="state" />
  </start>
  <state name="state">
    <transition to="node" />
  </state>
  <script name="node" expr="#{ActionHandler}" lang="juel">
    <transition to="task-node" />
  </script>
  <task assignee="Alex" name="task-node">
    <transition name="end-state" to="end-state" />
  </task>
  <end name="end-state" />
</process>

```

当然，在转换过程中会出现很多警告（WARN），提示您 jPDL3 与 jPDL4 的不兼容之处……所以，您何不亲自去验证一下这个转换呢？

11.3 jBPM3 到 jBPM4 的语义变更及翻译

本节展示了 jBPM3 版本到 jBPM4 版本最重要的一批流程语义变更，及其转义映射关系，分别如表 11-1、表 11-2 和表 11-3 所示。

表 11-1 jBPM3 到 jBPM4 主要术语/称谓的修改

jBPM3	jBPM4	说明
Node	Activity	组成流程 (process) 的元素的称谓, 从节点 (Node) 改为活动 (Activity)
Token	Execution	令牌 (Token) 这个概念变成了 Execution——执行。 需要注意的是: <ul style="list-style-type: none"> 在 jBPM4 中“根 Execution 对象”和流程实例 (ProcessInstance) 对象合二为一。在 jBPM3 中, 流程实例对象和 Token 对象是完全不同的两种数据结构, 流程实例对象中总有一个指向“根 Token 对象”的引用 不同于 jBPM3, jBPM4 中的 Execution 对象可以在不被激活的情况下, 创建一个子 Execution 对象, 并让这个子 Execution 对象继续流程的执行, 并且在逻辑展现上表现出只有一个单独的执行路径。这种情况一般会发生在了声明了定时器 (timer) 的活动中
Action-Handler	Event-Listener	jBPM3 的“行为 - 处理句柄”模式已经变成 jBPM4 的“事件 - 监听器”模式, 请牢记这个变化

表 11-2 jPDL3 到 jPDL4 流程定义语言 XML 元素名称的映射

jBPM3	jBPM4
process-definition	process
event type="..."	on event="..."
action	event-listener
node	custom
process-state	sub-process
super-state	group (jBPM4.3 版本在图形化方面还未完全支持 group)

表 11-3 jPDL3 到 jPDL4 行为/事件的默认传播机制修改

jBPM3	jBPM 4
默认传播的事件触发器行为在所属元素的外部定义	默认传播的事件不会触发所属元素外部的 事件监听器，只会触发位于所属元素中的 事件监听器

11.4 小结

本章从整体局限性到实施细节介绍了如何将 jBPM3 版本的工作流管理系统升级到 jBPM4 版本。最后提供了 jBPM3 到 jBPM4 版本语义及其功能变化的对照表，方便 jBPM 的老用户延续概念、对照升级。

读者通过本章的知识可以最大程度地将遗留的 jBPM3 系统重用，从而保证系统的连续性和节省一定的开发成本，使得企业前期的投资不至于全部废弃。



jBPM4 PVM (Process Virtual Machine, 流程虚拟机) 的设计初衷是通过实现接口和定制插件等方式兼容多种流程定义语言和流程活动场景, 为“世界上”所有的业务流程定义提供一套通用 API 平台。那么, 无论是需要对 jBPM 原有流程定义语言进行扩展, 或者是重新实现一套专用的流程定义语言, 都可以通过实现 PVM 制定的接口规范完成了。

本质上, PVM 是一个特定可执行的图形化开发框架。基于 PVM 规范的流程定义可以实例化为“执行流”, 它拥有可以表现为图形的数据结构。

从另一个角度来说, 您可以把 PVM 看做类似 JVM (Java Virtual Machine, Java 虚拟机) 和 JPA (Java Persistence API, Java 持久化应用程序接口) 之类的引擎或标准, 它们具有如下相通之处:

- 如同 JVM, 只要您完全实现了 PVM 的接口规范, PVM 的流程虚拟机就可以通过这些接口执行您“设计”的任何业务流程定义, 就像 JVM 可以通过其特定的实现在任何平台上执行无差别的 Java 代码一样。
- 如同 JPA, PVM 制定了一套 Java 接口 (Java Interface) 标准, 供您根据需要实现。jPDL 就是这套接口标准的官方实现, 就像 Hibernate3 对于 JPA 标准的实现一样。

12.1 PVM 的架构

具体来说, PVM 将流程定义从具体的活动行为中剥离了出来。PVM 仅负责从流程的一个活动到下一个活动中提取流程的执行上下文, 而将活动的具体行为委派给可定制、可插拔的 Java 类, 并提供一组活动行为 API 用来作为 PVM 和活动行为实现代码的接口。因此从本质上说, jPDL 流程定义语言及其相关解释代码仅仅是一系列流程活动行为的实现和解析器。

下面我们从 PVM 的角度来看一条业务流程的全生命周期过程。首先业务流程被定

义出来，如图 12-1 所示是这个流程定义的图形化表达。

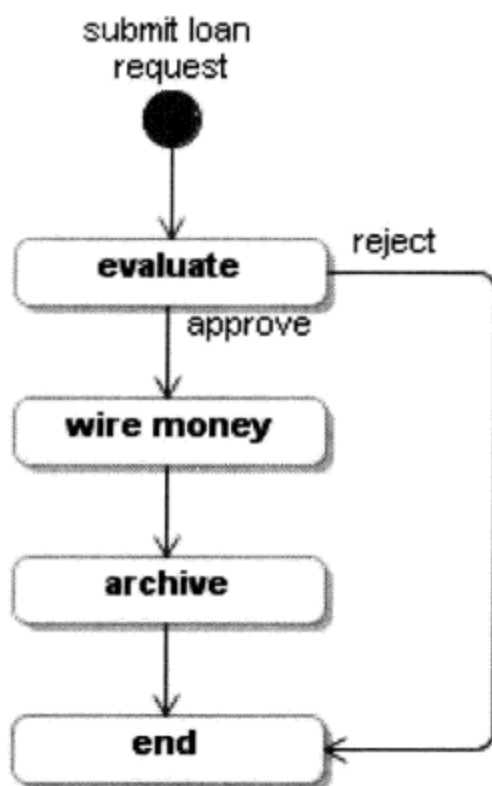


图 12-1 流程定义的图形化表达

流程定义可以看做静态的业务过程模板。流程定义由活动（Activity）和转移（Transition）组成。活动的行为被封装在该类型活动定义的实现中，各种类型的活动定义组成了整个流程结构。组成 PVM 流程定义的几个要素的结构类图如图 12-2 所示。

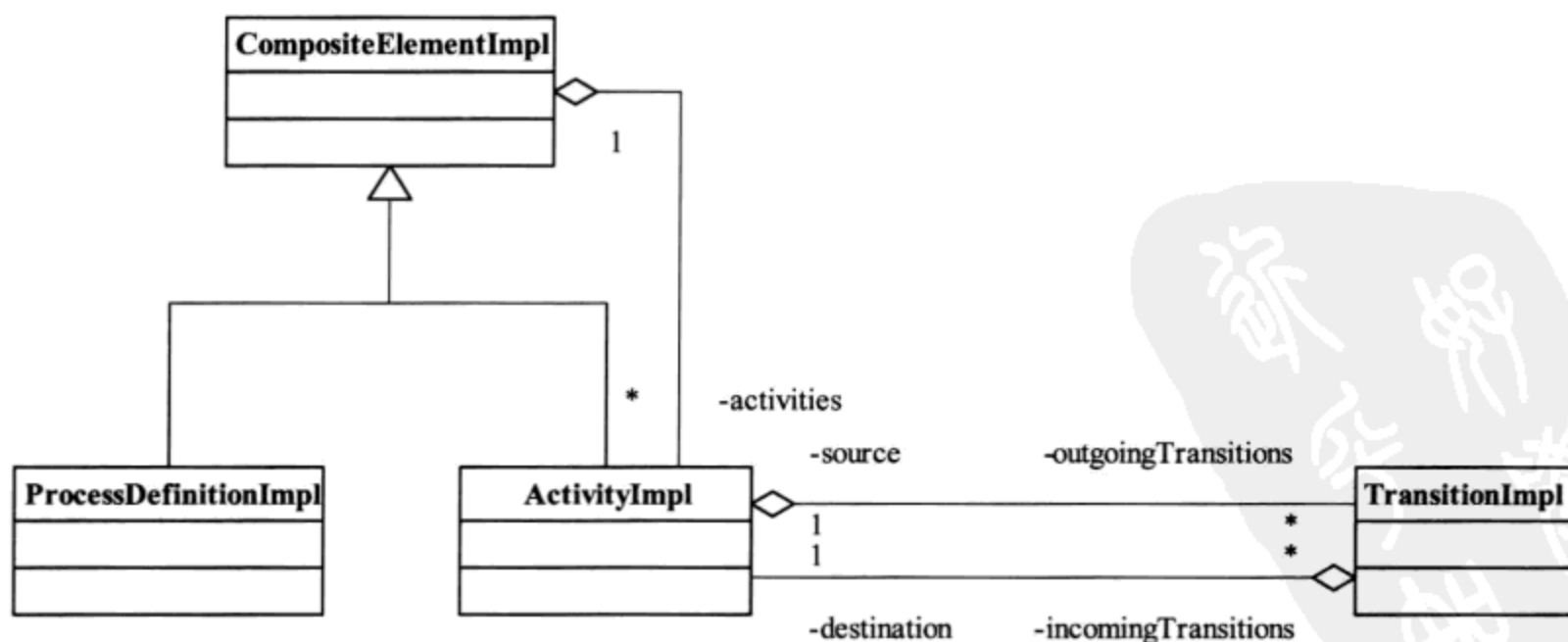


图 12-2 PVM 流程定义的结构类图

我们看到：流程定义（org.jbpm.pvm.internal.model.ProcessDefinitionImpl）和活动定义（org.jbpm.pvm.internal.model.ActivityImpl）都需要继承抽象类型 CompositeElementImpl。而活动定义内聚于 CompositeElementImpl，则使得流程定义拥有了包含多个活动定义的能力，同时也赋予了活动可以包含活动的特性（例如，group 活动就可以包含多个不同类型的活动）。而活动定义与转移定义（org.jbpm.pvm.internal.model.TransitionImpl）的互聚关系则表明了：

- 活动可以具有多个流入转移（incomingTransitions）。
- 活动可以具有多个流出转移（outgoingTransitions）。
- 转移只能从一个源活动（source）到一个目的地活动（destination）。

这个类图很好地证明了：流程定义由活动（Activity）和转移（Transition）组成 这句话。您可以查看 jBPM4 PVM 的 Java 源代码获取更多的证据。

PVM 没有包含任何具体的活动类型的实现，它只提供了上述的执行环境和一些 API，这些 API 用来被具体活动类型实现。活动可以是等待状态，这意味着把活动的控制权交给流程控制系统的外部，例如人工任务活动或异步执行。当流程实例陷入等待状态，则表示流程实例的上下文需要被持久化到数据库中了，相关的事务也要被提交了。

我们之前提到过，在 jBPM4 中，流程的“根执行（Root Execution）”已经和流程实例合二为一了，但是一个流程实例仍然可以生成多个“子执行”。每个执行都可以视为一个指向当前活动的指针。图 12-3 是正在运行中的流程实例示意图。

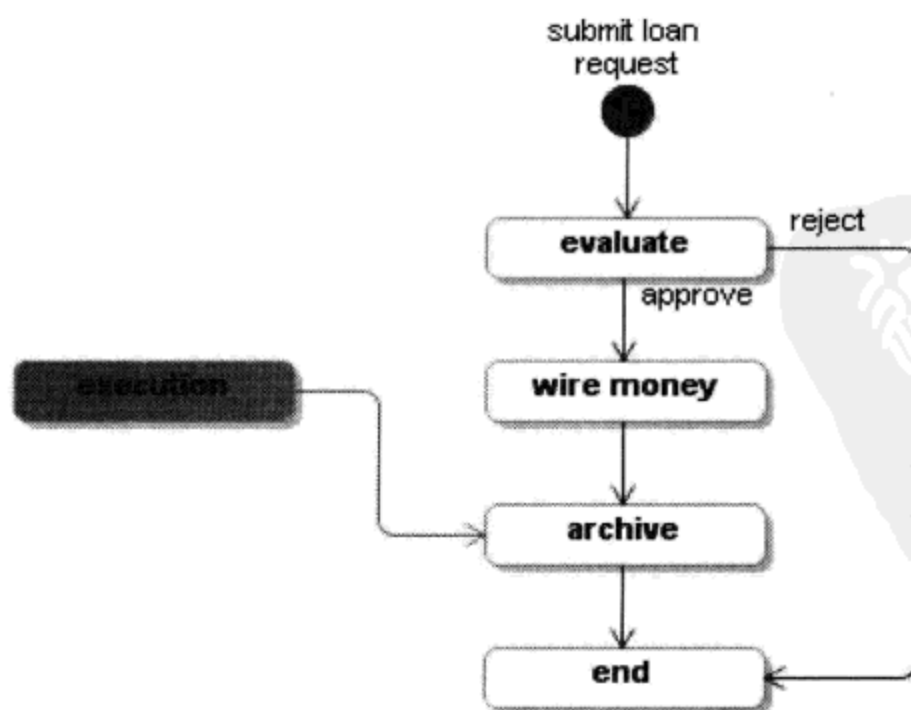


图 12-3 正在运行的流程实例

如图 12-3 所示，红色的 **execution**（执行）总是会忠实地指向流程实例当前的活动。

图 12-4 的结构类图展示了 **jBPM4** 中执行的父子关系，以及和活动的关联关系。

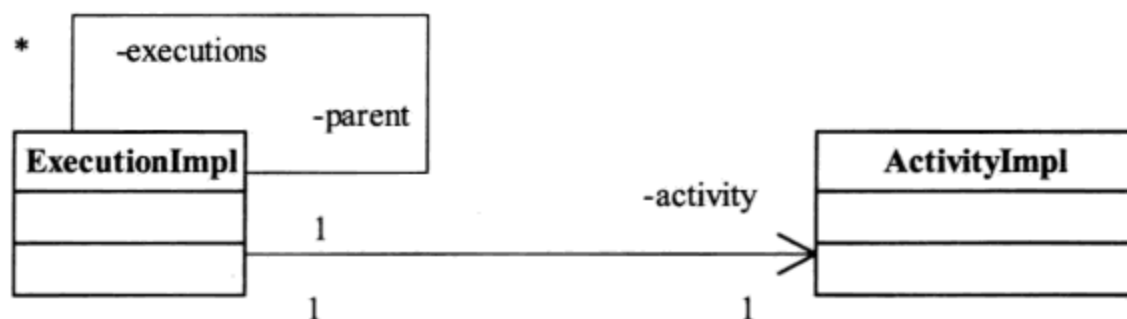


图 12-4 执行关系的结构类图

图 12-4 中 **ExecutionImpl** 的自我依赖关系证明了执行可以具有多个子执行，也证明了在必要的情况下流程实例（即根执行）可以分解为若干个子执行（**executions**）并行运行，但是每个子执行必然具有对其父执行（**parent**）的引用。每个执行（`org.jbpm.pvm.internal.model.ExecutionImpl`）都和一个（当前）活动（`org.jbpm.pvm.internal.model.ActivityImpl`）关联，这表示执行会指向当前活动，即执行可以看做指向当前活动的“指针”。

12.2 PVM 的实现

现在，让我们来看看 **PVM** 的构想具体如何实现。

目前，**jBoss** 系列产品已经在 **PVM** 的基础上完成 **jPDL**，**BPEL** 和 **Seam PageFlow** 三个实现，本书就是基于 **jPDL** 实现——这个 **jBPM4** 默认的流程定义标准来介绍的。**jPDL** 对 **PVM** 实现的源代码按包划分的功能分别如下：

- `org.jbpm.pvm.internal.ant`
 - 提供使用 **ant** 发布流程、启动应用服务器等任务的实现。
- `org.jbpm.pvm.internal.builder`
 - 用来构造各种 workflow 模型，包括活动、活动行为、事件、事件处理器、流程定义、变量……
- `org.jbpm.pvm.internal.cal`
 - 提供工作日历相关的实现和工具。

- org.jbpm.pvm.internal.cfg
 - JbpmConfiguration 和 SpringConfiguration 实现了 org.jbpm.api.Configuration 接口，分别用来从默认配置文件和 Spring 配置文件构造 ProcessEngine 对象。
- org.jbpm.api.client
 - 提供给 jBPM4 客户端应用程序使用的 API。
- org.jbpm.pvm.internal.cmd
 - Command 和 CommandService 接口及其所有的实现，是 jBPM4 命令设计模式的基础，所有的命令都在这里。
- org.jbpm.pvm.internal.email
 - 提供对电子邮件服务的支持。
- org.jbpm.pvm.internal.env
 - jBPM4 的系统环境实现，依赖于 IOC 框架，包括通用事务、依赖绑定、权限认证以及与环境上下文相关的类。
- org.jbpm.pvm.internal.hibernate
 - jBPM4 对 Hibernate 的封装，所有与数据库的操作都源于此。
- org.jbpm.pvm.internal.history
 - 流程历史服务的相关 API。
- org.jbpm.pvm.internal.identity
 - 提供身份认证服务相关的 API。
- org.jbpm.pvm.internal.jms
 - 封装对 JMS 消息服务的实现，也提供一些 JMS 相关的工具。
- org.jbpm.pvm.internal.job
 - 提供了对 Job, Message 和 Timer 的实现。
- org.jbpm.pvm.internal.jobexecutor
 - 提供了 Job 的执行器，这包括 JobExecutorServlet 和相应的线程池、命令和处理器。Job, Message 和 Timer 最终都会被这里的类执行。
- org.jbpm.pvm.internal.lob
 - 处理与流程相关的富文本、字节码或二进制资源，例如 jPDL 流程定义的 XML 文件、图片、Java Class、序列化的 Java 对象等。
- org.jbpm.pvm.internal.model
 - 包含了 jPDL 流程定义语言 XML 节点模型的所有 Java 实现。这包括 Activity, CompositeElement, Condition, ObservableElement, OpenProcessDefinitionTransition

等基础接口，及这些接口的各种实现类。

- `org.jbpm.pvm.internal.query`
 - 包含 `History`, `Job`, `ProcessDefintion`, `ProcessInstance` 这 4 种流程要素的查询实现类。还提供了 `Page` 相关的类，用来实现查询结果分页。
- `org.jbpm.pvm.internal.repositor`
 - 负责流程定义的发布及获取，提供缓存功能。
- `org.jbpm.pvm.internal.script`
 - 提供多种脚本解释引擎，这包括 `BeanShell`, `Groovy`, `Xpath`, 以及默认支持的 `jUEL`。
- `org.jbpm.pvm.internal.session`
 - 提供对各种会话的支持，这包括 `DbSession`, `MessageSession`, `RepositorySession` 和 `TimerSession`。
- `org.jbpm.pvm.internal.spring`
 - 目前这里只有一个类 `CommandTransactionCallback`，实现了从 `Spring` 框架的配置中获取事务，并在事务中执行 `jBPM4` 的各种命令。
- `org.jbpm.pvm.internal.stream`
 - 提供从各种途径获取“资源”的工具。这些资源包括 `jBPM4` 配置文件和流程定义引用的图片、类、表单等，都要通过此包中的工具转换成流的形式提供给 workflow 引擎使用。
- `org.jbpm.pvm.internal.svc`
 - 包名 `svc` 是 `service` 的缩写。提供了 `CommandService` 接口和 `org.jbpm.api` 包下所有的服务接口的实现，以及一些命令拦截器（`Interceptor`）。
- `org.jbpm.pvm.internal.task`
 - 提供所有与任务相关的东西。这包括任务的定义和实例，任务的参与者、分配者和任务泳道，任务分配处理器，历史任务……
- `org.jbpm.pvm.internal.test`
 - 只有一个 `JobTestHelper`。显然是用来帮助在测试环境下运行 `Job` 的。
- `org.jbpm.pvm.internal.tx`
 - 提供各种事务处理机制。包括对 `JTA` 事务、`Spring` 事务以及标准事务的支持。
- `org.jbpm.pvm.internal.type`
 - 提供 `jBPM4` 变量类型定义以及各种数据类型的转换工具。子包有 `converter`（类型转换器）、`matcher`（类型匹配器）和 `variable`（变量类型

定义)。

- `org.jbpm.pvm.internal.util`
 - 提供 jBPM4 常用工具以及一些常用接口。这包括类加载工具、反射工具、字符串工具、XML 工具等，以及可关闭接口 (Closable)、监听接口 (Listener)、观察接口 (Observable) 等。
- `org.jbpm.pvm.internal.wire`
 - jBPM4 的 IOC 框架。jBPM4 的依赖注入与控制反转机制的实现。
- `org.jbpm.pvm.internal.xml`
 - jBPM4 的 XML 解析模块。用来解析 jBPM4 配置文件和 jPDL4 流程定义文件的 XML 内容。

12.3 小结

本章首先向读者介绍了 PVM (流程虚拟机) 的理念，并说明了为什么要设计出 PVM。然后通过一条典型业务流程，引出了 PVM 的关键实现类及其结构类图，使读者对于 PVM 的基础架构原理有了深入了解，这对于读者进行基于 PVM 的扩展开发工作能起到很大的帮助。

最后本章列出了 jPDL4 对于 PVM 规范实现的 Java 包结构说明，这对读者查看 PVM 源代码、学习 PVM 架构以及扩展 PVM 功能起到了相应的索引和指导作用。



本章将从引擎服务 API 到数据持久化层自上而下地深入介绍 jBPM4 的架构设计思想。研究 jBPM4 架构设计的意义在于：

- jBPM 经过多年发展，到目前的第 4 个大版本，其架构设计越来越凝练和具有伸缩性，非常值得有志于向架构师发展的开发者学习。另外，具有“引擎架构”的思想对于开发者解决大型应用系统的设计问题十分重要，作者认为各种所谓“引擎”的程序架构都有相通的地方。
- 只有了解 jBPM4 的设计，才能明白应用 jBPM 系列产品的核心思想，即不要试图修改 jBPM，尝试扩展 jBPM，因为 jBPM 的架构是开放的——只有真正明白了这一点，才能把 jBPM4 运用得心应手，实现各种您所能想到、遇到的“奇怪”需求。

13.1 API 设计

jBPM4 workflow 引擎的核心 PVM，主要依靠 4 组服务 API 支持各种不同执行模式下的流程工作。这 4 组“核心”服务 API 包括：

- 流程定义服务——Process Service。
- 流程执行服务——Execution Service。
- 流程管理服务——Managerment Service。
- 指令服务——Command Service。

以一个运行时的流程实例为例，这 4 组服务 API 的作用关系如图 13-1 所示。

应用程序通过这 4 组服务 API 与 PVM 进行数据交互，**这些数据交互都在支持事务的持久化模式下运行**。这 4 组服务 API 中很多需要被 jBPM4 客户端应用开发者经常使用，例如：

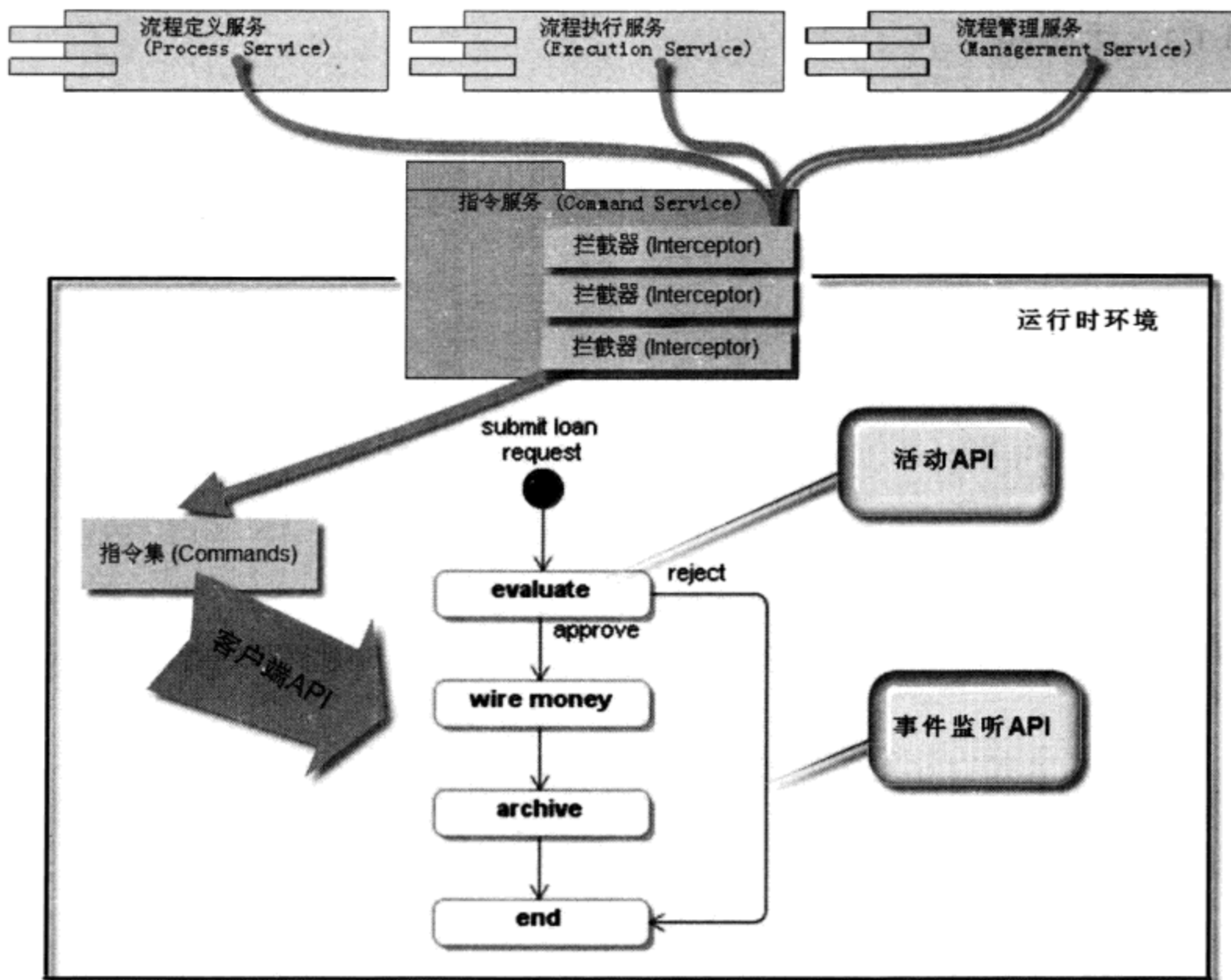


图 13-1 PVM 中 4 组核心服务 API 在运行时的作用关系

- `ExecutionService.startProcessInstanceByKey` ——发起流程实例。
- `TaskService.completeTask` ——完成任务。

开发者还可以直接使用客户端 API 来执行一些流程操作。客户端 API 是核心工作流模型对象对外暴露的公共方法，客户端 API 不会进行任何持久化操作。例如：

- `ProcessInstance.getName` ——获取流程实例名称。
- `Task.setAssignee` ——设置任务分配者。

实际上，客户端 API 是直接操作工作流模型上的执行对象的，操作结果需要通过调用相应的服务 API 保存才能持久化在数据库中。

以下将介绍的活动 API 和事件监听 API，属于流程定义的范畴。

13.1.1 活动 API

活动 API 被用来实现提供流程活动在运行时的行为，因此每一个类型的活动实际上是一种流程组件。所有的活动类型都需要实现 `ActivityBehaviour` 接口，这个接口提供了控制流程执行的方法。

`ActivityBehaviour` 接口定义如下：

```
public interface ActivityBehaviour extends Serializable {  
    //实现：当执行对象到达活动时的行为，参数即为执行对象  
    void execute(ActivityExecution execution) throws Exception;  
}
```

通过提供给活动**执行对象**使这个活动具有了“有所作为”的能力。执行对象的类型必须要实现 `ActivityExecution` 接口，这个接口定义了活动所能执行的控制流程推进的所有方法：

```
public interface ActivityExecution extends OpenExecution {  
    //获取活动名称  
    String getActivityName();  
    //等待执行信号  
    void waitForSignal();  
    //以下是选定转移路径的相关方法  
    void takeDefaultTransition();  
    void take(String transitionName);  
    //执行子活动  
    void execute(String activityName);  
    //以下是完成活动执行的相关方法  
    void end();  
    void end(String state);  
    //设置活动优先级  
    void setPriority(int priority);  
}
```

13.1.2 事件监听 API

前面的章节多次介绍到事件监听器，事件监听 API 主要被用来实现用户定制的事件监听器。事件监听器可以用来处理被监听到的流程事件，它与活动 API 唯一的差别

是不能控制流程的执行。来解释一下为什么会有这个差别：例如当一个活动通过执行（execution）决定了一个转移，那么对应的事件监听器会被触发，但是因为这个转移已经注定发生了，因此流程推进路径必然无法被事件监听器所改变。事件监听器需要实现 `EventListener` 接口：

```
public interface EventListener extends Serializable {  
    void notify(EventListenerExecution execution) throws Exception;  
}
```

事件监听器 `notify` 方法的参数是 `EventListenerExecution` 接口，它与活动 API 中提供的参数 `ActivityExecution` 接口相同之处在于它们继承的都是 `OpenExecution` 接口，但不同的是 `EventListenerExecution` 中的方法就只剩一个了：

```
public interface EventListenerExecution extends OpenExecution {  
    //设定优先级  
    void setPriority(int priority);  
}
```

如您所见，正像本节开始所说的：流程推进路径必然无法被事件监听器所改变。原本 `ActivityExecution` 接口中控制流程推进的相关方法在 `EventListenerExecution` 接口中都没有了。

13.2 执行环境设计

为了使流程在不同的事务环境下执行，如 Java EE，Spring 等，PVM 提供了一个执行环境对象，根据配置的环境，执行服务延迟加载、获取事务管理服务等操作。

执行环境是个 `EnvironmentFactory`（环境工厂）对象，目前 jBPM4 有两个环境工厂的实现：

- `ProcessEngineImpl` ——默认的 Java EE 环境支持。
- `SpringProcessEngine` ——基于 Spring 框架的环境支持。

获取默认环境工厂对象的代码如下：

```
ConfigurationImpl cfg = new ConfigurationImpl();  
//指定当前环境的配置文件  
cfg.setResource("jbpm.cfg.xml");
```

```
//创建环境工厂对象
EnvironmentFactory environmentFactory = new ProcessEngineImpl(cfg);
```

有了 EnvironmentFactory 对象，则可执行任意流程操作：

```
Environment environment = environmentFactory.openEnvironment();
try {
    //在这里可以获取 PVM 资源，执行任意操作
    //例如 environment.get(Session.class); .....
} finally {
    environmentFactory.close();
}
```

提示：通过 Environment 对象获取的流程服务受到事务控制。

也可以通过 Configuration 类加载默认的配置文件的，方便地获取各项流程服务，而不需要直接操作 Environment 对象：

```
ProcessEngine processEngine = Configuration.getProcessEngine();
RepositoryService repositoryService = processEngine.getRepositoryService();
...
```

13.3 命令设计

jBPM4 采用命令 (Command) 设计模式作为其实现流程逻辑的核心思想。所有“命令”都需要继承 Command 接口，在这个接口提供的“环境块”（即 execute 方法）中实现逻辑：

```
public interface Command<T> extends Serializable {
    T execute(Environment environment) throws Exception;
}
```

提示：每个“命令”都是一个独立的事务操作，即每一个 execute 方法的实现都被一个 Hibernate 事务所包含。

jBPM4 鼓励用户定制自己的“用户命令”，去实现特定的业务需求。如下例：

```

public class MyUserCommand implements Command<Void> {
    ...
    public Void execute(Environment environment) throws Exception {
        //从环境对象中获取执行服务
        ExecutionService executionService = environment.get
        (ExecutionService.class);
        //利用执行服务完成逻辑。这里的 executionId 可以由类构造方法设置的成员域
        executionService.signalExecutionById(executionId);
        ...
        return null;
    }
}

```

可使用流程引擎对象来执行命令：

```

ProcessEngine processEngine = Configuration.getProcessEngine();
processEngine.execute(new MyUserCommand());

```

13.4 服务设计

jBPM4 的服务 API 提供给外部（如客户端应用程序）作为操作 workflow 引擎的通道，也被用来对外暴露 PVM 持久化操作的调用。

目前 jBPM4 对外提供很多服务，随着版本的升级，相信服务的数量仍然会增加。以下是 jBPM4 中 3 个基本服务的接口定义。

● RepositoryService

```

//提供流程定义及其相关资源的服务
public interface RepositoryService {
    Deployment createDeployment();
    ProcessDefinitionQuery createProcessDefinitionQuery();
    ...
}

```

● ExecutionService

```

//提供流程实例及其执行相关的服务
public interface ExecutionService {
    ProcessInstance startProcessInstanceById(String processDefinitionId);
}

```

```

        ProcessInstance signalExecutionById(String executionId);
        ...
    }

```

● ManagementService

//目前主要提供 Job 相关的服务

```

public interface ManagementService {
    JobQuery createJobQuery();
    void executeJob(long jobDbid);
    ...
}

```

因为 jBPM4 中所有的流程逻辑都被封装成为命令 (Command)，因此这 3 个服务的方法实现其实都是在执行命令。所有 PVM 命令都统一委派给 CommandService 去调用（这样做的目的后面会有解释），CommandService 接口定义如下：

```

public interface CommandService {
    //用来标识一般情况下的 CommandService 实现：在同一个线程中使用一个事务执行所有命令
    String NAME_TX_REQUIRED_COMMAND_SERVICE = "txRequiredCommandService";
    //用来标识 CommandService 的另一种实现：为每个命令的执行新开启一个事务
    String NAME_NEW_TX_REQUIRED_COMMAND_SERVICE = "newTxRequiredCommandService";
    //执行命令。返回的参数由模板定义
    <T> T execute(Command<T> command);
}

```

jBPM4 默认的配置文件中 jbpm.default.cfg.xml 中设置了如下服务：

```

<jbpm-configuration>
    ...
    <process-engine-context>
        <repository-service />
        <repository-cache />
        <execution-service />
        <history-service />
        <management-service />
        <identity-service />
        <task-service />
    ...

```

CommandService 的实现也需要在配置文件中指定。CommandService 的整体实现可以看做环绕在命令周围的一群拦截器组成的一条“职责链”。这也是职责链设计模式

的运用。我们可以组合不同的拦截器，按照不同的顺序，在不同的环境下实现不同的持久化事务策略。

jBPM4 在配置文件 `jbpm.tx.hibernate.cfg.xml` 中描述其 `CommandService` 的各种实现策略：

```
<jbpm-configuration>
  <process-engine-context>
    <!-- 一般情况下的 CommandService 实现 -->
    <command-service name="txRequiredCommandService">
      <skip-interceptor />
      <retry-interceptor />
      <environment-interceptor />
      <standard-transaction-interceptor />
    </command-service>
    <!-- 对每个命令新启事务的 CommandService 实现 -->
    <command-service name="newTxRequiredCommandService">
      <retry-interceptor />
      <environment-interceptor policy="requiresNew" />
      <standard-transaction-interceptor />
    </command-service>
    ...
  </process-engine-context>
</jbpm-configuration>
```

各个服务（例如默认配置的 `repository-service`, `execution-service`, `management-service` 等）将会按照需要选择合适的 `command-service` 来执行命令。事实上，此时读者可能意识到了 `CommandService` 本身就是拦截器。没错，各种拦截器（例如 `retry-interceptor`）实现的正是 `CommandService` 接口，若干个 `CommandService` 被配置成为职责链用来拦截命令的调用，这样各个服务就可以选择不同的策略（由不同的职责链决定）去执行命令，而无须变更命令本身。

以 “`newTxRequiredCommandService`” 的实现为例。根据配置，`newTxRequiredCommandService` 在调用一条命令时会依次执行。

- 1) `retry-interceptor` —— 会在数据库乐观锁失败时，捕获 `Hibernate` 的 `StaleObjectExceptions`，并重新尝试调用命令。
- 2) `environment-interceptor` —— 会为命令的调用提供一个环境对象（`EnvironmentImpl`）。
- 3) `standard-transaction-interceptor` —— 会初始化一个标准事务对象（`StandardTransaction`），`Hibernate` 的会话/事务会被这个标准事务作为一个资

源使用。

- 4) 最后，由默认的 `DefaultCommandService` 来调用命令。当然，jBPM4 也支持其他方式调用命令，通过配置可以使用：
 - a) `EjbLocalCommandService` ——把命令委派给一个本地的 EJB 调用，这样可以启动一个 EJB 内容管理事务。
 - b) `EjbRemoteCommandService` ——把命令委派给一个远程的 EJB 调用，这样命令将在另一个 JVM 上被执行。
 - c) `AsyncCommandService` ——把命令包装成一个异步消息，这样命令会在一个新的事务中被异步执行。

13.5 历史流程处理原理

正如前面章节提到过的，jBPM3 版本的数据库表一直是饱受使用者诟病的设计。因为 jBPM3 没有“流程历史库”的概念，所有处于完成状态的流程实例数据（这包括关联的活动实例、任务实例、流程变量……）与正在运行的流程实例数据混合在一起，而没有归入流程历史库，因此“运行时流程实例”的数据库表就会随着时间的推移无限地膨胀，这会严重影响当前运行流程的执行效率。

在 jBPM4 版本中，流程历史库的概念被实现了。jBPM4 在整个流程实例执行过程的各个关键阶段，都加入了将流程实例数据存入历史库的历史事件（`HistoryEvent`）的触发器，实现了将已完成的流程实例及其数据归入专门的历史流程数据表的机制。从而真正实现了将运行中的流程数据和历史流程数据的物理分离，彻底解决了 jBPM 系统长时间使用后效率严重下降的问题。

具体实现原理如下：

历史事件在流程实例运行的过程中被触发，然后根据分类被分发到配置好的 `HistorySession` 中（见 `HistoryEvent` 类的静态方法 `fire`）——所有的历史事件都会委派给一个 `HistorySession` 接口处理。`HistorySession` 默认实现的 `HistorySessionImpl` 将调用相应历史事件对象（`HistoryEvent` 对象）的 `process` 方法执行历史事件处理逻辑。

`HistoryEvent` 是临时事件，即此类型的事件本身不会被持久化。在 `HistoryEvent` 对象的 `process` 方法中，历史实体被构建，例如 `HistoryProcessInstance`，`HistoryActivityInstance` 等。

同时在 `process` 方法中，历史事件创建的历史实体与当前的流程实体是对应、归并

的关系。例如 `ProcessInstanceCreate` 事件会创建 `HistoryProcessInstance` 对象和持久化数据记录, `ProcessInstanceEnd` 事件会设置对应的 `HistoryProcessInstance` 对象和持久化数据记录的状态属性值为“结束”。

再来说说流程历史库。

流程历史库是维护用于查询的过往流程信息归档的数据库表。当流程实例或活动实例结束的时候, 就会向流程历史库中写入数据, 这些数据对于运行中的流程来说, 都是过时的。

流程历史库使用 5 张数据表维护着 4 种实体的历史信息:

- (历史) 流程实例。
- (历史) 活动实例。
- (历史) 任务。
- (历史) 流程变量。

可以使用 `HistoryService` 的 `createHistoryXxxxQuery` 方法获取相应实体的查询对象, 查询历史流程实体。

另外, 每种流程实体的历史运行细节都被专门的“历史细节”表保存。`HistoryService` 根据历史细节表提供诸如 `avgDurationPerActivity (processDefinitionId)`——获取活动平均执行时间、`choiceDistribution (processDefinitionId, activityName)`——获取活动转移选择次数等方法供流程数据分析者调用。当然, 开发者也可以根据历史细节表扩展出自己的流程数据分析 API。

13.6 数据持久化设计

目前 `jBPM4` 的数据持久化实现是基于 `Hibernate` 框架的。

在未来 `jBPM4` 可能会按照 `JPA` 规范去实现持久化层, 这样可能更标准一些 (`Hibernate` 也对 `JPA` 规范有自己的一套实现); 也有可能 `jBPM4` 会额外提供一套 `JDBC` 的实现或接口, 这样用户就可以为大规模数据库集群 (例如在互联网的海量访问场景下) 来定制 `jBPM4` 的持久化层实现了。

因此我们应该尽量使用 `jBPM4` 的持久化 API 去操作数据库, 这些 API 可以帮助您“无视”上述的变革。

下面以 jBPM4.3 版本为例，按关联类别分别列出 jBPM4 数据库表结构的 ER 图，并说明每张数据表的作用。

13.6.1 jBPM4 流程定义资源和实例运行时数据表

jBPM4 流程定义资源和实例运行时数据表结构如图 13-2 所示，这些数据库表用来保存所有和流程定义相关的资源，以及运行中的流程实例数据。

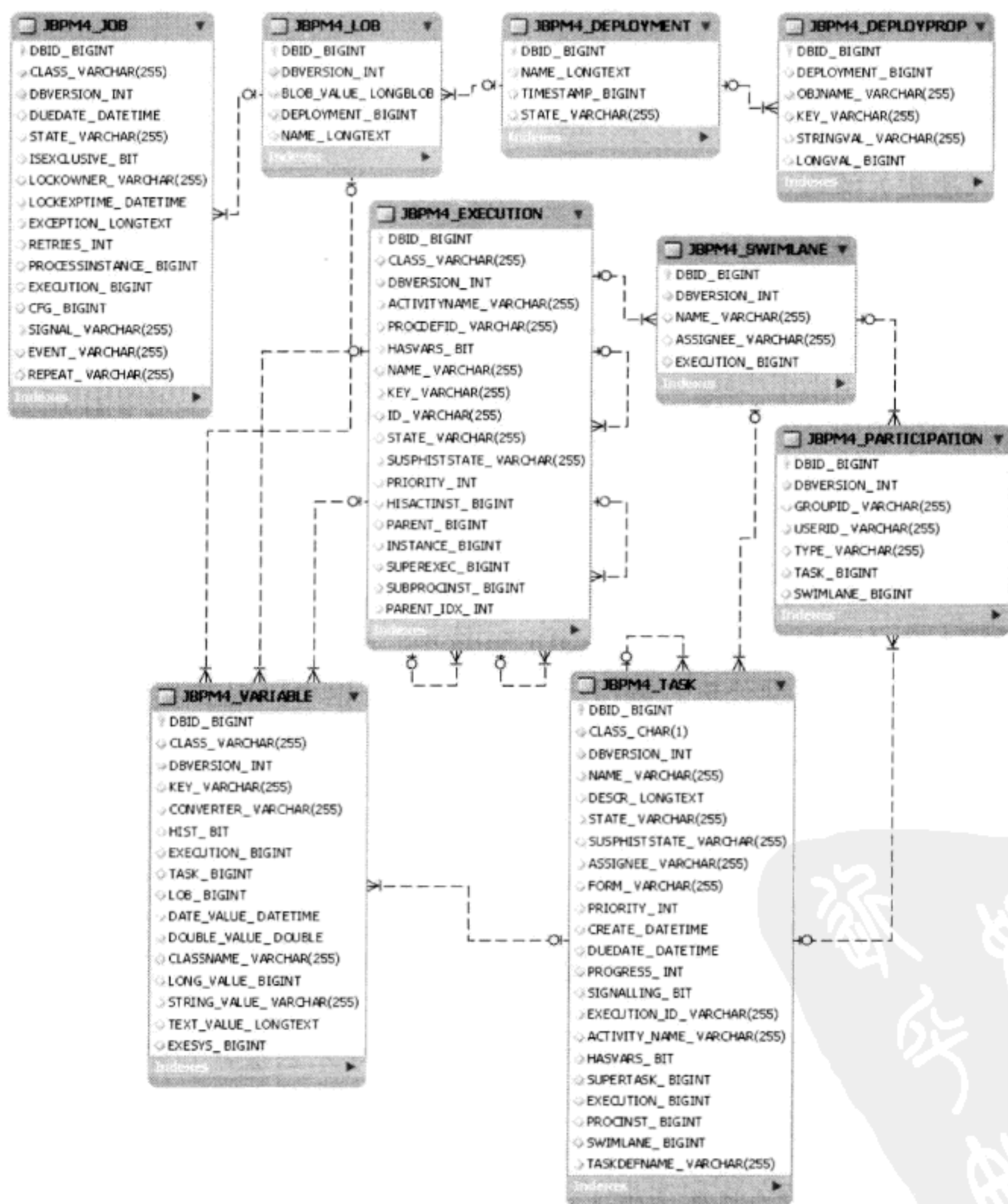


图 13-2 jBPM4 流程定义资源和实例运行时数据表结构 ER 图

- JBPM4_DEPLOYMENT
 - 流程定义的部署记录。
- JBPM4_DEPLOYPROP
 - 已部署的流程定义的具体属性。
- JBPM4_LOB
 - 流程定义的相关资源，包括 jPDL XML、图片、用户代码 Java 类等，以二进制格式统一存储在此表中。
- JBPM4_JOB
 - 异步活动或定时执行的 Job 记录。
- JBPM4_VARIABLE
 - 流程实例的变量。
- JBPM4_EXECUTION
 - 流程实例及执行对象。
- JBPM4_SWIMLANE
 - 任务泳道。这属于流程定义的数据。
- JBPM4_PARTICIPATION
 - 任务参与者（任务的相关用户，区别于任务的分配人）。这属于流程实例的数据。
- JBPM4_TASK
 - 流程实例的任务记录。

13.6.2 jBPM4 流程历史数据表

jBPM4 将运行中的流程实例数据与已完成的流程实例数据严格分离，从而形成了一系列流程历史数据表，如图 13-3 所示。

- JBPM4_HIST_PROCINST
 - 保存历史的流程实例记录。
- JBPM4_HIST_ACTINST
 - 保存历史的活动实例记录。
- JBPM4_HIST_TASK
 - 保存历史的任务实例记录。
- JBPM4_HIST_VAR
 - 保存历史的流程变量数据。

● JBPM4_HIST_DETAIL

- 保存流程实例、活动实例、任务实例运行过程中历史明细数据，例如起止时间、平均处理时间、任务注释等，为效率分析等流程数据挖掘服务提供基础数据支持。

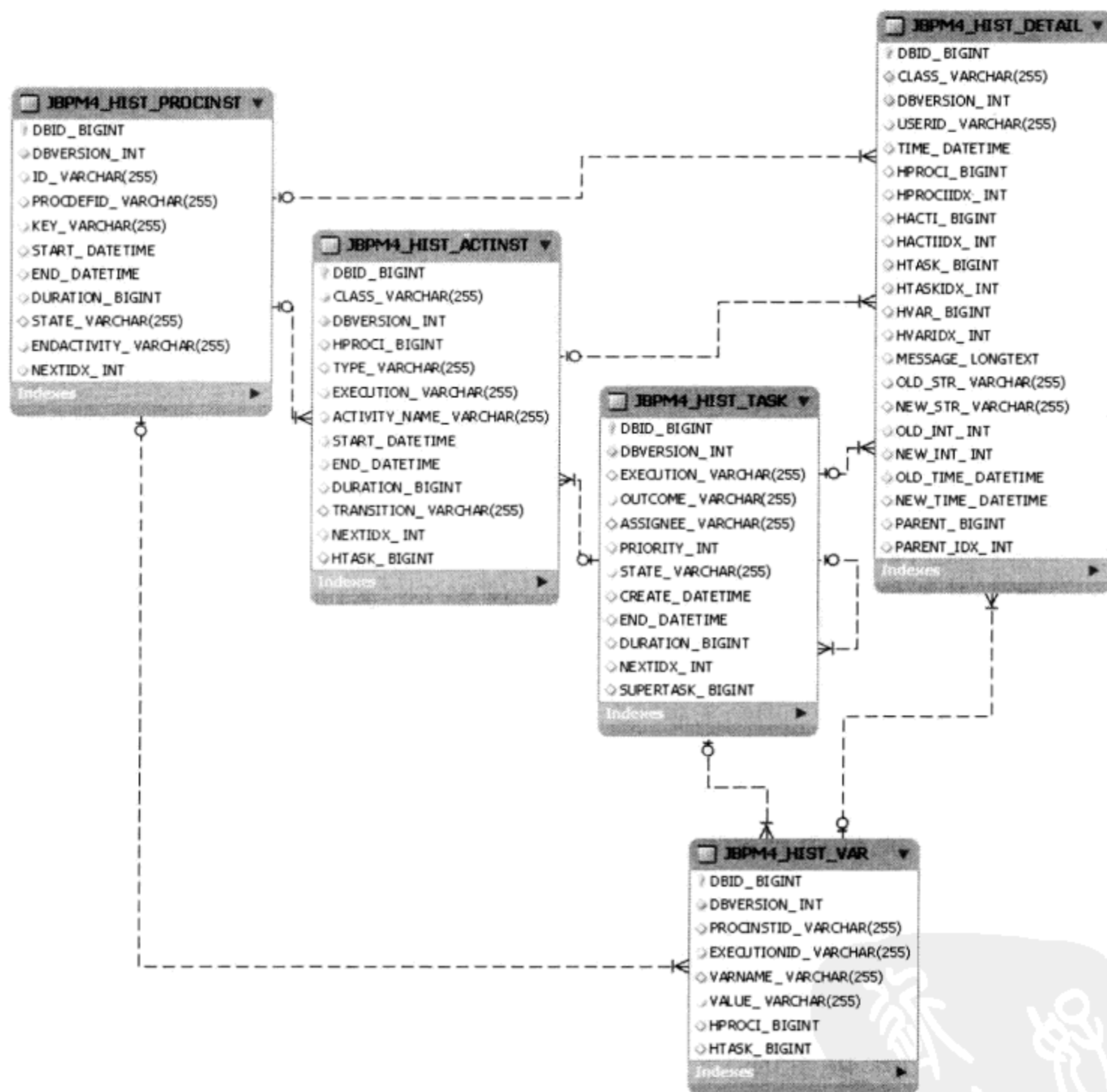


图 13-3 jBPM4 流程历史数据表结构 ER 图

13.6.3 jBPM4 身份认证数据表

如图 13-4 所示的 3 表提供一套非常简单的组织结构数据，供 jBPM4 的任务分派、

权限认证所使用。

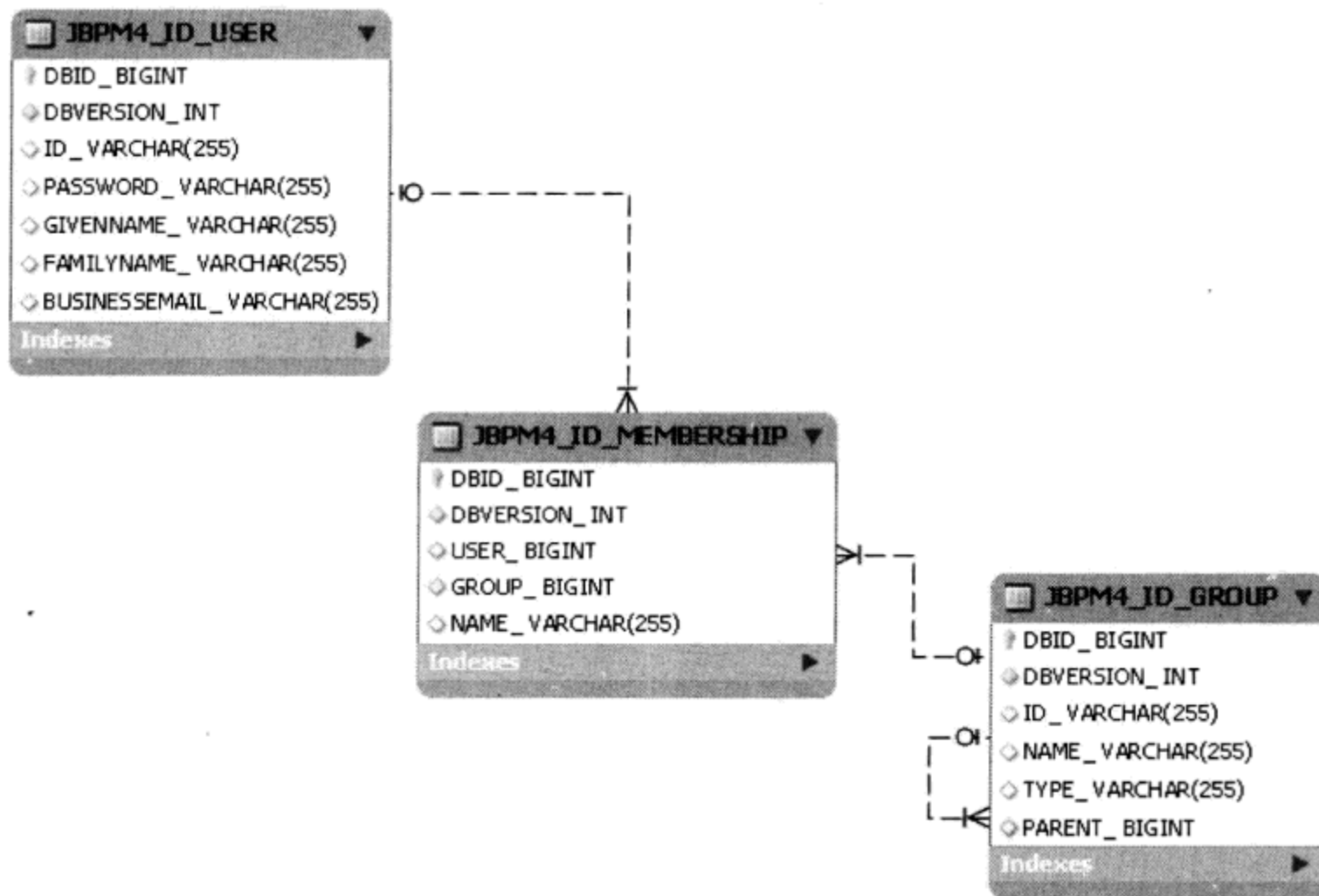


图 13-4 JBPM4 身份认证数据表结构 ER 图

- **JBPM4_ID_USER**
 - 保存用户记录。
- **JBPM4_ID_MEMBERSHIP**
 - 保存用户和用户组之间的关联关系。
- **JBPM4_ID_GROUP**
 - 保存用户组记录。

13.6.4 JBPM4 引擎属性数据表

JBPM4 引擎属性数据表结构如图 13-5 所示，JBPM4_PROPERTY 表会保存一些初始化设定的种子数据，例如当前的 JBPM 引擎版本（key = db.version）、ID 生成器版本（key = next.dbid）等。

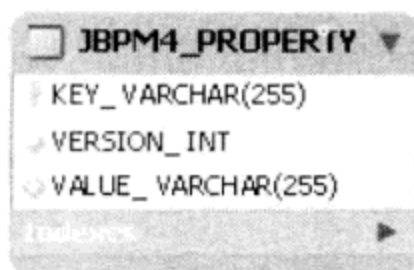


图 13-5 jBPM4 引擎属性数据表结构 ER 图

参见 2.8 安装 jBPM 数据库，可以了解这个数据表存在的意义。

13.7 例程：扩展 jBPM4 的 API 满足客户化的需求

jBPM 本身提供的 API 不能满足世界上所有的业务流程需求——这是一个再简单不过的道理。

所以，当“变态”的客户化需求被提出来时，您可以尝试去扩展 jBPM 的 API 而不是搞出一套复杂或混乱的流程定义模型。

jBPM 是非常易扩展的。想想那些用户代码吧：事件监听器、自定义活动、Java 代码活动。灵活地运用用户代码替代某些内置活动可以使事情在一些时候变得更简单。

但这里要说的是定制您的查询。假设您想用很多属性来过滤任务列表，例如按照优先级、日期、执行人、任务名称等条件来获取任务列表。

显然，利用目前 jBPM4 提供的任务服务（TaskService）的查询接口要实现上述需求比较麻烦。我们需要一种更简单的、有效率的、灵活的方式实现之，何不扩展 jBPM4 的命令服务，实现一个“客户化任务查询命令”呢？

请看 CustomTaskQueryCommand 的实现：

```
//实现 org.jbpm.api.cmd.Command 接口，成为定制的 jBPM4 命令
public class CustomTaskQueryCommand implements Command<List<Task>> {
    private int taskPriority;
    private String taskDescLike;
    private Date taskCreateFrom;
    private Date taskCreateTo;
    //利用构造方法获得查询过滤的参数
    //在这里我们通过任务优先级（taskPriority），任务描述（taskDescLike，模糊查询），
```


任务创建时间范围 (taskCreateFrom, taskCreateTo) 去过滤查询任务列表

```
public CustomTaskQueryCommand(int taskPriority, String taskDescLike,
    Date taskCreateFrom, Date taskCreateTo) {
    this.taskPriority = taskPriority;
    this.taskDescLike = taskDescLike;
    this.taskCreateFrom = taskCreateFrom;
    this.taskCreateTo = taskCreateTo;
}
@Override
public List<Task> execute(Environment environment) throws Exception {
    //注意这里,从 Environment 对象中可以获取包括 Hibernate 会话在内的所有工作流
    引擎运行时资源
    Session session = environment.get(Session.class);
    //使用 Hibernate3 的条件查询 API
    Criteria criteria = session.createCriteria(TaskImpl.class);
    //任务的字段属性在 Hibernate 配置文件 jbpm.task.hbm.xml 中可以看到
    //设置任务优先级参数到对应字段 priority
    if (taskPriority != 0)
        criteria.add(Restrictions.eq("priority", taskPriority));
    //设置任务描述参数到对应字段 description, 注意这里是模糊查询
    if (taskDescLike != null)
        criteria.add(Restrictions.like("description", "%" + taskDescLike
+ "%"));
    //设置任务的创建时间范围, 即设定 createTime 所在的区间范围
    if (taskCreateFrom != null && taskCreateTo != null)
        criteria.add(Restrictions.between("createTime",
taskCreateFrom, taskCreateTo));
    //执行查询
    List<?> list = criteria.list();
    //返回任务列表
    return (List<Task>) list;
}
}
```

编写如下单元测试代码验证上面的自定义命令。首先,在重载的 JbpmTestCase 的 setUp 方法中凭空创建出 3 个任务,分别给它们设置优先级和描述属性,创建时间即为当前系统时间。setUp 方法的实现如下:

```
super.setUp();
//创建名为 laundry 的任务
Task task = taskService.newTask();
task.setName("laundry");
```

```

task.setPriority(10);
task.setDescription("This is a laundry task.");
taskLaundryId = taskService.saveTask(task);
//创建名为 dishes 的任务
task = taskService.newTask();
task.setName("dishes");
task.setPriority(20);
task.setDescription("This is a dishes task.");
taskDishesId = taskService.saveTask(task);
//创建名为 iron 的任务
task = taskService.newTask();
task.setName("iron");
task.setPriority(30);
task.setDescription("This is an iron task.");
taskIronId = taskService.saveTask(task);

```

单元测试代码如下：

```

//任务的创建时间范围自 1999 年 12 月 22 日起
Calendar createFrom = Calendar.getInstance();
createFrom.set(1999, 11, 22);
//至当前时间为止
Calendar createTo = Calendar.getInstance();
//构造“自定义任务查询命令”对象，在构造方法中设置查询条件：任务优先级为 20，
任务描述中包含“dishes task”，时间范围……
Command<List<Task>> taskQueryCmd = new CustomTaskQueryCommand(20,
    "dishes task", createFrom.getTime(), createTo.getTime());
//通过 Configuration.getProcessEngine 方法拿到当前的流程引擎对象，执行“自定义任务查询”命令
List<Task> tasks = Configuration.getProcessEngine().execute(taskQueryCmd);
//以下都是在断言查询到的是预期的 dishes 任务
assertEquals(1, tasks.size());
assertEquals("dishes", tasks.get(0).getName());
assertEquals("This is a dishes task.", tasks.get(0).getDescription());

```

13.8 小结

本章和上一章系统地、集中地介绍了 jBPM4 的设计理念和思想，这包括：

- 流程虚拟机——PVM 的设计原理。

- Service API 的设计。
- jBPM4 执行环境（Runtime Environment）的设计。
- jBPM4 独特的基于“命令设计模式”的底层服务调用设计。
- 对客户端统一的服务层规划和设计。
- jBPM4 的数据库表设计。

读者可以通过这两章的介绍更好地扩展 jBPM4 的功能、定制自己的业务流程实现——因为“知其然，亦知其所以然”。同时，熟悉这些“大师级”的架构设计思想对于提升开发者自身的架构设计能力、系统建模能力有着极好的催化作用。无论是自己重新设计一个工作流引擎或是基于 PVM 再实现一套新的流程框架，乃至设计和架构其他领域的应用系统，jBPM4 框架的核心设计思想都可以起到极好的借鉴作用。



在第 2 章 安装和配置 jBPM4 中介绍了如何安装 jBPM4 以及配置其运行环境。其中涉及的配置技巧比较初级和简单，但十分稳定。下面将介绍一些比较高级的 jBPM4 配置修改技巧，这些技巧在一定程度上可以满足您特定的业务需求，但如果运用在生产系统中，您需要承担一定风险，请在配置完成时仔细测试后再使用。

在官方发布包 `jbpm.jar` 的根路径中包含了一些默认提供的配置文件。用户可以选择包含或排除某些功能，通过向 `jbpm.cfg.xml` 配置文件中导入选定的配置文件，例如：

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

这些默认提供的配置文件如下（按功能分组）：

- `jbpm.default.cfg.xml` ——流程引擎的主配置文件。
- `jbpm.identity.cfg.xml`, `jbpm.jboss.idm.cfg.xml` ——身份认证相关的配置文件。
- `jbpm.jbossremote.cfg.xml` ——基于 JBoss 应用服务器实现分布式远程调用的配置文件。
- `jbpm.jobexecutor.cfg.xml` ——Job 执行器配置文件。用于配置异步活动和定时器 Job 的执行策略。
- `jbpm.task.lifecycle.xml` ——任务生命周期状态定义的配置文件。默认的有开启（open）状态、挂起（suspended）状态、取消（cancelled）状态、完成（completed）状态。
- `jbpm.tx.hibernate.cfg.xml` , `jbpm.tx.jta.cfg.xml` , `jbpm.tx.spring.cfg.xml` ——Hibernate 事务，JTA 事务和 Spring 事务的配置文件。
- `jbpm.variable.types.xml` ——流程变量数据类型映射的配置文件。
- `jbpm.wire.bindings.xml`, `jbpm.jpdl.bindings.xml` ——基于 jBPM4 的 IOC 架构，将引擎组件绑定在运行环境中的配置文件（通过依赖注入）。
- `jbpm.businesscalendar.cfg.xml` ——工作日历的配置文件。

以下将介绍主要的配置技巧。

14.1 配置文件设计概要

一个典型的 jBPM4 总体配置文件 `jbpm.cfg.xml` 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<jbpm-configuration>
  <import resource="jbpm.default.cfg.xml" />
  <import resource="jbpm.businesscalendar.cfg.xml" />
  <import resource="jbpm.tx.hibernate.cfg.xml" />
  <import resource="jbpm.jpdl.cfg.xml" />
  <import resource="jbpm.identity.cfg.xml" />
  <import resource="jbpm.jobexecutor.cfg.xml" />
</jbpm-configuration>
```

其中 `jbpm.default.cfg.xml` 包含了默认的关键配置，例如 `Service`，`Hibernate` 会话工厂和持久化对象（导入自 `jbpm.hibernate.cfg.xml` 文件），工作日历（导入自 `jbpm.businesscalendar.cfg.xml` 文件）等。

当您想增加现有的功能时，首先可以考虑导入一些其他的配置文件或修改配置，例如，假设您需要在 JTA 环境中运行 jBPM4，则可以考虑导入 `jbpm.tx.jta.cfg.xml` 配置文件替换 `jbpm.tx.hibernate.cfg.xml` 配置文件。您还可以在 `jbpm.cfg.xml` 中直接指定配置项替换导入的配置文件，例如，本章的“配置身份认证组件”节将介绍如何设置客户化的身份认证适配器来替换 `jbpm.identity.cfg.xml` 配置文件。

默认地，jBPM4 采用 `Hibernate` 管理的事务持久化机制，在 `jbpm.tx.hibernate.cfg.xml` 中导入 `jbpm.hibernate.cfg.xml` 配置文件：

```
</jbpm-configuration>
...
<hibernate-configuration>
  <cfg resource="jbpm.hibernate.cfg.xml" />
</hibernate-configuration>
...
</jbpm-configuration>
```

`jbpm.hibernate.cfg.xml` 配置文件是一个标准的 `Hibernate3` 配置文件，它负责将如下 jBPM4 持久化实体集映射到指定的关系型数据库中：

- `jbpm.execution.hbm.xml`

- jbpmm.history.hbm.xml
- jbpmm.identity.hbm.xml
- jbpmm.repository.hbm.xml
- jbpmm.task.hbm.xml

通常情况下，作为 jBPM4 的应用开发者没必要深入到解析配置文件的原理，您仅仅需要了解如何选择需要的配置文件和修改之即可。

14.2 配置工作日历

如果想自定义工作日历的实现，可以在 jbpmm.cfg.xml 配置文件中这样配置：

```
<jbpmm-configuration>
  <import resource="jbpmm.default.cfg.xml" />
  ...
  <process-engine-context>
    <object class="yourpackage.CustomBusinessCalendar" />
  </process-engine-context>
  ...
</jbpmm-configuration>
```

以上自定义工作日历 CustomBusinessCalendar 将覆盖默认导入的 jbpmm.businesscalendar.cfg.xml 配置文件中设定的工作日历规则。

关于 CustomBusinessCalendar 的实现细节，在 10.1.2 工作日历中已经介绍过了。

14.3 配置身份认证组件（组织适配器）

目前，jBPM4 提供两套身份认证组件的实现：

- IdentitySessionImpl
 - jBPM4 默认的身份认证组件。
- JBossIdmIdentitySessionImpl
 - 针对 JBoss 应用服务器 IDM 身份认证机制的实现。

如果要使用定制的身份认证组件，步骤如下：

1) 删除 jbpm.cfg.xml 文件中的如下配置:

```
<import resource="jbpm.identity.cfg.xml" />
```

2) 在 jbpm.cfg.xml 文件中加入如下配置:

```
<transaction-context>
    <object class="yourpackage.YourIdentitySessionImpl" />
</transaction-context>
```

YourIdentitySessionImpl 正是身份认证组件的客户化定制实现, 它需要实现 org.jbpm.pvm.internal.identity.spi.IdentitySession 接口。

即定制身份认证组件, 需要实现如下的接口方法:

```
public interface IdentitySession {
    //创建一个用户对象 (User), 返回用户 ID
    String createUser(String userId, String givenName, String familyName,
String businessEmail);
    //根据用户 ID 获取用户对象
    User findUserById(String userId);
    //根据用户 ID 数组获取用户列表
    List<User> findUsersById(String... userIds);
    //获取所有用户的列表
    List<User> findUsers();
    //删除用户
    void deleteUser(String userId);
    //创建一个用户组, 返回用户组 ID
    String createGroup(String groupName, String groupType, String parentGroupId);
    //根据用户组 ID 获取此组内的用户列表
    List<User> findUsersByGroup(String groupId);
    //根据用户组 ID 获取用户组对象 (Group)
    Group findGroupById(String groupId);
    //根据用户 ID 和用户组类型获取用户组列表
    List<Group> findGroupsByUserAndGroupType(String userId, String groupType);
    //根据用户 ID 获取其所属的用户组列表 (此 API 表明 jBPM4 支持一个用户属于多个用户组)
    List<Group> findGroupsByUser(String userId);
    //删除用户组
    void deleteGroup(String groupId);
    //关联用户和用户组, 并设置用户在组中的角色 (这个角色属性可以不设置)
    void createMembership(String userId, String groupId, String role);
    //删除用户和用户组的关联关系
```

```
void deleteMembership(String userId, String groupId, String role);  
}
```

同时，还要实现客户化的 User 和 Group 类型才可完成身份认证组件的定制。

jbpm4 的 User 接口定义如下：

```
public interface User {  
    //获取用户 ID  
    String getId();  
    //获取用户名字  
    String getGivenName();  
    //获取用户姓氏（用户的姓名分离，这是典型的西方用户习惯设定，权且接受吧）  
    String getFamilyName();  
    //获取用户电子邮件地址  
    String getBusinessEmail();  
}
```

jbpm4 的 Group 接口定义如下：

```
public interface Group {  
    //获取用户组 ID  
    String getId();  
    //获取用户组名称  
    String getName();  
    //获取用户组类型  
    String getType();  
}
```

事实上，以上的 User 接口和 Group 接口正是“最小工作流组织模型”的定义，我们完全可以根据业务系统的需求扩展这两个接口，例如：

- 实现自定义的 IUser 接口继承 User 接口，添加如下方法。
 - String getAddress()——获取用户的地址。
 - String getLeader()——获取用户的直属上级 ID。
 -
- 实现自定义的 IGroup 接口继承 Group 接口，添加如下方法。
 - String getDescription()——获取关于用户组的描述。
 - String getSuperGroup()——如果用户组被视为组织结构中的“部门”的话，那么获取“上级部门”这个需求应该也很合理。

IdentitySession, User 和 Group 这 3 个接口可以视为 jbpm4 的“组织适配器”体系，

这也是一个最小组织权限系统的定义。用户可以通过实现、扩展这 3 个接口来适配任何已有的组织权限系统。事实上，在 workflow 系统实施之前，用户的组织权限系统早已存在是很常见的业务场景。与已有的组织权限系统结合正是这个“组织适配器”的用武之地。

那么，通过这个“组织适配器”来定制实现组织权限系统也绝非难事，您可以很方便地抛弃 jBPM4 自带组织数据表：JBPM4_ID_USER，JBPM4_ID_MEMBERSHIP，JBPM4_ID_GROUP，而基于自己的实现重建整个组织权限系统的持久化层。

您可以参考前面提到的 jBPM4 身份认证组件的默认实现及其对 JBoss 应用服务器的定制实现，来快速地获得自定义身份认证组件的灵感。

14.4 小结

本章首先详细介绍了 jBPM4 的配置文件体系，有众多灵活、可配置的功能是一个具有良好扩展性系统所必须具备的特点。

本章重点介绍了基于配置文件定制工作日历和身份认证组件的方法，这可以帮助很多开发者打消选择 jBPM4 作为 workflow 管理系统框架的疑虑。特别是后者——通过自实现“组织适配器”，jBPM4 几乎能与任何组织权限系统无缝整合，无疑对于想保持已有 IT 系统投资的企业具有很大吸引力。



对于 jPDL 中定义的异步活动和定时工作，jBPM 是利用事务性的异步消息和定时器 Job 来实现的。这二者并不属于标准 Java 平台提供的功能，因此，jBPM4 使用了支持集群环境的 JodExecutor 组件来执行异步消息和定时器 Job。

15.1 设计原理

默认情况下，当调用一个服务（例如 TaskService，ExecutionService）的方法时，整个调用的生命周期会在同一个线程中，并且这个线程由客户端应用发起，例如一个 HTTP 请求。

大多数情况下，上述处理方式完全满足需要，因为流程中的大多数自动活动执行不需要花费很多时间。这意味着向一个流程实例下达执行命令，从一个等待活动到达另一个等待活动，其间经过流程中的若干（自动活动）步骤是可以在一个事务中完成的。

然而，在另外一些业务场景中，流程使用异步执行的方式会更有效，例如运行时长达数小时的自动活动、需要等到指定时间执行的工作等。

将一个活动标记为异步， workflow 引擎会在执行到此活动时，不在主线程（即由客户端启动调用的线程）中执行它，而是开启另一个线程执行它，此机制也被用于执行定时器和异步邮件（所谓异步邮件，就是使用单独的线程发送邮件）工作。图 15-1 示意了 jBPM4 如何利用 JodExecutor 组件执行异步活动。

在流程执行过程中，当遭遇**异步活动**或定时器工作时， workflow 引擎首先会创建一个相应的 Job 对象到**数据库**中，这个 Job 对象主要描述执行时间和需要执行的逻辑。需要注意这个创建 Job 的机制是可插拔的，这意味着在以后可以使用 JMS，JCR 等技术来替换。

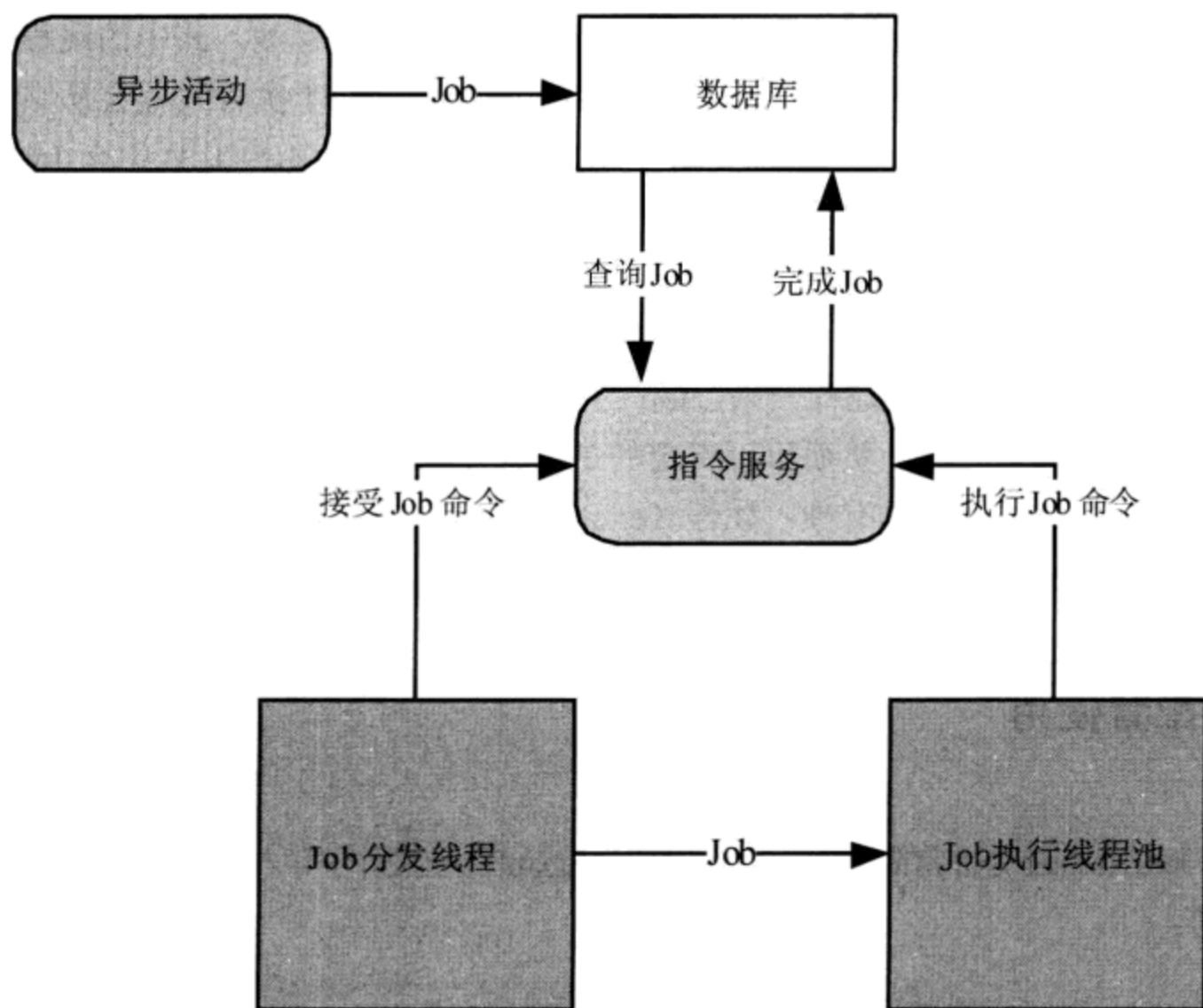


图 15-1 JobExecutor 组件执行异步活动的示意图

JobExecutor（即对应图 15-1 中的**指令服务**）随 workflow 引擎启动开始运行，它会不断地扫描数据库中的 Job 对象，识别可以被执行的 Job。实际上，JobExecutor 也是一个管理器，它管理着以下几个组件：

- 一个共享的阻塞队列，它用来临时存储可执行 Job 的 ID。何谓可执行？例如当前时间已经到达或超过了该 Job 的执行时间。
- 一个 **Job 分发线程**。这个线程会从数据库中不断扫描“可执行的 Job”。这个线程通过 **CommandService** 调用一个预设的命令。因为使用了 **CommandService**，所以该命令（默认）自动启用事务，即被配置好的事务拦截器封装。如果该 Job 暂时没有得到执行机会，分发线程就会把该 Job 的 ID 放到上面提到的共享阻塞队列中去，直到此队列满。当共享的阻塞队列满时，线程会自动被 JVM 锁定，直到队列中的 Job 被执行完成从而空出，或数据库中已经扫描不到可以执行的 Job（Job 分发线程会等待一段配置好的时间“idle time”来确认是否扫描不到可以执行的 Job）。

- 一个 Job 执行线程池。即用来传送和执行 Job 的线程池，其中的线程数量可以配置，这个数量会影响共享阻塞队列的大小。每个执行线程会从共享阻塞队列中获得 1 个 Job 来处理，队列会阻塞执行线程，直到队列中有 Job，队列中一旦有了 Job 就会立即被一个等待的执行线程处理。在从共享阻塞队列中获得 Job 之后，执行线程会利用 CommandService 执行一个专用的命令，在事务的控制中完成这个 Job，因此，Job 会被在各个执行线程（而非主线程）中完整地执行。但是这样一来，Job 的完成顺序就不可预知了，因为可能有多个竞争的执行线程，然而可以肯定的是，一个事务中只会有一个 Job 被完成。要注意的是 Job 具有独占属性（exclusive），所有设定了独占属性的 Job 会按照触发顺序依次执行完成，可参见 6.5 异步执行了解独占属性的细节。

15.2 配置使用

启用 JobExecutor 是非常简单的，只要在 jbpm.cfg.xml 配置文件中导入 jbpm.jobexecutor.cfg.xml 配置即可：

```
<import resource="jbpm.jobexecutor.cfg.xml" />
```

在 jbpm.jobexecutor.cfg.xml 配置文件中，可以通过 job-executor 配置节点调整异步工作执行器的属性。有如下参数可供调整：

- threads
 - 定义 Job 执行线程池的容量（默认为 3 个线程）。
- idle
 - 定义 Job 分发线程在数据库中扫描可执行 Job 的间隔时间（以毫秒为单位，默认 5000 毫秒）。
- idle-max
 - 每当扫描数据库出现一个异常时，idle 参数设定的间隔时间会翻倍，直到达到 idle-max 设定的时间值为止。这其实是一个补偿机制，用来避免从一个有问题的数据库一直读取数据。
- lock-millis
 - 定义一个 Job 被 Job 分发线程读取之后经过多久会被执行线程锁定。以毫秒为单位。这可以预防多个 Job 执行线程访问同一个 Job 时（例如在集群的情况下），会出现死锁的情况。

以下是经过上述参数调整后的 jbpmm.jobexecutor.cfg.xml 配置文件示例：

```
<jbpm-configuration>
  <process-engine-context>
    <!-- 参数的意义依次为：线程池容量大小为 4，扫描可执行 Job 的时间间隔为 1.5 秒，
    Job 扫描异常的“容忍时间”上限为 6 秒，Job 被取出后的生存时间上限为 1 小时。 -->
    <job-executor threads="4" idle="15000" idle-max="60000" lock-millis=
    "3600000" />
  </process-engine-context>
</jbpm-configuration>
```

15.3 小结

本章详细介绍了 jBPM4 异步执行功能的实现原理，帮助读者进一步理解异步执行功能适用的业务场景。同时，详细介绍了如何配置和使用异步执行的总控制器——异步工作执行器（JobExecutor）。

根据作者对于一些通信类企业实施 jBPM workflow 管理系统的了解，异步执行功能的应用是必不可少的。

在现代企业中，电子邮件的重要性不言而喻，所以，一个成熟的企业应用框架必须要对电子邮件服务具有良好的支持。

本章将深入介绍 jBPM4 对于电子邮件功能的实现和支持。jBPM 4 利用 JavaMail API 来实现电子邮件服务。

提示：JavaMail API 为构建电子邮件和消息应用提供一套平台无关及协议无关的框架。JavaMail API 可以作为 Java SE 平台的一个可选组件包，并且被默认包含在 Java EE 平台中。

JavaMail 项目的网址为 <http://java.sun.com/products/javamail/>。

16.1 电子邮件的产生

jBPM4 使用“邮件生产者”来生成电子邮件对象。所谓的邮件生产者都需要实现 `org.jbpm.pvm.internal.email.spi.MailProducer` 接口。默认地，邮件生产者应该可以接受外部提供的电子邮件收件人地址。

jBPM4 提供了默认的邮件生产者实现——`MailProducerImpl`。默认的邮件生产者可以根据“模板”生成文本或 HTML 内容的邮件体和附件。模板有两种设置方式：

- 在 jPDL 流程定义的 mail 活动内部定义。
- 在 jBPM4 配置文件中的 `process-engine-context` 节点配置。

模板可以包含表达式、流程变量。工作流引擎负责解释流程变量和调用脚本引擎去执行表达式。

以下是在 mail 活动内部定义电子邮件模板的 jPDL 示例（即第一种方式）：

```
<process name="Process4Mail" xmlns="http://jbpm.org/4.3/jpdl">
```

```

...
<!-- 在这里指定电子邮件模板中表达式使用的脚本语言类型。如果没有指定，会使用默认的表
达式语言 juel。 -->
<mail name="MailAct" language="juel">
    <!-- 以下开始电子邮件模板定义： -->
    <!-- 发件人列表（以英文半角逗号分隔）。发件人可以是直接电子邮件地址或具有电子邮
件属性的 jBPM4 用户或用户组（即身份认证组件中的 User 或 Group 对象）名称。 -->
    <from addresses='alex@renren.com' />
    <!-- 收件人列表（以英文半角逗号分隔）。收件人包括以下分类：to - 普通收件人，cc -
抄送收件人，bcc - 密送收件人。收件人可以是直接电子邮件地址或具有电子邮件属性的 jBPM4 用
户或用户组（即身份认证组件中的 User 或 Group 对象）名称。 -->
    <to addresses='joe@qq.com, nick@msn.com' />
    <cc users='dick' />
    <bcc groups='fansgroup, innerparty' />
    <!-- 电子邮件的主题（Mail Subject）。在这里引用了流程变量。 -->
    <subject>Hello, Part ${part} Chapter ${chapter}</subject>
    <!-- 纯文本格式的电子邮件内容（Mail Body）。在这里引用了流程变量。 -->
    <text>Daily ${date} reporting by ${username}, refs ${persons} to deal
with something.</text>
    <!-- 在 html 元素中定义一段 XHTML 的文本，可作为 HTML 格式的电子邮件内容。 -->
    <html>
        <table>
            <tr>
                <td>Daily</td>
                <td>${date}</td>
                <td>reporting by ${username}, refs ${persons} to deal
with something.</td>
            </tr>
        </table>
    </html>
    <!-- 电子邮件的附件。获取附件资源的方式可以是：url - Web 地址上的资源，resource
- classpath 中的资源，file - 本地文件系统中的资源。 -->
    <attachments>
        <attachment
url='http://www.baidu.com/search/baike_help.html' />
        <attachment resource='org/example/sample.jpg' />
        <attachment file='${user.home}/.cfg' />
    </attachments>
    <!-- 电子邮件模板定义结束。 -->
</mail>
...
</process>

```

注意：电子邮件模板的每个自定义部分都可以使用表达式脚本、引用流程变量。

现在来介绍第二种使用电子邮件模板的方式。

在 jBPM4 配置文件的 `process-engine-context` 节点可以配置“全局电子邮件模板”。上面流程定义中 `mail` 活动内的所有元素都可以原封不动地移到全局电子邮件模板的配置中。例如下面的配置片段：

```
<jbpm-configuration>
  <process-engine-context>
    <!-- 全局电子邮件模板的名称 my-template -->
    <mail-template name="my-template">
      <!-- 以下同 mail 活动的定义 -->
      <from addresses='alex@renren.com' />
      <to addresses='joe@qq.com, nick@msn.com' />
      <cc users='dick' />
      <bcc groups='fansgroup, innerparty' />
      <subject>Hello, Part ${part} Chapter ${chapter}</subject>
      <text>Daily ${date} reporting by ${username}, refs ${persons}
to deal with something.</text>
    </mail-template>
  </process-engine-context>
</jbpm-configuration>
```

每个全局电子邮件模板必须具有一个在全应用程序内唯一的名称，例如上例的 `my-template`。然后，流程定义的 `mail` 活动即可在其 `template` 属性中通过引用模板名称来使用全局电子邮件模板，如下面的 jPDL 流程定义片段所示：

```
<process name="Process4Mail" xmlns="http://jbpm.org/4.3/jpdl">
  ...
  <mail name="MailAct" template="my-template" />
  ...
</process>
```

您可能需要定义更复杂的电子邮件或自由地生成电子邮件附件（而不是利用一个已有的资源），那么请参考本章的“电子邮件扩展”部分。

16.2 电子邮件服务器

jBPM4 使用 JavaMail API 来支持电子邮件服务，但是 JavaMail 只提供邮件的构建服务，并不能提供邮件服务器。所以我们需要在配置文件中为 jBPM 指定邮件服务器。

邮件服务器的配置需要在 jBPM4 配置文件的 mail-server 节点定义。在 mail-server 指定一个 SMTP 邮件服务器来发送电子邮件消息。

所有标准 JavaMail 支持的属性都可以在 jBPM4 配置文件中使用。先看一个最简单的邮件服务器配置，使用 session-properties 子节点指定 SMTP 属性：

```
<jbpm-configuration>
<transaction-context>
...
<mail-session>
  <mail-server>
    <session-properties>
      <!-- SMTP 服务器的地址 -->
      <property name="mail.smtp.host" value="localhost" />
      <!-- SMTP 服务器的端口 -->
      <property name="mail.smtp.port" value="2525" />
      <!-- 默认的发件人电子邮件地址。即如果在电子邮件构建中没有指定 From 属性，则会使用此参数的值作为发件人电子邮件地址 -->
      <property name="mail.from" value="noreply@jbpm4.com" />
    </session-properties>
  </mail-server>
</mail-session>
...
</transaction-context>
</jbpm-configuration>
```

您也可以使用 properties 文件来配置电子邮件服务器，例如 jbpm.mail.properties：

```
mail.smtp.host localhost
mail.smtp.port 2525
mail.from noreply@jbpm4.com
```

然后在 jBPM4 配置文件中引用此 properties 文件：

```
</jbpm-configuration>
...
```

```

<transaction-context>
    ...
    <mail-session>
        <mail-server>
            <session-properties resource="jbpm.mail.properties" />
        </mail-server>
    </mail-session>
    ...
</transaction-context>
...
</jbpm-configuration>

```

jBPM4 已经支持了多 SMTP 服务器的配置，来适应不同类型邮件发送的需求。例如，对于那些既需要使用内部邮箱又需要使用外部邮箱的企业就很有用。

配置多 SMTP 服务器，需要在配置文件中定义多个 mail-server 节点。同时需要使用 address-filter（地址过滤器）节点定义每个 mail-server（电子邮件服务器）的处理域。address-filter 节点使用正则表达式来过滤收件人电子邮件地址，使得不同类型的收件人电子邮件地址被不同的 mail-server 发送。

提示：address-filter 节点使用的正则表达式语法请参考 Sun 正则表达式规范。

多 SMTP 服务器配置片段如下：

```

<jbpm-configuration>
    ...
    <transaction-context>
        <mail-session>
            <mail-server>
                <!-- 此电子邮件服务器将处理后缀为 "@jbpm4.com" 的收件人地址 -->
                <address-filter>
                    <include>.+@jbpm4.com</include>
                </address-filter>
                <session-properties>
                    <property name="mail.smtp.host" value="1.smtp.jbpm4.com" />
                    <property name="mail.from" value="noreply@jbpm4.com" />
                </session-properties>
            </mail-server>
            <mail-server>

```

```

        <!-- 此电子邮件服务器将处理后缀非 "@jbpm4.com" 的收件人地址 -->
        <address-filter>
            <exclude>.+@jbpm4.com</exclude>
        </address-filter>
        <session-properties>
            <property name="mail.smtp.host" value="2.smtp.google.org" />
            <property name="mail.from" value="noreply@google.org" />
        </session-properties>
    </mail-server>
</mail-session>
</transaction-context>
...
</jbpm-configuration>

```

address-filter（地址过滤器）按照以下规则来处理电子邮件的分发（优先级由高至低）：

- 1) 属于 include 且非 exclude 的电子邮件地址。
- 2) 属于非 include 和非 exclude 的电子邮件地址。
- 3) 属于非 exclude 的电子邮件地址。

您可以参考 JavaMail API 的 SMTP 属性获得更多邮件服务器参数配置的信息：

<http://java.sun.com/products/javamail/javadocs/com/sun/mail/smtp/package-summary.html>。

16.3 电子邮件扩展

在本章开始时，提到过 jBPM4 使用“邮件生产者”来生成电子邮件对象。如果 jBPM4 默认邮件生产者实现的功能不能满足您的需求，则可以考虑创建自定义的邮件生产者来生成所需的特定邮件。

自定义的邮件生产者需要实现 `org.jbpm.pvm.internal.email.spi.MailProducer` 接口：

```

public interface MailProducer {
    Collection<Message> produce(Execution execution);
}

```

此接口只有唯一的方法——`produce`。`produce` 方法被注入流程实例的执行对象

execution, execution 对象能够使您获取足够多的流程实例数据, 从而结合业务需求生成标准的 Java 电子邮件消息——javax.mail.Message 对象。这些 Message 对象将会被您配置的 mail-session 发送出去。

一个最常见的电子邮件扩展需求是: 自定义电子邮件附件。这在 jBPM4 里很容易做到, 通过实现 MailProducer 接口, 或者直接扩展 jBPM4 默认的电子邮件生产者实现类 (MailProducerImpl) 都可以。

下面就是一个通过扩展 jBPM4 默认的电子邮件生产者实现类的示例, 用来实现向每条分发的电子邮件添加自定义的附件:

```
public class CustomMailProducer extends MailProducerImpl {
    //覆盖默认的添加附件方法
    @Override
    protected void addAttachments (Execution execution, Multipart multipart)
    {
        try {
            //在这里简单地自定义一个附件。当然, 您可以根据流程提供的参数设计更复杂的
            //附件获取逻辑, 例如将附件的获取逻辑用单独的方法或类去实现
            DataHandler[] attachmentDatas = new DataHandler[] { new DataHandler(
                new URL("http://jbpm4.com/jbpm.gif")) };
            // 首先需要执行默认的逻辑
            super.addAttachments(execution, multipart);
            for (DataHandler attachmentData : attachmentDatas) {
                //创建一个附件的容器
                BodyPart attachmentPart = new MimeBodyPart();
                //在容器中设置附件
                attachmentPart.setDataHandler(attachmentData);
                //将附件加入当前邮件对象
                multipart.addBodyPart(attachmentPart);
            }
        } catch (Exception e) {
            //在这里做异常处理
            e.printStackTrace();
        }
    }
}
```


16.4 小结

现代型企业的日常工作，怎能离开电子邮件？

尽管有众多的信息管理系统可用，但只有收发电子邮件是几乎所有企业的员工都必须做的“功课”。因此，即使我们的 workflow 管理系统无比强大，但很多情况下也不得不使用电子邮件这种通用的异步消息技术来完成通知、提醒任务的办理等工作。

通过本章的介绍，您对 **6.3.5 mail（邮件活动）** 的应用和定制有了更深入的了解，您将掌握如何配置电子邮件的内容、配置电子邮件的发送服务器（SMTP），以及扩展、自定义电子邮件的整个生成过程。

相信通过了解本章的内容，读者能将电子邮件发送服务在 jBPM4 系统中使用得得心应手。



对于信息化“系统”来说，日志是一个非常重要的功能部分。日志可以记录下系统产生的所有行为，并按照某种规范表达出来。我们可以使用日志的信息为系统排查异常、优化系统性能或者根据这些日志信息优化系统的功能。在安全领域，日志的地位尤其重要，可以说是支持系统安全审计工作最主要的工具。

因此，但凡软件产品上升到“系统”的高度，日志必然不可缺少。

反之，日志缺少或日志不完备的信息化应用不配称之为系统。

17.1 配置日志

jBPM4 的系统日志由 PVM 管理。PVM 支持两套不同的日志工具：

- Java Logging API——jBPM4 默认。由 `java.util.logging` 工具包提供。

提示：Java Logging API 是 Sun 公司于 2002 年 5 月正式发布的。它是自 J2SE 1.4 版本开始提供的一个应用程序接口。它能够很方便地控制和输出日志信息到文件系统、控制台或其他用户定义的位置，如数据库系统、电子邮件地址、Socket 端口等。所以它是为最终用户、系统管理员、软件服务工程师和开发人员提供的一种捕捉安全漏洞、检查配置正确性、跟踪调查系统运行瓶颈和系统运行异常的工具。

这个项目的主页位于 <http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/>。

- log4j——最流行的开源日志项目。由 Apache 开放源代码组织提供。

提示: log4j 是 Apache 旗下的一个开放源代码项目。使用 log4j, 我们可以将日志信息打印到控制台、文件系统、GUI 组件, 甚至是 Socket 端口、Windows NT 事件记录器、UNIX Syslog 守护进程等目的地。用户可以定制每条日志信息的输出格式。用户可以定制每条日志信息的输出级别。总之, 用户能够细致地控制日志生成的全生命周期过程。这些工作都可以通过一个配置文件来灵活地设定, 而几乎不需要修改程序代码。

这个项目的主页位于 <http://logging.apache.org/log4j/>。

jBPM4 提供了一个统一的日志工具——org.jbpm.internal.log.Log 类, 您可以使用如下代码获取系统日志对象:

```
// YourClass 是以 Java Class 区分的日志对象归属类型
private static Log log = Log.getLog(YourClass.class.getName());
```

jBPM4 的系统日志分为 5 个输出级别, 根据严重程度由低至高依次为:

- 1) TRACE ——一般用于程序输出的跟踪。
- 2) DEBUG ——一般用于程序调试。
- 3) INFO ——一般用于程序运行的关键信息打印。
- 4) WARN ——一般用于在系统发生不致命异常时, 给予警告。
- 5) ERROR ——当系统发生致命异常 (例如事务失败、线程中断、系统崩溃等) 时, 记录异常信息。

有了 log 对象, 您就可以选择上述 5 个输出级别之一记录系统日志了, 例如:

```
//非 ERROR 的日志输出级别, 都提供 isXxxxEnabled 的预判方法。先行预判是否允许此
级别的日志, 是个编写高效率程序的好习惯
if (log.isDebugEnabled()) {
    log.debug("deleting process instance "+processInstanceId);
}
```

PVM 会根据如下规则选择日志工具:

- 1) 如果 Context ClassLoader 在 classpath 根下发现了 logging.properties 文件, 则 PVM 会使用 Java Logging API 来记录系统日志。logging.properties 文件是 Java Logging API 的默认配置文件。

- 2) 如果 Context ClassLoader 在 classpath 中发现了 log4j 的类库, 则 PVM 会使用 log4j 来记录系统日志。对 log4j 类库的检测根据对 org.apache.log4j.LogManager 类的存在与否来进行判定。
- 3) 最后, 如果上述两个特征都没被发现, 则 PVM 会根据默认的配置, 使用 Java Logging API 来记录系统日志。

17.2 日志输出级别

PVM 使用 Java 类名作为日志的归属类型。

如果您需要了解 PVM 运行时的底层行为, 请打开 DEBUG 级别(对应 Java Logging API 配置的 FINE 级别)。如果您打开 TRACE 级别(对应 Java Logging API 配置的 FINEST 级别), 那么您将看到“漫山遍野”的 PVM 系统日志输出。

以下是 logging.properties 文件中日志输出级别定义的片段:

```
org.jbpm.level=FINE
# 对日志输出级别作用域的定义以“类全名片段+level”的格式标识。您可以根据需要扩大和缩小
# 作用域的范围。
# 以下是设置 PVM 日志输出范围为 FINE 级别的片段:
# org.jbpm.pvm.internal.tx.level=FINE
# org.jbpm.pvm.internal.wire.level=FINE
# org.jbpm.pvm.internal.util.level=FINE
...
```

17.3 Java Logging API 日志

PVM 的默认日志工具是 Java Logging API。

在 Java Logging API 日志的输出级别配置中, 和 PVM 内置的 5 个输出级别存在如下映射关系(左侧是 PVM 的内置输出级别, 右侧是 Java Logging API 的输出级别):

- TRACE —— FINEST
- DEBUG —— FINE
- INFO —— INFO

- WARN —— WARNING
- ERROR —— SEVERE

事实上, Java Logging API 还有 CONFIG 和 FINER 两个日志输出级别的定义, 但在 PVM 中没有使用。

Java Logging API 提供众多类型的 Handler (对应 log4j 里的 Appender) 来处理不同的日志“落盘”目的地。例如:

- `java.util.logging.ConsoleHandler` —— 将日志打印到控制台。
- `java.util.logging.FileHandler` —— 将日志打印到文件系统。
-

另外, `org.jbpm.pvm.internal.log.LogFormatter` 是 PVM 类库的一部分, 它可以为 Java Logging API 日志创建一个优美的单行格式输出, 同时它还可以为每个线程的日志加上一个唯一标识。

以下是一个典型的 `logging.properties` 配置文件中 Handler 部分的片段, 同时配置和使用了 `ConsoleHandler` 和 `FileHandler`, 将日志输出到控制台和文件系统:

```
# 指定要用到的 Handler。
handlers = java.util.logging.ConsoleHandler, java.util.logging.FileHandler
# 将 Apache Common-Logging 的日志也定向过来。
redirect.commons.logging = enabled
# 以下是对控制台日志的全局定义。
java.util.logging.ConsoleHandler.level = FINEST
# 使用 PVM 的日志格式化器。
java.util.logging.ConsoleHandler.formatter =
org.jbpm.internal.log.LogFormatter
# 以下是对文件系统日志的全局定义。
java.util.logging.FileHandler.level = FINEST
# 使用 Java Logging API 提供的日志格式化器。
java.util.logging.FileHandler.formatter =
java.util.logging.SimpleFormatter
# 设置日志文件的存储路径。
java.util.logging.FileHandler.pattern = /jBPM4.log
# 如果日志文件已存在, 则以追加的方式写入。
java.util.logging.FileHandler.append= true
# 设置每个日志文件的最大容量, 单位: 字节 (byte)。
```

```
# 如果日志文件满了，则将此日志文件加上整数编号（从 0 开始）的扩展名归档。然后产生一个新的日志文件。  
# 如果此参数值为 0，则表示日志文件的容量无限。这也是默认的日志文件容量策略。  
java.util.logging.FileHandler.limit = 50 000  
# 设置日志文件的最大数量（如果数量达到了，则循环覆盖）。默认值为 1。  
java.util.logging.FileHandler.count = 10
```

17.4 利用持久化层日志进行调试

不知大家有没有这样的感觉，在调试程序的时候，“隐藏最深的 Bug”往往需要跟踪到数据持久化层才能发现。因此掌握 jBPM4 持久化层，即 Hibernate 层日志的调试技巧，对于 jBPM 业务流程开发者来说是有必要的。

● 技巧一

打开包 `org.hibernate.sql` 的跟踪日志，可以显示所有执行的 SQL 语句。打开包 `org.hibernate.type` 的跟踪日志，可以显示 SQL 查询中设置的参数值。配置如下：

```
org.hibernate.SQL.level=FINEST  
org.hibernate.type.level=FINEST
```

● 技巧二

可以关闭 Hibernate 的批处理机制，这样 SQL 语句会立即得到执行，方便同步跟踪代码；如果有异常也会在第一时间抛出，方便定位异常。

这需要在 Hibernate 的配置文件（一般名称是“*hibernate.cfg.xml”）里做如下设置：

```
hibernate.jdbc.batch_size = 0
```

● 技巧三

在 Hibernate 的配置文件中，设置允许输出执行的 SQL 语句细节到控制台：

```
hibernate.show_sql = true  
hibernate.format_sql = true  
hibernate.use_sql_comments = true
```

17.5 小结

本章集中介绍了所谓“信息系统”中必不可少的一环——jBPM4 的系统日志。读者通过本章的介绍，应该能理解 jBPM4 系统日志的工作原理，掌握配置日志的技巧。

这对于开发者在开发过程中的 Debug、实施过程中的调试，以及上线运行过程中的业务追查和分析等工作，都是必不可少的。



jBPM4 与 Spring 框架集成

支持嵌入到各种框架环境中使用一直是 jBPM workflow 引擎的核心竞争力之一。自 jBPM3 版本开始, jBPM workflow 引擎就在很多应用中被集成到 Spring 等框架中使用, 但这通常需要额外的依赖库和非正式的解决方案来支持。从 jBPM4 版本开始, jBPM workflow 引擎可以支持开发者很自然地将其集成到 Spring 框架中使用。

Spring 框架 (Spring Framework) 是当前最流行的 Java EE 应用框架之一。Spring 框架既广泛应用在企业级系统中, 也大量应用在互联网环境中。SSH (Struts, Spring, Hibernate) 架构的应用在国内有着极为广泛的市场, 对于 jBPM4 来说, 其持久层已经采用了 Hibernate 框架实现, 其客户端 (控制层) 使用 Struts 框架也不会遇到什么技术问题, 因为 jBPM 框架不绑定任何客户端实现。那么, 如何使 jBPM 与 Spring 框架集成在一起使用, 例如使用 Spring 框架的 Auto-Ware 机制为客户端注入 jBPM 服务, 特别是把 jBPM 的 Hibernate 持久层实现与 Spring 框架完美地结合, 则是一个需要认真解决的问题。

提示: Spring Framework 是一个解决了许多 Java EE 开发中常见问题的强大应用程序框架。Spring 框架提供了管理业务对象的统一方法, 并且鼓励依赖注入和针对接口编程而不是在类内部硬编码的良好开发习惯。Spring 框架的架构基础是基于使用 JavaBean 属性的控制反转 (Inversion of Control, IoC) 容器解决业务逻辑类之间的耦合。Spring 框架在使用 IoC 容器作为构建所有架构层次的完整解决方案方面是独一无二的。Spring 框架提供了独一无二的数据持久层抽象, 包括简单高效的 JDBC 框架, 这极大地改进了应用程序的执行效率并且防止了很多错误。Spring 框架的数据持久层架构还集成了 Hibernate 框架和许多其他的“对象-关系”映射解决方案。Spring 框架还提供了独一无二的事务管理抽象, 它能够在各种不同的事务管理技术 (例如 JTA 事务管理器或 JDBC 事务管理器) 基础上提供一个一致的编程模型。Spring 还提供了一个使用标准 Java 语言编写的 AOP (面向切面编程) 框架, 它可以为基于简单 Java 对象的持久层提供声明式事务管理机制和其他企业级事务管理机制——如果您需要还可以自定义 AOP 拦截器。

总之，Spring 这个框架足够强大，使得应用程序能够抛开 EJB（企业级 JavaBean）的复杂性，同时享受着和传统 EJB 相似的关键服务。Spring 也提供了与其 IoC 容器集成的强大灵活的 MVC 应用开发框架。

Spring Framework 项目网址位于 <http://www.springsource.org/>。

18.1 集成的目标

Spring 框架集成 jBPM4，只要达成两个目标，就可以基本成功了：

- 持久化集成
 - 默认地，jBPM4 为每个客户端操作开启一个事务，在此事务中调用服务 API。而在通常的 Spring 应用中，应用的访问一般来自 Web 的 HTTP 请求，然后在此 HTTP 请求线程中，通过调用 Spring Bean 的方法进入事务边界。这与标准的 jBPM4 事务处理方式是不同的。因此 jBPM4 提供了相应的工具将自身的持久化事务管理权交给 Spring 框架。
- 服务集成
 - 默认地，jBPM4 的客户端通过硬编码获取各种工作流服务接口。现在，需要将这些 jBPM4 服务接口集成到 Spring 的 IoC 架构中，作为 Spring Bean，经由依赖注入等方式供客户端应用调用。

18.2 为集成配置 jBPM4

下面以 Spring2.X 版本为例，说明如何集成 jBPM4 到 Spring 框架中去。

首先，需要将 jBPM4 默认的 Hibernate 事务管理配置替换为 Spring 事务管理配置，即在 jBPM4 主配置文件 jbpm.cfg.xml 中将：

```
<import resource="jbpm.tx.hibernate.cfg.xml" />
```

替换为：

```
<import resource="jbpm.tx.spring.cfg.xml" />
```

jbpm.tx.spring.cfg.xml 的内容如下（已标识出与默认的 jbpm.tx.hibernate.cfg.xml 不

同之处):

```
<!-- 支持 Spring 框架标识 -->
<jbpm-configuration spring="enabled">
  <process-engine-context>
    <command-service name="newTxRequiredCommandService">
      <retry-interceptor />
      <environment-interceptor policy="requiresNew" />
      <!-- 将默认的 standard-transaction-interceptor 替换为 spring-
transaction-interceptor (Spring 事务拦截器) -->
      <spring-transaction-interceptor policy="requiresNew" />
    </command-service>
    <command-service name="txRequiredCommandService">
      <retry-interceptor />
      <environment-interceptor />
      <!-- 将默认的 standard-transaction-interceptor 替换为 spring-
transaction-interceptor (Spring 事务拦截器) -->
      <spring-transaction-interceptor />
    </command-service>
  </process-engine-context>
  <transaction-context>
    <transaction type="spring" />
    <hibernate-session current="true" />
  </transaction-context>
</jbpm-configuration>
```

是否配置 hibernate-session 节点的属性 current="true", 这取决于您是否使用了 Spring 中的“current session”策略。而且, 如果您希望事务只由 Spring 管理, 请将 transaction 节点从 transaction-context 节点中删除, 这会强制让 workflow 引擎从 Spring 中搜索当前持久化会话。

根据上面 jbpm.tx.spring.cfg.xml 中的配置, workflow 引擎将会使用默认 Spring 配置文件 applicationContext.xml 中的事务管理器。如果需要为 jBPM4 指定其他 Spring 配置文件, 请在 jbpm.cfg.xml 中做如下配置, 例如使用 applicationContext2.xml 中的事务管理器:

```
</jbpm-configuration>
...
<process-engine-context>
  <string name="spring.cfg" value="applicationContext2.xml" />
</process-engine-context>
```

```
...
</jbpm-configuration>
```

工作流引擎使用 `spring.cfg` 这个上下文常量来定位 Spring 配置文件，在类 `org.jbpm.pvm.internal.processengine.SpringProcessEngine` 的代码片段中您可以发现 `spring.cfg` 的默认值正是 `applicationContext.xml`：

```
...
String springCfg = (String) configuration.getProcessEngineWireContext().
get("spring.cfg");
//当 jBPM4 工作流引擎未找到 spring.cfg 常量时，则默认 classpath 根路径下的
applicationContext.xml 为 Spring 配置文件
if (springCfg == null) {
    springCfg = "applicationContext.xml";
}
applicationContext = new ClassPathXmlApplicationContext(springCfg);
...
```

`jbpm.tx.spring.cfg.xml` 中的 `spring-transaction-interceptor` 节点默认会使用 Spring 配置文件中名为 `transactionManager` 的事务管理器（实现 `org.springframework.transactionPlatformTransactionManager` 接口）。但如果 Spring 配置文件中有多多个事务管理器的话，我们需要通过如下配置指明到底要使用哪个 Spring 事务管理器，例如 `transactionManager2`：

```
<spring-transaction-interceptor transaction-manager="transactionManager2" />
```

18.3 为集成配置 Spring

上节介绍了在 jBPM4 中需要完成的配置工作，下面的工作都是在 Spring 配置文件里做的。

在上面的配置中，jBPM4 已经将事务管理权交给了 Spring。那么接下来，就要把 jBPM4 的服务配置在 Spring 框架中，以使客户端能够通过 Spring 框架的 `get Bean` 机制调用 jBPM4 的服务。

以下是一个完整的集成了 jBPM4 的 Spring 配置文件（`applicationContext.xml`，基于 Spring2.X 版本）：

```
<beans xmlns=http://www.springframework.org/schema/beans
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
```

```

xmlns:aop=http://www.springframework.org/schema/aop
xmlns:tx=http://www.springframework.org/schema/tx
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
    <!-- Hibernate 会话工厂, jBPM4 持久化必备。 -->
    <bean id="sessionFactory"

        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation">
            <!-- jbpmm.hibernate.cfg.xml 是 Hibernate 配置文件, jBPM4 中必有。 -->
            <value>classpath:jbpm.hibernate.cfg.xml</value>
        </property>
    </bean>
    <!-- 将 jBPM 配置导入 Spring 控制。 -->
    <bean id="jbpmSpringHelper" class="org.jbpm.pvm.internal.processengine.
SpringHelper"
        lazy-init="default" autowire="default" dependency-check="default">
        <!-- 指定 jBPM 主配置文件。如果 jbpm.cfg.xml 存在 classpath 根下 (默认值),
jbpmCfg 属性的配置是可以删除的。 -->
        <property name="jbpmCfg">
            <value>jbpm.cfg.xml</value>
        </property>
    </bean>
    <!-- 通过工厂方法, 将 workflow 引擎对象配置为一个 Spring Bean, 方便在 Spring 环境下使
用。 -->
    <bean id="processEngine" factory-bean="jbpmSpringHelper" factory-method=
"createProcessEngine" />
    <!-- 使用 processEngine Bean 的工厂方法创建 RepositoryService Bean——流程资
源库服务的 Spring Bean -->
    <bean id="repositoryService" factory-bean="processEngine" factory-method=
"getRepositoryService" />
    <!-- 使用 processEngine Bean 的工厂方法创建 ExecutionService Bean——流程执行
服务的 Spring Bean。按照此思路, 您可以继续创建 HistoryService Bean、TaskService
Bean…… -->
    <bean id="executionService" factory-bean="processEngine" factory-method=
"getExecutionService" />
    <!-- Spring 的事务管理器 -->
    <bean id="transactionManager"

```



```

    class="org.springframework.orm.hibernate3.HibernateTransactionManage
r">
    <property name="sessionFactory">
        <ref bean="sessionFactory" />
    </property>
</bean>
<!-- 在这里指定事务策略 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    ...
</tx:advice>
<!-- 在这里指定事务切入点 -->
<aop:config>
    ...
</aop:config>
</beans>

```

18.4 使用

当集成配置完毕后，就可以在 jPDL 流程定义中使用 Spring 服务了。

下面是在一个 java 活动的定义中通过表达式引用 Spring Bean 的示例：

```

<java name="echo" expr="#{helloService}" method="sayHello">
    <transition name="to accept" to="accept"/>
</java>

```

当 jBPM4 与 Spring 集成后执行到上面的活动，脚本引擎会在所有的上下文中查找名为 **helloService** 的对象，最先是在流程变量中查找，最后会到 Spring Context 中查找。

事实上，实施本章描述的集成方案后，Spring 的配置和 jBPM4 的配置已然“互通”。您可以配置一个 jBPM4 对象到 Spring Bean，例如 `IdentitySessionImpl`，设置其 Spring Bean id 为 `identitySession`，那么，就可以在 jBPM4 配置中使用 “`<env class="identitySession" />`” 引用这个 Spring Bean。

关于如何在基于 Spring 的客户端应用中使用 jBPM 服务，则是纯 Spring 范畴的应用技术了，本书将不做说明，请参考 Spring 编程开发的相关资料。

18.5 测试

jBPM4 提供了基于 JUnit 的 `AbstractTransactionalJbpmTestCase` 来对与 Spring 集成的业务流程进行隔离测试。`AbstractTransactionalJbpmTestCase` 继承了 Spring 框架提供的 `AbstractTransactionalDataSourceSpringContextTests` 类，被用来代替 `JbpmTestCase` 作为单元测试的父类。那么在这种情况下，编写一个 jBPM4 业务流程的单元测试和一个 Spring DAO 的单元测试是无差别的。

18.6 小结

Spring 作为一种优雅的、轻量级的 Java EE 框架受到广大“人民群众”的广泛欢迎，这也不得不“逼着”jBPM4 的开发者为集成 Spring 和 jBPM 框架开发出若干工具和设计出一些解决方案，而这正是本章向读者介绍的内容。

本章从集成的目标谈起，首先给读者一个最终能集成到何种程度的心理预期，然后分别介绍如何在 jBPM4 配置体系中和 Spring 配置体系中需要为集成所做的前期工作，再说明集成后 jBPM4 框架如何使用在 Spring 中配置的“成果”，最后介绍了由 jBPM4 提供的 Spring 集成测试工具及其使用方法。

通过本章的介绍，读者应该了解 jBPM4 良好的框架适应性。事实上，jBPM4 对 Spring 集成的实现完全可以对读者起到“抛砖引玉”的效果，如果有必要，读者应该尝试将 jBPM4 对 Spring 框架集成的方法引深到其他 Java 应用框架与 jBPM4 的集成。



jBPM4 与 JBoss 应用服务器集成

JBoss 4.2.X 和 JBoss 5.0.0.GA 版本应用服务器可以支持将 jBPM workflow 引擎和流程定义发布工具安装成一个 JBoss 服务，这样，该 JBoss 上所有的应用程序都可以使用 jBPM workflow 引擎了。

作为安装脚本的一部分，执行 `install.jbpm.into.jboss` 目标任务会自动安装 jBPM4 到指定的 JBoss 应用服务器上，具体的安装步骤请参见 2.4 安装到 JBoss。

在成功安装之后，启动 JBoss 应用服务器，在其启动日志中应该可以看到 workflow 引擎也随之启动，并绑定到一个 JDNI 服务：

```
...
12:00:22,301 INFO [JBPMService] jBPM 4 - Integration JBoss 4
12:00:22,301 INFO [JBPMService] 4.0.0.Beta1
12:00:22,301 INFO [JBPMService] ProcessEngine bound to: java:/ProcessEngine
...
```

19.1 流程定义打包部署

在 JBoss 应用服务器上部署一个流程定义打包很简单，创建一个 *.jpd1 归档文件（使用 zip 压缩）即可。这个归档文件需要包含所有的流程定义资源，这主要有：

- *.jpd1.xml 流程定义 XML 文件。
- 流程定义所引用的用户代码类文件 (*.class)。
- 其他资源，如属性文件 (*.properties)、图片文件 (*.gif, *.png, *.jpg) ……

将这些资源集中在一起，然后使用 `jar` 命令将之打包，示例如下：

```
jar -tf MyProcess.jpd1
```

流程定义打包 MyProcess.jpd1 内部结构示例如下：

- META-INF/MANIFEST.MF

- Meta 信息文件
- MyProcess.jpdl.xml - jPDL
 - 流程定义文件
- com/something/mine/Customer1.class
 - 用户代码 Java 类 1
- com/something/mine/Customer2.class
 - 用户代码 Java 类 2
-

打包成功后，接着就需要把流程定义打包部署到 JBoss 应用服务器上。

直接把流程定义打包复制到 \$JBOSS_HOME/server/<config>/deploy/ 目录下即可，例如在 Linux 下执行：

```
cp MyProcess.jpdl $JBOSS_HOME/server/default/deploy
```

JBoss 应用服务器会实时监测 deploy 目录下的变化，当复制完成后，JBoss 应用服务器会使用自带的 JBPMDeployer 工具将流程定义打包立即部署，可以使用如下命令从 JBoss 系统日志中监测部署过程：

```
less $JBOSS_HOME/server/default/log
```

执行上面的命令，则在部署时会看到类似如下的日志在控制台打出：

```
...
2010-05-01 12:22:24,947 INFO [org.jbpm.integration.jboss4.JBPMDeployer]
Deploy file:/Users/huqi/JBoss_4_2_2_GA
/opt/jboss-4.2.2.GA/server/default/deploy/MyProcess.jpdl
...
```

如果想删除一个流程定义的部署，直接在 \$JBOSS_HOME/server/<config>/deploy/ 目录下删除流程定义归档文件即可。

19.2 在 JBoss 企业级编程模型中使用 jBPM4

我们强调过，当 JBoss 集成 jBPM4 完成后， workflow 引擎会被安装成 JBoss 的 JNDI 服务。这意味着 JBoss 上的任何企业级应用程序、组件（例如 Servlet, EJB）都可以使用 JNDI 查找服务来获取工作引擎并使用它。如下例所示（jBPM4 工作流引擎的 JNDI

名称为“java:/ProcessEngine”):

```
private static ProcessEngine processEngine;
public static void main(String[] args) {
    try {
        InitialContext ctx = new InitialContext();
        //使用 JNDI 服务获取 ProcessEngine 对象就这么简单
        processEngine = (ProcessEngine) ctx.lookup("java:/ProcessEngine");
        //在这里尽情使用工作流吧
        ...
    } catch (Exception e) {
        throw new RuntimeException("Failed to lookup process engine.");
    }
}
```

得到了工作流引擎对象, 就可以使用所有的 jBPM4 服务 API。以下是在 JBoss 环境下调用 jBPM4 服务的完整示例代码:

```
public class Main {
    private static ProcessEngine processEngine;
    public static void main(String[] args) {
        try {
            InitialContext ctx = new InitialContext();
            processEngine = (ProcessEngine) ctx.lookup("java:/
ProcessEngine");
            //在这里使用 JNDI 获取事务服务是可选的。事实上, 在真正的生产环境中, 对 jBPM
服务的调用一般来自 CMT 组件 (例如一个 EJB), 这时候调用已经在事务中了, 则可以不用再新启一
个 UserTransaction 事务
            UserTransaction tx = (UserTransaction) ctx.lookup("UserTransaction");
            //获取工作流环境对象
            EnvironmentImpl env = ((EnvironmentFactory) processEngine).
openEnvironment();
            try {
                //UserTransaction 事务的上边界, 事务开始
                tx.begin();
                //获取工作流执行服务
                ExecutionService executionService = (ExecutionService)
processEngine.get(ExecutionService.class);
                //以下是具体的业务逻辑调用
                executionService.signalExecutionById("ICL.88888");
                // UserTransaction 事务的下边界, 事务提交、结束
                tx.commit();
            } catch (Exception e) {
```



```

        if (tx != null) {
            try {
                //事务失败, 回滚
                tx.rollback();
            } catch (SystemException e1) {
                e1.printStackTrace();
                //在这里可以设定对事务系统异常的处理逻辑
            }
        }
        throw new RuntimeException("...", e);
    } finally {
        //这里要手工关闭工作流环境
        env.close();
    }
} catch (Exception e) {
    throw new RuntimeException("Failed to lookup process engine.");
}
}
}
}

```

19.3 小结

由于目前 jBPM 是 JBoss 旗下的子产品, 因此 jBPM 不仅可以“部署”在 JBoss 应用服务器上, 而且可以“集成”在 JBoss 应用服务器里, 作为 JBoss 提供的服务存在于整个应用范围内, 并且能很方便地在 JBoss 企业级编程模型中被调用。这正是本章向读者介绍的内容。

通过本章的介绍, 读者应该认识到 jBPM4 不仅能简单地部署在 Web 应用服务器(这是非常容易的)之上, 如 Tomcat, Resin 等, 而且能够被“企业级”应用服务器所集成, 例如通过 JNDI 等方式被 JBoss 集成。因此, 如果有必要, 读者可以对 jBPM4 与 Oracle WebLogic, IBM WebSphere 等企业级应用服务器的部署甚至集成进行大胆尝试, 因为从理论上来说都是可行的。

目前，国内对于 workflow 管理系统的应用主要还是集中于“人工流程”，也就是以人工任务密集型的工作流应用为主。主要原因在于国内的信息化系统建设还远不及欧美发达国家成熟。系统多是以新建和推倒重来为主，很少有经过长期稳定运行的“沉淀”系统，因此 workflow 管理系统中用于应用集成的相关自动活动功能就很少能得到施展，workflow 管理系统大部分的应用还是集中在需要人工干预的“任务”上。这也可以解释为何在欧美国家大行其道的 EAI（企业应用集成）产品在国内市场并不普及的原因。

所以，当前国内的工作流管理系统主要应用在以下业务领域：

- 电子政务，最常见的场景是各种各样的行政审批流程。
- 企业协同办公，例如电信、电力等企业的工单。
- 企业的采购合同、销售合同等常用业务流程的审批和管理。

同时从 jBPM 项目本身来说，其作为一款基于西方业务流程管理思想设计的工作流框架，更多的是关注“如何辅助开发者更容易地让流程运行完成”，而不是关注“记录流程运行的历史和轨迹”。

因此，jBPM 项目从设计上就没有考虑“回退”、“取回”、“会签”、“委派”等业务场景。这也是因为东西方文化的差异之所在。例如回退，西方人认为“往回流转的情况肯定也是一种业务流程规则的定义，那么肯定可以通过分支或条件流转的设计来解决”，而国内则常常把回退作为一个“人性化管理和处理的潜在规则”来看待，并且认为这是一个合理的需求。

本章的目的就是：提出这些具有中国特色的（当然这并不是中国所特有的）业务流程问题，在 jBPM4 的架构基础上分析解决这些问题的思路，并给出一种解决方案供参考。

20.1 退回

退回 (Rollbak) 又名“回退”。退回是针对当前用户“待办任务 (task)”的操作，即当前用户主动退回待办任务到上一步骤。

为什么会有退回这种需求？

可以想象，当前用户接收任务后，发现不应由自己办理此任务或上一步任务的办理有严重业务错误，那么，他就需要将此任务退回给上一步任务的办理者重新办理。

在 jBPM4 架构下，一个解决思路是：

- 1) 识别出“需要具有退回能力”的任务活动。
- 2) 在流程定义中为具有退回能力的任务活动设置专门用于处理退回逻辑的监听器。
- 3) “退回监听器”接受一个参数，用于指定退回目的地的活动 ID——这样设计的目的是因为退回发起点和退回目的地之间可能相隔若干个活动，必须在定义中指定退回的目的地。当然这个参数可以使用流程变量，在流程运行时动态算出。
- 4) 退回监听器动态创建出一条转移路径，指向退回目的地。
- 5) 编写“退回任务”的服务 API，供客户端调用。
- 6) 如果退回操作引起了业务“损失”，还需要在退回服务 API 的实现中考虑“补偿”。如果退回操作需要清除“历史痕迹”，还需要在退回服务 API 的实现中删除相关的历史活动记录。

现在介绍详细的实现过程。退回场景的示例流程定义如图 20-1 所示。

对应的 jPDL：

```
<process name="Rollback" xmlns="http://jbpm.org/4.3/jpdl">
  <start name="start">
    <transition name="to outlay apply" to="outlay apply" />
  </start>
<!-- 费用申请 (outlay apply) 活动。作为退回的目的地。 -->
  <task assignee="Jack" name="outlay apply">
    <transition name="to leader audit" to="leader audit" />
  </task>
</process>
```

```

</task>
<!-- 领导审批 (leader audit) 活动。定义为“具有退回到 费用申请活动 的能力”。 -->
<task assignee="Alex" name="leader audit">
  <on event="start">
    <event-listener
class="com.examples.jbpm4.n3_cn_rollback.RollbackListener">
      <field name="m_rollbackTo">
        <string value="outlay apply" />
      </field>
    </event-listener>
  </on>
  <transition to="end" />
</task>
<end name="end" />
</process>

```

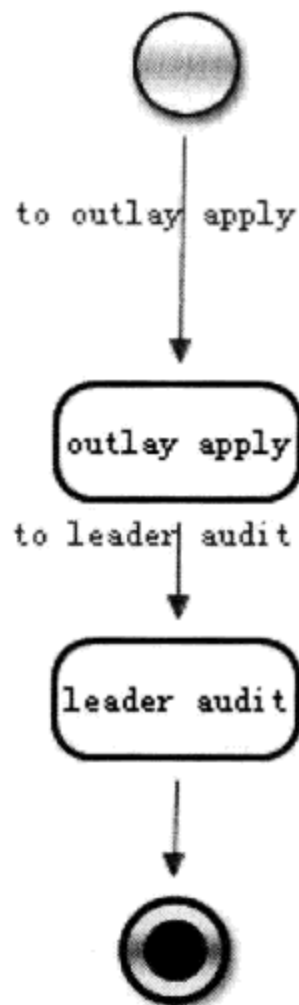


图 20-1 退回场景的流程定义

RollbackListener 正是所谓的退回监听器,它根据注入的 `m_rollbackTo` 成员域值动

态创建一条通向退回目的地的转移路径。RollbackListener 的实现如下：

```
public class RollbackListener implements EventListener {
    private static ProcessEngine processEngine = Configuration.
getProcessEngine();
    //退回目的地，通过流程定义注入
    private String m_rollbackTo;
    /**
     * 增加一条用于退回的路径
     */
    @Override
    public void notify(EventListenerExecution execution) throws Exception
    {
        // 首先要获取流程定义对象——processDefinition
        ProcessInstance processInstance = execution.getProcessInstance();
        String processDefinitionId = processInstance.getProcessDefinitionId();
        ProcessDefinitionImpl processDefinition = (ProcessDefinitionImpl)
processEngine
        .getRepositoryService().createProcessDefinitionQuery()

        .processDefinitionId(processDefinitionId).uniqueResult();
        //获取退回目的地的活动定义对象
        ActivityImpl toActivityImpl = processDefinition.findActivity
(m_rollbackTo);
        if (toActivityImpl == null) {
            //如果退回目的地活动不存在，则属于流程定义错误。在这里处理之
            String msg = "In " + processDefinitionId + " no " + m_rollbackTo;
            Log.getLog(this.getClass().getName()).error(msg);
            throw new Exception(msg);
        }
        //获取当前的活动定义对象
        ActivityImpl fromActivityImpl=((ExecutionImpl)execution).getActivity();
        //在这里建立退回的转移路径
        TransitionImpl transition = fromActivityImpl.createOutgoingTransition();
        transition.setName(fromActivityImpl.getName() + " to " + m_rollbackTo);
        transition.setDestination(toActivityImpl);
    }
}
```

最后，编写退回任务的服务 API。以下是一个最简单的实现：

```
public class TaskRollbackService {
```

```

//先获取流程引擎和任务服务。如果采用 jBPM4 的命令模式去实现，则会更方便和受到事务
控制
private static final ProcessEngine processEngine = Configuration.
getProcessEngine();
private static final TaskService taskService = processEngine.
getTaskService();
//这个方法用来处理客户端要求退回任务的请求
public void completeTaskRollback(String taskId, String rollbackToActName) {
    Task task = taskService.getTask(taskId);
    //调用 TaskService 的 completeTask 方法完成任务
    taskService.completeTask(task.getId(), task.getActivityName() + "
to " + rollbackToActName);
    //下面可以考虑清除历史痕迹、补偿业务损失……
}
}

```

对 jBPM4 的扩展工作到此为止。以下单元测试代码可以验证此退回流程：

```

//发起流程实例
ProcessInstance processInstance = executionService.
startProcessInstanceByKey("Rollback");
final String pid = processInstance.getId();
//正常办理第一个任务
final Task taskOutlayApply = taskService.findPersonalTasks("Jack").
get(0);
taskService.completeTask(taskOutlayApply.getId());
//断言到达第二个任务
processInstance = executionService.findProcessInstanceById(pid);
assertTrue(processInstance.isActive("leader audit"));
//“领导审批”未通过，调用退回任务服务API
Task taskLeaderAudit = taskService.findPersonalTasks("Alex").
get(0);
TaskRollbackService taskRollbackService = new TaskRollbackService();
taskRollbackService.completeTaskRollback(taskLeaderAudit.getId(),
"outlay apply");
//断言已经退回到“费用申请”活动
processInstance = executionService.findProcessInstanceById(pid);
assertTrue(processInstance.isActive("outlay apply"));
//重新办理费用申请
Task taskOutlayApply2 = taskService.findPersonalTasks("Jack").
get(0);
taskService.completeTask(taskOutlayApply2.getId());

```



```

        //领导审批通过
        Task taskLeaderAudit2 = taskService.findPersonalTasks("Alex").
get(0);
        taskService.completeTask(taskLeaderAudit2.getId());
        //断言流程实例已经结束
        assertProcessInstanceEnded(processInstance);
        //断言流程实例已经成为历史
        HistoryProcessInstance hProcInst = historyService

        .createHistoryProcessInstanceQuery().processInstanceId(pid).uniqueRe
sult();
        assertNotNull(hProcInst);

```

事实上，以上解决方案还有很多可以优化的地方，例如退回任务的 API 方法 `completeTaskRollback` 需要传入退回目的地的活动名称作为参数：

```

        taskRollbackService.completeTaskRollback(taskLeaderAudit.getId(),
"outlay apply");

```

由于退回目的地已经在流程定义中指定，这个参数完全可以不需要（需要修改 `RollbackListener`），简化客户端的调用。但是如果存在多个退回转移路径的情况呢？这时候就必须要求客户端传入一个参数，指明具体的退回目的地了。所以，该怎么实现，完全取决于您的业务需求，这也是 jBPM4 令人着迷的灵活扩展优势。

另外，如果基于 jBPM4 提供的命令模型去实现 `completeTaskRollback` 的逻辑，将会受到事务控制，这在要加入处理清除历史痕迹和业务补偿的逻辑时尤为重要。自定义命令可以如此使用：

```

        //使用 Configuration.getProcessEngine() 获取流程引擎对象执行自定义命令
是规范的做法
        Configuration.getProcessEngine().execute(new Command<Void>() {
            @Override
            public Void execute(Environment environment) throws Exception {
                //流程引擎会为自定义命令注入流程环境对象（environment），通过这个流程环
境对象可以获取到任何引擎服务
                HistoryService historyService = environment.get
(HistoryService.class);
                ...
            }
        });

```

20.2 取回

取回（Withdraw）又名“撤销”，是针对当前用户已办任务的操作，即将已办的历史任务重新置回当前待办状态。

为什么会有取回这种需求？

假设有如下串行的简单流程定义：

活动 1 → 任务活动 2（王二） → 任务活动 3（李三、张三） → 活动 4

- 任务活动 2 的办理人王二提交任务后，则流程到达了任务活动 3，这时任务办理人李三就会收到一个待办任务，但在李三还没有办理之前，王二突然从历史任务里发现，自己填写的任务表单有错误（或者粘贴了错误的附件），这时王二需要将李三的待办任务撤销，取回自己已经成为历史的任务重新办理后再交给李三。
- 如果任务活动 3 有两个候选人——李三和张三，那么王二则需要在办理任务时根据业务需要选择下一步任务是提交给李三还是张三办理。但由于王二的判断失误，把本应该提交给张三办理的任务错误地提交给了李三，那么此时，在李三办理任务之前，王二需要将提交给李三的任务撤销，取回后重新提交给张三办理。

可以想象，当用户办理完一个任务后，发现自己办理的任务有业务错误或需要等待一段时间后再提交，而此时此任务已经进入历史状态，新的任务已经在下一步办理者的待办列表中……在这种情况下，当前用户从自己的历史任务列表中“取回”某些任务重新办理的需求是合理的。

但需要注意的是，如果下一步办理者已经办理完成了任务，或流程已经流转了若干步骤（甚至还可能走了分支），那么，此时执行取回操作的业务损失就比较大了，是否值得，是一个要权衡的问题。

在 jBPM4 架构下，假设只能取回到上 1 步活动，实现这种“取回”需求的思路如下：

- 1) 为取回的目的地活动设计一个“取回监听器”，使此活动具有从下一步活动转移回来的路径，即赋予此活动的任务具有被取回的能力。
- 2) 遵循 jBPM4 的命令设计模式，基于用户的历史任务设计一个“取回命令”，

在这个命令中：结束当前活动的所有任务并清除当前活动的历史痕迹，然后转移到取回目的地活动。在这里我们必须使用 jBPM4 命令设计模式，因为这涉及两步数据库写操作，一定要受到事务控制。

- 3) 如果取回操作引发了重要业务的“损失”，则需要在取回命令中“补偿”之。当然，如果业务损失太严重（例如下一步任务已经开始办理了），则可以在取回命令中“拒绝”。

具体的实现过程如下。首先是支持取回的流程定义，如图 20-2 所示。

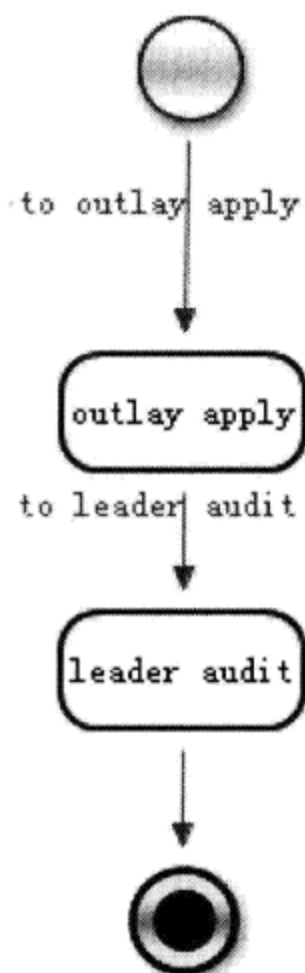


图 20-2 取回场景的流程定义

对应的 jPDL 如下：

```
<process name="Withdraw" xmlns="http://jbpm.org/4.3/jpdl">
  <start name="start">
    <transition name="to outlay apply" to="outlay apply" />
  </start>
  <task assignee="Jack" name="outlay apply">
    <!-- 在取回目的地活动的流出转移中设置“取回监听器”，使得此活动的历史任务具有被取回的能力 -->
  </task>
  <task assignee="Jack" name="leader audit">
  </task>
</process>
```

```

        <transition name="to leader audit" to="leader audit">
            <event-listener
class="com.examples.jbpm4.n3_cn_withdraw.WithdrawListener" />
        </transition>
    </task>
    <!-- 当流程运行到此活动时，任务可以被取回 -->
    <task assignee="Alex" name="leader audit">
        <transition to="end" />
    </task>
    <end name="end" />
</process>

```

取回监听器 WithdrawListener 的实现代码如下：

```

public class WithdrawListener implements EventListener {
    /**
     * 这个监听器的唯一目标就是：动态创建一条用于取回的路径。
     */
    @Override
    public void notify(EventListenerExecution execution) throws Exception
    {
        TransitionImpl transition = ((ExecutionImpl) execution).getTransition();
        //获得取回发生点活动定义的对象
        ActivityImpl withdrawFrom = transition.getDestination();
        //获得取回目的地活动定义的对象
        ActivityImpl withdrawBackto = transition.getSource();
        //在这里做一个简单的取回合理性判断。在实际应用中，应该加入更多的判断逻辑
        if (withdrawFrom == null || withdrawBackto == null) {
            String msg = "Not support withdraw.";
            Log.getLog(this.getClass().getName()).error(msg);
            throw new Exception(msg);
        }
        //为取回动态创建一条转移路径
        TransitionImpl newTran = withdrawFrom.createOutgoingTransition();
        newTran.setName(withdrawFrom.getName() + " to " + withdrawBackto.
getName());
        newTran.setDestination(withdrawBackto);
    }
}

```

取回命令的实现代码如下：

```

//作为 jBPM4 的命令，首先必须要实现 Command 接口

```

```

public class WithdrawTaskCommand implements Command<Void> {
    private static Log logger = Log.getLog(WithdrawTaskCommand.class.
getName());
    //流程实例 ID 和取回目的地活动名称这两个成员域的值由命令构造方法注入
    private String pid;
    private String withdrawToActName;
    /**
     * 命令构造方法的两个参数 流程实例 ID 和 取回目的地活动名称,由客户端的历史任务列表
提供——这应该不难获取。
     * @param pid
     * @param withdrawToActName
     */
    public WithdrawTaskCommand(String pid, String withdrawToActName) {
        this.pid = pid;
        this.withdrawToActName = withdrawToActName;
    }
    @Override
    public Void execute(Environment environment) throws Exception {
        //从流程引擎环境中获取执行、任务、历史 3 种服务
        ExecutionService executionService = environment.get
(ExecutionService.class);
        TaskService taskService = environment.get(TaskService.class);
        HistoryService historyService = environment.get(HistoryService.class);
        //这里比较特殊,由于 jBPM4 没有清除历史数据的服务 API 直接提供,所以我们要获取
Hibernate Session 对象直接操作持久层
        Session session = environment.get(Session.class);
        //获取当前活动名称集合
        Execution exec = executionService.findExecutionById(pid);
        Set<String> actNames = exec.findActiveActivityNames();
        //在这里做取回的合法性判断:如果当前活动不唯一则不允许取回。在实际业务中,应该
加入更多的判断逻辑
        if (actNames == null || actNames.size() == 0) {
            String msg = "没有可以取回的活动";
            logger.error(msg);
            throw new Exception(msg);
        }
        if (actNames.size() > 1) {
            String msg = "存在多个活动的节点";
            logger.error(msg);
            throw new Exception(msg);
        }
        String actName = actNames.iterator().next();

```



```

String withdrawPath = actName + " to " + withdrawToActName;
//获取当前活动的任务
List<Task> tasks = taskService.createTaskQuery().processInstanceId(pid)
    .activityName(actName).list();
for (Task task : tasks) {
    //结束任务（即取回任务）。如果 completeTask API 执行失败，很可能是因为
    withdrawPath 不存在，这说明当前活动已非取回目的地活动的下一步了.....实际应用中可以据此规
    则做进一步的异常处理
    taskService.completeTask(task.getId(), withdrawPath);
}
//在这里清除当前活动的历史痕迹，即删除历史活动实例及其历史任务
HistoryActivityInstanceImpl hActInst = (HistoryActivityInstanceImpl)
historyService
    .createHistoryActivityInstanceQuery().activityName(actName)
        .executionId(pid).uniqueResult();
//使用 Hibernate Session 对象直接操作持久层
session.delete(hActInst);
return null;
}
}

```

对 jBPM4 的扩展工作到此为止。以下单元测试代码可以验证此取回流程：

```

//发起流程实例
ProcessInstance processInstance = executionService.
startProcessInstanceByKey("Withdraw");
String pid = processInstance.getId();
//首先正常完成第 1 任务，即完成费用申请活动
Task taskOutlayApply = taskService.findPersonalTasks("Jack").
get(0);
taskService.completeTask(taskOutlayApply.getId());
//断言流程实例已经执行到第 2 任务，即领导审批活动
processInstance = executionService.findProcessInstanceById(pid);
assertTrue(processInstance.isActive("leader audit"));
//在这里模拟来自客户端历史任务列表中的操作：取回任务——这个操作应该是由用户
Jack 发起的
Configuration.getProcessEngine().execute(new
WithdrawTaskCommand(pid, "outlay apply"));
//取回成功。验证历史痕迹已经消除
List<HistoryTask> hTasks = historyService.createHistoryTaskQuery().
assignee("Alex").list();
//断言任务的办理人 Alex 已无历史任务

```



```

        assertEquals(0, hTasks.size());
        List<HistoryActivityInstance> hActInsts = historyService

        .createHistoryActivityInstanceQuery().activityName("leader
audit").list();
        //断言领导审批活动的历史不存在
        assertEquals(0, hActInsts.size());
        processInstance = executionService.findProcessInstanceId(pid);
        //断言任务已经取回，即费用申请活动重新（被激活）成为当前活动
        assertTrue(processInstance.isActive("outlay apply"));
        //完成费用申请任务
        List<Task> taskOutlayApply2s = taskService.findPersonalTasks("Jack");
        assertEquals(1, taskOutlayApply2s.size());
        taskService.completeTask(taskOutlayApply2s.get(0).getId());
        //这次不取回。完成领导审批任务
        Task taskLeaderAudit = taskService.findPersonalTasks("Alex").get(0);
        taskService.completeTask(taskLeaderAudit.getId());
        //此时可断言流程实例已经结束
        assertProcessInstanceEnded(pid);

```

20.3 会签

会签，又称会审，也即流程中某个业务需要经过多人表决，并且根据表决意见的汇总结果，匹配设定的规则，决定流程的走向。会签是审批流程中常见的需求。

会签可以分为单步会签（只使用一个活动处理会签业务），以及多步会签（会签业务由多个活动组成）。

单步会签比较常见，也较为容易实现，主要的解决方案是在会签活动的“主任务”基础上动态创建若干子任务来实现。在 jBPM4 架构下，具体的实现思路是：

- 1) 编写专门用于会签活动的任务分配处理器（实现 AssignmentHandler 接口），在这个任务分配处理器中从流程变量获取参加会签的用户 ID，并为这些用户动态地创建“会签任务”对象。
- 2) 编写专门用于完成“会签任务”的命令，取代 TaskService.completeTask 方法。在这个命令中，需要设定会签的业务逻辑，例如以下 4 种情况。
 - a) 一票否决制——参加会签的用户中任何一个人不同意，会签活动就结束，即进入“会签否决”转移；若所有会签用户都同意，则进入“会签通过”

转移。

- b) 一票通过制——与一票否决制完全相反。
- c) 按比例否决制——等全部参加会签的用户提交任务后，根据他们提交的意见，按比例（例如“少数服从多数”原则、“20%否定”制度、“80%通过”制度等）决定是否进入“会签否决”转移。
- d) 意见收集制——这种场景比较简单，仅仅是收集全部参加会签用户的意见（一般是通过任务表单），即全部会签用户提交任务后，会签活动就结束。

而多步会签，则相对较为复杂，建议使用“动态创建子流程”的方法实现。对于更复杂的业务场景，例如将第三方业务系统（例如财务系统、其他工作流系统，甚至其他组织的系统等）接入会签，则可以考虑使用 jms 活动发送消息并监听第三方业务系统应答的方式，异步地实现会签需求。

下面以单步会签为例，依据“一票否决制”的逻辑，来说明如何实现会签这种业务场景。示例流程定义如图 20-3 所示。

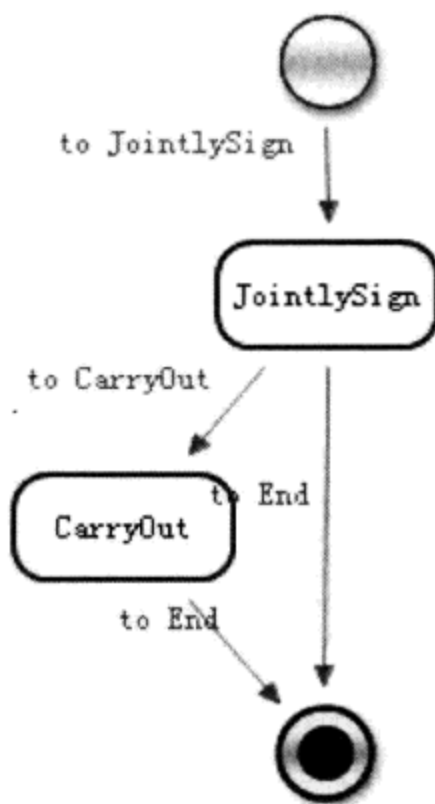


图 20-3 会签场景的流程定义

对应的 jPDL 如下：

```
<process name="JointlySign" xmlns="http://jbpm.org/4.3/jpdl">
  <start name="Start">
```

```

        <transition name="to JointlySign" to="JointlySign" />
    </start>
    <task name="JointlySign">
        <!-- JointlySignAssignment 是会签活动的任务分配处理器，这个处理器根据
        participants 成员域的定义的“会签参与者”动态创建出相应的子任务 -->
        <assignment-handler
        class="com.examples.jbpm4.n3_cn_jointlysign.JointlySignAssignment">
            <!-- 在这里定义参与会签的 3 个用户。当然，您可以赋予 participants 一个流
            程变量的值，这样就可以动态决定参与会签的人了，例如可以通过会签的上一步任务表单选定会签
            的用户。 -->
            <field name="participants">
                <list>
                    <string value="Alex" />
                    <string value="Jack" />
                    <string value="Vance" />
                </list>
            </field>
        </assignment-handler>
        <!-- 会签通过，则流向下面这个转移（执行会签决议）——此规则由客户端指定，当然
        您也可以考虑在配置中“约定”。 -->
        <transition name="to CarryOut" to="CarryOut" />
        <!-- 会签否决，则流向下面这个转移（流程结束）——此规则由客户端指定，同样您也
        可以考虑在配置中“约定”。 -->
        <transition name="to End" to="End" />
    </task>
    <!-- 执行会签决议 -->
    <state name="CarryOut">
        <transition name="to End" to="End" />
    </state>
    <!-- 流程结束 -->
    <end name="End" />
</process>

```

专门用于会签活动的任务处理器实现如下：

```

public class JointlySignAssignment implements AssignmentHandler {
    private final static ProcessEngine processEngine = Configuration.
getProcessEngine();
    //这里只需要使用任务服务的 API
    private final static TaskService taskService = processEngine.
getTaskService();
    //会签参与者 ID 列表——在流程定义 (jPDL) 中注入

```



```

    private List<String> participants;
    @Override
    public void assign(Assignable assignable, OpenExecution execution)
throws Exception {
        String pid = execution.getProcessInstance().getId();
        //获取会签活动的“主”任务对象，因为我们需要基于此主任务为所有会签参与者创建子
任务
        Task task = taskService.createTaskQuery().processInstanceId(pid)
            .activityName(execution.getName()).uniqueResult();
        //在这里创建会签子任务
        createSubTasks(task);
        return;
    }

```

如何创建会签子任务值得商榷。我们可以看如下 3 种实现方法。

1) 使用任务服务 API 基于主任务创建。

```

void createSubTasks(Task task) {
    if (participants != null) {
        for (String participant : participants) {
            //基于主任务为该会签参与者创建子任务
            Task subTask = taskService.newTask(task.getId());
            //设置该会签参与者为子任务的分配者
            subTask.setAssignee(participant);
            taskService.addTaskParticipatingUser(task.getId(),
participant, Participation.CLIENT);
        }
    }
}

```

这样做是不行的。因为 `TaskService.newTask` 方法会立即持久化子任务及其历史，而此时主任务的事务还未提交，因此在保存 `subTask` 时会出现主任务历史无法关联的异常。

2) 使用任务服务 API 脱离主任务直接创建。

```

void createSubTasks(Task task) {
    if (participants != null) {
        for (String participant : participants) {
            //为该会签参与者创建“孤立”的任务
            Task subTask = taskService.newTask();

```



```

        //设置该会签参与者为任务的分配者
        subTask.setAssignee(participant);
        taskService.addTaskParticipatingUser(task.getId(),
participant, Participation.CLIENT);
    }
}
}

```

这样做也是不行的。虽然使用 TaskService “凭空” 创建出的任务不会在持久化时出现异常，但是其与主任务无关联，因此会造成此会签子任务无归属，不受流程实例控制，成为“孤岛任务”，在后续的会签操作以及级联删除、历史分析时会存在重大隐患。

3) 使用主任务 Task 对象的 API 创建。

```

void createSubTasks(Task task) {
    if (participants != null) {
        for (String participant : participants) {
            //使用主任务对象的方法 createSubTask(不会触发持久化操作)为该会签
参与者创建子任务
            Task subTask = ((OpenTask) task).createSubTask();
            //设置该会签参与者为子任务的分配者
            subTask.setAssignee(participant);
            //以下操作的意义在于：关联会签用户到主任务。Participation.CLIENT
表示此用户是主任务的使用者之一，虽然不可以直接操作主任务，但可以对主任务添加注释，可以
用做将会签意见保存为主任务注释的目的。关于 Participation.CLIENT 的详尽解释，请参见附
录中的“参与者 (Participant)”，或 jBPM4 的 Java Doc 文档。
            taskService.addTaskParticipatingUser(task.getId(),
participant, Participation.CLIENT);
        }
    }
}

```

这样做既可以和主任务（及流程实例）关联，又可以随主任务一同持久化而避免异常，是创建会签子任务可取的实现方案。

基于“一票否决制”业务逻辑的会签任务提交命令，实现代码如下：

```

//首先，需要实现 jBPM4 的标准命令接口 Command
public class SubmitJoinsignTaskCmd implements Command<Boolean> {
    //这个常量用来标识传递会签意见的“任务变量”名称
    public static final String VAR_SIGN = "Sign";
    //当会签通过时，流向的转移名称。由客户端通过构造方法传入
}

```

```

private String transitionNamePass;
//当会签否决时，流向的转移名称。由客户端通过构造方法传入
private String transitionNameNoPass;
//主任务 ID。由客户端通过构造方法传入
private String parentTaskId;
//主任务对象。运行时计算得出
private Task parentTask;
//流程实例 ID。运行时计算得出
private String pid;
//会签任务对象。由客户端通过 setter 方法注入
private Task joinsignTask;
public void setJoinsignTask(Task joinsignTask) {
    this.joinsignTask = joinsignTask;
}
public String getPid() {
    return pid;
}
//命令的构造方法。从客户端收集一些关键数据
public SubmitJoinsignTaskCmd(String parentTaskId,
    String transitionNamePass, String transitionNameNoPass) {
    this.transitionNamePass = transitionNamePass;
    this.transitionNameNoPass = transitionNameNoPass;
    this.parentTaskId = parentTaskId;
}
@Override
public Boolean execute(Environment environment) throws Exception {
    //此处仅需使用到任务服务
    TaskService taskService = environment.get(TaskService.class);
    //计算出主任务和流程实例 ID
    this.parentTask = taskService.getTask(parentTaskId);
    this.pid = parentTask.getExecutionId();
    //获取当前的会签任务
    String joinsignTaskId = joinsignTask.getId();
    //从当前会签任务的任务变量中获取“会签意见”
    String sign = (String) taskService.getVariable(joinsignTaskId,
VAR_SIGN);
    //在这里设定一个规则：如果会签意见为“false”、“no”或“不同意”，则表示否决
    if (sign != null &&
        (sign.toLowerCase().equals("false") || sign.toLowerCase().equals("no")
|| sign.equals("不同意")))
    {
        //根据“一票否决制”的业务逻辑，一旦会签存在否定意见，则会签活动结束。具体实现
如下：

```

```

        //首先结束会签任务
        taskService.completeTask(joinsignTaskId);
        //然后, 为主任务增加一条会签意见的注释记录, 以备日后查询
        taskService.addTaskComment(parentTaskId, "User: "
            + joinsignTask.getAssignee() + ", Sign: " + sign);
        //最后, 结束主任务并流向预先定义的会签否决转移
        taskService.completeTask(parentTaskId, transitionNameNoPass);
        //返回 true 到客户端。表示会签活动结束
        return true;
    }
    //以下是会签任务为肯定意见时的处理逻辑:
    //完成会签任务
    taskService.completeTask(joinsignTaskId);
    //为主任务增加一条会签意见的注释记录
    taskService.addTaskComment(parentTaskId, "User: "
        + joinsignTask.getAssignee() + ", Sign: " + sign);
    //在这里判断, 是否还有会签“子任务”
    if (taskService.getSubTasks(parentTaskId).size() == 0) {
        //如已无会签“子任务”, 则表明会签全体同意, 会签活动结束, 需要:
        //结束主任务, 并流向会签通过转移
        taskService.completeTask(parentTaskId, transitionNamePass);
        //返回 true 到客户端。表示会签活动结束
        return true;
    }
    //返回 false 到客户端。表示会签活动未结束, 继续进行
    return false;
}
}

```

对 jBPM4 的扩展工作到此为止。以下单元测试代码可以验证此会签流程:

首先, 发起会签流程的实例:

```

//发起流程实例
ProcessInstance processInstance = executionService.startProcessInstanceByKey(
    "JointlySign");
String pid = processInstance.getId();
//获取会签活动的主任务
Task task = taskService.createTaskQuery().processInstanceId(pid)
    .activityName(processInstance.findActiveActivityNames().iterator()
        .next()).uniqueResult();

```



```

        List<Task> subTasks = taskService.getSubTasks(task.getId());
        //断言当前活动为会签活动
        assertTrue(processInstance.isActive("JointlySign"));
        //断言会签活动的主任务产生了 3 条子任务。这是在流程定义中设置的，由“会签任务
        处理器”负责生成
        assertEquals(3, subTasks.size());

```

下面我们可以模拟两种情况并验证。

第一种情况，会签被否决：

```

        //获取主任务 ID
        String taskId = task.getId();
        //构建会签任务提交命令。指定会签通过的转移和会签否决的转移，这两个参数可以由客
        户端提供
        SubmitJoinsignTaskCmd submitJoinsignTaskCmd =
        new SubmitJoinsignTaskCmd(taskId, "to CarryOut", "to End");
        //获取会签用户 Jack 的任务
        Task jackTask = taskService.findPersonalTasks("Jack").get(0);
        Map<String, Object> jackTaskVarMap = new HashMap<String, Object>();
        jackTaskVarMap.put(SubmitJoinsignTaskCmd.VAR_SIGN, "不同意");
        //模拟用户 Jack 否决会签，即在其任务中设置变量值“不同意”
        taskService.setVariables(jackTask.getId(), jackTaskVarMap);
        submitJoinsignTaskCmd.setJoinsignTask(jackTask);
        //提交会签任务。使用工作流引擎执行自定义的“会签任务提交”命令
        boolean result = Configuration.getProcessEngine().execute
        (submitJoinsignTaskCmd);
        //断言会签活动已经完成
        assertTrue(result);
        //断言流程实例已经结束。参见流程定义：会签否决的转移指向流程的结束活动
        assertProcessInstanceEnded(submitJoinsignTaskCmd.getPid());

```

第二种情况，会签通过（需要全体会签参与者提交任务，并且无否决意见）：

```

        //获取主任务 ID
        String taskId = task.getId();
        //构建会签任务提交命令
        SubmitJoinsignTaskCmd submitJoinsignTaskCmd =
        new SubmitJoinsignTaskCmd(taskId, "to CarryOut", "to End");
        //获取会签用户 Jack 的任务，不设置否决意见（即通过），并提交
        Task jackTask = taskService.findPersonalTasks("Jack").get(0);
        submitJoinsignTaskCmd.setJoinsignTask(jackTask);
        boolean result = Configuration.getProcessEngine().execute

```



```

(submitJoinsignTaskCmd);
    //断言会签活动未完成
    assertFalse(result);
    //获取会签用户 Alex 的任务, 不设置否决意见并提交
    Task alexTask = taskService.findPersonalTasks("Alex").get(0);
    submitJoinsignTaskCmd.setJoinsignTask(alexTask);
    result = Configuration.getProcessEngine().execute(submitJoinsignTaskCmd);
    //断言会签活动未完成
    assertFalse(result);
    //获取会签用户 Vance 的任务, 不设置否决意见并提交
    Task vanceTask = taskService.findPersonalTasks("Vance").get(0);
    submitJoinsignTaskCmd.setJoinsignTask(vanceTask);
    result = Configuration.getProcessEngine().execute(submitJoinsignTaskCmd);
    //此时 3 个“会签子任务”已经提交, 且皆无否决意见, 则可断言会签活动完成
    assertTrue(result);
    String pid = submitJoinsignTaskCmd.getPid();
    ProcessInstance processInstance = executionService.
findProcessInstanceById(pid);
    //因为会签通过, 则根据流程定义, 可断言当前活动为“CarryOut”
    assertTrue(processInstance.isActive("CarryOut"));
    //完成 CarryOut 活动
    String executionId = processInstance.findActiveExecutionIn
("CarryOut").getId();
    executionService.signalExecutionById(executionId);
    //断言流程实例已经结束
    assertProcessInstanceEnded(pid);

```

20.4 委派

委派, 又称代理, 是一种很常见的任务再分配机制。

在实际业务中, 经常会有这样的场景: 任务已经分配给 A 用户, 但由于某种原因 A 用户不方便办理, 需要“委派”给 B 用户 (也即代理人) 代为办理。这就是委派的业务场景。

解决委派问题有两种基本思路:

- 1) 不新创建任务。直接修改原始任务的分配人 (assignee) 属性, 即将“assignee = A”修改为“assignee = B”。但这样做会彻底断绝任务与原始分配人的关联

关系，因此还需要调用 TaskService 的 addTaskParticipatingUser 方法将任务的原始分配人作为一种特定的参与者类型与此任务建立关联；以及调用 TaskService 的 addTaskComment 方法为任务添加关于代理的注释，以备历史追溯之需。同时客户端还需要提供相应的“我委派的任务”列表，供收回委派需求使用。这种方式可以方便地支持任务被委派多次。

- 2) 新创建委派任务。在原始任务的基础上为委派创建子任务，供代理人用户办理。这需要注意同步原始任务和其子任务的业务数据，另外需要单独为委派任务的提交操作自定义一个“委派任务提交”命令，在这个命令中同时完成原始任务及其子任务。这种方式可以方便地支持任务被委派给多人办理。

下面按照思路 1 实现任务的委派需求。示例流程定义如图 20-4 所示。

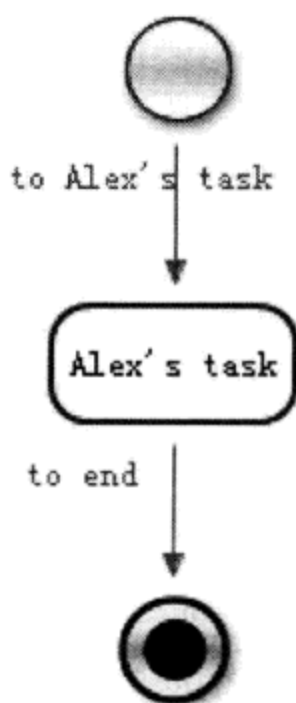


图 20-4 委派场景的流程定义

对应的 jPDL 如下：

```
<process name="Delegate" xmlns="http://jbpm.org/4.3/jpdl">
  <start name="start">
    <transition name="to Alex's task" to="Alex's task" />
  </start>
  <!-- 这个任务被定义分配给用户 Alex。在后面的单元测试中，我们将模拟客户端调用，将此任务委派给用户 Jack -->
  <task assignee="Alex" form="testTask.form" name="Alex's task">
    <transition name="to end" to="end" />
  </task>
</process>
```

```

</task>
<end name="end" />
</process>

```

现在，我们需要自定义一个供客户端调用的“任务委派”命令。实现代码如下：

```

//首先需要实现 Command 接口
public class TaskDelegateCmd implements Command<Void> {
    private String taskId;
    private String delegateUserId;
    //通过命令的构造方法，注入任务 ID 和被委派用户 ID
    public TaskDelegateCmd(String taskId, String delegateUserId) {
        this.taskId = taskId;
        this.delegateUserId = delegateUserId;
    }
    @Override
    public Void execute(Environment environment) throws Exception {
        //获取 TaskService 服务
        TaskService taskService = environment.get(TaskService.class);
        //获取需要委派的任务对象
        Task task = taskService.getTask(taskId);
        // Participation.REPLACED_ASSIGNEE 的官方解释：表示此用户是该任务原始的
        // 分配者，但由于缺席或其他某些原因被替换了。在这里将其设置为 REPLACED_ASSIGNEE 类型参与
        // 者保存的原因是，为了日后当此用户方便时将该任务“还”给他
        // 以下调用将任务的原分配者设置为 REPLACED_ASSIGNEE 类型的参与者保存
        taskService.addTaskParticipatingUser(taskId, task.getAssignee(),
            Participation.REPLACED_ASSIGNEE);
        //为任务增加一条注释，记录委派的明细，以备历史查询所用
        taskService.addTaskComment(taskId, "Delegate task from "
            + task.getAssignee() + " to " + delegateUserId);
        //将任务的分配者设置为委派用户
        task.setAssignee(delegateUserId);
        //保存任务对象的改动保存，持久化之
        taskService.saveTask(task);
        return null;
    }
}

```

至此，为实现委派而对 jBPM4 做的扩展工作已经完成。接下来，编写单元测试代码验证此实现：

```

//发起委派流程的实例

```



```

        ProcessInstance processInstance = executionService.
startProcessInstanceByKey("Delegate");
        String pid = processInstance.getId();
        //获取需要被委派的任务。根据定义，这个任务初始是被分配给用户 Alex 的
        Task oTask = taskService.findPersonalTasks("Alex").get(0);
        // 执行“任务委派”命令。将该任务委派给用户 Jack
        Configuration.getProcessEngine().execute(new
TaskDelegateCmd(oTask.getId(), "Jack"));
        //现在，用户 Jack 获取自己的委派任务
        Task delegateTask = taskService.findPersonalTasks("Jack").get(0);
        List<Participation> participations = taskService.getTaskParticipations
(delegateTask.getId());
        //断言该任务具有一个参与者，即任务的原始分配者
        assertEquals(1, participations.size());
        //断言该参与者的类型为 REPLACED_ASSIGNEE
        assertEquals(Participation.REPLACED_ASSIGNEE,
participations.get(0).getType());
        //(用户 Jack) 完成委派任务
        taskService.completeTask(delegateTask.getId());
        //根据流程定义，可断言此时流程实例已经结束
        assertProcessInstanceEnded(pid);
        //以下代码对流程历史进行验证：
        //获取历史任务查询接口
        HistoryTaskQuery historyTaskQuery = historyService.
createHistoryTaskQuery();
        //由于用户 Alex 已将任务委派给用户 Jack 办理，因此在历史查询中无用户 Alex 的历
史任务，只有用户 Jack 的历史任务。
        assertEquals(0, historyTaskQuery.assignee("Alex").list().size());
        assertEquals(1, historyTaskQuery.assignee("Jack").list().size());
        HistoryTask hTask = historyTaskQuery.assignee("Jack").uniqueResult();
        //获取该任务的注释对象
        HistoryComment comment = (HistoryComment) historyService.
createHistoryDetailQuery().
        taskId(hTask.getId()).uniqueResult();
        //断言此任务的注释如预期。证明是用户 Alex 委派给用户 Jack 办理的
        assertEquals("Delegate task from Alex to Jack", comment.getMessage());
        //获取此任务的参与者
        List<Participation> hParticipations = taskService.
getTaskParticipations(hTask.getId());
        //断言参与者数量为 0。即证明任务提交后，其对应的参与者信息即被删除，因此无法从
任务的参与者记录中追溯历史痕迹
        assertEquals(0, hParticipations.size());

```



```
return;
```

以上是解决委派问题第 1 种思路的全部实现过程。对于第 2 种思路，实现起来也不困难，只需重点关注以下两点：

- 1) 在“任务委派”命令中根据原始任务新创建一个子任务作为委派任务分配给代理用户。代码片段如下：

```
public Task execute(Environment environment) throws Exception {
    TaskService taskService = environment.get(TaskService.class);
    Task task = taskService.getTask(taskId);
    String delegateMsg = "Delegate task from " + task.getAssignee() +
" to " + delegateUserId;
    taskService.addTaskComment(taskId, delegateMsg);
    //在这里创建用于委派的子任务
    TaskImpl delegateTask = ((TaskImpl) task).createSubTask();
    //分配给代理用户
    delegateTask.setAssignee(delegateUserId);
    delegateTask.setDescription(delegateMsg);
    //以下操作将原始任务的关键数据（例如任务名称、任务表单）复制给委派任务使用
    delegateTask.setName(task.getName());
    delegateTask.setFormResourceName(task.getFormResourceName());
    // .....
    //分别保存原始任务和委派任务
    taskService.saveTask(task);
    taskService.saveTask(delegateTask);
    return delegateTask;
}
```

- 2) 需要提供一个“委派任务提交”命令，以同时完成委派任务和原始任务，结束活动实例。代码片段如下：

```
public Void execute(Environment environment) throws Exception {
    TaskService taskService = environment.get(TaskService.class);
    //根据委派任务获取原始任务
    TaskImpl delegateTask = (TaskImpl) taskService.getTask(delegateTaskId);
    Task oTask = delegateTask.getSuperTask();
    //分别完成委派任务和原始任务
    taskService.completeTask(delegateTask.getId());
    taskService.completeTask(oTask.getId());
    return null;
}
```

当然，对委派场景的实现并不能到此为止，在实际业务中，您很可能需要考虑如下延伸问题：

- “收回委派任务”命令的实现。
- 支持委派给多个用户，这又分多个候选者以及用户组。
-

作者认为，只要您掌握了上面两种思路的实现方法，再实现这些延伸问题应该是水到渠成的事情。

20.5 自由流

何谓自由流？就是流程运行过程中需要在原本并没有转移路径的活动之间进行“自由”流转。

根据作者的经验，在国内很多工作流应用的实施过程中都会或多或少地被自由流（或其相似需求）所困扰。对于一名深刻理解工作流管理技术理念的架构师来说，他一般不会推荐使用自由流的方式解决业务问题，因为既然使用了工作流架构，在流程定义阶段应该能识别业务活动之间的所有转移路径并加以连接，这样，流程才是透明的、规范的，是可以监控、管理和优化的。而如果在运行过程中，根据个人的意志（相对流程定义的“公共”约定），在单个流程实例中动态地决定和创造活动之间的转移路径，无疑这样的流程定义是无法有效监控和管理的，即很容易出现所谓的“拍脑袋决策”、“特殊处理”甚至“暗箱操作”，后续的“业务流程再造”和“业务流程优化”也会相当困难。

当然，在特殊的业务情况及客户需求下，对“僵化的流程”进行人为的干预，自由地选择和创建流转的路径，以便“低成本”地使得业务流程具有一定的灵活性与自主性，采用“自由流”的方式也是一种变通的方法。

下面是一个简单的自由流应用场景示意图，如图 20-5 所示。

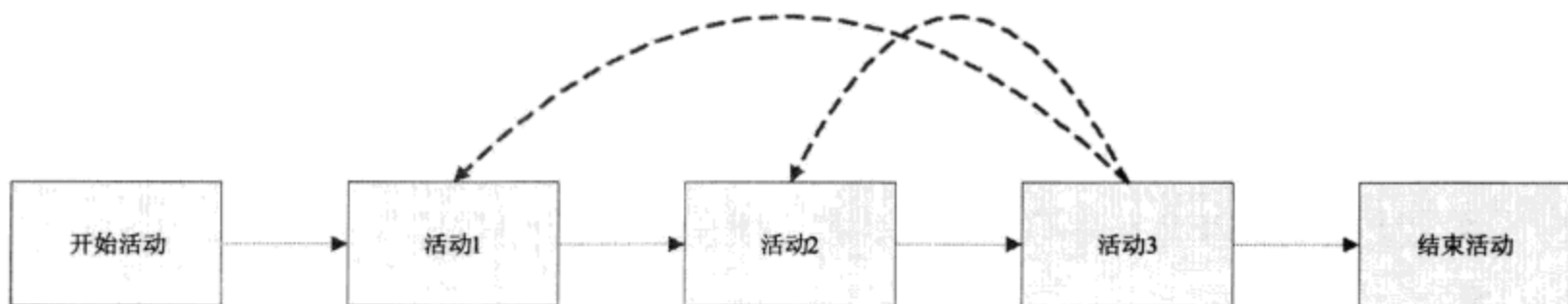


图 20-5 自由流应用场景示意图

假设有上述串行流程定义。根据定义，活动 3 到活动 2、活动 3 到活动 1 之间是没有转移路径的，但是如果有一类“特殊”的用户需要这么一种特权——可以在活动 3 中将流程的执行直接向后跳转，是到活动 2 还是到活动 3（或是到其他什么地方）并不确定，那么我们就需要开发出这么一种功能，使活动 3 具有向活动 1、活动 2 乃至任意活动转移的路径，且设定为最高优先级（如果需要的话）。

事实上，如何在 jBPM4 架构下实现自由流在 20.1 退回和 20.2 取回章节中已有答案。参照取回的实现，我们扩展实现自由流的功能可按照以下步骤进行：

- 1) 对客户端应用暴露一个“执行自由流”功能的操作，并要求客户端提供一个“转移目的地活动名称（destActName）”作为参数。
- 2) 实现 Command 接口，自定义一个“自由转移（FreeTransitionCmd）”命令。在调用时为这个自由转移命令注入 destActName 参数。然后自由转移命令使用 destActName 作为目的地活动名称，使用当前活动名称作为源活动名称，动态地创建出一条自由转移路径，在完成当前活动/任务后通过此转移。

需要注意的是，无论如何实现自由流，都要考虑对业务数据造成的影响或损失，然后根据需要设计一定的机制进行“补偿”。

20.6 小结

其实所谓“中国特色工作流”就是在国内比较常见的工作流业务场景，本章介绍的 5 种场景也是 jBPM 在实际应用中面临的最为常见的挑战。另外，中国特色工作流的应用场景还可以列出如下几种：

- 自由分配任务
 - 例如，由领导在上一步骤的任务指定下一步骤的任务办理者。对于 jBPM4 来说这很简单，提供一个修改任务分配者操作接口给客户端应用即可，参见“委派”。但这会在一定程度上固化某些功能，使得客户端应用丧失一定的通用性。
- 催办
 - 例如，任务活动在一段时间内得不到响应，需要系统通过某种方式催促办理人办理。解决方案是时间设定可以交由 timer（定时器，参见相关章节）元素来处理，催促手段可以交给任务提醒邮件或 mail 活动、jms 活动去实现，即使需要短信等方式催促，通过编写 timer 事件处理器调用第三方的短信发送接口来实现也不难。
- 任意消除历史痕迹
 - 典型的“毁尸灭迹”类需求，客户一般会以变通的方式提出，因此需求分析师和开发者务必领会其真实目的。但 jBPM4 本身提供的 HistoryService 完全不支持任何对流程历史记录的写操作 API，因此我们需要获取 Hibernate 的 Session 对象直接操作持久化层，删除需要“抹去”的历史记录，这种方法的实现请参见“取回”。删除流程历史记录并非 jBPM 设计者的初衷，因此删除后可能会引发“历史流程追溯”的一些异常，开发者需要做相应的考虑和处理。

在这里请允许我讲个故事作为本书常规章节的结束：

某企业实施 workflow 管理系统后……

高层：“恩，这个 workflow 管理很重要，体现了企业管理在向透明化和流程化的规范道路上前进。”

中层：“为了更好地管理和监控，请在流程上加入我这个步骤，另外我还有些需求要提……”

工作人员：“无所谓，走流程也好，省得得罪人。”或者“这样一来，我只是流程中的一个环节啊……”

最后是开发人员：“客户的需求很 BT。”

上面这个故事也许并非典型，仅供放松。

最后，希望本章的内容能起到抛砖引玉的效果，读者能想到更多更好的扩展 jBPM 的方法，并用来帮助客户解决他们的**真正需求**，从而实现个人、企业与客户的双赢！

业务流程 (Business Process)

“一系列结构化的、可度量的活动，设计它的目标是为特定客户或市场产生规定的输出。”——Davenport (1993)

“一种活动的集合，具有一种或多种输入和确定的输出，这些输出对客户产生价值。”——Hammer 与 Champy (1993)

“业务流程是为产生产品或服务而设计的一系列步骤。多数的流程 (……) 跨越职能，贯穿组织机构图上矩形之间的空白。一些流程的结果是由组织外的客户所接受的产品或服务，称为主要流程；另一些流程的产出不为外部客户所见，但它是有效管理所必需的，称为支持流程。”——Rummler 与 Brache (1995)

“互相连接的活动集合，它们将输入转换为输出。理想情况下，在流程中发生的转换将为输入增加价值，并形成对接受者更有效用的输出，无论接受者处于上游还下游。”——Johansson 等 (1999)

综上所述：业务流程是为实现特定客户或市场的生产、服务等行为而设计的一组相互关联的活动或过程，业务流程通常在具有功能角色或关系的组织结构内进行。

jBPM 流程定义语言 (jBPM Process Define Language, jPDL)

jPDL 是一种拥有出色建模能力、整洁 Java 接口和强大任务管理功能的可执行业务流程定义语言。jPDL 能很好地支持业务分析人员与软件开发人员的合作。首先业务分析人员通过基于图形表达的流程设计器可视化地定义业务流程；然后在不改变流程定义图型的情况下，开发人员通过一系列用户代码机制将技术实现与业务过程绑定。这样就实现了业务分析模型到应用实现的转变。

在第 6 章 掌握 jBPM 流程定义语言中会详细介绍 jPDL 的内容。

业务流程执行语言 (Business Process Execution Language, BPEL)

BPEL (发音为 'bipple' 或 'bee-pell') 是一种基于 XML 的、用来描述业务过程的编程语言，其描述的业务过程的每个单一步骤一般由 Web Service 技术来实现。

2002 年 IBM, BEA 和微软等公司一起开发和引入了 BPEL 作为描述和编排 Web Service 的语言。通过 BPEL 可以描述和编排参加 Web 服务过程的 Web Service 的接口，例如信息需要按照怎样的顺序被输入。

jBPM 的 BPEL 运行引擎与 WS-BPEL 2.0 标准和 BPEL4WS 1.1 标准兼容。因此它可以执行 Eclipse BPEL Designer 或其他遵循 BPEL 标准的图形化流程设计器所定义的流程。

在 9.1.2 扶择，是否使用 BPEL 中会详细分析 BPEL 之于业务流程设计。

图形化流程设计器 (Graph Process Designer, GPD)

jBPM 提供的一个支持图形化“编程”的流程定义设计工具。它是一组 Eclipse 插件，用来可视化地支持业务流程的定义、测试、发布（在 jBPM4 中，图形化流程设计器取消了发布功能）和监控。

在第 3 章 使用 jBPM 图形化流程设计器设计流程中会详细介绍如何使用 GPD 及其相关工具定义流程。

Web 控制台 (Web Console)

jBPM 的每个发行版本都包括一个基于浏览器的管理平台。它是一个标准的 Java EE 应用程序，可以实时地控制和监视当前 jBPM 系统内的流程定义部署情况和流程实例运行情况，这会细致到流程变量的值、流程实体的状态、用户的任务列表等信息。在浏览器中，使用 Web 控制台可以实时地调度流程实例，管理不同版本的流程定义，以及分析流程运行数据（jBPM4 Web 控制台新加入的功能）。

工作流引擎 (Workflow Engine)

工作流引擎主要用来解释流程定义和为流程实例提供运行环境，并且对外（客户端应用）提供执行流程的 API。工作流引擎是整个工作流管理系统的核心。

在 1.2.1 工作流管理系统参考模型中会详细介绍工作流引擎及其相关组件的定义。

流程虚拟机 (Process Virtual Machine, PVM)

PVM 是 jBPM4 提出的一个重要概念。PVM 为构建流程定义图形和执行流程实例提供 Java 库和 API。PVM 可以消除各种不同类型工作流管理系统和流程定义语言的差异，只要遵循它的规范，实现它的接口。因此在 PVM 的基础上可以“创造”出一种流程定义语言。jPDL4 就是 jBPM4 基于 PVM 的默认实现。

在第 12 章 流程虚拟机原理中会详细介绍 PVM 的设计理念。

流程定义 (Process Definition / Process Model)

流程定义又称流程模型，是用来描述业务过程的规定性文档。一条流程定义主要由一系列活动定义和转移组成。流程定义需要遵循特定的语法规则，一般流程定义以 XML 文档的形式保存。jBPM4 流程定义示意如图 A-1 所示。

流程定义由工作流引擎负责解释执行。

jBPM 的流程定义需要遵循 jPDL 语法规则（由 <http://jbpm.org/4.x/jpdl> 的 XML Schema 约束），文件需要遵循 *.jpdl.xml 的名称规范。

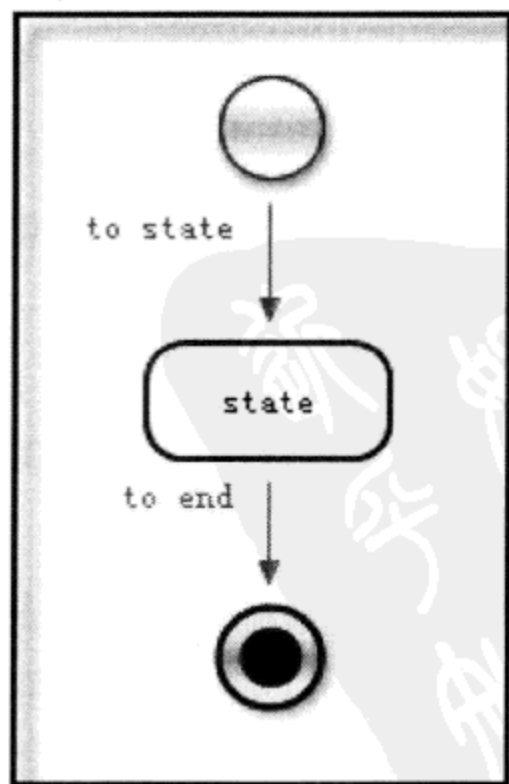


图 A-1 jBPM4 流程定义示意图

活动定义 (Activity Definition)

活动定义是最小的业务描述单位，是流程定义的重要组成部分。我们说一条业务流程一般由若干个业务步骤组成，则活动定义就是这里所说的业务步骤。jBPM4 活动定义示意如图 A-2 所示。

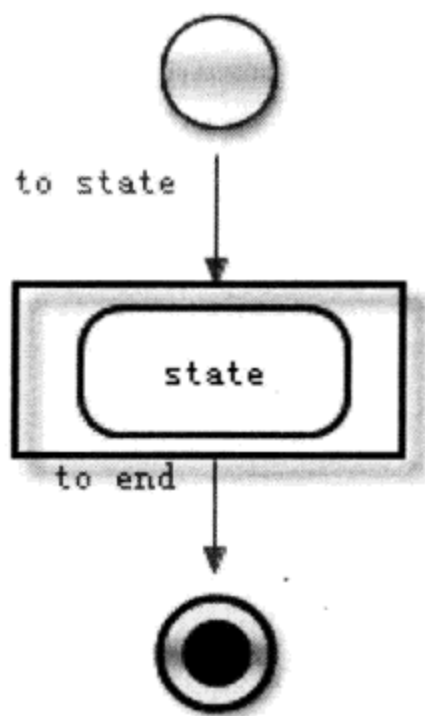


图 A-2 jBPM4 活动定义示意图

活动分为人工活动（需要人工干预完成）和自动活动（由系统自动执行完成）。任务活动可以产生任务，并由此任务的分配者（Assignee）办理、提交，从而完成任务，结束活动。

转移 (Transition)

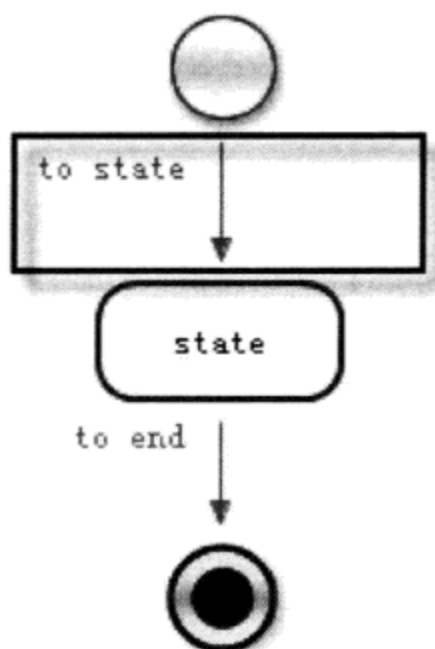
转移负责将各个活动定义连接起来，组成流程定义。

对于活动定义来说，转移可分为流入转移（Incoming Transition）和流出转移（Outgoing Transition）。

流入转移 (Incoming Transition)

流入转移的概念是针对特定的活动定义而言的。

由其他活动流出，（图形上的箭头）指向特定活动（如图 A-3 中的 state 活动）的转移，被称为流入转移。



A-3 jBPM4 流入转移示意图

描述流入转移的 XML 片段在其他活动中定义。

流出转移 (Outgoing Transition)

流出转移的概念是针对特定的活动定义而言的。

由特定活动（如图 A-4 中的 state 活动）流出，（图形上的箭头）指向其他活动的转移，被称为流出转移。

描述流出转移的 XML 片段需要在特定活动中定义。

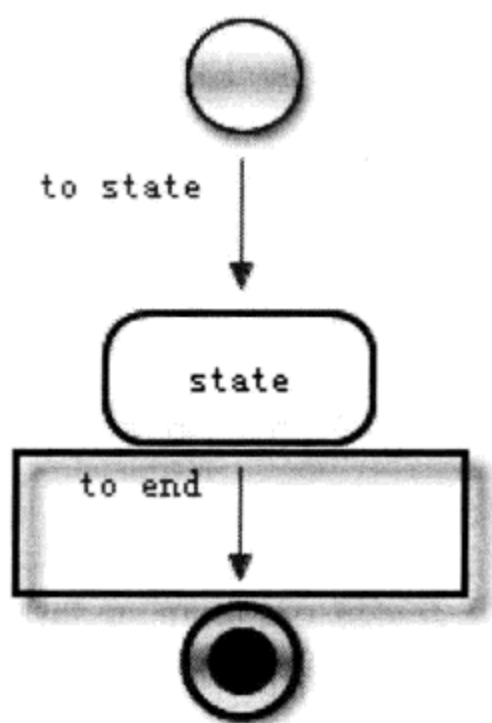


图 A-4 jBPM4 流出转移示意图

流程实例 (Process Instance)

流程实例是在流程运行时根据流程定义产生的实体，是实例化的流程定义。我们说一条流程执行完毕，意思也就是流程实例的生命周期结束。

jBPM4 中处于运行状态的流程实例实现 `org.jbpm.api.ProcessInstance` 接口。已经完成的流程实例即为历史流程实例，实现 `org.jbpm.api.history.HistoryProcessInstance` 接口。

活动实例 (Activity Instance)

活动实例也是运行时的概念。当流程运行到相应活动的定义时，就会产生活动实例。流程实例是由一系列其生命周期中经过的活动实例所组成的。

任务 (Task)

任务也是运行时的概念。任务在任务活动中定义，随任务活动的实例产生而产生，用来支持人工干预流程的操作和记录。例如当需要一个人填写表单，为业务流程提供数据时，就会产生任务。

任务一般由 workflow 客户端应用程序（例如 Web 控制台、任务列表等）推送给用户进行办理。任务的生命周期在其被用户提交后结束。

jBPM4 中处于运行状态的任务实现 `org.jbpm.api.task.Task` 接口。已经结束的任务即为历史任务，实现 `org.jbpm.api.history.HistoryTask` 接口。

分配者 (Assignee)

分配者是任务的实际办理人，一条任务最多只能有一个分配者。当用户是任务的分配者时，才能办理任务。

在 jBPM4 的持久化设计中，分配者的用户 ID 作为任务实体数据表的一个列，被保存在数据库中。

候选者 (Candidate User)

候选者是 jBPM4 已经支持的一种任务参与者类型，1 条任务可以有 0 个或多个候选者。候选者被规定为可以浏览任务，但无法办理和提交任务。如果一个候选者想要办理任务，他必须先成为任务的分配者（一般是通过客户端调用 `TaskService.assignTask` 方法实现）。

在 jBPM4 的持久化设计中，候选者的 ID 被保存在参与者实体数据表中，通过任务 ID 与任务实体数据表关联。

参与者 (Participant)

除任务分配者外，与任务有关的用户、用户组都被称为任务的参与者，这包括候选者。

目前，jBPM4 支持的任务参与者类型如下：

- Participation.CANDIDATE——候选者。只能查看该任务并发表注释。
- Participation.OWNER——所有者。可以对该任务做任何事。
- Participation.CLIENT——客户。被定义为该任务结果的受用者，只能查看该任务并发表注释。
- Participation.VIEWER——查看者。顾名思义，只能查看该任务。
- Participation.REPLACED_ASSIGNEE ——被取代的分配者。表示此用户曾经是任务的分配者，但是出于某种原因，现在被替代了，因此被记录在这。这种参与者类型是为了能够追溯任务的过往分配者，以便帮助其重新“夺回”任务而设计的。在 20.4 委派中有涉及到此类型参与者的扩展应用。

事件监听器 (Event Listener)

事件-监听器模式是流程定义时的概念，由触发条件（即流程事件）和触发操作（即监听处理器，需要实现 `org.jbpm.api.listener.EventListener` 接口）组成。用来支持将用户自定义的业务逻辑代码“嵌入”流程生命周期的特定阶段，从而达到扩展流程定义功能的目的。

详细使用方法请参见 6.4 事件章节。

定时器 (Timer)

定时器是流程定义时的概念，用来支持在预先定义的时间点或等待指定的时间段后执行的工作（Job）。定时器的触发可以是一次性的，也可以是周期性的。所谓周期性的周期范围可以是流程实例的整个生命周期过程或是某一等待活动的等待时间内。

详细使用方法请参见 10.1 TIMER (定时器) 能为您做什么章节。

工作 (Job)

工作 (Job) 是与异步执行有关的概念。工作是由定时器或异步活动产生的一组流程操作, jBPM4 workflow 引擎提供一个可配置的工作执行器 (job-executor), 在需要的时候自动执行工作。当然工作也可以通过 JMS 消息、手工编程调用等方式被触发执行。

关于工作的应用, 您可以在 6.5 异步执行和 10.1 TIMER (定时器) 能为您做什么中详细了解。

