

京东购买地址：<https://item.jd.com/12246565.html>

扫描关注作者公众号，免费获取多种技术的教学视频以及配套代码：



扫码获取视频、电子书



目录

第1章 Activiti 介绍.....	4
1.1 工作流介绍.....	5
1.2 BPMN2.0 规范简述.....	5
1.2.1 BPMN2.0 概述.....	6
1.2.2 BPMN2.0 元素.....	6
1.2.3 BPMN2.0 的 XML 结构.....	8
1.3 Activiti 介绍.....	8
1.3.1 Activiti 的出现.....	8
1.3.2 Activiti 的发展.....	9
1.3.3 选择 Activiti 还是 jBPM.....	9
1.4 本章小结.....	9
2 安装与运行 Activiti.....	9
2.1 下载与运行 Activiti.....	10
2.1.1 下载和安装 JDK.....	10
2.1.2 下载和安装 MySQL.....	11
2.1.3 下载和安装 Activiti.....	12
2.2 运行官方的 Activiti 示例.....	13
2.2.1 本小节流程概述.....	14
2.2.2 新建用户.....	14
2.2.3 定义流程.....	16
2.2.4 发布流程.....	19
2.2.5 启动与完成流程.....	20
2.2.6 流程引擎管理.....	23
3 Activiti 开发环境搭建.....	24
3.1 安装开发环境.....	24
3.1.1 下载 Eclipse.....	25
3.1.2 安装 Activiti 插件.....	25
3.2 编写第一个 Activiti 程序.....	26
3.2.1 如何运行本书案例.....	27
3.2.2 建立工程环境.....	27
3.2.3 创建配置文件.....	28
3.2.4 创建流程文件.....	29
3.2.5 加载流程文件与启动流程.....	30
3.3 小结.....	31
4 配置文件读取与数据源配置.....	31
4.1 流程引擎配置对象.....	32
4.1.1 读取默认的配置.....	32
4.1.2 读取自定义的配置.....	33
4.1.3 读取输入流的配置.....	33
4.1.4 使用 createStandaloneInMemProcessEngineConfiguration 方法.....	34
4.1.5 使用 createStandaloneProcessEngineConfiguration 方法.....	34
4.2 数据源配置.....	35

4.2.1	Activiti 支持的数据库.....	35
4.2.2	Activiti 与 Spring.....	35
4.2.3	JDBC 配置.....	36
4.2.4	DBCP 数据源配置.....	36
4.2.5	C3P0 数据源配置.....	37
4.2.6	Activiti 其他数据源配置.....	38
4.2.7	数据库策略配置.....	38
4.2.8	databaseType 配置.....	40
5	流程引擎的创建.....	41
5.1	ProcessEngineConfiguration 的 buildProcessEngine 方法.....	41
5.2	ProcessEngines 对象.....	41
5.2.1	init 与 getDefaultProcessEngine 方法.....	42
5.2.2	registerProcessEngine 和 unregister 方法.....	42
5.2.3	retry 方法.....	43
5.2.4	destroy 方法.....	43
5.3	ProcessEngine 对象.....	44
5.3.1	服务组件.....	44
5.3.2	关闭流程引擎.....	45
5.3.3	流程引擎名称.....	46
5.4	本章小节.....	47
6	邮件服务器与 history 配置.....	47
6.1	history 配置.....	47
6.2	邮件服务器配置.....	48
7	Activiti 的设计模式.....	48
7.1	Activiti 的命令拦截器.....	49
7.1.1	命令模式.....	49
7.1.2	责任链模式.....	51
7.1.3	编写自定义拦截器.....	53
8	Activiti 数据查询（一）.....	55
8.1	Activiti 数据查询.....	55
8.1.1	查询对象.....	55
8.1.2	list 方法.....	56
8.1.3	listPage 方法.....	57
8.1.4	count 方法.....	57
9	Activiti 数据查询.....	58
9.1	排序方法.....	58
9.2	ID 排序问题.....	60
9.3	多字段排序.....	62
9.4	singleResult 方法.....	63
9.5	用户组数据查询.....	64
9.6	原生 SQL 查询.....	65
10	特别子流程.....	66
	特别子流程.....	67
11	流程控制逻辑.....	69

11.1 概述.....	69
11.2 设计流程对象.....	69
11.3 创建流程节点行为.....	71
11.4 编写业务处理类.....	72
11.5 流程 XML 转换为 Java 对象.....	73
11.6 编写客户端代码.....	74
12 DMN 规范概述.....	75
DMN 的出现背景.....	75
Activiti 与 Drools.....	76
DMN 的 XML 样例.....	76
13 DMN 的 XML 规范.....	77
决策.....	77
决策表.....	78
输入参数.....	78
输出结果.....	79
规则.....	79
14 Activiti 运行第一个 DMN 应用.....	80
建立项目.....	81
规则引擎配置文件.....	81
编写 DMN 文件.....	82
加载与运行 DMN 文件.....	83

第 1 章 Activiti 介绍

在计算机尚未普及时，许多工作流程采用手工传递纸张表单的方式，一级一级审批签字，工作效率非常低下，对于数据统计以及生成报表的功能，需要经过大量的手工操作才能实现。随着电脑的普及，这些工作的参与者只需要在电脑的系统中填入工作内容，系统就会按照定义好的流程自动执行，各级审批者可以得到工作的信息并作出相应的审批和管理操作，数据统计和报表的生成均由系统代为完成，这样大大提高了工作效率，在这种背景下，各种的工作流应用以及中间件应运而生。

工作流应用在日常工作中的应用越来越广泛，JavaEE 领域出现了许多优秀的工作流引擎，例如 JBoss 社区的 jBPM、OpenSymphony 的 OSWorkflow 等，在 2010 年 5 月 17 日，以 Tom Baeyens 为首的工作流小组发布了一个全新的工作流引擎——Activiti，该工作流引擎的第一个版本为 5.0alpha1，由于 Tom Baeyens 是 jBPM 的创始人（由于意见分歧离开 JBoss），因此 Activiti 的团队希望该流程引擎是 jBPM4 的延伸，希望在 jBPM 中积累的经验知识的基础上，继续进行新一代工作流解决方案的建设，因此将第一个 Activiti 版本定义为 5.0alpha1。

Activiti 经过多年的发展，已经发布了多个版本，随着 DMN（决策模型与图形）规范的推出，Activiti 开始实现自己的规则引擎，本书将以 Activiti6.0 为基础，深入了解 Activiti 工作流引擎以及规则引擎的特性。

1.1 工作流介绍

工作流（Workflow），是对工作流程及其各操作步骤之间业务规则的抽象、概括、描述。工作流建模，即将工作流程中的工作如何前后组织在一起的逻辑和规则在计算机中以恰当的模型进行表示并对其实施计算。工作流要解决的主要问题是：为实现某个业务目标，在多个参与者之间，利用计算机，按某种预定规则自动传递文档、信息或者任务。工作流管理系统（Workflow Management System, WfMS）的主要功能是通过计算机技术的支持去定义、执行和管理工作流，协调工作流执行过程中工作之间以及群体成员之间的信息交互。工作流需要依靠工作流管理系统来实现。工作流属于计算机支持的协同工作（Computer Supported Cooperative Work, CSCW）的一部分。工作流管理系统是普遍地研究一个群体，如何在计算机的帮助下实现协同工作。注：本段内容来自维基百科。

早在 20 世纪 70 年代，办公自动化概念出现的时候，工作流思想就已经出现，人们希望新的技术可以改善办公效率，但是由于当时计算机并没有普及，网络技术还不普遍等原因，70 年代工作流技术仅仅停留在研究领域。到了 90 年代以后，各种的技术条件逐渐成熟，工作流技术被应用于电信、软件、制造、金融和办公自动化领域。随着工作流技术的兴起，为了给全部业务的参与者提供易于理解的标准标记法，由业务流程管理倡议组织（BPMI）开发出了“业务流程建模标记法”（BPMN, Business Process Modeling Notation），BPMI 组织于 2005 年并入 OMG 组织，当前 BPMN 规范由 OMG 组织进行维护。

1.2 BPMN2.0 规范简述

BPMN 规范 1.0 版本由 BPMI 组织于 2004 年发布，全称是 Business Process Modeling Notation，BPMN 规范的发布是为了让业务流程的全部参与人员对流程可以进行可视化管理，提供一套让所有参与人员都易于理解的语言和标记，为业务流程的设计人员（非技术人

员)和流程的实现人员(技术人员)建立起一座桥梁。BPMI 组织于 2005 合并到 OMG(Object Management Group) 组织中，2008 年 1 月发布 BPMN1.1 规范。BPMN2.0 规范于 2011 年 1 月正式发布，并且全称改为 Business Process Model And Notation（业务流程模型和符号）。

在 1.0 版本的 BPMN 规范中，只注重流程元素的图形，这使其在流程分析人员中非常受欢迎，而 BPMN2.0 版本则继承了 1.0 版本的内容，并且注重流程执行语法和标准交换格式。

1.2.1 BPMN2.0 概述

BPMN2.0 规范定义了业务流程的符号以及模型，并且为流程定义设定了转换格式，目的是为了流程的定义实现可移植性，那么用户可以在不同的供应商环境中定义流程，并且这些流程可以移植到其他遵守 BPMN2.0 规范的供应商环境中。BPMN2.0 在以下方面扩展了 BPMN1.2：

- ❑ 规范了流程元素的执行语法；
- ❑ 定义了流程模型和流程图的扩展机制；
- ❑ 细化了事件的组成；
- ❑ 扩展了参与者的交互定义；
- ❑ 定义了编排模型。

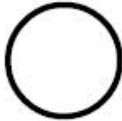

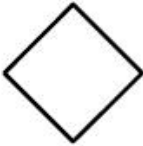








1.2.2 BPMN2.0 元素

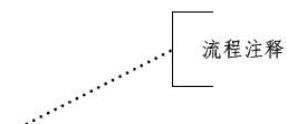
BPMN2.0 的目的是建立简单的并且易懂的业务流程模型，但是同时又需要处理高度复杂的业务流程，因此要解决这两个矛盾的要求，需要在规范中定义标准的图形和符号。BPMN 中定义了 5 类基础的元素分类：

- ❑ 流对象（Flow Objects）：在一个业务流程中，流对象是用于定义行为的图形元素，主要有事件（Events）、活动（Activities）和网关（Gateways）三种流对象。
- ❑ 数据（Data）：主要有数据对象（Data Objects）、数据输入（Data Inputs）、数据输出（Data Outputs）和数据存储（Data Stores）4 种元素。
- ❑ 连接对象（Connecting Objects）：用于连接流对象，主要有 4 种连接流对象的方式，包括顺序流（Sequence Flows）、消息流（Message Flows）、关联（Associations）和数据关联（Data Associations）。
- ❑ 泳道（Swimlanes）：泳道提供了有 2 种途径组织基础的模型元素，分别是池（Pools）和道（Lanes）。
- ❑ 制品（Artifacts）：制品主要用于为流程提供附加信息，当前制品包括组（Group）和注释（Text Annotation）。

以上的元素分类以及其下面的元素，均是 BPMN 规范中元素的组成部分，每个对象均有自己对应的图形，以下表格为各个元素的图形及其描述。

元素	图形	描述
----	----	----

事件 (Events)		用于描述流程中发生的事件，事件会对流程产生影响，事件会被触发或者会产生结果。
活动 (Activities)		活动是工作流中一个通用的术语，活动包括任务 (Task) 和子流程 (Sub-Process)。
网关 (Gateways)		网关主要用于控制流程中的顺序流的走向，使用网关可以控制流程进行分支与合并。
顺序流 (Sequence Flow)		顺序流显示流程将会执行哪个活动。
消息流 (Message Flows)		消息流主要用于显示消息在流程参与者之间的传递情况。
关联 (Association)		主要用于连接流程元素及其制品 (流程信息)。
池 (Pool)		存放道的容器。
道 (Lane)		用于区分流程参与人的职能范围。
数据对象 (Data Object)		数据对象主要表示活动需要的或者产生的信息。
消息 (Message)		消息主要用于描述流程参与者之间的沟通内容。
组 (Group)		主要用于存放一些流程信息，包括流程文档、流程分析信息等。

注释（Text Annotation）		主要为阅读流程图的人提供附加的文字信息。
---------------------	---	----------------------

以上为 BPMN 规范中定义的基本元素，在这些元素的基础上，还会产生多种子元素，例如网关（Gateways）元素，还可以细分为单向网关、并行网关等，这些细分的元素将会在本书的 BPMN2.0 规范章节详细讲解。

1.2.3 BPMN2.0 的 XML 结构

BPMN2.0 规范中除了定义流程元素的图形外，还对流程描述文件作了语法上的定义，例如在定义一个 `userTask` 的时候，BPMN2.0 规范中定义了需要有 `id` 和 `name` 属性，定义一个顺序流，需要提供 `id`、`name`、`sourceRef` 和 `targetRef` 属性。BPMN2.0 定了 XML 规范，这样的话，一份流程描述文件可以在不同的流程引擎中使用（流程引擎需要遵守 BPMN2.0 规范）。

除了 BPMN2.0 规定的元素以及属性外， workflow 引擎的供应商还可以在这些规范的基础上添加额外的属性，但是这些扩展的属性不允许与任何的 BPMN2.0 元素产生冲突，除些之外，在对属性进行扩展时，所产生的流程模型与流程图，必须要让流程的参与者能够轻松看懂，而且规范中最基础的流程元素不允许发生改变，因为这是 BPMN2.0 规范的初衷。

BPMN 定义的 XML 元素以及相关众多，各个元素的作用以及其使用，将会在本书后面章节中讲述。

1.3 Activiti 介绍

当 BPMN2.0 规范在 2011 年发布时，各个 workflow 引擎的供应商均向其靠拢，包括 jBPM5 和本书所介绍的 Activiti。Activiti 的第一个版本为 5.0alpha1，一直到 2010 年 12 月发布了 Activiti5.0 的正式版，此过程经历了 4 个 alpha 版本、2 个 beta 版本和 1 个 rc 版本，直到 5.0 正式版本才出现对 BPMN2.0 规范的支持。Activiti6.0 于 2017 年 5 月发布，已经开始实现 DMN 规范。

1.3.1 Activiti 的出现

Activiti 的创始人 Tom Baeyens 是 jBPM 的创始人，由于在 jBPM 的未来架构上产生意见分歧，Tom Baeyens 在 2010 年离开了 JBoss 并加入 Alfresco 公司，Tom Baeyens 的离开使得 jBPM5 完全放弃了 jBPM4 的架构，基于 Drools Flow 重新开发，而在 2010 年的 5 月，Tom Baeyens 发布了第一个 Activiti 版本（5.0alpha1），由此看来，Activiti 更像是 jBPM4 的延续，也许为了让其看起来更像 jBPM4 的延续，Activiti 团队直接将 Activiti 的第一版本定义为 5.0。

1.3.2 Activiti 的发展

从 2010 年 5 月发布第一个 Activiti 版本至今（2017 年），Activiti 经历了近几十个版本的演化，笔者成书时版本已经发布到 6.0.0.RC1。Activiti 采用了宽松的 Apache Licence2.0 开源协议，因此 Activiti 一出，就得到了开源社区的大力支持，在开源社区的支持下，Activiti 可以吸引到更多的工作流专家参与到该项目中，并且可以促使 Activiti 在工作流领域的创新。

1.3.3 选择 Activiti 还是 jBPM

根据前面的内容可以得知，jBPM5 和 Activiti 同样支持 BPMN2.0 规范，但是实际上 jBPM5 已经推翻了 jBPM3 和 jBPM4 的架构，使用了 Drools Flow 作为工作流架构，这对于原来使用 jBPM3 和 jBPM4 的用户来说是非常郁闷的一件事（从零开始重新学习 jBPM5），而 Activiti 更像是原来 jBPM4 的延续，因此对于原来使用 jBPM3 和 jBPM4 的用户来说，更推荐使用 Activiti，但是由于 JBoss 中有一些优秀的项目（例如规则引擎 Drools、Seam 等），jBPM5 与这些项目进行整合具有先天的优势，因此如何进行选择还需要进行权衡。

除了原来的架构有所改变之外，还需要考虑的是，jBPM5 采用的是 LGPL 开源协议，如果要在其基础上使用修改和衍生的方式做二次开发的商业软件，涉及的修改部分需要使用 LGPL 协议，因此对于这些商用的软件来说，如果对 jBPM5 的源代码进行修改并做二次开发，显然不是明智的选择。相对于 jBPM5 来说，Activiti 采用了更为宽松的 Apache License2.0 协议，该协议鼓励代码共享和尊重原作者的著作权，允许对代码进行修改、再发布而不管其用途。

1.4 本章小结

本章对工作流的起源以及发展作了简单的介绍，其中主要介绍了在工作流领域的 BPMN2.0 规范，BPMN2.0 规范为工作流应用提供了语言以及图形的标准，在 1.2 小节介绍 BPMN2.0 规范目标以及该规范的部分内容，在 1.3 小节讲述了 Activiti 的产生背景，简述了 Activiti 目前所拥有的优势，并且与“成熟”的 jBPM 进行对比，在经过对工作流领域以及 Activiti 的简单介绍后，本书将带领读者开始 Activiti 之旅。

2 安装与运行 Activiti

要点

- 安装 JDK 与 MySQL

- 安装 Eclipse 以及 Activiti 插件
- 运行官方的 Activiti 例子

Activiti 的第一个正式版本发布于 2010 年 12 月 1 日，经过多年的发展，Activiti 已经成为一个较为成熟的工作流引擎，作为一个开源的工作流引擎，它在工作流领域吸引了众多开发者的目光，在当前的工作流框架角逐中，慢慢成为众多企业的首选。在 2017 年 5 月 26 日，Activiti 迎来全新篇章：6.0 版本正式发布。本书将以 6.0 版本为基础，讲解基于 Activiti 的工作流应用开发。

本章将介绍 Activiti 的安装与运行、Activiti 开发环境的搭建等内容，本书除了最后一章的项目案例外，其他所有的案例均以本章的开发环境为基础。搭建 Activiti 的开发环境，需要安装 JDK、Eclipse、MySQL 等软件，除此之外，还会编写第一个 Activiti 应用，让大家对 Activiti 有一个初步的了解。

注：本书全部的案例均在 Windows7 下开发和运行。

2.1 下载与运行 Activiti

如果仅仅是运行 Activiti，看下工作流的例子，可以只下载 JDK、Tomcat 和 Activiti，Activiti 的开发包中，已经含有 Activiti 的 web 应用例子。本书使用的 Tomcat 版本为 7.0.42，可以到以下地址下载该版本的 Tomcat：

<http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.42/bin/apache-tomcat-7.0.42.zip>

本书所使用的 Tomcat 是非安装版本，下载后解压即可使用，在运行 Tomcat 需要先安装 JDK。本书全部需要使用浏览器的程序，均使用 Google Chrome 浏览器，笔者也建议读者使用该浏览器。

注意：已有 JDK 与 MySQL 环境的，可跳过相应章节。

2.1.1 下载和安装 JDK

Activiti6.0 要求在 JDK7 以上版本运行，本书所使用的是 JDK8（32 位），大家可以到以下网址下载 JDK：

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

选择合适的版本进行下载后即可进行安装，在 Win7 下，默认的安装目录是 C:\Program Files (x86)\Java\jdk1.8.0_131，安装完成后，需要配置环境变量，新建 JAVA_HOME 变量，值为 JDK 的安装目录，如图 2-1 所示。



图 2-1 JAVA_HOME 环境变量

添加了 JAVA_HOME 环境变量后，修改系统的 Path 变量，加入“%JAVA_HOME%\bin”，

如图 2-2 所示。



图 2-2 Path 环境变量

为了验证 JDK 是否成功安装，打开系统命令行，输入“java -version”可看到 JDK 的版本信息，笔者安装的 JDK 信息如图 2-3 所示。



图 2-3 JDK 版本信息

看到图 2-3 信息，即表示 JDK 成功安装。如果大家的机器需要不同版本的 JDK，可以为 JAVA_HOME 设置不同值来实现切换。

2.1.2 下载和安装 MySQL

MySQL 作为市面上关系型数据库的佼佼者，一直受到各大企业及开发人员的青睐，笔者之前就职的公司一直使用 MySQL，因此本书选用 MySQL 作为 Activit 的数据库。目前 MySQL 的版本发展到 5.7，由于数据库并不是本书的重点，因此笔者选用了较为成熟的 5.6 版本（64 位），大家可到以下网址下载 5.6 版本的 MySQL 数据库：

<https://dev.mysql.com/downloads/installer/5.6.html>

下载并安装了 MySQL 数据库后，将 MySQL 的 bin 目录添加到环境变量中，以便可以在命令行中使用 MySQL 命令。

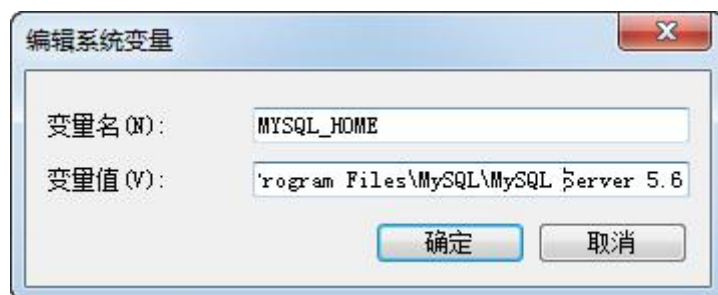


图 2-4 添加系统变量

修改系统变量的 Path 属性，添加 “%MYSQL_HOME%\bin”，如图 2-5 所示。



图 2-5 修改 Path 变量

完成以后的步骤后，打开命令行，输入 “mysql -V”，可以看到输出如图 2-6 所示。



图 2-6 查看 MySQL 安装

关于 MySQL 数据库的客户端工具，本书使用的是 Navicat，读者也可以使用其他工具，这些工具目的是为了更加方便操作 MySQL 数据库，大家可根据个人习惯来选用。本书的开发环境为 Windows7，安装完 MySQL5.6 后，笔者建议将 MySQL 配置成区分大小写（默认不区分），修改 MySQL 的配置文件 my.ini（如果在 Windows7 下安装 MySQL，则修改 C:\ProgramData\MySQL\MySQL Server 5.6\my.ini），加入 “lower_case_table_names = 0” 配置，重启 MySQL 服务即可。

2.1.3 下载和安装 Activiti

安装了 JDK 和 MySQL 后，现在可以下载 Activiti，Activiti 的主页为：<http://www.activiti.org/>，本书使用的 Activiti 版本为 6.0 版本，以下为 Activiti6.0 版本的下载

地址：

<https://github.com/Activiti/Activiti/releases/download/activiti-6.0.0/activiti-6.0.0.zip>

由于某些非技术原因，以上链接可能在国内无法打开，需要借助其他方法进行下载。

下载解压后得到 **activiti-6.0.0** 目录，该目录下有三个子目录：**database**、**libs** 和 **wars**。

以下为各个目录的作用描述：

- ❑ **database**: 用于存放 Activiti 数据表的初始化脚本（**create** 子目录）、删除脚本（**drop** 子目录）和升级脚本（**upgrade** 子目录）。从各个目录中的脚本可得知，目前 Activiti 支持各大主流的关系型数据库，包括 DB2、MySQL、Oracle 等等。
- ❑ **libs**: 存放本版本 Activiti 所发布的 jar 包，也包含对应的源码包。
- ❑ **wars**: 存放 Activiti 官方提供的 war 包，当前版本有 **activiti-app.war**、**activiti-admin.war**、**activiti-rest.war** 三个 war 包。

需要注意的是，这三个 war 包默认情况下使用的是 H2 数据库，该数据库是一个内存数据库，因此部署前不需要进行任何的数据库配置，但如果重启了应用服务器，那么之前的数据将会丢失，大家在使用时请注意这个小细节。将三个 war 包复制到 Tomcat/webapps 目录并启动 Tomcat，在浏览器中打开以下链接，即可以看到 Activiti 的演示界面：

<http://localhost:8080/activiti-app/>

Activiti 的演示界面如图 2-7 所示。

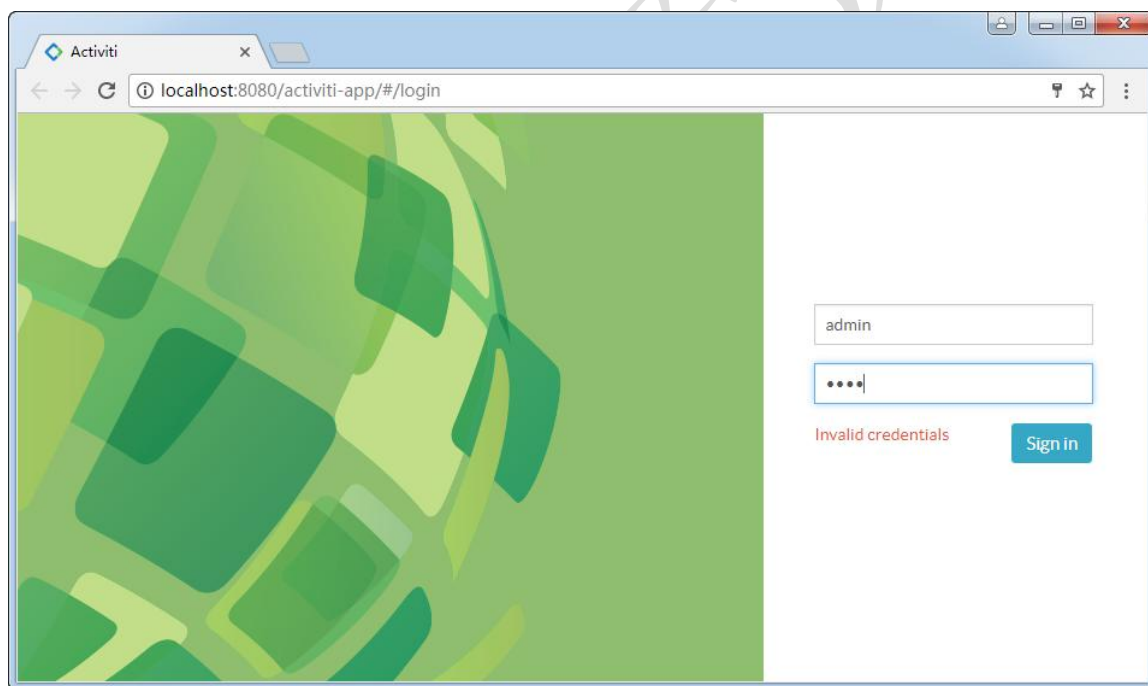


图 2-7 Activiti Demo 的登录界面

注：见到图 2-7 的界面，表示已经启动成功，如果需要登录及使用其他功能，请见 2.2 小节。

2.2 运行官方的 Activiti 示例

Activiti 官方发布的 **activiti-app**，可以说是一个较为完善的样例，用户可通过试用该应用来了解 Activiti 的大部分功能。随 **Activiti6.0** 版本发布的官方示例，包括流程图定义、流

程发布、动态表单等一系列功能，在笔者看来，这个示例的功能已经相当强大。但由于该示例偏向技术，如果需要开发更贴近某个特定业务的产品，我们还是需要掌握 **Activiti** 的核心。

本小节将以一个简单的请假流程为基础，向大家展示该 **Activiti** 示例的功能，以便大家对工作流引擎有一个初步的了解。

注：activiti-app 的登录用户名为 admin，默认密码为 test，该应用的功能将在本小节讲述。

2.2.1 本小节流程概述

我们先定一个简单的请假流程，主要是由员工发起请假，然后再由他的经理审批，最后流程结束，流程图如图 2-8 所示。



图 2-8 员工请假流程

本小节目的是为了让大家初步了解 **activiti-app** 的功能，对 **Activiti** 有一个初步的认识，因此设计的流程较为简单。

2.2.2 新建用户

根据我们前面定义的请假流程，需要有一个员工的用户，然后需要有一个经理的用户，在实际业务中，普通员工、经理可能就是用户，在此为了简单起见，只定义有一个员工与一个经理，不涉及用户组数据。使用 **admin** 账号登录 **activiti-app**（默认密码是 **test**），主界面如图 2-9 所示。

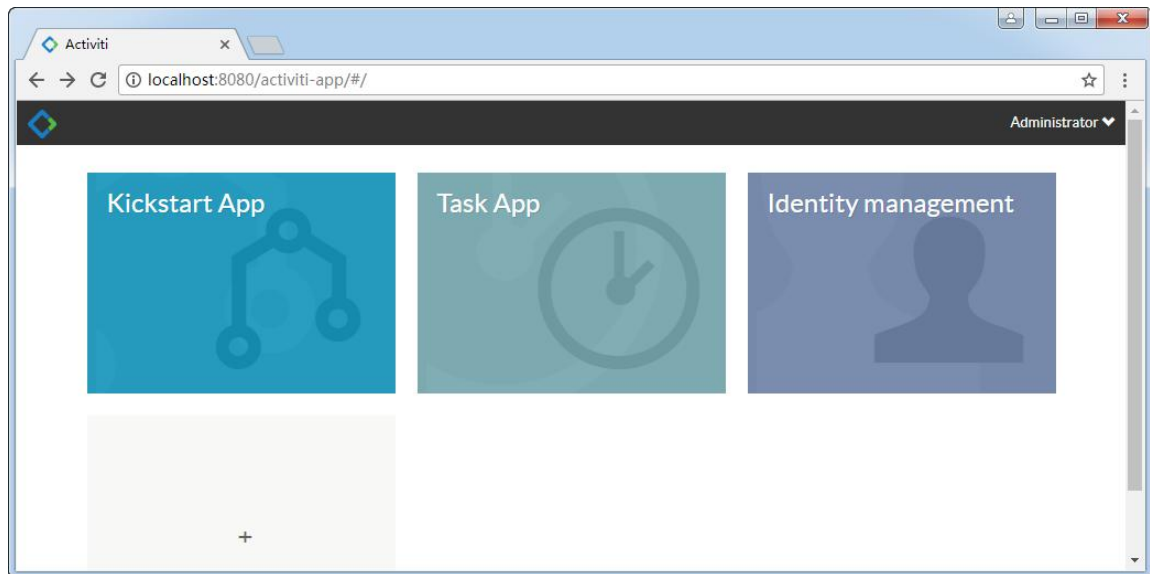


图 2-9 登录 activiti-app 后的主界面

主界面的三个菜单主要承担以下功能：

- ❑ **Kickstart App**: 主要用于流程模型管理、表单管理及应用（App）管理，一个应用可以包含多个流程模型，应用可发布给其他用户使用。
 - ❑ **Task App**: 用于管理整个 activiti-app 的任务，在该功能里面也可以启动流程。
 - ❑ **Identity management**: 身份信息管理，可以管理用户、用户组等数据。
- 点击“Identity Management”菜单，再点击 Users 菜单，界面如图 2-10 所示。

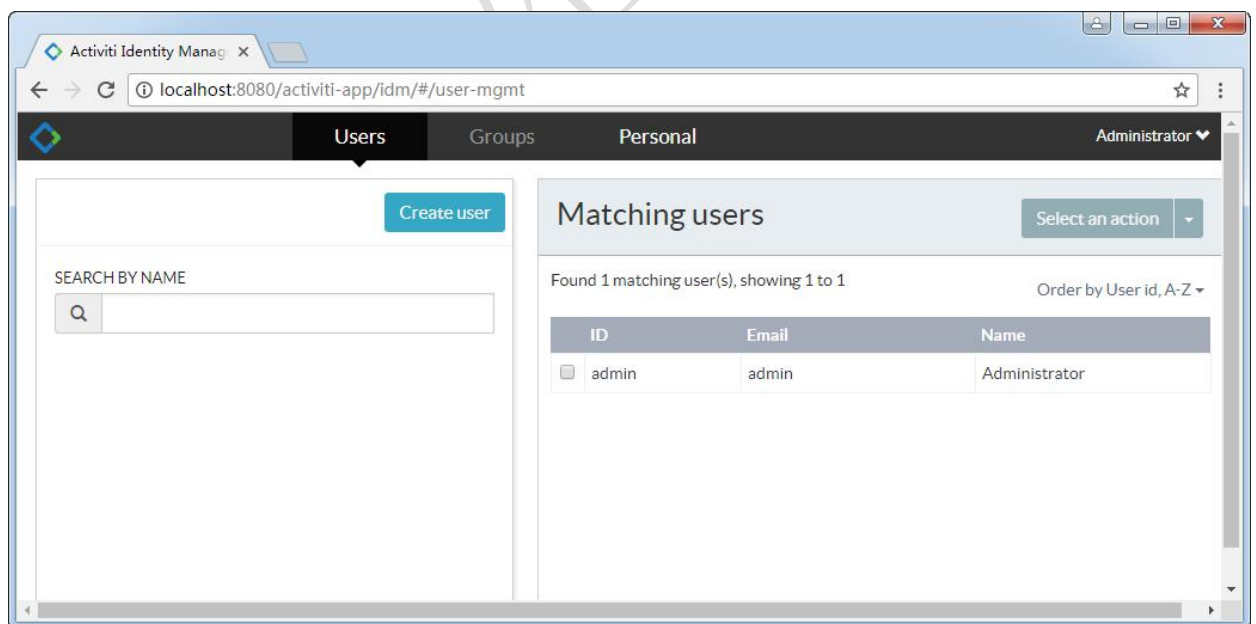


图 2-10 进入用户管理

点击“Create user”按钮，弹出输入新用户信息的界面，根据我们定义的请假流程，需要新建一个员工用户。新建用户名为“employee”的用户，信息如图 2-11 所示。

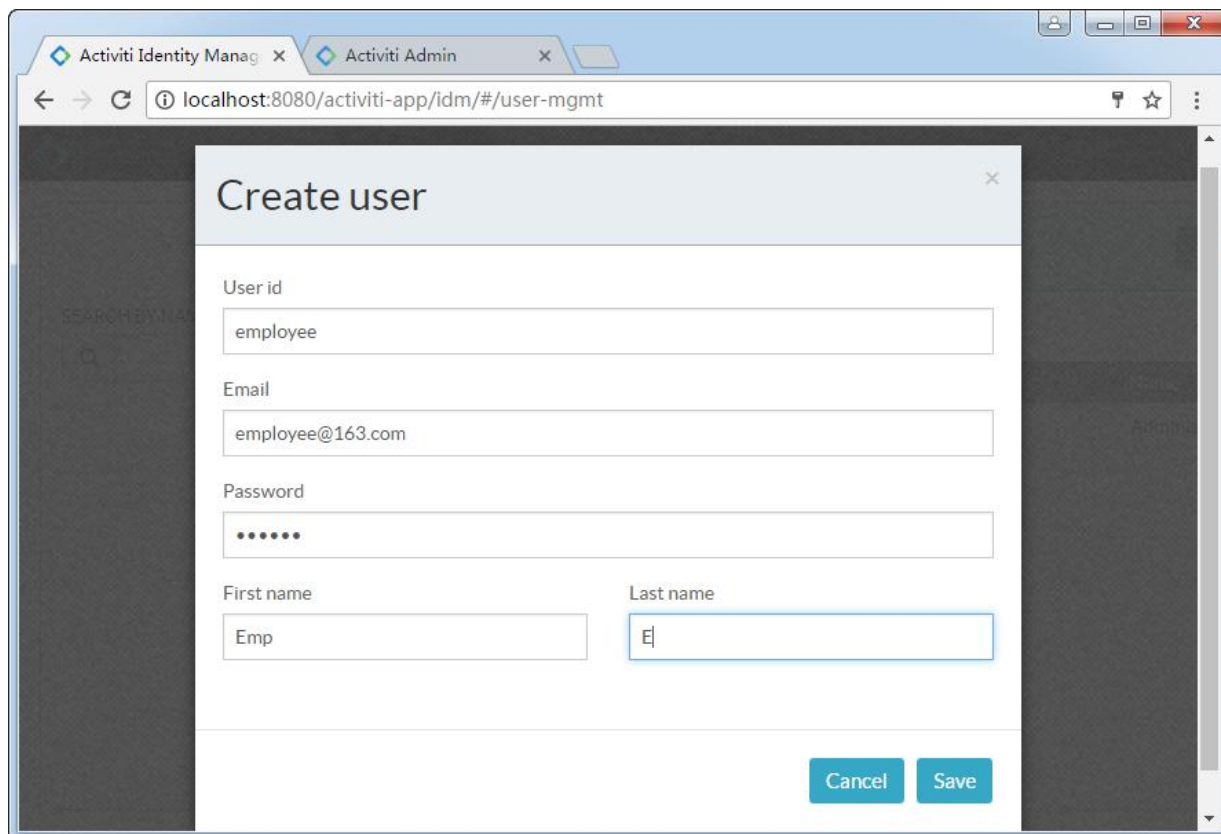


图 2-11 新建用户

需要注意的是，Email 等信息虽然不是必填的，但如果不填，则在登录时会出现异常，笔者建议将全部信息填完，以便减少遇到的问题。以同样的方法，再创建一个“manager”的用户作为经理，用于审核请假任务。

2.2.3 定义流程

点击“Kickstart App”菜单，进入流程模型管理的主界面，点击“Create Process”按钮，弹出新建流程模型界面，如图 2-12 所示。

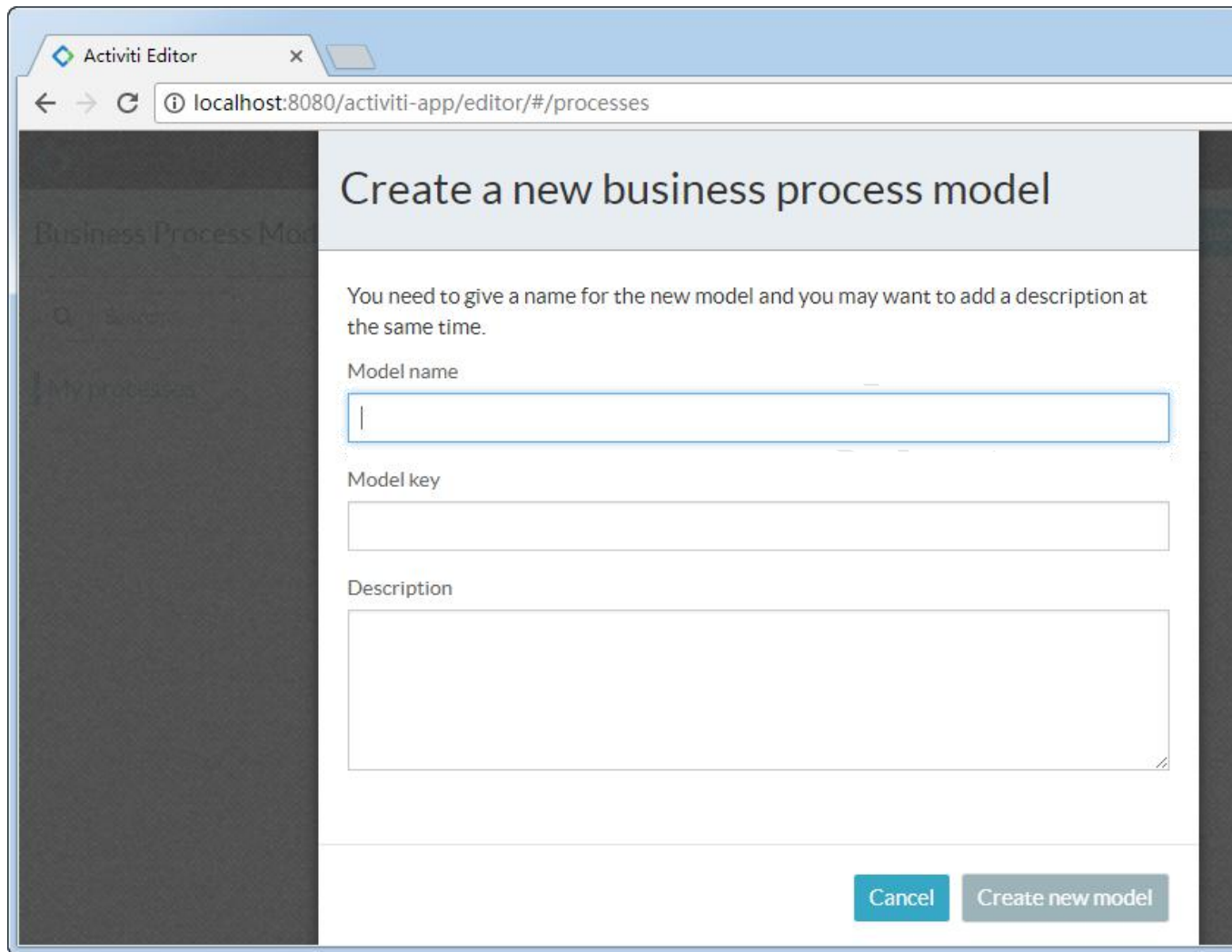


图 2-12 新建流程模型界面

新建模型后，会进入流程模型设计界面，在流程设计界面中，只需要普通的鼠标拖拉操作，即可完成流程模型的定义，该编辑器也可以开放给业务人员使用。根据前面定义的请假流程，在编辑器中“拖拉”一下，定义请假流程模型，如图 2-13 所示。

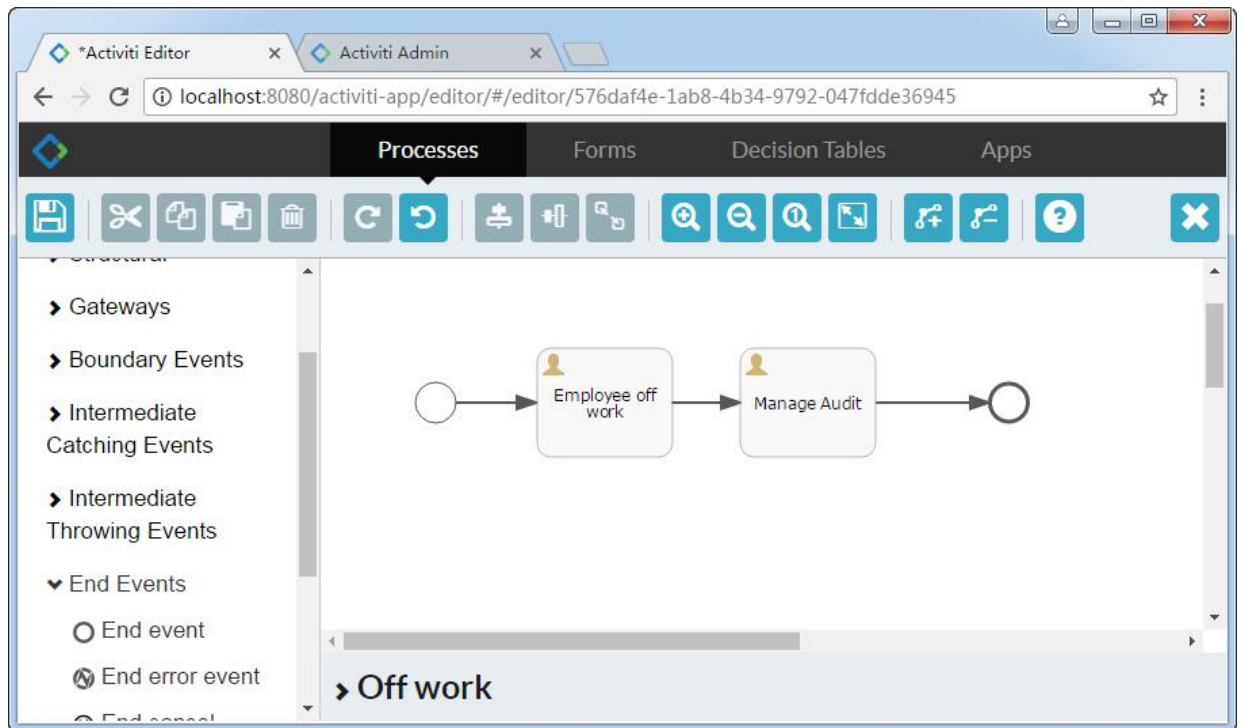


图 2-13 设计流程模型

在图 2-13 中，定义了一个开始事件、两个用户任务、一个结束事件。我们定义的请假业务，需要将该用户任务分配给 **employee** 用户。点击第一个用户任务，并修改“Assignment”属性，如图 2-14 所示。

The screenshot shows the 'Assignment' configuration dialog in the Activiti Editor. The dialog has a sidebar on the left with a tree view of process elements. The main area contains the following fields and controls:

- Type:** Two tabs, 'Identity store' (selected) and 'Fixed values'.
- Assignment:** A text field containing 'Candidate users'.
- Candidate users:** A text field containing 'employee employee'.
- Search:** A text field containing 'employee'. Below it, a message says 'Use ↑ and ↓ to select and press Enter to confirm or use the mouse'. Below that, a list box shows 'employee employee'.
- Email:** A text field containing 'Enter an email address' and an 'Accept' button.
- Allow process initiator to complete task:** A checkbox that is currently unchecked.
- Buttons:** 'Cancel' and 'Save' buttons at the bottom right.

图 2-14 为任务分配给用户

如图 2-14 可知，将“Employee off work”任务分配给“Emp E”用户，需要注意的是，Emp 是用户的真实名称，登录系统的用户名是 employee。保存成功后，再使用同样的方法将“Manage Audit”任务分配给 manager 用户，保存流程模型后，就可以将流程发布。

2.2.4 发布流程

在 activiti-app 中，一个 App 可包含多个流程模型，因此在发布流程前，先新建一个 App 并为其设置流程模型。点击 Apps 菜单，再点击“Create App”按钮，新建一个 App，如图 2-15 所示。

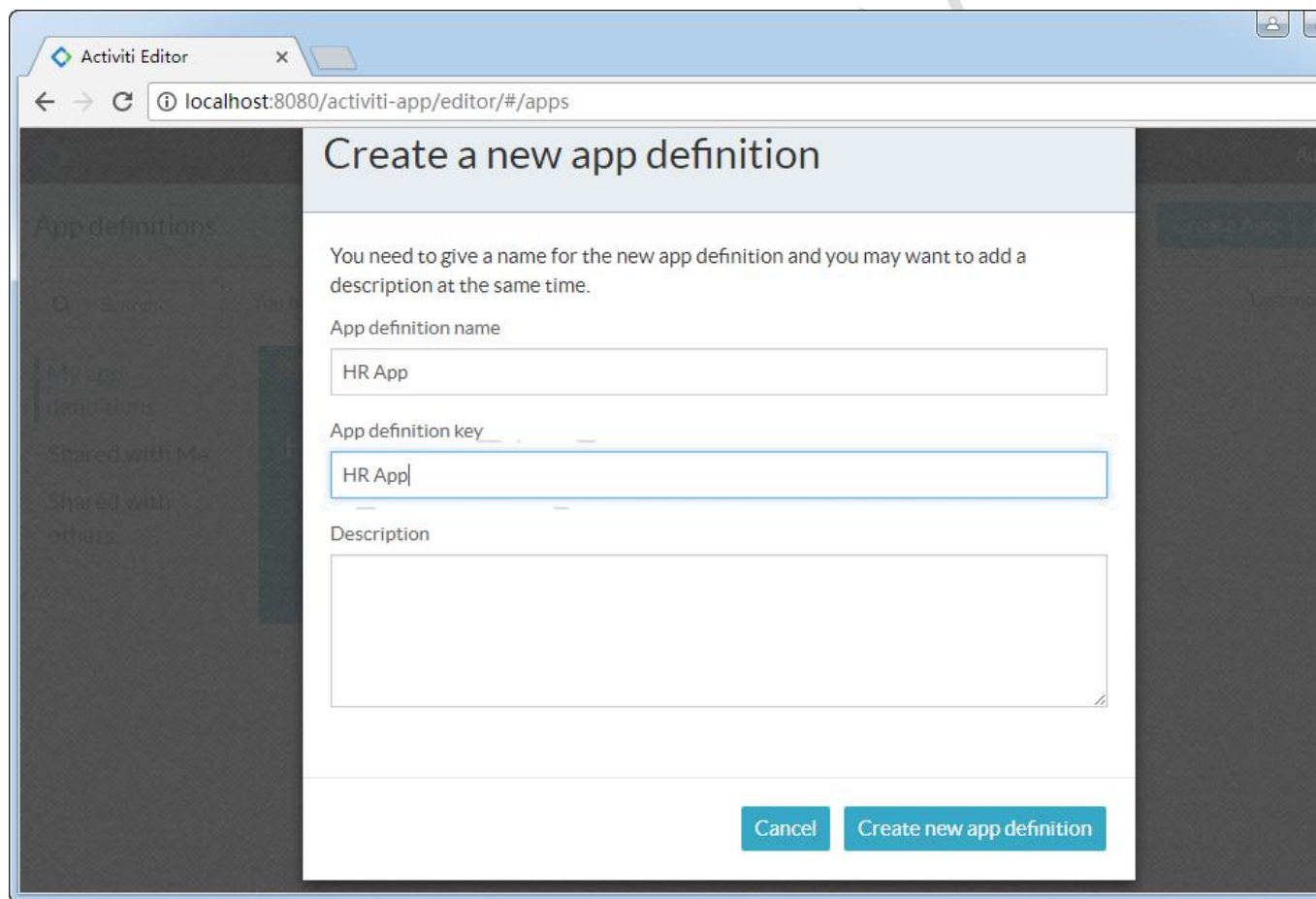


图 2-15 新建 App

由于我们设计的请假流程，属于人事管理领域的，因此新建一个给 HR 使用的 App（应用），该 App 就包含我们前面所设计的请假流程模型。创建 App 成功后，再为其设置流程模型并发布 App，点击修改 App，显示界面如图 2-16 所示。

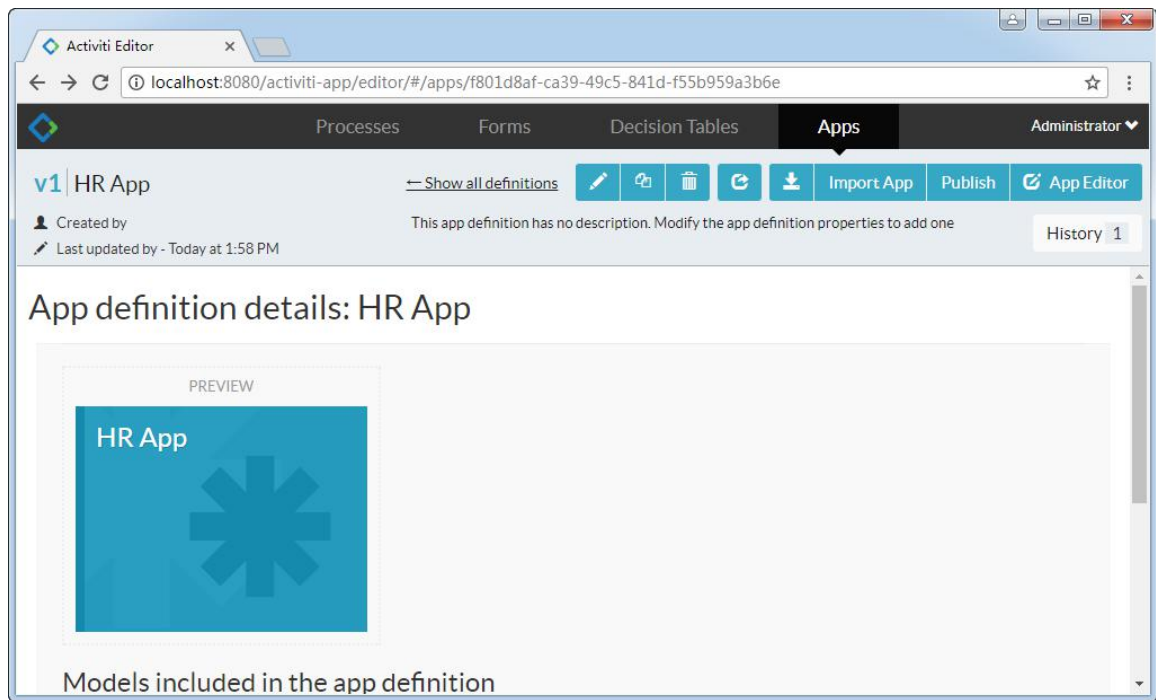


图 2-16 查看 App

图 2-16 为查看 App 的界面，右上角的“Publish”按钮可以发布 App。点击“App Editor”可以进行 App 的模块修改，本例中已经将前面定义的流程绑定到“HR App”中。

2.2.5 启动与完成流程

发布了 App，再使用之前新建的 employee 用户进行登录，登录后可以看到 HR App 的菜单，如图 2-17 所示。

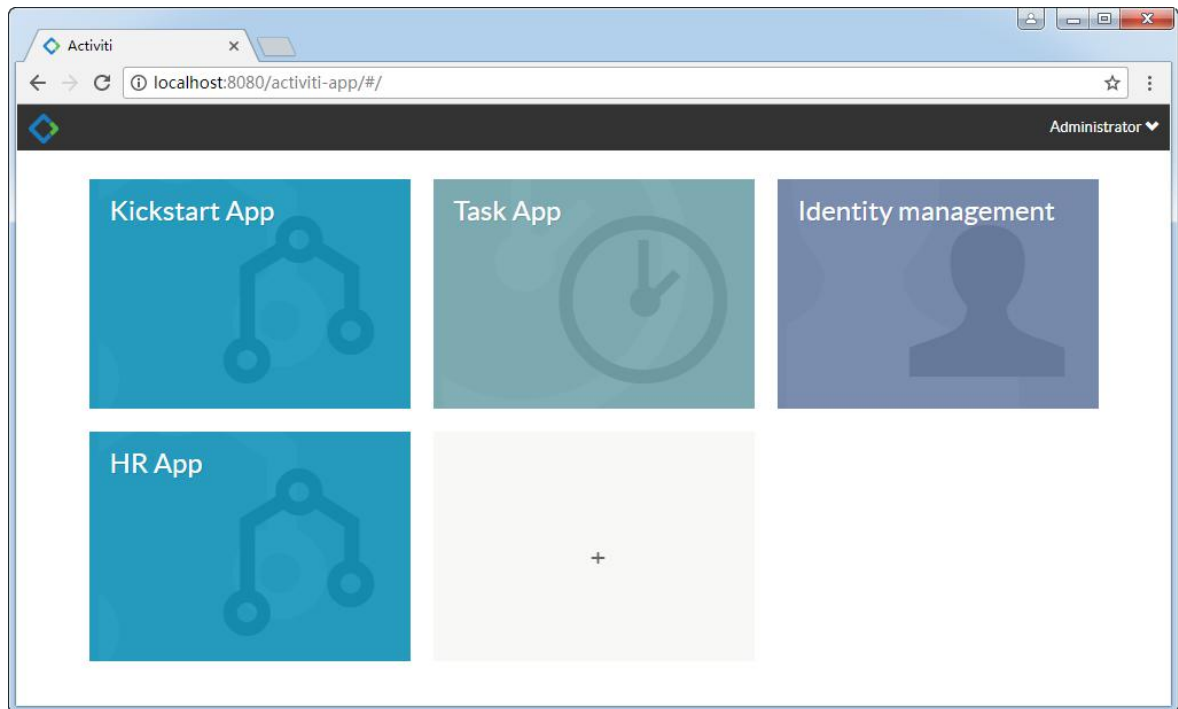


图 2-17 员工登录后主界面

进入 HR App 并且点击“Processes”菜单，在界面左上角，可以看到“Start a process”按钮，点击启动请假流程后，可以看到界面如图 2-18 所示。

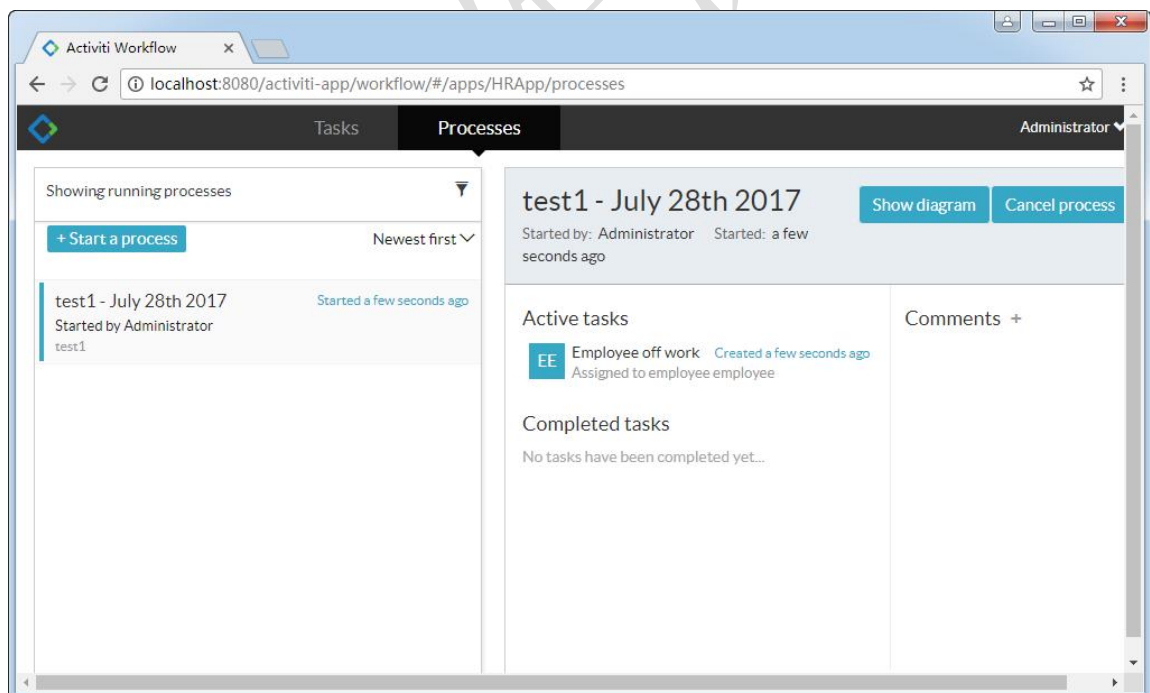


图 2-18 启动流程

根据流程模型的定义可知，启动流程后，就由 **employee** 来完成第一个用户任务，点击图 2-18 右边的任务列表，进行任务操作。如图 2-19 所示。

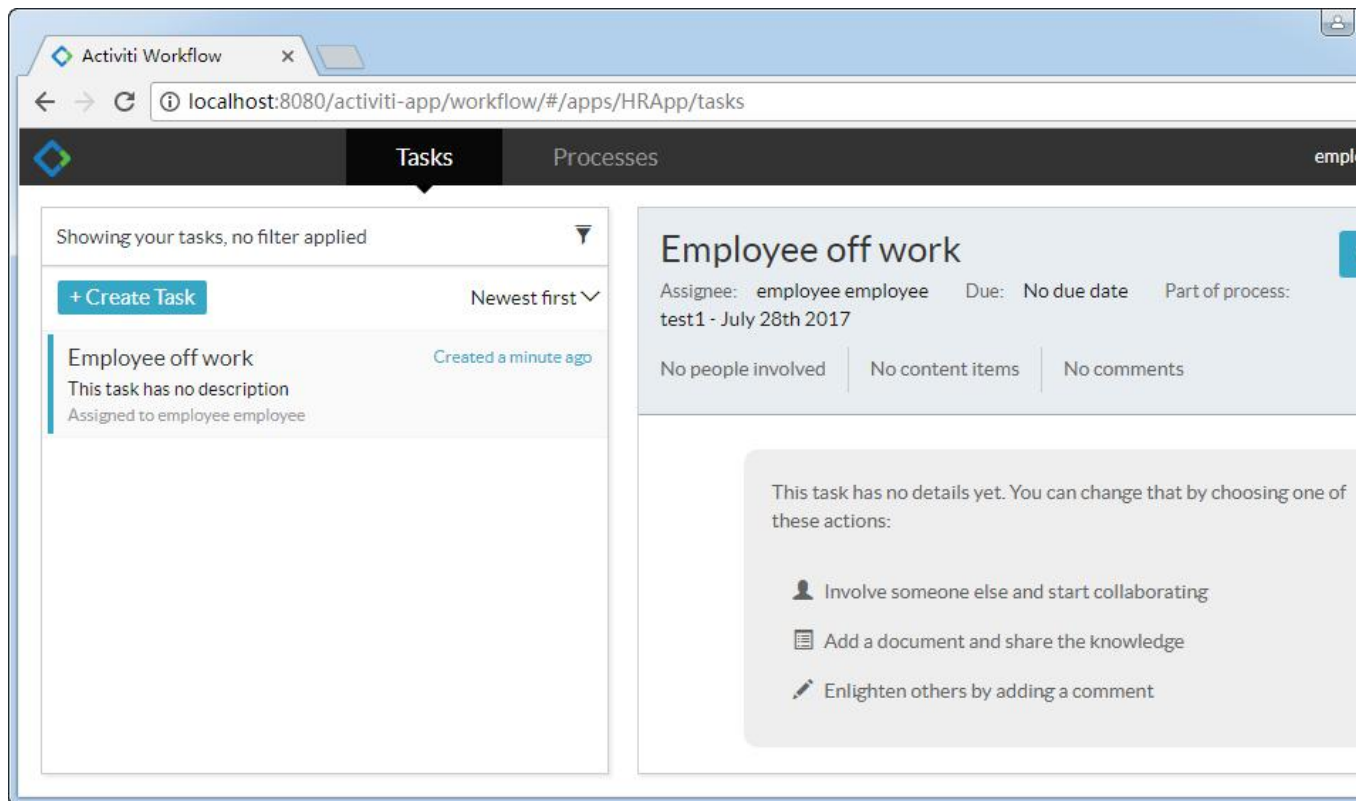


图 2-19 查看用户任务

图 2-19 右上角的“Complete”按钮，点击后即可完成当前的用户任务。按照流程设计，employee 完成任务后，就到 manager 用户审核请假。使用 manager 用户登录系统，同样进入“HR App”的 Processes 菜单，可以同样看到分配到 manager 用户下面的任务，以同样的方式完成任务后，流程结束，如图 2-20。

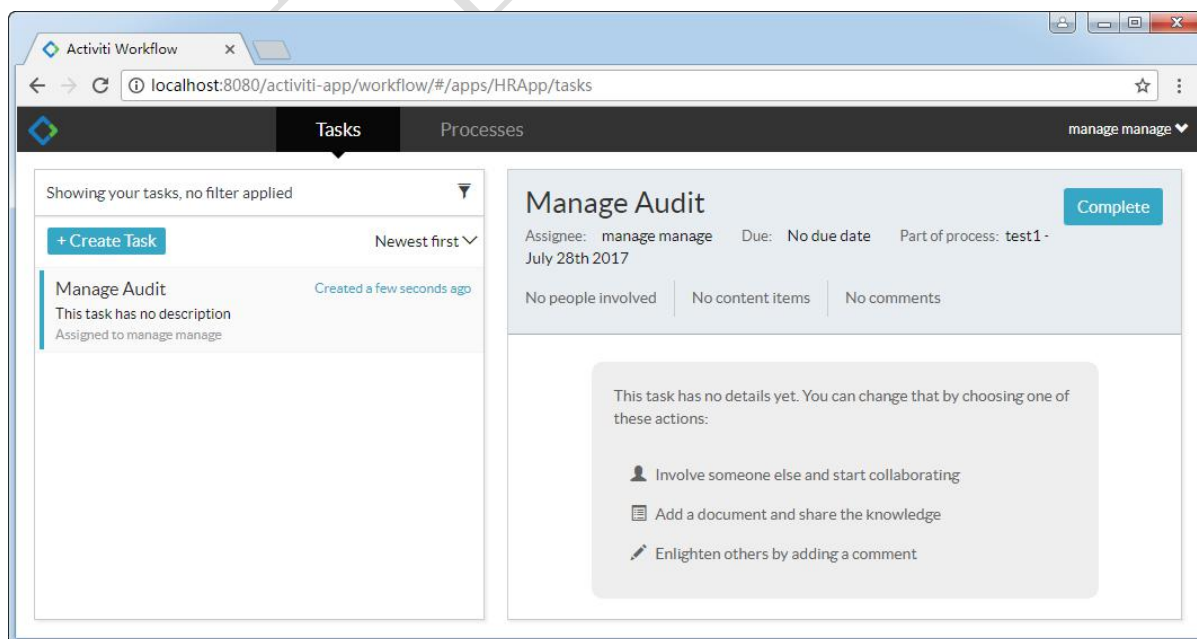


图 2-20 manager 完成任务

至此，这个简单的请假流程，已经在 activiti-app 上面运行成功。

2.2.6 流程引擎管理

除了 **activiti-app** 这个 war 包外，还有一个 **activiti-admin** 的 war 包，在部署时也放到 Tomcat 的应用目录下，**activiti-admin** 用于查看流程引擎的主要数据，包括流程引擎的部署信息、流程定义、任务等数据。启动了 Tomcat 后，在浏览器中打开以下链接：

<http://localhost:8080/activiti-admin>

打开上面链接后，可以看到 **activiti-admin** 的登录界面，内置的用户名为 **admin**，密码为 **admin**。登录成功后，点击“**Configuration**”菜单，先配置管理的对象信息，由于 **activiti-app** 也是部署在 Tomcat 中，因此只需要修改一下端口即可，将默认的 **9999** 端口改为我们的 Tomcat 端口（**8080**），修改界面如图 2-21 所示。

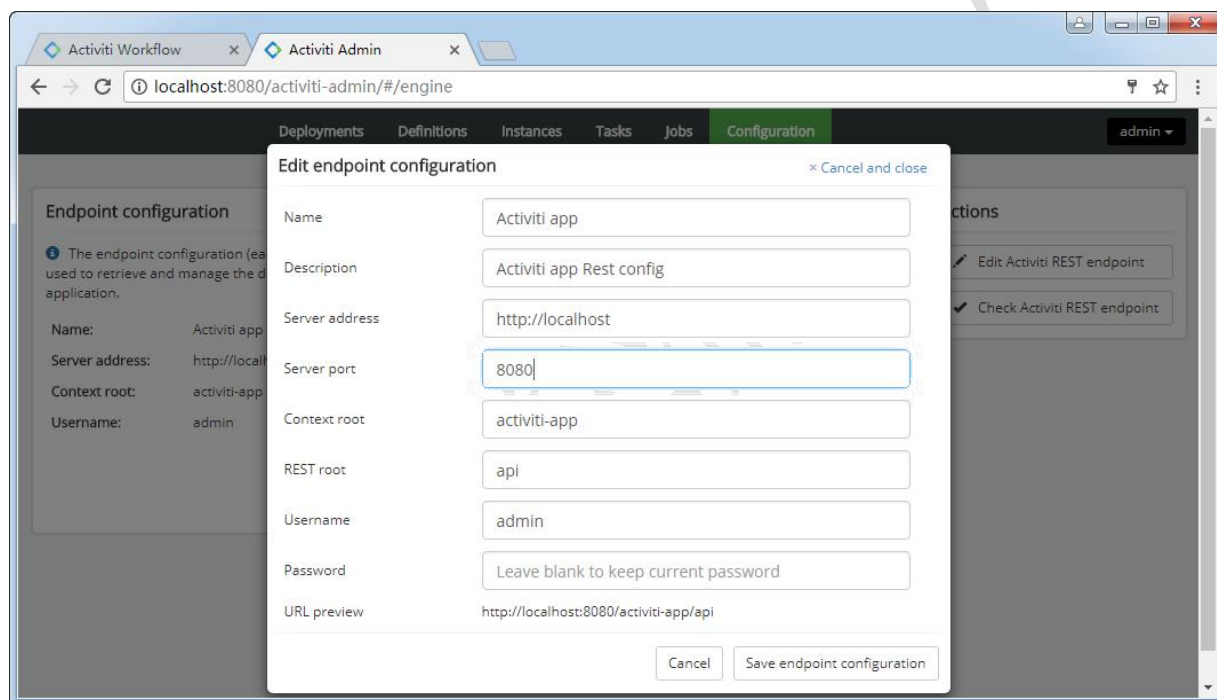


图 2-21 修改 **activiti-admin** 的配置信息

修改完成配置后，可以点击“**Check Activiti REST endpoint**”来测试是否可以连接到 **activiti-app** 的接口，连接成功后会有提示。

点击“**Instances**”，可以看到我们之前完成的请假流程实例，再点击流程实例，可以查看到流程的全部信息，请假流程的全部信息如图 2-22 所示。

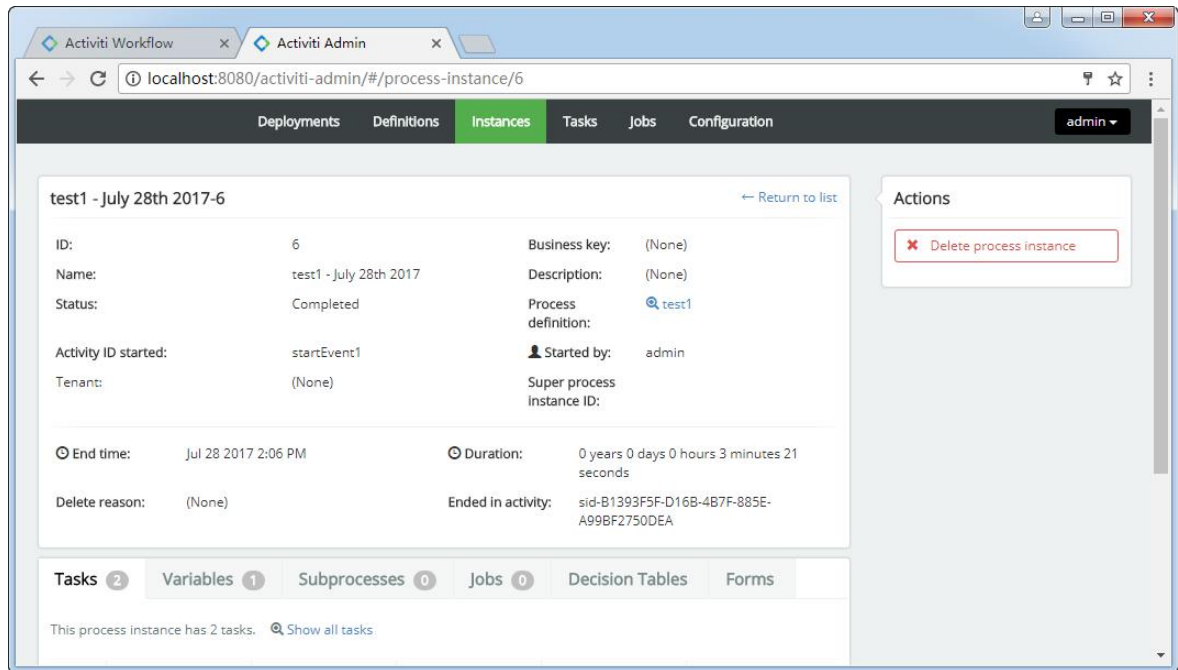


图 2-22 查看请假流程信息

在 **activiti-admin** 应用中的数据，可以通过 **Activiti** 发布的接口获取，接口的使用将在本书后面的章节中讲述。

根据本小节的内容可知，**Activiti** 官方提供的 **activiti-app**、**activiti-admin** 两个应用，包含了流程设计、流程发布、流程引擎管理等功能，本书后面的章节，将会深入讲解 **Activiti** 的功能，学习后再回头看官方的两个应用，即可明白它们的实现原理。

3 Activiti 开发环境搭建

本章要点

- 安装 JDK 与 MySQL
- 安装 Eclipse 以及 Activiti 插件
- 编写第一个 Activiti 程序

3.1 安装开发环境

本小节所说的 **Activiti** 开发环境包括以下内容：

□ Eclipse IDE

❑ Eclipse 的 Activiti 插件

3.1.1 下载 Eclipse

本书使用 Eclipse 作为开发工具，如果想使用 Activiti 的 Eclipse 设计器，官方建议使用 Kepler（4.3）或者 Luna（4.4）版本，本书所使用的版本为 Luna，大家可以从以下的地址得到该版本的 Eclipse：

<http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/lunasr2>

目前 Eclipse 已经发展到 4.7 版本，本书所使用的 Eclipse 功能不多，主要使用 Activiti 的设计器插件，本书的全部代码，理论上可以在高版本的 Eclipse 中运行。

安装 Eclipse 插件的过程较为漫长，如果想直接使用已经安装好插件的 Eclipse，可以到以下链接下载：<https://pan.baidu.com/s/1bpEh1Gr>，如以上链接失效，可以与笔者联系，笔者邮箱地址：yangenxiong@163.com。如果下载到了安装好插件的 Eclipse，则可以跳过相应章节。

3.1.2 安装 Activiti 插件

使用 Eclipse 的 Luna 版本，在安装 Activiti 插件前，要安装 EMF 插件（2.6.0 版本）。打开 Eclipse，在“Help”菜单中选择“Install New Software”，点击“Add”按钮，弹出窗口如图 2-23 所示。

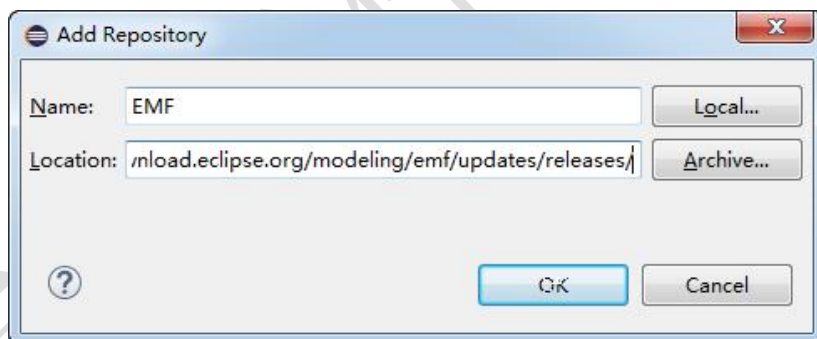


图 2-23 安装 EMF 插件

输入的名称为 EMF，位置 URL 为：<http://download.eclipse.org/modeling/emf/updates/releases/>，注意在选择版本时，要选择 2.6.0 版本，安装完成后重启 Eclipse，再进行 Activiti 插件安装。

使用 Activiti 的 Eclipse 插件，开发者可以对流程模型进行可视化操作，对于流程元素可以进行拖拉，插件会自动生成相应的 XML 代码。安装方法与 EMF 插件安装方法一样，输入的插件信息如图 2-24 所示，输入的位置 URL 为：<http://activiti.org/designer/update/>。

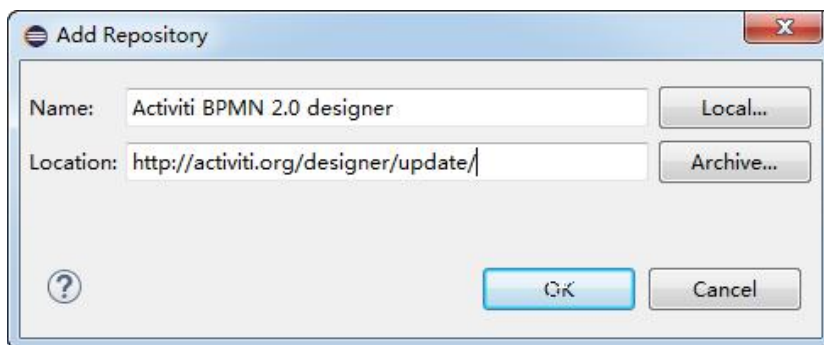


图 2-24 Eclipse 中添加软件仓库

插件安装完成后重启 Eclipse，在新建文件的对话框，如看到图 2-25 的选项，则表示已经安装成功。

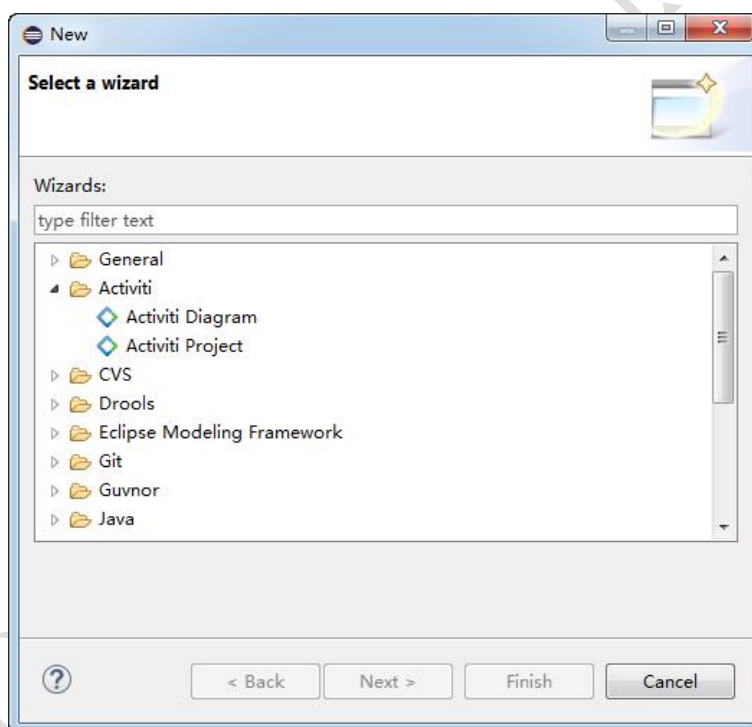


图 2-25 成功安装 Activiti 插件

再次强调一下，在安装过程中，由于网络的原因，会导致安装中断或者安装时间过长，如果不是为了体验插件安装，笔者建议直接下载安装好插件的 Eclipse，笔者提供的下载地址为：<https://pan.baidu.com/s/1bpEh1Gr>。

3.2 编写第一个 Activiti 程序

完成了 Activiti 的开发环境搭建后，可以进行第一个 Activiti 程序的开发，开发 Activiti 应用，基本上只需要 Eclipse 即可，但是为了能更加方便设计流程，还要求使用 Eclipse 的 Activiti 插件，在编写 Activiti 程序前，请先确认 Eclipse 和 Activiti 插件已经成功安装。

注：Activiti 的可视化插件，是为了更加方便进行流程模型设计，笔者建议还是要认真学习 BPMN 规范，明白插件的工作原理。

3.2.1 如何运行本书案例

使用 Eclipse 导入 codes\common-lib 项目，该项目用于存放本书全部例子所使用的第三方 jar 包，导入该项目后，可以选择某一章的案例进行导入，例如要查看第 4 章的案例，就可以在 Eclipse 可选择 codes\04 目录，将第 4 章全部的案例项目导入。每一个项目中都有相应的运行类，绝大部分的运行类都有 main 方法，直接运行相应案例的 main 方法即可以看到效果。

为了能在每个案例运行后看到数据库的变化，因此大部分的案例均会将 Activiti 的 databaseSchemaUpdate 属性配置为 drop-create（详细请见第 4 章），该属性会在相应案例运行前将原有的数据表删除，再创建 Activiti 的数据表，请读者注意该细节。

注：本书除 OA 系统、第 15 章的 Web 项目和第 16 章的 Web 项目外，全部的案例所使用的第三方包均存放在 codes\common-lib\lib 目录下，因此成功编译和运行全部案例的前提，是先导入 codes\common-lib 项目。

3.2.2 建立工程环境

打开 Eclipse，将 common-lib 项目导入到 Eclipse 中，然后新建一个普通的 Java 项目，在项目的根目录下建立一个 resource 的源文件目录。修改项目的“Java Build Path”，在“Libraries”中点击“Add JARs”，选中 common-lib/lib 目录下的全部 jar 包。项目结构如图 2-26 所示。

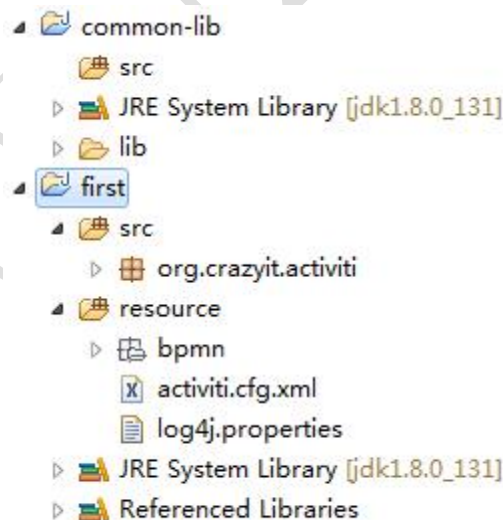


图 2-26 项目结构

注：项目 common-lib/lib 目录下面的 jar 包，是从 activiti-6.0.0/libs 目录下复制过去的，并且也含有其他框架的 jar 包。本书后面章节代码，如没有特别说明，也是使用该方法引入 jar 包。

3.2.3 创建配置文件

如果没有指定 Activiti 的配置文件，那么默认情况下将会到 CLASSPATH 下读取 `activiti.cfg.xml` 文件作为 Activiti 的配置文件，该文件主要用于配置 Activiti 的数据库连接等属性（详细请见第 4 章）。将 `activiti-5.10\setup\files\cfg\activiti\standalone` 目录下的 `activiti.cfg.xml` 文件复制到项目的 `resource` 目录下，修改该文件，内容如代码清单 2-1 所示。

代码清单 2-1: `codes\02\first\resource\activiti.cfg.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  <!-- 流程引擎配置的 bean -->
  <bean id="processEngineConfiguration"
        class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/act" />
    <property name="jdbcDriver" value="com.mysql.jdbc.Driver" />
    <property name="jdbcUsername" value="root" />
    <property name="jdbcPassword" value="123456" />
    <property name="databaseSchemaUpdate" value="true" />
  </bean>
</beans>
```

代码清单 2-1 中的 `activiti.cfg.xml` 是一份标准的 XML 文档，该 XML 中只配置了一个名称为 `processEngineConfiguration` 的 bean 元素，代码清单 2-1 中的粗体部分，配置了连接的数据库中 `act`，因此需要在 MySQL 中建立一个名称为“act”的数据库，数据库的属性如图 2-27 所示。

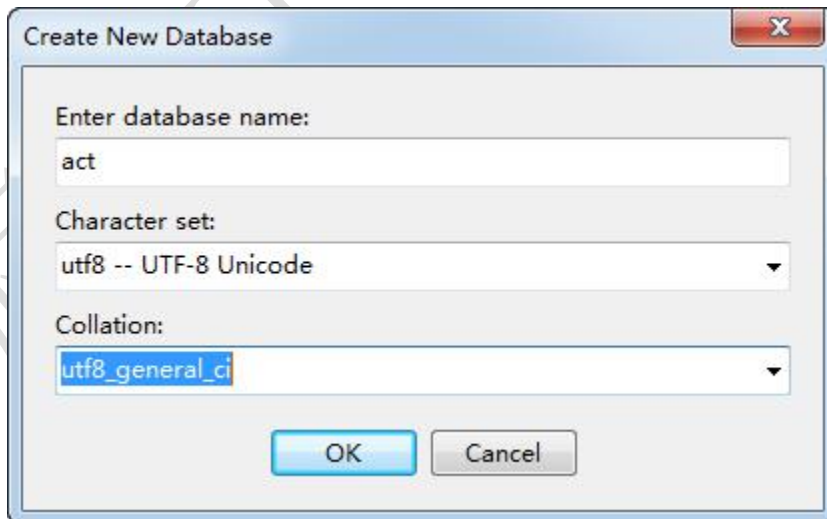


图 2-27 数据库属性

本书的数据库字符集均使用“utf8”，项目中的源文件也使用“UTF-8”编码，如出现乱码问题，请检查数据库及源文件编码。配置文件中的 `processEngineConfiguration` 的各个属性及其作用，请见第 4 章。

3.2.4 创建流程文件

流程描述文件是用 XML 语言去描述业务流程的文件，Activiti 的流程文件需要遵守 BPMN2.0 规范。使用 Activiti 的 Eclipse 插件新建一个流程文件，该流程与 Activiti 的 demo 中的费用申请单一致，图 2-28 为流程图，代码清单 2-2 为该流程的 XML 配置。

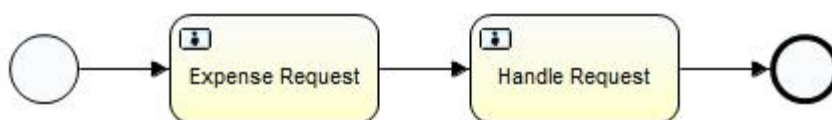


图 2-28 第一个 Activiti 流程

代码清单 2-2: codes\02\first\resource\bpmn\First.bpmn

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:activiti="http://activiti.org/bpmn"
  xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
  xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DC"
  xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI"
  typeLanguage="http://www.w3.org/2001/XMLSchema"
  expressionLanguage="http://www.w3.org/1999/XPath"
  targetNamespace="http://www.activiti.org/test">
  <process id="process1" name="process1">
    <startEvent id="startevent1" name="Start"></startEvent>
    <userTask id="usertask1" name="Expense Request"></userTask>
    <userTask id="usertask3" name="Handle Request"></userTask>
    <endEvent id="endevent1" name="End"></endEvent>
    <sequenceFlow id="flow1" name="" sourceRef="startevent1"
      targetRef="usertask1"></sequenceFlow>
    <sequenceFlow id="flow2" name="" sourceRef="usertask1"
      targetRef="usertask3"></sequenceFlow>
    <sequenceFlow id="flow3" name="" sourceRef="usertask3"
      targetRef="endevent1"></sequenceFlow>
  </process>
  <bpmndi:BPMNDiagram id="BPMNDiagram_process1">
    <bpmndi:BPMNPlane bpmnElement="process1" id="BPMNPlane_process1">
      <bpmndi:BPMNShape bpmnElement="startevent1"
        id="BPMNShape_startevent1">
        <omgdc:Bounds height="35" width="35" x="150" y="190"></omgdc:Bounds>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="usertask1" id="BPMNShape_usertask1">
        <omgdc:Bounds height="55" width="105" x="230" y="180"></omgdc:Bounds>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="usertask3" id="BPMNShape_usertask3">
        <omgdc:Bounds height="55" width="105" x="380" y="180"></omgdc:Bounds>
      </bpmndi:BPMNShape>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>
  
```

```

        </bpmndi:BPMNShape>
        <bpmndi:BPMNShape bpmnElement="endevent1" id="BPMNShape_endevent1">
            <omgdc:Bounds height="35" width="35" x="530" y="190"></omgdc:Bounds>
        </bpmndi:BPMNShape>
        <bpmndi:BPMNEdge bpmnElement="flow1" id="BPMNEdge_flow1">
            <omgdi:waypoint x="185" y="207"></omgdi:waypoint>
            <omgdi:waypoint x="230" y="207"></omgdi:waypoint>
        </bpmndi:BPMNEdge>
        <bpmndi:BPMNEdge bpmnElement="flow2" id="BPMNEdge_flow2">
            <omgdi:waypoint x="335" y="207"></omgdi:waypoint>
            <omgdi:waypoint x="380" y="207"></omgdi:waypoint>
        </bpmndi:BPMNEdge>
        <bpmndi:BPMNEdge bpmnElement="flow3" id="BPMNEdge_flow3">
            <omgdi:waypoint x="485" y="207"></omgdi:waypoint>
            <omgdi:waypoint x="530" y="207"></omgdi:waypoint>
        </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
</bpmndi:BPMNDiagram>
</definitions>

```

代码清单 2-2 中为一份流程描述文件，该文件中的 **process** 元素用于描述流程信息，而 **bpmndi:BPMNDiagram** 元素则用于描述这些流程节点的位置信息。代码清单 2-2 中，定义了两个 **userTask** 元素，分别表示图 2-26 中的两个用户任务。

注：在本书的代码清单中，为了减少篇幅，一般情况下不会将这些流程节点的位置信息配置贴出。

3.2.5 加载流程文件与启动流程

有了流程引擎的配置文件和流程文件后，就可以编写代码启动流程引擎并加载该流程文件，运行类如代码清单 2-3 所示。

代码清单 2-3: codes\02\first\src\org\crazyit\activiti\First.java

```

public class First {
    public static void main(String[] args) {
        // 创建流程引擎
        ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
        // 得到流程存储服务组件
        RepositoryService repositoryService = engine.getRepositoryService();
        // 得到运行时服务组件
        RuntimeService runtimeService = engine.getRuntimeService();
        // 获取流程任务组件
        TaskService taskService = engine.getTaskService();
        // 部署流程文件
        repositoryService.createDeployment()
            .addClasspathResource("bpmn/First.bpmn").deploy();
        // 启动流程
        runtimeService.startProcessInstanceByKey("process1");
        // 查询第一个任务
        Task task = taskService.createTaskQuery().singleResult();
        System.out.println("第一个任务完成前，当前任务名称: " + task.getName());
        // 完成第一个任务
        taskService.complete(task.getId());
    }
}

```

```
// 查询第二个任务
task = taskService.createTaskQuery().singleResult();
System.out.println("第二个任务完成前，当前任务名称：" + task.getName());
// 完成第二个任务（流程结束）
taskService.complete(task.getId());
task = taskService.createTaskQuery().singleResult();
System.out.println("流程结束后，查找任务：" + task);
// 退出
System.exit(0);
}
}
```

代码清单 2-3 中，使用 **ProcessEngines** 类加载默认的流程引擎配置文件（**activiti.cfg.xml**），再获取 **Activiti** 的各个服务组件的实例，**RepositoryService** 主要用于管理流程的资源（请见第 7 章），**RuntimeService** 主要用于进行流程运行时的流程管理（请见第 9 章），**TaskService** 主要用于管理流程任务（请见第 8 章）。代码清单 2-3 中使用 **RepositoryService** 部署流程文件，使用 **RuntimeService** 启动流程，然后使用 **TaskService** 进行流程任务查找，并对结束查找到的任务。关于这些服务对象的使用以及流程文件的定义，将会在本书后面章节中详细讲解。运行代码清单 2-3，输出结果如下：

```
第一个任务完成前，当前任务名称：Expense Request
第二个任务完成前，当前任务名称：Handle Request
流程结束后，查找任务：null
```

3.3 小结

工欲善其事，必先利其器。本章主要讲解进行 **Activiti** 开发的准备工作，包括 **Activiti** 的下载和安装，**Activiti** 开发环境的搭建，带领读者试用了 **Activiti** 的官方应用，并且开发了第一个 **Activiti** 程序。本章作为 **Activiti** 开发实践的第一课，学习完本章内容后，将有助于提升学习信心。从下一章开始，我们将一起遨游 **Activiti** 与 **BPMN2.0** 的世界。

4 配置文件读取与数据源配置

要点

- 掌握 **Activiti** 的配置文件读取方式
- 掌握 **Activiti** 的数据源配置

4.1 流程引擎配置对象

ProcessEngineConfiguration 对象代表一个 Activiti 流程引擎的全部配置，该类提供一系列创建 ProcessEngineConfiguration 实例的静态方法，这些方法用于读取和解析相应的配置文件，并返回 ProcessEngineConfiguration 的实例。除这些静态方法外，该类为其他可配置的引擎属性提供相应的 setter 和 getter 方法。本小节主要讲解如何使用这些静态方法创建 ProcessEngineConfiguration 实例。

4.1.1 读取默认的配置文

ProcessEngineConfiguration 的 createProcessEngineConfigurationFromResourceDefault 方法，使用 Activiti 默认的方式创建 ProcessEngineConfiguration 的实例。这里所说的默认方式，是指由 Activiti 决定读取配置文件的位置、文件的名称和配置 bean 的名称这些信息。Activiti 默认到 ClassPath 下读取名为“activiti.cfg.xml”的 Activiti 配置文件，启动并获取名称为“processEngineConfiguration”的 bean 的实例。解析 XML 与创建该 bean 实例的过程，由 Spring 代为完成。

使用过 Spring 的朋友可以知道，只需要指定 Spring 的 XML 配置文件，创建相应的 BeanFactory 实例，再通过 getBean(bean 名称)方法即可获取相应对象的实例，ProcessEngineConfiguration 使用 Spring 框架的 DefaultListableBeanFactory 作为 BeanFactory。

代码清单 4-1 使用 createProcessEngineConfigurationFromResourceDefault 方法创建 ProcessEngineConfiguration 实例。

代码清单 4-1: codes\04\4.1\create-default\src\org\crazyit\activiti\CreateDefault.java

```
//使用 Activiti 默认的方式创建 ProcessEngineConfiguration
ProcessEngineConfiguration config =
ProcessEngineConfiguration.createProcessEngineConfigurationFromResourceDefault();
```

代码清单 4-1 中，Activiti 默认到 ClassPath 下读取 activiti.cfg.xml 文件，如果找不到该配置文件则抛出 FileNotFoundException，如果找不到名称为 processEngineConfiguration 的 bean，则抛出 org.springframework.beans.factory.NoSuchBeanDefinitionException，本例中的 activiti.cfg.xml 内容如代码清单 4-2 所示。

代码清单 4-2: codes\04\4.1\create-default\resource\activiti.cfg.xml

```
<!-- 只配置相应的数据库属性 -->
<bean id="processEngineConfiguration"
      class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/act" />
  <property name="jdbcDriver" value="com.mysql.jdbc.Driver" />
  <property name="jdbcUsername" value="root" />
  <property name="jdbcPassword" value="123456" />
</bean>
```

此处需要注意的是，代码清单 4-2 中所使用的 ProcessEngineConfiguration 为 StandaloneProcessEngineConfiguration 类，ProcessEngineConfiguration 为抽象类，不能直接作为 bean 的 class 进行配置，ProcessEngineConfiguration 的子类将在下面章节进行描述。

4.1.2 读取自定义的配置文件

在 4.1.1 章节中可知，默认情况下 Activiti 将到 ClassPath 下读取 activiti.cfg.xml 文件，如果希望 Activiti 读取另外名称的配置文件，可以使用 createProcessEngineConfigurationFromResource 方法创建 ProcessEngineConfiguration，该方法参数为一个字符串对象，调用该方法时，需要告诉 Activiti 配置文件位置。代码清单 4-3 调用 createProcessEngineConfigurationFromResource(String resource)方法。

代码清单 4-3：
codes\04\4.1\create-resource\src\org\crazyit\activiti\CreateFromResource_1.java

```
// 指定配置文件创建 ProcessEngineConfiguration
ProcessEngineConfiguration config = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResource("my-activiti1.xml");
```

代码清单 4-3 中，Activiti 会到 ClassPath 下查找 my-activiti1.xml 配置文件，并创建名称为“processEngineConfiguration”的 bean，此处创建 bean 的过程与 4.1.1 中描述一致，my-activiti1.xml 文件的配置内容与代码清单 4-2 一致。

ProcessEngineConfiguration 中还有一个 createProcessEngineConfigurationFromResource 的重载方法，该方法需要提供两个参数来创建 ProcessEngineConfiguration，第一个参数为 Activiti 配置文件的位置，第二个参数为创建 bean 的名称。代码清单 4-4 调用 createProcessEngineConfigurationFromResource(String resource, String beanName)的方法。

代码清单 4-4：
codes\04\4.1\create-resource\src\org\crazyit\activiti\CreateFromResource_2.java

```
// 指定配置文件创建 ProcessEngineConfiguration
ProcessEngineConfiguration config = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResource(
        "my-activiti2.xml", "test");
System.out.println(config.getProcessEngineName());
```

代码清单 4-4 中，告诉 Activiti 需要到 ClassPath 下查找 my-activiti2.xml 文件，并且创建名字为“test”的 bean。如果找不到名称为“test”的 bean，则抛出 NoSuchBeanDefinitionException，以下的代码会抛该异常，因为找不到名称为 test2 的 bean。

```
ProcessEngineConfiguration config = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResource(
        "my-activiti2.xml", "test2");
```

4.1.3 读取输入流的配置

ProcessEngineConfiguration 中提供了一个 createProcessEngineConfigurationFromInputStream 方法，该方法使得 Activiti 配置文件的加载不再局限于项目的 ClassPath，只要得到配置文件的输入流，即可创建 ProcessEngineConfiguration。

同样的，createProcessEngineConfigurationFromInputStream 方法也提供了两个重载的方法，可以指定在解析 XML 时 bean 的名称。代码清单 4-5 中使用

createProcessEngineConfigurationFromInputStream 方法（没有指定 bean 名称）。

代 码 清 单 4-5 :

codes\04\4.1\create-stream\src\org\crazyit\activiti\CreateInputStream.java

```
File file = new File("resource/input-stream.xml");
// 得到文件输入流
InputStream fis = new FileInputStream(file);
// 使用 createProcessEngineConfigurationFromInputStream 方法创建
ProcessEngineConfiguration
ProcessEngineConfiguration config = ProcessEngineConfiguration
.createProcessEngineConfigurationFromInputStream(fis);
```

4.1.4 使用 createStandaloneInMemProcessEngineConfiguration 方法

使用该方法创建 ProcessEngineConfiguration，并不需要指定任何参数，该方法直接返回一个 StandaloneInMemProcessEngineConfiguration 实例，该类为 ProcessEngineConfiguration 的子类。使用该方法创建 ProcessEngineConfiguration，并不会读取任何的 Activiti 配置文件，这意味着流程引擎配置的全部属性，都会使用默认值，与其他子类不一样的是，创建的 StandaloneInMemProcessEngineConfiguration 实例，只特别指定了 databaseSchemaUpdate 属性和 jdbcUrl 属性，详情请见代码清单 4-6。

代码清单 4-6:

codes\04\4.1\create-standalone-inmem\src\org\crazyit\activiti\CreateStandaloneInMem.java

```
ProcessEngineConfiguration config = ProcessEngineConfiguration
.createStandaloneInMemProcessEngineConfiguration();
// 值为 create-drop
System.out.println(config.getDatabaseSchemaUpdate());
// 值为 jdbc:h2:mem:activiti
System.out.println(config.getJdbcUrl());
```

该方法不需要读取任何的配置文件，ClassPath 下也没有任何的 Activiti 配置文件，如果需要改变相关的配置，可以调用 ProcessEngineConfiguration 中相应的 setter 方法进行修改。

方法 createStandaloneInMemProcessEngineConfiguration 返回的是一个 org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration 实例，如果使用配置文件的方式创建 ProcessEngineConfiguration，可以将该类配置为 bean 的 class，但使用时需要注意该类中属性的默认值。

注：ProcessEngineConfiguration 的各个属性及其作用，将在下面章节中逐一描述。

4.1.5 使用 createStandaloneProcessEngineConfiguration 方法

与 4.1.4 类似的是，createStandaloneProcessEngineConfiguration 方法返回的是一个 StandaloneProcessEngineConfiguration 实例，并且需要注意的是，4.1.4 中的 StandaloneInMemProcessEngineConfiguration 类是本小节 StandaloneProcessEngineConfiguration 类的子类。代码清单 4-7 中输出了 StandaloneProcessEngineConfiguration 类的 databaseSchemaUpdate 和 jdbcUrl 的值。

代 码 清 单 4-7 :

codes\04\4.1\create-standalone\src\org\crazyit\activiti\CreateStandalone.java

```
ProcessEngineConfiguration config = ProcessEngineConfiguration
    .createStandaloneProcessEngineConfiguration();
// 默认值为 false
System.out.println(config.getDatabaseSchemaUpdate());
// 默认值为 jdbc:h2:tcp://localhost/~/activiti
System.out.println(config.getJdbcUrl());
```

从代码清单 4-7 中可以明显看出，父类 `StandaloneProcessEngineConfiguration` 的 `databaseSchemaUpdate` 和 `jdbcUrl` 属性值分别为 “false” 和 “jdbc:h2:tcp://localhost/~/activiti”，而其子类 `StandaloneInMemProcessEngineConfiguration`（4.1.4 章节），这两个属性值分别为 “create-drop” 和 “jdbc:h2:mem:activiti”。

4.2 数据源配置

Activiti 在启动时，会读取数据源配置，用于对数据库进行相应的操作。在前面章节中得知，Activiti 会先读取配置文件，然后取得配置的 bean，并对其进行初始化，本小节将讲解配置 bean 的一系列参数，并了解其作用。

4.2.1 Activiti 支持的数据库

Activiti 默认 H2 数据库，H2 是一个开源的嵌入式数据库，使用 Java 语言编写。使用 H2 数据库并不需要另外安装服务器或者客户端，只需要提供一个 jar 包即可使用。在实际的企业应用中，很少会使用这种轻量级的嵌入式数据库，因此 H2 数据更适合使用于单元测试。除 H2 数据库，Activiti 还为以下的数据库提供支持：

- ❑ **MySQL**：主流数据库之一，它是一个开源的小型关系型数据库，由它体积小、速度快，得到相当多开发者的青睐，并且最重要的是，它是免费的。
- ❑ **Oracle**：目前世界上最流行的商业数据库，价格昂贵，但是它高效的性能、可靠的数据管理，仍令不少企业心甘情愿为其掏钱。
- ❑ **Postgres**：PostgreSQL 是另外一款开源的数据库。
- ❑ **DB2**：由 IBM 公司研发的一款关系型数据库，其良好的伸缩性、数据库的高效性，让它成为继 Oracle 之后，又一应用广泛的商业数据库。
- ❑ **MSSQL**：微软研发的一款数据库产品，目前也支持在 Linux 下使用。

4.2.2 Activiti 与 Spring

Spring 是目前非常流行的一个轻量级 J2EE 框架，它提供了一套轻量级的企业应用解决方案，它包括 IoC 容器、AOP 面向切面技术以及 Web MVC 框架等。

使用 Activiti 的项目，并不意味着一定要使用 Spring，Activiti 可以在没有 Spring 的环境中使用。虽然 Activiti 并不需要使用 Spring 环境，但是 Activiti 在创建流程引擎时，使用了 Spring 的 XML 解析与依赖注入功能，`ProcessEngineConfiguration` 对应的配置，即为 Spring 中的一个 bean。

使用过 Spring 的读者，看到 ProcessEngineConfiguration 对应的配置，会感到非常熟悉，没有使用过的读者也不必感到气馁，因为 Activiti 也可以在一个完全没有 Spring 的环境中运行。

4.2.3 JDBC 配置

JDBC 连接数据库，需要使用 jdbc url、jdbc 驱动、数据库用户名和密码，以下代码为连接 MySQL 的配置：

```
<bean id="processEngineConfiguration"
class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
  <!-- JDBC url -->
  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/act" />
  <!-- JDBC 驱动 -->
  <property name="jdbcDriver" value="com.mysql.jdbc.Driver" />
  <!-- 数据库用户名 -->
  <property name="jdbcUsername" value="root" />
  <!-- 数据库密码 -->
  <property name="jdbcPassword" value="123456" />
</bean>
```

以上代码配置一个 bean，表示一个 ProcessEngineConfiguration，并且使用“设值注入”的方式将四个数据库属性设置到该 bean 中，换言之，该 ProcessEngineConfiguration 类中，肯定有相应属性的 setter 方法。该 bean 的实现类以及这些属性，将在下面章节中作详细讲解。

4.2.4 DBCP 数据源配置

DBCP 是 Apache 提供的一个数据库连接池。ProcessEngineConfiguration 中提供了一个 dataSource 属性，如果用户不希望将 JDBC 的相关连接属性交给 Activiti，可以自己创建数据库连接，然后通过这个 dataSource 属性设置到 ProcessEngineConfiguration 中。为 Activiti 的 ProcessEngineConfiguration 设置 dataSource，可以采用配置或者编写代码的方式。代码清单 4-8 为使用配置方式使用 DBCP 数据源。

代码清单 4-8: codes\04\4.2\ds-dbcpl\resource\dbcpl-config.xml

```
<!-- 使用 DBCP 数据源 -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost:3306/act" />
  <property name="username" value="root" />
  <property name="password" value="123456" />
</bean>
<bean id="processEngineConfiguration"
class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
  <property name="dataSource" ref="dataSource" />
</bean>
```

代码清单 4-8 中的粗体部分，配置了一个 DBCP 的 dataSource bean，然后在 processEngineConfiguration 的 bean 中注入该 dataSource。在初始化流程引擎配置时，只需根据情况调用 ProcessEngineConfiguration 的 createXXX 方法即可，如以下代码所示：

```
// 读取 dbcp-config.xml 配置
ProcessEngineConfiguration config = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResource("dbcp-config.xml");
// 能正常输出，即完成配置
DataSource ds = config.getDataSource();
// 查询数据库元信息，如果能查询则表示连接成功
ds.getConnection().getMetaData();
// 结果为 org.apache.commons.dbcp.BasicDataSource
System.out.println(ds.getClass().getName());
```

本例使用了 `createProcessEngineConfigurationFromResource` 方法读取 Activiti 的配置文件。除使用配置外，也可以通过编码方式设置相应的 `dataSource`，只需要先创建一个 `DataSource` 对象，然后设置到 `ProcessEngineConfiguration` 中即可，代码清单 4-9 为通过编码方式设置 DBCP 数据源。

代码清单 4-9: codes\04\4.2\ds-dbc\src\org\crazyit\activiti\DBCPCongig.java

```
// 创建 DBCP 数据源
BasicDataSource ds = new BasicDataSource();
// 设置 JDBC 连接的各个属性
ds.setUsername("root");
ds.setPassword("123456");
ds.setUrl("jdbc:mysql://localhost:3306/act");
ds.setDriverClassName("com.mysql.jdbc.Driver");
// 验证是否连接成功
ds.getConnection().getMetaData();
// 读取 Activiti 配置文件
ProcessEngineConfiguration config = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResource("dbcp-coding.xml");
// 为 ProcessEngineConfiguration 设置 dataSource 属性
config.setDataSource(ds);
System.out.println(config.getDataSource());
```

代码清单 4-9 中，先创建 `DataSource` 对象，然后为该对象设置相应的数据库连接属性，然后读取 Activiti 配置文件，得到 `ProcessEngineConfiguration` 对象，并将 `DataSource` 设置到该对象中。`ProcessEngineConfiguration` 的 bean 配置不需要设置任何属性：

```
<!-- 不初始化任何属性 -->
<bean id="processEngineConfiguration"
    class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
</bean>
```

在笔者成书时，DBCP 项目已经发展到 2.1 版本，本小节的案例就是基于该版本。

4.2.5 C3P0 数据源配置

与 DBCP 类似，C3P0 也是一个开源的数据库连接池，它们都被广泛的应用到开源项目以及企业应用中。与 DBCP 类似，可以在 Activiti 中使用 C3P0 数据源，配置方式大致相同，代码清单 4-10 为 C3P0 bean 的配置。

代码清单 4-10: codes\04\4.2\ds-c3p0\resource\config\c3p0-config.xml

```
<!-- 使用 C3P0 数据源 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/act" />
    <property name="user" value="root" />
```

```

        <property name="password" value="123456" />
    </bean>

    <bean id="processEngineConfiguration"
        class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
        <property name="dataSource" ref="dataSource" />
    </bean>

```

注：此处需要注意的是，DBCP 与 C3P0 的属性名称不一样，可以到两个数据源的官方文档查看更详细的配置。

除了配置方式外，也可以像 DBCP 一样使用编码方式创建数据源，设置方式基本与 DBCP 一致，只是创建 DataSource 实例的方式不一样而已。代码清单 4-11 展示如何创建 C3P0 数据源。

代码清单 4-11: codes\04\4.2\ds-c3p0\src\org\crazyit\activiti\C3P0Coding.java:

```

// 创建 C3P0 数据源
ComboPooledDataSource ds = new ComboPooledDataSource();
// 设置 JDBC 连接的各个属性
ds.setUser("root");
ds.setPassword("123456");
ds.setJdbcUrl("jdbc:mysql://localhost:3306/act");
ds.setDriverClass("com.mysql.jdbc.Driver");
// 验证是否连接成功
ds.getConnection().getMetaData();
// 读取 Activiti 配置文件
ProcessEngineConfiguration config = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResource("config/c3p0-coding.xml");
// 为 ProcessEngineConfiguration 设置 dataSource 属性
config.setDataSource(ds);
System.out.println(config.getDataSource());

```

4.2.6 Activiti 其他数据源配置

如果不使用第三方数据源，直接使用 Activiti 提供的数据库属性。Activiti 默认使用的是 myBatis 的数据连接池，因此 ProcessEngineConfiguration 中也提供了一些 MyBatis 的配置：

- ❑ jdbcMaxActiveConnections: 在数据库连接池内最大的活跃连接数，默认值为 10。
- ❑ jdbcMaxIdleConnections: 连接池最大的空闲连接数。
- ❑ jdbcMaxCheckoutTime: 当连接池内的连接耗尽，外界向连接池请求连接时，创建连接的等待时间，单位为毫秒，默认值为 20000，即 20 秒。
- ❑ jdbcMaxWaitTime: 当整个连接池需要重新获取连接的时候，设置等待时间，单位为毫秒，默认值为 20000，即 20 秒。

4.2.7 数据库策略配置

ProcessEngineConfiguration 提供了 databaseSchemaUpdate 属性，该项可以设置流程引擎启动和关闭时数据库执行的策略。Activiti 的官方文档中，databaseSchemaUpdate 有以下三个值：

- ❑ **false**: **false** 为默认值，设置为该值后，Activiti 在启动时，会对比数据库表中保存的版本，如果没有表或者版本不匹配时，将在启动时抛出异常。
- ❑ **true**: 设置为该值后，Activiti 会对数据库中所有的表进行更新，如果表不存在，则 Activiti 会自动创建。
- ❑ **create-drop**: Activiti 启动时，会执行数据库表的创建操作，在 Activiti 关闭时，执行数据库表的删除操作。

代码清单 4-12 将 databaseSchemaUpdate 配置为 false。

代码清单 4-12: codes\04\4.2\schema-update\resource\schemaUpdate-false.xml

```
<!-- 将 databaseSchemaUpdate 设置为 false -->
<bean id="processEngineConfiguration"
      class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/act" />
  <property name="jdbcDriver" value="com.mysql.jdbc.Driver" />
  <property name="jdbcUsername" value="root" />
  <property name="jdbcPassword" value="123456" />
  <property name="databaseSchemaUpdate" value="false"/>
</bean>
```

使用以下代码启动 Activiti 流程引擎：

```
//读取 Activiti 配置
ProcessEngineConfiguration config = ProcessEngineConfiguration

.createProcessEngineConfigurationFromResource("schemaUpdate-false.xml");
//启动 Activiti
config.buildProcessEngine();
```

以上代码的粗体部分，如果没有数据表，则会抛出异常。这里需要注意的是，如果想看到抛出异常的效果，需要将相应数据库里面的表全部删除。如果想执行数据库表结构更新，可以将该配置设置为 **true**，将全部数据库表删除后，再启动 Activiti，即可看到 Activiti 已经建好全部的表，值为 **true** 的配置与代码在此不再赘述，读者可看以下配置文件与 Java 类：

- ❑ 配置文件: codes\04\4.2\schema-update\resource\schemaUpdate-true.xml
- ❑ Java 类: codes\04\4.2\schema-update\src\org\crazyit\activiti\DatabaseSchemaUpdateTrue.java

将 databaseSchemaUpdate 设置为 create-drop 后，Activiti 会先检查数据表是否存在，如果表已经存在，则抛出异常并停止创建流程引擎。代码清单 4-13 使用 create-drop 属性启动 Activiti。

代码清单 4-13:

codes\04\4.2\schema-update\src\org\crazyit\activiti\DatabaseSchemaUpdateCreateDrop.java

```
// 读取 Activiti 配置
ProcessEngineConfiguration config = ProcessEngineConfiguration

.createProcessEngineConfigurationFromResource("schemaUpdate-create-drop.xml");
// 启动 Activiti
ProcessEngine engine = config.buildProcessEngine();
// 关闭流程引擎
engine.close();
```

注意代码清单 4-13 中的粗体部分，如果想要 Activiti 执行“drop”操作，必须要调用 ProcessEngine 的 close 方法，否则将不会删除表。一般情况下，将 databaseSchemaUpdate

配置为 `create-drop`，更适合在单元测试中使用。

除了 `false`、`true` 和 `create-drop` 三个值外，`databaseSchemaUpdate` 还有一个 `drop-create` 值，跟 `create-drop` 类似，`drop-create` 会在流程引擎启动时，先将原来全部的数据表删除，再进行创建，与 `create-drop` 不同的是，不管是否调用 `ProcessEngine` 的 `close` 方法，都会执行 `create` 操作。同样地，该值在单元测试中使用比较合适，在流程引擎初始化时将原有的数据删除，在实际应用中，此举会带来较大的风险，Activiti 的官方文档并没有提供该项配置，读者知道即可。

注：使用各种方法读取 Activiti 配置，均不会创建数据库表，Activiti 的数据库表只会在流程引擎创建的时候，才会按照配置的策略进行创建（代码清单 4-12 的粗体部分）。

4.2.8 databaseType 配置

根据前一小节得知，将 `databaseSchemaUpdate` 设置为 `create-drop` 或者 `drop-create` 时，Activiti 在启动和初始化时，会执行相应的创建表和删除表操作，Activiti 支持多种数据库，每种数据库的创建表与删除表的语法有可能不一样，因此，需要指定 `databaseType` 属性，来告诉 Activiti，目前使用了何种数据库（当然，如果设置 `true` 而数据库中没有表的话，也需要知道使用哪种数据库）。`databaseType` 属性支持这些值：`h2`、`mysql`、`oracle`、`postgres`、`mssql`、`db2`，没有指定值时，`databaseType` 为 `null`。指定 `databaseType` 属性，目的是为了确定执行创建（或删除）表的 SQL 脚本。

代码清单 4-14，将该属性设置为 `oracle`。

代码清单 4-14: codes\04\4.2\db-type\resource\database-type.xml

```
<!-- 将 databaseType 设置为 oracle -->
<bean id="processEngineConfiguration"
      class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/act" />
  <property name="jdbcDriver" value="com.mysql.jdbc.Driver" />
  <property name="jdbcUsername" value="root" />
  <property name="jdbcPassword" value="123456" />
  <property name="databaseSchemaUpdate" value="create-drop"/>
  <b>property name="databaseType" value="oracle"/>
</bean>
```

使用以上配置，然后启动和关闭 Activiti，会抛出 MySQL 异常，因为 Activiti 会根据该值去使用 Oracle 创建表和删除表的脚本，Oracle 的 SQL 脚本在 MySQL 上面执行，肯定会出错。

实际上，可以根本不需要指定该属性，Activiti 就可以知道使用的是哪种数据库，因为配置数据源时就提供了 JDBC 连接属性给 Activiti，根据这些属性创建 JDBC 连接，得到 `Connection` 对象后，可以调用 `getMetaData` 方法获取当前数据库的元数据，完全可以判断出当前所使用的数据库。的确，Activiti 也是这样做的，但是为什么另外提供一个 `databaseType` 属性如此多此一举呢？笔者认为，Activiti 为防止适配数据库类型出现异常，就提供多一个这样的值来给使用者选择，确保能适配到准确的数据库类型。

注：没有配置 `databaseType` 属性，Activiti 会使用 `Connection` 的 `getMetaData` 方法获取数据库元数据，但是一旦配置了 `databaseType` 属性，将会以该值为准。

本文节选自《疯狂 Workflow 讲义（第2版）》。

5 流程引擎的创建

本章要点

- 流程引擎的创建方法
- 流程引擎的初始化、销毁以及关闭
- Activiti 的服务组件简述

前面章节，讲述了 Activiti 的配置，根据这些配置，可以创建相应的流程引擎。Activiti 提供了多种创建流程引擎的方式供研发人员选择，可以通过 `ProcessEngineConfiguration` 的 `buildProcessEngine` 方法，也可以使用 `ProcessEngines` 的 `init` 方法来创建 `ProcessEngine` 实例，可以根据项目的不同需要来选择不同的创建方式。

5.1 ProcessEngineConfiguration 的 buildProcessEngine 方法

前面的章节，使用 `ProcessEngineConfiguration` 的 `create` 方法可以得到 `ProcessEngineConfiguration` 的实例。`ProcessEngineConfiguration` 中提供了一个 `buildProcessEngine` 方法，该方法返回一个 `ProcessEngine` 实例。代码清单 5-1 中使用 `buildProcessEngine` 方法。

代 码 清 单 5-1 :
codes\05\5.1\build-engine\src\org\crazyit\activiti\BuildProcessEngine.java

```
// 读取配置
ProcessEngineConfiguration config = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResource("build_engine.xml");
// 创建 ProcessEngine
ProcessEngine engine = config.buildProcessEngine();
```

得到流程引擎的相关配置后，`buildProcessEngine` 方法会根据这些配置，初始化流程引擎的相关服务和对象，包括数据源、事务、拦截器、服务组件等等。这个流程引擎的初始化过程，实际上也可以看作是一个配置检查的过程。

5.2 ProcessEngines 对象

除了 `ProcessEngineConfiguration` 的 `buildProcessEngine` 方法外，`ProcessEngines` 类提供了创建 `ProcessEngineConfiguration` 实例的方法。`ProcessEngines` 是一个创建流程引擎与关闭流程引擎的工具类，所有创建（包括其他方式创建）的 `ProcessEngine` 实例均被注册到 `ProcessEngines` 中。这里所说的注册，实际上是 `ProcessEngines` 类中维护一个 `Map` 对象，该对象的 `key` 为 `ProcessEngine` 实例的名称，`value` 为 `ProcessEngine` 的实例，当向 `ProcessEngines` 注册 `ProcessEngine` 实例时，实际上是调用 `Map` 的 `put` 方法，将该

实例缓存到 Map 中。

5.2.1 init 与 getDefaultProcessEngine 方法

ProcessEngines 的 init 方法，会读取 Activiti 的默认配置文件，然后将创建的 ProcessEngine 实例缓存到 Map 中。这里所说的默认配置文件，一般情况下是指 ClassPath 下的 activiti.cfg.xml，如果有与 Spring 进行整合，则读取 ClassPath 下的 activiti-context.xml 文件。代码清单 5-2 为调用 ProcessEngines 的 init 方法。

代码清单 5-2: codes\05\5.1\init-engine\src\org\crazyit\activiti\Init.java

```
// 初始化 ProcessEngines 的 Map,
// 再加载 Activiti 默认的配置文​​件（classpath 下的 activiti.cfg.xml 文件）
// 如果与 Spring 整合，则读取 classpath 下的 activiti-context.xml 文件
ProcessEngines.init();
// 得到 ProcessEngines 的 Map
Map<String, ProcessEngine> engines = ProcessEngines.getProcessEngines();
System.out.println(engines.size());
System.out.println(engines.get("default"));
```

调用了 init 方法后，Activiti 会根据默认配置创建 ProcessEngine 实例，此时 Map 的 key 值为“default”，代码清单输出结果如下：

```
1
org.activiti.engine.impl.ProcessEngineImpl@a23610
```

此处的 init 方法并不会返回任何的 ProcessEngine 实例，该方法只会加载 classpath 下全部的 Activiti 配置文件并且将创建的 ProcessEngine 实例保存到 ProcessEngines 中。如果需要得到相应的 ProcessEngine 实例，可以使用 getProcessEngines 方法拿到 ProcessEngines 中全部的 ProcessEngine 实例，getProcessEngines 返回的是一个 Map，只需要根据 ProcessEngine 的名称，即可得到相应的 ProcessEngine 实例。

另外，ProcessEngines 提供了一个 getDefaultProcessEngine 方法，用于返回 key 为“default”的 ProcessEngine 实例，该方法会判断 ProcessEngines 是否进行初始化，如果没有，则会调用 init 方法进行初始化。

5.2.2 registerProcessEngine 和 unregister 方法

注册和注销方法，registerProcessEngine 方法向 ProcessEngines 中注册一个 ProcessEngine 实例，unregister 方法则向 ProcessEngines 中注销一个 ProcessEngine 实例。注册与注销 ProcessEngine 实例，均会根据该 ProcessEngine 实例的名称进行操作，因为 Map 的 key 使用的是 ProcessEngine 的名称。使用代码清单 5-3 读取自定义配置，然后创建 ProcessEngine 实例，并注册到 ProcessEngines 中，最后调用 unregister 方法注销该实例。

代码清单 5-3: codes\05\5.2\register-engine\src\org\crazyit\activiti\Register.java

```
//读取自定义配置
ProcessEngineConfiguration config = ProcessEngineConfiguration.
    createProcessEngineConfigurationFromResource("register.xml");
//创建 ProcessEngine 实例
ProcessEngine engine = config.buildProcessEngine();
//获取 ProcessEngine 的 Map
```

```
Map<String, ProcessEngine> engines = ProcessEngines.getProcessEngines();
System.out.println("注册后引擎数: " + engines.size());
//注销 ProcessEngine 实例
ProcessEngines.unregister( engine);
System.out.println("调用 unregister 后引擎数: " + engines.size());
```

代码清单 5-3 中，使用 `ProcessEngineConfiguration` 的 `buildProcessEngine` 方法，即将创建的 `ProcessEngine` 实例注册到 `ProcessEngines` 中，并不需要再次调用 `registerProcessEngine` 方法。使用了 `ProcessEngineConfiguration` 的 `buildProcessEngine` 方法后，可以获取 `ProcessEngines` 的 `Map`，打印出 `ProcessEngine` 的实例数，最后调用注销方法，再打印出实例数，代码清单 5-3 的结果如下：

```
注册后引擎数: 1
调用 unregister 后引擎数: 0
```

默认情况下，创建的 `ProcessEngine` 名称为“default”，如果需要设置名称，可调用引擎配置类的 `setProcessEngineName` 方法。`ProcessEngines` 里面维护的 `Map` 对象，key 就是引擎的名称。

注意：`unregister` 方法只是单纯的将 `ProcessEngine` 实例从 `Map` 移除，并不会调用 `ProcessEngine` 的 `close` 方法。

5.2.3 retry 方法

如果 `Activiti` 在加载配置文件时出现异常，可以调用 `ProcessEngines` 的 `retry` 方法重新加载配置文件，重新创建 `ProcessEngine` 实例并加入到 `Map` 中。代码清单 5-4 使用 `retry` 方法加载配置文件。

代码清单 5-4: `codes\05\5.2\retry-engine\src\org\crazyit\activiti\Retry.java`

```
//得到资源文件的 URL 实例
ClassLoader cl = Retry.class.getClassLoader();
URL url = cl.getResource("retry.xml");
//调用 retry 方法创建 ProcessEngine 实例
ProcessEngineInfo info = ProcessEngines.retry(url.toString());
//得到流程实例保存对象
Map<String, ProcessEngine> engines = ProcessEngines.getProcessEngines();
System.out.println("调用 retry 方法后引擎数: " + engines.size());
```

最后输出结果为 1，成功使用 `retry` 方法加载资源，创建 `ProcessEngine` 实例。在此需要注意的是，`retry` 方法返回的是一个 `ProcessEngineInfo` 实例。

5.2.4 destroy 方法

`ProcessEngines` 的 `destroy` 方法，顾名思义，是对其所有维护的 `ProcessEngine` 实例进行销毁，并且在销毁时，会调用全部 `ProcessEngine` 实例的 `close` 方法。代码清单 5-5 使用 `destroy` 方法。

代码清单 5-5: `codes\05\5.2\destroy-engine\src\org\crazyit\activiti\Destroy.java`

```
// 进行初始化并且返回默认的 ProcessEngine 实例
ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
System.out.println("调用 getDefaultProcessEngine 方法后引擎数量: "
    + ProcessEngines.getProcessEngines().size());
// 调用销毁方法
```

```

ProcessEngines.destroy();
// 最终结果为 0
System.out.println("调用 destroy 方法后引擎数量: "
    + ProcessEngines.getProcessEngines().size());

// 得到资源文件的 URL 实例
ClassLoader cl = Destroy.class.getClassLoader();
URL url = cl.getResource("activiti.cfg.xml");
// 调用 retry 方法创建 ProcessEngine 实例
ProcessEngines.retry(url.toString());
System.out.println("只调用 retry 方法后引擎数量: "
    + ProcessEngines.getProcessEngines().size());
// 调用销毁方法，没有效果
ProcessEngines.destroy();
System.out.println("调用 destory 无效果，引擎数量: "
    + ProcessEngines.getProcessEngines().size());

```

ProcessEngine 实例销毁的前提是 ProcessEngines 的初始化状态为 true，如果为 false，则调用 destory 不会有效果。例如调用 retry 方法，再调用 destory 方法，则不会有销售效果，因为 retry 方法并没有设置初始化状态。代码清单 5-5 的输出结果：

```

调用 getDefaultProcessEngine 方法后引擎数量：1
调用 destroy 方法后引擎数量：0
只调用 retry 方法后引擎数量：1
调用 destory 无效果，引擎数量：1

```

在 destory 执行时，会调用全部 ProcessEngine 实例 close 方法，该方法会将异步执行器（AsyncExecutor）关闭，如果流程引擎配置的数据库策略为 create-drop，则会执行数据库表的删除操作。（数据库策略请见 4.2.7 章节）。

5.3 ProcessEngine 对象

在 Activiti 中，一个 ProcessEngine 实例代表一个流程引擎，ProcessEngine 保存着各个服务组件的实例，根据这些服务组件，可以操作流程实例、任务、系统角色等数据。本小节将简述 ProcessEngine 以及其维护的各个服务组件实例。

5.3.1 服务组件

当创建了流程引擎实例后，在 ProcessEngine 中会初始化一系列服务组件，这些组件提供了很多控制流程引擎的业务方法，它们就好像 J2EE 中的 Service 层，可以使用 ProcessEngine 中的 getXXXService 方法得到这些组件的实例。一个 ProcessEngine 主要有以下实例：

- ❑ RepositoryService: 提供一系列管理流程定义和流程部署的 API。
- ❑ RuntimeService: 在流程运行时对流程实例进行管理与控制。
- ❑ TaskService: 对流程任务进行管理，例如任务提醒、任务完成和创建任务等。
- ❑ IdentityService: 提供对流程角色数据进行管理的 API，这些角色数据包括用户组、用户及它们之间的关系。
- ❑ ManagementService: 提供对流程引擎进行管理和维护的服务。

- ❑ **HistoryService**: 对流程的历史数据进行操作，包括查询、删除这些历史数据。
- ❑ **DynamicBpmnService**: 使用该服务，可以不需要重新部署流程模型，就可以实现对流程模型的部分修改。

代码清单 5-6 中，展示了如何获取这些服务组件实例及它们的实现类。

代码清单 5-6: codes\05\5.3\engine-service\src\org\crazyit\activiti\GetService.java

```
//读取流程引擎配置
ProcessEngineConfiguration config = ProcessEngineConfiguration.
    createProcessEngineConfigurationFromResource("service.xml");
//创建流程引擎
ProcessEngine engine = config.buildProcessEngine();
//得到各个业务组件实例
RepositoryService repositoryService = engine.getRepositoryService();
RuntimeService runtimeService = engine.getRuntimeService();
TaskService taskService = engine.getTaskService();
IdentityService identityService = engine.getIdentityService();
ManagementService managementService = engine.getManagementService();
HistoryService historyService = engine.getHistoryService();
DynamicBpmnService dynamicBpmnService = engine.getDynamicBpmnService();
// 输入类名
System.out.println(repositoryService.getClass().getName());
System.out.println(runtimeService.getClass().getName());
System.out.println(taskService.getClass().getName());
System.out.println(identityService.getClass().getName());
System.out.println(managementService.getClass().getName());
System.out.println(historyService.getClass().getName());
System.out.println(dynamicBpmnService.getClass().getName());
```

代码清单 5-6 中的粗体部分，使用 **ProcessEngine** 得到各个服务组件实例，这些服务组件的详细使用，将会在第 6 章开始作详细的讲解，运行代码清单 5-6，可以看到每一个业务组件的实现类，实现类的命名规则为接口名称加 “Impl”：

```
org.activiti.engine.impl.RepositoryServiceImpl
org.activiti.engine.impl.RuntimeServiceImpl
org.activiti.engine.impl.TaskServiceImpl
org.activiti.engine.impl.IdentityServiceImpl
org.activiti.engine.impl.ManagementServiceImpl
org.activiti.engine.impl.HistoryServiceImpl
org.activiti.engine.impl.DynamicBpmnServiceImpl
```

注：**ProcessEngine** 中还维护表单相关的服务组件，我们将在表单引擎一章中讲述。

5.3.2 关闭流程引擎

根据前面章节可知，**ProcessEngines** 实例在销毁时，会调用全部 **ProcessEngine** 的 **close** 方法，会对流程引擎进行关闭操作，这些操作包括关闭异步执行器（**AsyncExecutor**）和执行数据库表删除（**drop**），需要让其删除数据表，前提是要将流程引擎配置的 **databaseSchemaUpdate** 属性设置为 **create-drop**（请见 4.2.7 章节）。代码清单 5-7 为执行 **close** 方法，运行前请将全部数据表删除。

代码清单 5-7: codes\05\5.3\close-engine\src\org\crazyit\activiti\Close.java

```
//读取配置
ProcessEngineConfiguration config = ProcessEngineConfiguration.
```

```

        createProcessEngineConfigurationFromResource("close.xml");
//创建流程引擎
ProcessEngine engine = config.buildProcessEngine();
System.out.println("完成流程引擎创建");
Thread.sleep(10000);
//执行 close 方法
engine.close();

```

注意代码清单使用了 `Thread.sleep` 方法，可以利用暂停的这 10 秒钟时间去查看数据库的表是否已经创建成功，10 秒后执行 `close` 方法，以下为代码清单 5-7 输出的日志信息：

```

22:21:43,348 INFO DbSqlSession - performing create on engine with resource
org/activiti/db/create/activiti.mysql.create.engine.sql
22:21:43,348 INFO DbSqlSession - Found MySQL: majorVersion=5 minorVersion=6
22:21:43,355 INFO DefaultManagementAgent - JMX Connector thread started and listening at:
service:jmx:rmi:///jndi/rmi://AY-PC:1099/jmxrmi/activiti
22:22:48,881 INFO DbSqlSession - performing create on history with resource
org/activiti/db/create/activiti.mysql.create.history.sql
22:22:48,882 INFO DbSqlSession - Found MySQL: majorVersion=5 minorVersion=6
22:23:04,069 INFO DbSqlSession - performing create on identity with resource
org/activiti/db/create/activiti.mysql.create.identity.sql
22:23:04,070 INFO DbSqlSession - Found MySQL: majorVersion=5 minorVersion=6
22:23:08,724 INFO ProcessEngineImpl - ProcessEngine default created
完成流程引擎创建
22:23:18,948 INFO DbSqlSession - performing drop on engine with resource
org/activiti/db/drop/activiti.mysql.drop.engine.sql
22:23:18,965 INFO DbSqlSession - Found MySQL: majorVersion=5 minorVersion=6
22:23:33,613 INFO DbSqlSession - performing drop on history with resource
org/activiti/db/drop/activiti.mysql.drop.history.sql
22:23:33,613 INFO DbSqlSession - Found MySQL: majorVersion=5 minorVersion=6
22:23:42,030 INFO DbSqlSession - performing drop on identity with resource
org/activiti/db/drop/activiti.mysql.drop.identity.sql
22:23:42,030 INFO DbSqlSession - Found MySQL: majorVersion=5 minorVersion=6

```

注意以上输出信息的粗体部分，完成了流程引擎创建后，由于调用了 `close` 方法，并且 `databaseSchemaUpdate` 属性设置为 `create-drop`，因此 Activiti 会执行相应数据库的 `drop` 脚本。

注：运行代码清单 5-7 前，需要先将数据库中的全部数据表删除。

5.3.3 流程引擎名称

根据 5.2.1 章节可以知道，每个 `ProcessEngine` 实例均有自己的名称，在 `ProcessEngines` 的 `Map` 中，会使用该名称作为 `Map` 的 `key` 值，如果不为 `ProcessEngine` 设置名称的话，Activiti 会默认的将其设置为“`default`”。`ProcessEngine` 本身没有提供设置名称的方法，该方法由 `ProcessEngineConfiguration` 提供。代码清单 5-8 为 `ProcessEngine` 设置名称。

代码清单 5-8: codes\05\5.3\engine-name\src\org\crazyit\activiti\Name.java

```

ProcessEngineConfiguration config = ProcessEngineConfiguration.
    createProcessEngineConfigurationFromResource("name.xml");
//设置流程引擎名称
config.setProcessEngineName("test");
ProcessEngine engine = config.buildProcessEngine();

```

```
//根据名称查询流程引擎
ProcessEngine engineTest = ProcessEngines.getProcessEngine("test");
System.out.println("创建的引擎实例: " + engine);
System.out.println("查询的引擎实例: " + engineTest);
```

代码清单 5-8 中的粗体字代码，调用了 `ProcessEngineConfiguration` 的 `setProcessEngineName` 方法将流程引擎名称设置为 `test`，然后根据该名称到 `ProcessEngines` 中查询相应的流程引擎，代码输出结果两个引擎均为同一对象。由此可知，`buildProcessEngine` 方法实际上完成了 `ProcessEngines` 的 `register` 操作，运行代码清单 5-8，输出结果如下：

```
创建的引擎实例: org.activiti.engine.impl.ProcessEngineImpl@16fe72b
查询的引擎实例: org.activiti.engine.impl.ProcessEngineImpl@16fe72b
```

5.4 本章小节

本章内容主要讲述了如何利用 `Activiti` 的配置创建流程引擎对象（`ProcessEngine`）。本章中所述的创建流程引擎对象的方法有两种：`ProcessEngineConfiguration` 的 `buildProcessEngine` 方法、`ProcessEngines` 的 `init` 方法，除此之外，如果项目中使用了 `Spring`，还可以将 `ProcessEngine` 作为一个 `bean` 配置到 `XML` 文件中，然后使用 `Spring` 的 `API` 获取。

本章内容较为简单，最基本要掌握 `ProcessEngines` 与 `ProcessEngine` 的使用就已经完成任务，对于本章中一些关于实现原理的描述，如不感兴趣可不必掌握。

6 邮件服务器与 history 配置

6.1 history 配置

在流程执行的过程中，会产生一些流程相应的数据，例如流程实例、流程任务和流程参数等数据，随着流程的进行与结束，这些数据将会从流程数据表中删除，为了能保存这些数据，`Activiti` 提供了历史数据表，可以让这些数据保存到历史数据表中。

对于这些历史数据，保存到何种粒度，`Activiti` 提供了 `history` 属性对其进行配置。`history` 属性有点像 `log4j` 的日志输出级别，该属性有以下四个值：

- ❑ `none`：不保存任何的历史数据，因此，在流程执行过程中，这是最高效的。
- ❑ `activity`：级别高于 `none`，保存流程实例与流程行为，其他数据不保存。
- ❑ `audit`：除 `activity` 级别会保存的数据外，还会保存全部的流程任务及其属性。`audit` 为 `history` 的默认值。
- ❑ `full`：保存历史数据的最高级别，除了会保存 `audit` 级别的数据外，还会保存其他全部流程相关的细节数据，包括一些流程参数等。

流程历史数据配置的用法以及效果，将在“历史数据管理”章节进行详细讲解，如果读

者想查看各个属性的效果，可以运行以下几个例子：

- codes\04\4.3\history-config\src\org\crazyit\activiti\Activity.java
- codes\04\4.3\history-config\src\org\crazyit\activiti\Audit.java
- codes\04\4.3\history-config\src\org\crazyit\activiti\Full.java
- codes\04\4.3\history-config\src\org\crazyit\activiti\None.java

注：在运行这几个例子前，先将 act 数据库的全部表删除。

6.2 邮件服务器配置

Activiti 支持邮件服务，当流程执行到某一个节点时，Activiti 会根据流程文件配置 (Email Task)，发送邮件到相应的邮箱。以下为 ProcessEngineConfiguration 中提供的邮件服务器配置项：

- ❑ mailServerHost: 邮件服务器地址，非必填，默认值为 localhost。
- ❑ mailServerPort: SMTP 发送邮件服务器端口，默认值为 25。
- ❑ mailServerDefaultFrom: 非必填，发送人的邮箱地址，默认值为 activiti@activiti.org。
- ❑ mailServerUsername: 邮箱登录用户名。
- ❑ mailServerPassword: 邮箱登录密码。
- ❑ mailServerUseSSL: 是否使用 SSL 协议通信，默认为 false。
- ❑ mailServerUseTLS: 是否使用 TLS 协议通信，默认为 false。

使用 SMTP 协议发送邮件，需要知道邮件服务器地址、SMTP 端口、邮箱登录用户名和密码，代码清单 4-15 中以网易邮箱为例子，列出如何设置这几个邮件配置项。

代码清单 4-15: codes\04\4.3\mail\resource\mail.xml

```
<bean id="processEngineConfiguration"
      class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/act" />
  <property name="jdbcDriver" value="com.mysql.jdbc.Driver" />
  <property name="jdbcUsername" value="root" />
  <property name="jdbcPassword" value="123456" />
  <property name="mailServerHost" value="smtp.163.com"></property>
  <property name="mailServerPort" value="25"></property>
  <property name="mailServerDefaultFrom" value="yangenxiong@163.com"></property>
  <property name="mailServerUsername" value="yangenxiong@163.com"></property>
  <property name="mailServerPassword" value="123456"></property>
</bean>
```

关于邮件发送任务 (Email Task) 的使用，请参看流程任务章节。

7 Activiti 的设计模式

本文要点

- 命令模式和责任链模式，以及 Activiti 如何使用这两种模式

7.1 Activiti 的命令拦截器

Activiti 提供了命令拦截器的功能，外界对 Activiti 流程中各个实例进行操作，实际可以看作是对数据进行相应的操作，在此过程中，Activiti 使用了设计模式中的命令模式，每一个操作数据库的过程，均可被看作为一个命令，然后交由命令执行者去完成。除此之外，为了能让使用者可以对这些命令进行相应的拦截（进行个性化处理），Activiti 还使用了设计模式中的责任链模式，使用者可以添加相应的拦截器（责任链模式中的处理者）。为了让读者对这些知识有更深入的了解，在本小节，将先讲解命令模式与责任链模式。

注：对于初学者，可跳过本小节，掌握前面的流程引擎配置即可。

7.1.1 命令模式

在 GoF 的设计模式中，命令模式属于行为型模式，它把一个请求或者操作封装到命令对象中，这些请求或者操作内容包括接收者信息，然后将该命令对象交由执行者执行，执行者不需要关心命令的接收人或者命令的具体内容，因为这些信息均被封装到命令对象中。命令模式中涉及的角色以及其作用如下：

- ❑ 命令接口（Command）：声明执行操作的接口。
- ❑ 接口实现（ConcreteCommand）：命令接口实现，需要保存接收者的相应操作，并执行相应的操作。
- ❑ 命令执行者（Invoker）：要求命令执行此次请求。
- ❑ 命令接收人（Receiver）：由命令的实现维护实例，并在命令执行时处理相应的任务。

接下来，编写一个最简单的命令模式。代码清单 4-19 为命令接口。

代码清单 4-19: codes\04\4.5\gof-command\src\org\crazyit\activiti\Command.java

```
public interface Command {  
  
    /**  
     * 执行命令，参数为命令接收人  
     * @param receiver  
     */  
    void execute(CommandReceiver receiver);  
}
```

然后创建命令接收者，请看代码清单 4-20。

代 码 清 单 4-20 :

codes\04\4.5\gof-command\src\org\crazyit\activiti\CommandReceiver.java

```
public interface CommandReceiver {  
  
    //命令执行者方法 A
```

```

void doSomethingA();

//命令执行者方法 B
void doSomethingB();
}

```

本例中命令接者只有一个实现，方法 A（doSomething）打印“命令接收人执行命令 A”，方法 B（doSomethingB）打印“命令接收人执行命令 B”，接下来创建命令执行者，如代码清单 4-21 所示。

代 码 清 单 4-21 :

codes\04\4.5\gof-command\src\org\crazyit\activiti\CommandExecutor.java

```

public class CommandExecutor {

    public void execute(Command command) {
        //创建命令接者可以使用其他设计模式
        command.execute(new CommandReceiverImpl());
    }
}

```

注意命令执行者的实现中，调使用命令的 execute 方法，并将相应的命令接收人设置到命令的 execute 方法参数中。此处创建命令接收者的方式，可以使用其他的设计模式完成，例如工厂模式，单态模式等等，此处为了更加简单，直接 new 一个 CommandReceiver 的实现类。接下来，为命令接口提供两个实现 CommandA 和 CommandB，代码清单 4-22 为 CommandA 的实现。

代 码 清 单 4-22 :

codes\04\4.5\gof-command\src\org\crazyit\activiti\impl\CommandA.java

```

public class CommandA implements Command {
    public void execute(CommandReceiver receiver) {
        receiver.doSomethingA();
    }
}

```

在代码清单 4-22 中，CommandA 的实现中，直接让命令接收执行方法 A（doSomethingA），CommandB 的实现与 CommandA 类似，只是执行命令接收者的方法 B（doSomethingB）。到此，命令模式的各个角色已经创建完毕，接下来编写客户端代码，让命令执行者执行相应的命令。代码清单 4-23 为客户端代码。

代码清单 4-23: codes\04\4.5\gof-command\src\org\crazyit\activiti\Client.java

```

public class Client {

    public static void main(String[] args) {
        //创建命令执行者
        CommandExecutor executor = new CommandExecutor();
        //创建命令 A，交由命令执行者执行
        Command commandA = new CommandA();
        executor.execute(commandA);
        //创建命令 B，交由命令执行者执行
        Command commandB = new CommandB();
        executor.execute(commandB);
    }
}

```

代码清单 4-23 中，先创建一个命令执行者，然后创建两个命令，并交由命令执行者执行，最终执行结果将输出“命令接收人执行命令 A”和“命令接收人执行命令 B”。

现在了解了 GoF 的命令模式，在 Activiti 中，每一个数据库的 CRUD 操作，均为一个命令的实现，然后交给 Activiti 的命令执行者执行。Activiti 使用了一个 `CommandContext` 类作为命令接收者，该对象维护一系列的 `Manager` 对象，这些 `Manager` 对象就像 J2EE 中的 DAO 对象。除了命令接收者外，Activiti 还使用一系列的 `CommandInterceptor`（命令拦截器），这些命令拦截器扮演命令模式中的命令执行者角色。那么这些命令拦截器是如何工作的呢？接下来需要了解责任链模式。

7.1.2 责任链模式

与命令模式一样，责任链模式也是 GoF 的设计模式之一，同样也是行为型模式。该设计模式为了让多个对象都有机会处理请求，从而避免了请求发送者和请求接收者之间的耦合。这些请求接收者将组成一条，并沿着这条链传递该请求，直到有一个对象处理这个请求为止，这就形成一条责任链。责任链模式有以下参与者：

- ❑ 请求处理者接口（`Handler`）：定义一个处理请求的接口，可以实现后继链。
- ❑ 请求处理者实现（`ConcreteHandler`）：请求处理接口的实现，如果它可以处理请求，就处理，否则就将该请求转发给它的后继者。

代码清单 4-24 中编写一个请求处理的抽象类。

代码清单 4-24: `codes\04\4.5\gof-chain\src\org\crazyit\activiti\Handler.java`

```
public abstract class Handler {

    //下一任处理者
    protected Handler next;

    public void setNext(Handler next) {
        this.next = next;
    }

    //处理请求的方法，交由子类实现
    public abstract void execute(Request request);
}
```

代码清单 4-24 定义了一个请求处理者的抽象类，并且定义了请求的处理方法，需要由子类实现，需要注意的是，处理请求方法（`execute`）的参数为一个 `Request` 对象，本例中的 `Request` 对象只是一个普通的类，若责任链模式结合命令模式一起使用的话，那么 `execute` 方法的参数可以是命令模式中的命令接口。除此之外，`Handler` 还定义了一个 `next` 属性，在这里表示下一任处理者的对象，此处提供 `setter` 方法，由客户端决定下一任请求处理者是谁。`Request` 对象如代码清单 4-25 所示。

代码清单 4-25: `codes\04\4.5\gof-chain\src\org\crazyit\activiti\Request.java`

```
public class Request {
    public void doSomething() {
        System.out.println("执行请求");
    }
}
```

`Request` 对象只有一个 `doSomething` 方法，如果将 `Request` 设置为一个接口的话，那么它也可以像命令模式的命令接口一样（见命令模式的 `Command`）有多个实现。接下来，为请求处理接口添加若干个实现，代码清单 4-26 为其中一个实现。

代码清单 4-26: `codes\04\4.5\gof-chain\src\org\crazyit\activiti\impl\HandlerA.java`

```

public class HandlerA extends Handler {

    public void execute(Request request) {
        //处理自己的事，然后交由下一任处理者继续执行请求
        System.out.println("请求处理者 A 处理请求");
        next.execute(request);
    }
}

```

代码清单 4-26 中，HandlerA 继承了 Handler 并且实现了 execute 方法，当该处理器处理完自己的事情后，再将请求交由下一任处理者继续执行请求。在责任链中，可以新建多个这样的请求处理者，本例中有两个这样的请求处理者，实现均与代码清单 4-26 中的 HandlerA 类似，在此不再赘述。

除了若干个请求处理者的实现外，还需要新建一个真实的请求处理者，通过代码清单 4-26 可以知道，实际上就算再多这样的请求处理者实现，依然没有对请求作任何处理，只是交由下一任处理者执行，因此需要一个真实的请求处理者来终结这条责任链。代码清单 4-27 为真实任务处理者。

代码清单 4-27: codes\04\4.5\gof-chain\src\org\crazyit\activiti\impl\ActualHandler.java

```

/**
 * 最终请求执行者，需要将其设置到责任链的最后一环
 */
public class ActualHandler extends Handler {

    public void execute(Request request) {
        //直接执行请求
        request.doSomething();
    }
}

```

如代码清单 4-27 所示，最终的请求处理者最终执行了请求，并且不再往下执行（不使用 next 属性）。下面编写客户端代码，使用这个责任链。客户端代码如代码清单 4-28 所示。

代码清单 4-28: codes\04\4.5\gof-chain\src\org\crazyit\activiti\Client.java

```

public static void main(String[] args) {
    //创建第一个请求处理者集合
    List<Handler> handlers = new ArrayList<Handler>();
    //添加请求处理者到集合中
    handlers.add(new HandlerA());
    handlers.add(new HandlerB());
    //将最终的处理者添加到集合中
    handlers.add(new ActualHandler());
    //处理集合中的请求处理者，按集合的顺序为它们设置下一任请求处理者，并返回第一任处
    理人
    Handler first = setNext(handlers);
    // 第一任处理开始处理请求
    first.execute(new Request());
}
//按照集合的顺序，设置下一任处理者，并返回第一任处理者
static Handler setNext(List<Handler> handlers) {
    for (int i = 0; i < handlers.size() - 1; i++) {
        Handler handler = handlers.get(i);
        Handler next = handlers.get(i + 1);
        handler.setNext(next);
    }
}

```

```

        return handlers.get(0);
    }

```

在代码清单 4-28 中，定义了一个请求处理者的集合，然后按照该集合顺序通过 `setNext` 方法为每一个请求处理器设置下一任的请求处理器，`setNext` 方法最后返回第一任处理器（`HandlerA`）。需要注意的是，由于定义了最终的请求处理器为 `ActualHandler`，因此需要将其放到集合的最后，作为终止整个责任链的角色。最终运行顺序为：“请求处理器 A 处理请求”，“请求处理器 B 处理请求”，“执行请求”。

7.1.3 编写自定义拦截器

前面讲解了命令模式与责任链模式，`Activiti` 的拦截器，就是结合这两种设计模式，达到拦截器的效果，每次 `Activiti` 进行业务操作，都会封装为一个 `Command` 放到责任链中执行。知道其原理后，可以在实现自定义配置类时，编写自己的拦截器。首先编写第一个拦截器实现，请看代码清单 4-29。

代 码 清 单 4-29 :
codes\04\4.5\custom-interceptor\src\org\crazyit\activiti\InterceptorA.java

```

/**
 * 拦截器实现 A
 *
 */
public class InterceptorA implements CommandInterceptor {

    private CommandInterceptor next;

    @Override
    public <T> T execute(CommandConfig config, Command<T> command) {
        // 输出字符串和命令
        System.out.println("this is interceptor A: "
            + command.getClass().getName());
        // 然后让责任链中的下一请求处理器处理命令
        return getNext().execute(config, command);
    }

    public CommandInterceptor getNext() {
        return this.next;
    }

    public void setNext(CommandInterceptor next) {
        this.next = next;
    }
}

```

代码清单 4-29 中的类 `InterceptorA` 实现 `CommandInterceptor` 接口，实现责任链模式时，在拦截器的 `execute` 方法中，执行完拦截器自己的程序后（输出业务命令），会执行责任链的下一个拦截器的 `execute` 方法。了解责任链模式后，不难发现，此处的 `next` 就是拦截器中的下一任请求处理器，而此处的请求，则是命令模式中的 `Command` 接口，编写的 `InterceptorA` 就是责任链模式中请求处理者的其中一个实现。

使用同样的方式，创建拦截器 B，与拦截器 A 类似，输出字符串与业务命令，再将请

求（此处为 **Command**）交由下一执行者执行。完成了两个拦截器后，再去实现父类的初始化拦截器方法，将我们的拦截器“侵入”到 **Activiti** 的责任链中，详情请见代码清单 4-30。

代 码 清 单 4-30 :
codes\04\4.5\custom-interceptor\src\org\crazyit\activiti\TestConfiguration.java

```
/**
 * 自定义配置类
 */
public class TestConfiguration extends ProcessEngineConfigurationImpl {

    public CommandInterceptor createTransactionInterceptor() {
        // 不实现事务拦截器
        return null;
    }

    /**
     * 重写初始化命令拦截器方法
     */
    public void initCommandInterceptors() {
        // 为父类的命令集合添加拦截器
        customPreCommandInterceptors = new ArrayList<CommandInterceptor>();
        // 依次将 A 和 B 两个拦截器加入集合（责任链）
        customPreCommandInterceptors.add(new InterceptorA());
        customPreCommandInterceptors.add(new InterceptorB());
        // 再调用父类的实始化方法
        super.initCommandInterceptors();
    }
}
```

代码清单 4-30 中 **initCommandInterceptors** 方法，用于初始化命令拦截器集合。我们的自定义集合，加上 **Activiti** 的默认集合，形成拥有多个拦截器集合，也就是一条责任链。**Activiti** 的默认集合，会加入日志拦截器、**createTransactionInterceptor** 方法返回的拦截器、包含业务操作的命令、事务拦截器。代码清单 4-31 为运行代码。

代码清单 4-31: codes\04\4.5\custom-interceptor\src\org\crazyit\activiti\MyConfig.java

```
ProcessEngines.getDefaultProcessEngine();
// 创建 Activiti 配置对象
ProcessEngineConfiguration config = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResource("my-config.xml");
// 初始化流程引擎
ProcessEngine engine = config.buildProcessEngine();
// 部署一个最简单的流程
engine.getRepositoryService().createDeployment()
    .addClasspathResource("bpmn/config.bpmn20.xml").deploy();
// 构建流程参数
Map<String, Object> vars = new HashMap<String, Object>();
vars.put("day", 10);
// 开始流程
engine.getRuntimeService().startProcessInstanceByKey("vacationProcess",
    vars);
```

代码清单 4-31 中，部署了一个简单的流程，此时运行该测试程序，可以看到拦截器打印的效果如图 4-2 所示。

```
08:03:43,585 INFO DbSqlSession - Found MySQL: majorVersion=5 minorVersion=6
08:03:44,223 INFO DefaultManagementAgent - JMX Connector thread started and listening at: service:jmx:rmi:///jndi/rmi://AY-F
08:05:32,835 INFO DbSqlSession - performing create on history with resource org/activiti/db/create/activiti.mysql.create.his
08:05:32,835 INFO DbSqlSession - Found MySQL: majorVersion=5 minorVersion=6
08:05:52,894 INFO DbSqlSession - performing create on identity with resource org/activiti/db/create/activiti.mysql.create.ic
08:05:52,895 INFO DbSqlSession - Found MySQL: majorVersion=5 minorVersion=6
08:06:03,516 INFO ProcessEngineImpl - ProcessEngine default created
this is interceptor A: org.activiti.engine.impl.cmd.ValidateExecutionRelatedEntityCountCfgCmd
this is interceptor B org.activiti.engine.impl.cmd.ValidateExecutionRelatedEntityCountCfgCmd
this is interceptor A: org.activiti.engine.impl.RepositoryServiceImpl$1
this is interceptor B org.activiti.engine.impl.RepositoryServiceImpl$1
this is interceptor A: org.activiti.engine.impl.cmd.DeployCmd
this is interceptor B org.activiti.engine.impl.cmd.DeployCmd
this is interceptor A: org.activiti.engine.impl.cmd.GetNextIdBlockCmd
this is interceptor B org.activiti.engine.impl.cmd.GetNextIdBlockCmd
this is interceptor A: org.activiti.engine.impl.cmd.GetProcessDefinitionInfoCmd
this is interceptor B org.activiti.engine.impl.cmd.GetProcessDefinitionInfoCmd
this is interceptor A: org.activiti.engine.impl.cmd.StartProcessInstanceCmd
this is interceptor B org.activiti.engine.impl.cmd.StartProcessInstanceCmd
```

图 4-2

在图 4-2 中可以看到，每执行一个命令，都会经过我们定义的拦截器 A 和 B。从命令名称不能看出，输出的命令与代码清单 4-31 基本吻合：部署流程模型、查询流程定义、启动流程。

8 Activiti 数据查询（一）

本文要点

- Activiti 的数据查询、排序机制

8.1 Activiti 数据查询

Activiti 提供了一套数据查询 API 供开发者使用，可以使用各个服务组件的 createXXXQuery 方法来获取这些查询对象。本小节将结合用户组数据来讲解 Activiti 的数据查询设计，这些设计应用于整个 Activiti 的数据查询体系。

8.1.1 查询对象

Activiti 的各个服务组件（XXXService）均提供了 createXXXQuery 方法，例如本章的 IdentityService 中的 createGroupQuery 方法和 createUserQuery 方法，TaskService 中的 createTaskQuery 方法等，这些方法返回的是一个 Query 实例，例如 createGroupQuery 返回的是 GroupQuery，GroupQuery 是 Query 的子接口。

Query 是全部查询对象的父接口，该接口定义了若干个基础方法，各个查询对象均可以使用这些公共方法，包括设置排序方式、数据量统计（count）、列表、分页和唯一记录查询。

这些方法描述如下：

- ❑ **asc**：设置查询结果的排序方式为升序。
- ❑ **count**：计算查询结果的数据量。
- ❑ **desc**：设置查询结果的排序方式为降序。
- ❑ **list**：封装查询结果，返回相应类型的集合。
- ❑ **listPage**：分页返回查询结果。
- ❑ **singleResult**：查询单条符合条件的数据，如果查询不到，则返回 **null**，如果查询到多条记录，则抛异常。

下面将以用户组数据为例，讲解这些方法的使用以及注意事项。

8.1.2 list 方法

Query 接口的 **list** 方法，将查询对象对应的实体数据以集合形式返回，返回的集合需要指定元素类型，如果没有查询条件，则会将表中全部的数据查出，默认按照主键（ID_列）升序排序。代码清单 6-4 中使用 **list** 方法。

代码清单 6-4：codes\06\6.2\list-data\src\org\crazyit\activiti>ListData.java

```
public static void main(String[] args) {
    // 创建流程引擎
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 得到身份服务组件实例
    IdentityService identityService = engine.getIdentityService();
    // 写入 5 条用户组数据
    createGroup(identityService, "1", "GroupA", "typeA");
    createGroup(identityService, "2", "GroupB", "typeB");
    createGroup(identityService, "3", "GroupC", "typeC");
    createGroup(identityService, "4", "GroupD", "typeD");
    createGroup(identityService, "5", "GroupE", "typeE");
    // 使用 list 方法查询全部的部署数据
    List<Group> datas = identityService.createGroupQuery().list();
    for (Group data : datas) {
        System.out.println(data.getId() + "---" + data.getName() + " ");
    }
}

// 将用户组数据保存到数据库中
static void createGroup(IdentityService identityService, String id,
    String name, String type) {
    // 调用 newGroup 方法创建 Group 实例
    Group group = identityService.newGroup(id);
    group.setName(name);
    group.setType(type);
    identityService.saveGroup(group);
}
```

在代码清单 6-4 中，先往数据库中写入 5 条用户组数据，然后调用 **Query** 的 **list** 方法将全部数据查出（代码清单 6-4 中的粗体字代码），需要注意的是，在不设置任何排序条件以及排序方式的情况下，将会以主键升序的方式返回结果，代码清单 6-4 的运行结果如下：

```
1---GroupA
2---GroupB
```



```
3---GroupC
4---GroupD
5---GroupE
```

8.1.3 listPage 方法

listPage 方法与 **list** 方法类似，最终也是以主键升序排序返回结果集，与 **list** 方法不同的是，**listPage** 方法需要提供两个 **int** 参数，第一个参数数据的开始索引，从 0 开始，第二个参数为结果数量，不难看出，该方法适用于分页查询。代码清单 6-5 使用 **listPage** 方法进行查询。

代码清单 6-5: codes\06\6.2\list-page\src\org\crazyit\activiti\ListPage.java

```
public static void main(String[] args) {
    //创建流程引擎
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 得到身份服务组件实例
    IdentityService identityService = engine.getIdentityService();
    // 写入 5 条用户组数据
    createGroup(identityService, "1", "GroupA", "typeA");
    createGroup(identityService, "2", "GroupB", "typeB");
    createGroup(identityService, "3", "GroupC", "typeC");
    createGroup(identityService, "4", "GroupD", "typeD");
    createGroup(identityService, "5", "GroupE", "typeE");
    //调用 listPage 方法，从索引为 2 的记录开始，查询 3 条记录
    List<Group> datas = identityService.createGroupQuery().listPage(2, 3);
    for (Group data : datas) {
        System.out.println(data.getId() + "---" + data.getName() + " ");
    }
}

// 将用户组数据保存到数据库中
static void createGroup(IdentityService identityService, String id,
    String name, String type) {
    // 调用 newGroup 方法创建 Group 实例
    Group group = identityService.newGroup(id);
    group.setName(name);
    group.setType(type);
    identityService.saveGroup(group);
}
```

代码清单 6-5 中，使用了 **listPage** 方法，查询用户组的数据，设置从第二条记录开始，查询 3 条记录，该方法与 MySQL 的 **LIMIT** 关键字类似。代码清单 6-5 运行结果如下：

```
3---GroupC
4---GroupD
5---GroupE
```

8.1.4 count 方法

该方法用于计算查询结果的数据量，类似于 SQL 中的 **SELECT COUNT** 语句，如果不加任何的条件，将会统计整个表的数据量。代码清单 6-6 使用 **count** 方法。

代码清单 6-6: codes\06\6.2\count-data\src\org\crazyit\activiti\Count.java

```
public static void main(String[] args) {
    // 创建流程引擎
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 得到身份服务组件实例
    IdentityService identityService = engine.getIdentityService();
    // 写入 5 条用户组数据
    createGroup(identityService, UUID.randomUUID().toString(), "GroupA", "typeA");
    createGroup(identityService, UUID.randomUUID().toString(), "GroupB", "typeB");
    createGroup(identityService, UUID.randomUUID().toString(), "GroupC", "typeC");
    createGroup(identityService, UUID.randomUUID().toString(), "GroupD", "typeD");
    createGroup(identityService, UUID.randomUUID().toString(), "GroupE", "typeE");
    // 使用 list 方法查询全部的部署数据
    long size = identityService.createGroupQuery().count();
    System.out.println("Group 数量: " + size);
}

// 将用户组数据保存到数据库中
static void createGroup(IdentityService identityService, String id,
    String name, String type) {
    // 调用 newGroup 方法创建 Group 实例
    Group group = identityService.newGroup(id);
    group.setName(name);
    group.setType(type);
    identityService.saveGroup(group);
}
```

9 Activiti 数据查询

本章要点

- Activiti 的数据查询、排序机制

9.1 排序方法

Query 中提供了 **asc** 和 **desc** 方法，这两个方法可以设置查询结果的排序方式，但是调用这两个方法的前提是，必须告诉 Query 对象，是按何种条件进行排序，例如要按照 ID 排序，就要调用相应查询对象的 **orderByXXX** 方法。例如 GroupQuery 的 **orderByGroupId**、**orderByGroupName** 等方法，如果不调用这些方法而直接使用 **asc** 或者 **desc** 方法，则会抛

出 `ActivitiException`，异常信息为：You should call any of the `orderBy` methods first before specifying a direction。要求 `Activiti` 进行排序，却不告诉它以哪个字段进行排序，因此会抛出该异常。代码清单 6-7 中调用 `asc` 和 `desc` 方法。

代码清单 6-7：codes\06\6.2\sort-data\src\org\crazyit\activiti\Sort.java

```
public static void main(String[] args) {
    //创建流程引擎
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 得到身份服务组件实例
    IdentityService identityService = engine.getIdentityService();
    // 写入 5 条用户组数据
    createGroup(identityService, UUID.randomUUID().toString(), "1", "typeA");
    createGroup(identityService, UUID.randomUUID().toString(), "2", "typeB");
    createGroup(identityService, UUID.randomUUID().toString(), "3", "typeC");
    createGroup(identityService, UUID.randomUUID().toString(), "4", "typeD");
    createGroup(identityService, UUID.randomUUID().toString(), "5", "typeE");
    //调用 orderByGroupId 和 asc 方法，结果为按照 ID 升序排序
    System.out.println("asc 排序结果：");

    List<Group> datas = identityService.createGroupQuery().orderByGroupName().asc().list();
    for (Group data : datas) {
        System.out.println("    " + data.getId() + "---" + data.getName());
    }
    System.out.println("desc 排序结果");
    //调用 orderByGroupName 和 desc 方法，结果为名称降序排序
    datas = identityService.createGroupQuery().orderByGroupName().desc().list();
    for (Group data : datas) {
        System.out.println("    " + data.getId() + "---" + data.getName());
    }
}

// 将用户组数据保存到数据库中
static void createGroup(IdentityService identityService, String id,
    String name, String type) {
    // 调用 newGroup 方法创建 Group 实例
    Group group = identityService.newGroup(id);
    group.setName(name);
    group.setType(type);
    identityService.saveGroup(group);
}
```

代码清单 6-7 中，调用了 `asc` 和 `desc` 方法（代码清单中的粗体部分），输出的结果如下：

```
asc 排序结果：
35987ec6-de7f-4d36-920f-71d27b586817---1
3273d754-a77f-4a7b-ac88-b529cc5e3d35---2
590f5597-d662-4c35-a35c-c2828468878d---3
f8decda9-ceb9-4172-ad61-ae3d2a8a4e8e---4
0f50f928-a7ff-4b77-b4fd-578773c0fb2f---5
desc 排序结果
0f50f928-a7ff-4b77-b4fd-578773c0fb2f---5
f8decda9-ceb9-4172-ad61-ae3d2a8a4e8e---4
590f5597-d662-4c35-a35c-c2828468878d---3
3273d754-a77f-4a7b-ac88-b529cc5e3d35---2
```

35987ec6-de7f-4d36-920f-71d27b586817---1

注意：调用 asc 或者 desc，只是让 Query 设置排序方式，orderByXXX 方法、asc 方法和 desc 方法均返回 Query 本身，如果需要得到最终结果集，还需要调用 list 或者 listPage 方法。

9.2 ID 排序问题

在 Activiti 的设计中，每个数据表的主键均设计为字符型，这样的设计使得 Activiti 各个数据表的主键可以灵活设置，但是如果使用数字字符串作为其主键，那么按照 ID 排序，就会带来排序问题，请看代码清单 6-8。

代码清单 6-8: codes\06\6.2\sort-data\src\org\crazyit\activiti\SortProblem.java

```
public static void main(String[] args) {
    //创建流程引擎
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 得到身份服务组件实例
    IdentityService identityService = engine.getIdentityService();
    // 写入 5 条用户组数据
    createGroup(identityService, "1", "GroupA", "typeA");
    createGroup(identityService, "12", "GroupB", "typeB");
    createGroup(identityService, "13", "GroupC", "typeC");
    createGroup(identityService, "2", "GroupD", "typeD");
    createGroup(identityService, "3", "GroupE", "typeE");
    //根据 ID 升序排序
    System.out.println("asc 排序结果");
    List<Group> datas = identityService.createGroupQuery().orderByGroupId().asc().list();
    for (Group data : datas) {
        System.out.print(data.getId() + " ");
    }
}

// 将用户组数据保存到数据库中
static void createGroup(IdentityService identityService, String id,
    String name, String type) {
    // 调用 newGroup 方法创建 Group 实例
    Group group = identityService.newGroup(id);
    group.setName(name);
    group.setType(type);
    identityService.saveGroup(group);
}
```

代码清单 6-8 中，加入了 5 条用户组数据，需要注意的是，这 5 条用户组数据，ID 分别为：1、12、13、2、3，然后调用 orderByGroupId 方法，并且设置为升序排序，期望的结果应该是：1、2、3、12、13，而此处输出结果却是：1、12、13、2、3，产生这种现象是由于 ID_列字段数据类型是字符型，以 MySQL 为例，如果字段类型为字符型，而实际存储的是数据的话，那么进行排序时，会将其看作字符型，因此会产生以上的 ID 顺序错乱，详细请看图 6-2 所示。

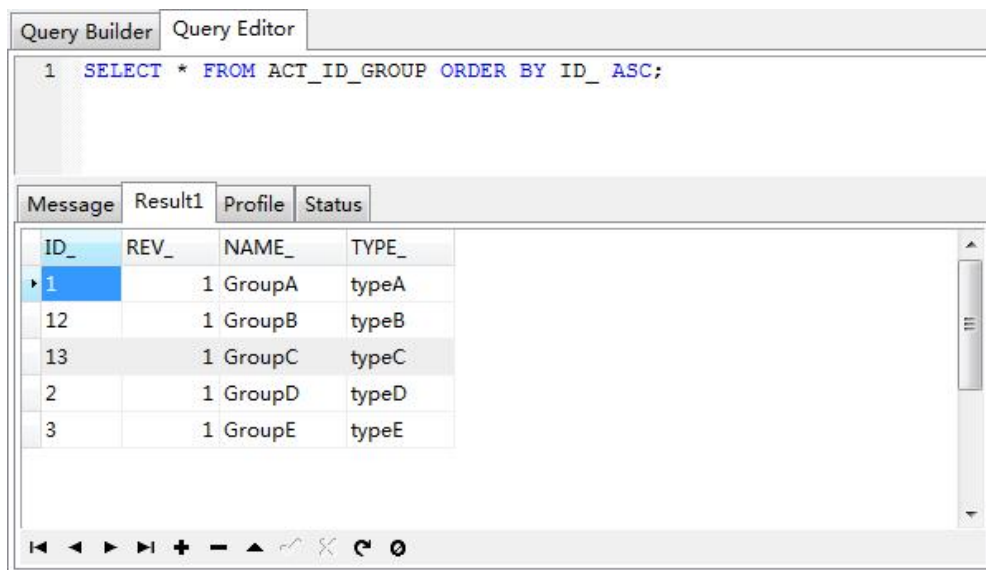


图 6-2 MySQL 的排序

如图 6-2 所示，在 MySQL 中执行普通的 ORDER BY 语句，可以看到数据库排序结果与程序结果一致，前面已经讲到，这样的顺序错乱是由于 ID_ 列数据类型为字符型导致，如果需要使用正确的排序，可以使用以下的 MySQL 语句进行排序：SELECT * FROM ACT_ID_GROUP ORDER BY ID_ ASC，此处 ORDER BY ID_ 后加了“+0”语句，再进行查询后，可以看到结果如图 6-3 所示。

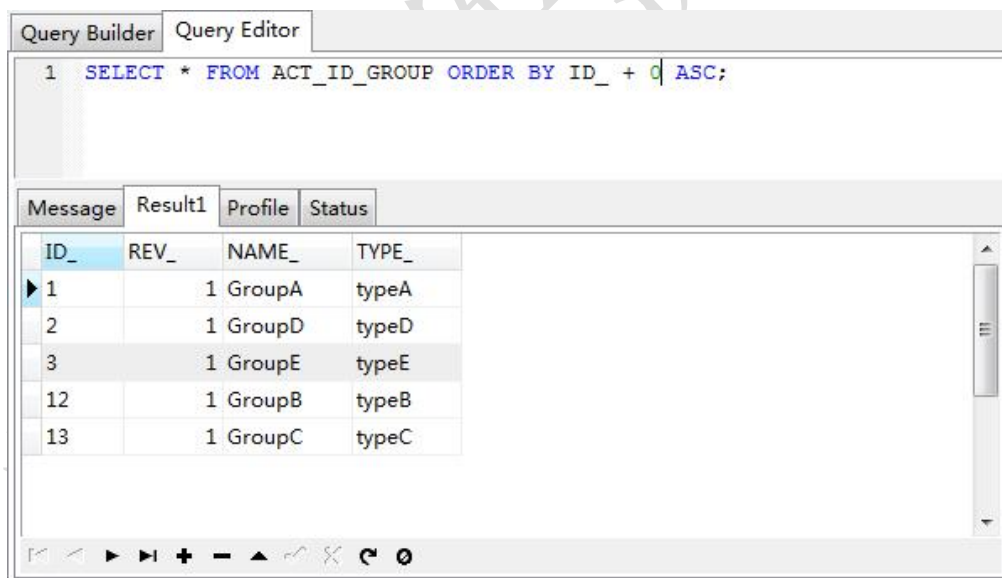


图 6-3 处理后的 MySQL 排序

如图 6-3 所示，MySQL 已经展示了“正确”的排序结果。如果想在代码中解决该排序问题，可以将 Query 转换为 AbstractQuery，再调用 orderBy 方法，请见以下代码片断：

```
AbstractQuery aq = (AbstractQuery)identityService.createGroupQuery();
List<Group> datas = aq.orderBy(new GroupQueryProperty("RES.ID_ + 0")).asc().list();
```

将 GroupQuery 转换为 AbstractQuery，再调用 orderBy 方法，构造一个 GroupQueryProperty，构造参数的字符串为“RES.ID + 0”。执行代码并输出结果后，可以发现结果正确。但笔者不建议使用该方式，因为官方 API 中并没有提供 AbstractQuery 与 GroupQueryProperty，一旦后面的 Activiti 版本中修改了这两个类，那我们的代码也需要进

行修改。除了该方法，也可以使用 Activiti 提供的原生 SQL 查询，详细请见 6.2.10 章节。

9.3 多字段排序

在进行数据查询时，如果想对多个字段进行排序，例如根据名称降序、根据 ID 升序这样的排序方式，那么在调用 `asc` 和 `desc` 方法时就需要注意，`asc` 和 `desc` 方法会根据 Query 实例（`AbstractQuery`）中的 `orderProperty` 属性来决定排序的字段，由于 `orderProperty` 是 `AbstractQuery` 的类属性，因此如果在第二次调用 `orderByXXX` 方法后，会覆盖第一次调用时所调置的值。具体测试结果如代码清单 6-9 所示。

代码清单 6-9: codes\06\6.2\sort-data\src\org\crazyit\activiti\SortMix.java

```
public static void main(String[] args) {
    //创建流程引擎
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 得到身份服务组件实例
    IdentityService identityService = engine.getIdentityService();
    // 写入 5 条用户组数据
    createGroup(identityService, "1", "GroupE", "typeB");
    createGroup(identityService, "2", "GroupD", "typeC");
    createGroup(identityService, "3", "GroupC", "typeD");
    createGroup(identityService, "4", "GroupB", "typeE");
    createGroup(identityService, "5", "GroupA", "typeA");
    //优先按照 id 降序、名称升序排序
    System.out.println("ID 降序排序: ");
    List<Group> datas = identityService.createGroupQuery()
        .orderByGroupId().desc()
        .orderByGroupName().asc().list();
    for (Group data : datas) {
        System.out.println("    " + data.getId() + "---" + data.getName() + " ");
    }
    System.out.println("名称降序排序: ");
    //下面结果将按名称排序
    datas = identityService.createGroupQuery().orderByGroupId()
        .orderByGroupName().desc().list();
    for (Group data : datas) {
        System.out.println("    " + data.getId() + "---" + data.getName() + " ");
    }
}

// 将用户组数据保存到数据库中
static void createGroup(IdentityService identityService, String id,
    String name, String type) {
    // 调用 newGroup 方法创建 Group 实例
    Group group = identityService.newGroup(id);
    group.setName(name);
    group.setType(type);
    identityService.saveGroup(group);
}
```

代码清单 6-9 中的粗体字代码，均使用了两个字段进行排序，第一个查询中，告诉 Query 实例，使用 `groupId` 进行降序排序，再使用名称升序排序，输出结果如下：

ID 降序排序：

```
5---GroupA
4---GroupB
3---GroupC
2---GroupD
1---GroupE
```

输出结果为优先按照 ID 进行降序排序，符合预期。第二个查询中，虽然也调用了 `orderByGroupId` 方法，但是由于没有马上调用 `desc` 方法，而是调用了其他的 `orderBy` 方法，因此原来的 `orderByGroupId` 方法所设置的排序属性（Query 的 `orderProperty` 属性）将会被 `orderByGroupName` 替换，最终输入结果如下：

名称降序排序：

```
1---GroupE
2---GroupD
3---GroupC
4---GroupB
5---GroupA
```

根据输出结果可以看出，最终按照名称降序排序。根据上面的测试可以看出，`asc` 与 `desc` 方法会生成（根据查询条件）相应的查询语句，如果调用了 `orderByXXX` 方法却没有调用一次 `asc` 或者 `desc` 方法，则该排序条件会被下一个设置的查询条件所覆盖。

9.4 singleResult 方法

该方法根据查询条件，到数据库中查询唯一的数据记录，如果没有找到符合条件的数据，则返回 `null`，如果找到多于一条的记录，则抛出异常，异常信息为：Query return 2 results instead of max 1，代码清单 6-10 中使用 `singleResult` 方法，并且体现三种查询结果。

代码清单 6-10: codes\06\6.2\single-result\src\org\crazyit\activiti\SingleResult.java

```
public static void main(String[] args) {
    //创建流程引擎
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 得到身份服务组件实例
    IdentityService identityService = engine.getIdentityService();
    // 写入 5 条用户组数据
    createGroup(identityService, UUID.randomUUID().toString(), "GroupA", "typeA");
    createGroup(identityService, UUID.randomUUID().toString(), "GroupB", "typeB");
    createGroup(identityService, UUID.randomUUID().toString(), "GroupC", "typeC");
    createGroup(identityService, UUID.randomUUID().toString(), "GroupD", "typeD");
    createGroup(identityService, UUID.randomUUID().toString(), "GroupE", "typeE");
    //再写入一条名称为 GroupA 的数据
    createGroup(identityService, UUID.randomUUID().toString(), "GroupA", "typeF");
    //查询名称为 GroupB 的记录
    Group groupB = identityService.createGroupQuery()
        .groupName("GroupB").singleResult();
    System.out.println(" 查询到 一条 GroupB 数据 : " + groupB.getId() + "---" +
groupB.getName());
    //查询名称为 GroupF 的记录
    Group groupF = identityService.createGroupQuery()
        .groupName("GroupF").singleResult();
    System.out.println("没有 groupF 的数据: " + groupF);
    //查询名称为 GroupA 的记录，这里将抛出异常
    Group groupA = identityService.createGroupQuery()
```

```

        .groupName("GroupA").singleResult();
    }

    // 将用户组数据保存到数据库中
    static void createGroup(IdentityService identityService, String id,
        String name, String type) {
        // 调用 newGroup 方法创建 Group 实例
        Group group = identityService.newGroup(id);
        group.setName(name);
        group.setType(type);
        identityService.saveGroup(group);
    }

```

在代码清单 6-10 中，写入了 6 条用户组数据，其中需要注意的是最后一条数据，名称与第一条数据名称一致，目的是为了测试使用 `singleResult` 方法，在查询到多条记录时抛出异常。代码清单 6-10 中的粗体字代码分别为三种情况：正常使用 `singleResult` 方法返回第一条数据，查询不到任何数据，查询出多于一条数据抛出异常。程序运行结果如下：

```

查询到一条 GroupB 数据: deed85ac-d76c-4e7a-b5f6-4d48eb8340ee---GroupB
没有 groupF 的数据: null
16:52:12,936 ERROR CommandContext - Error while closing command context
org.activiti.engine.ActivitiException: Query return 2 results instead of max 1

```

9.5 用户组数据查询

前面章节中，以用户组数据为基础，讲解了 **Activiti** 的数据查询机制以及一些公用的查询方法。**Activiti** 的每种数据均自己对应的查询对象，例如用户组的查询对象为 **GroupQuery**，它继承了 **AbstractQuery**，除了拥有基类的方法（6.2.2 至 6.2.8 的方法）外，它还拥有自己的查询以及排序方法：

- `groupId(String groupId)`：根据 ID 查询与参数值一致的记录。
- `groupMember(String groupMemberUserId)`：根据用户 ID 查询用户所在的用户组，用户组与用户为多对多关系，因此一个用户有可能属于多个用户组。
- `groupName(String groupName)`：根据用户组名称查询用户组。
- `groupNameLike(String groupName)`：根据用户组名称模糊查询用户组数据。
- `groupType(String groupType)`：根据用户组类型查询用户组数据。
- `orderByGroupId()`：设置排序条件为根据 ID 排序。
- `orderByGroupName()`：设置排序条件为根据名称排序。
- `orderByGroupType()`：设置排序条件为根据类型排序。
- `potentialStarter(String procDefId)`：根据流程定义的 ID，查询有权限启动该流程定义的用户组。

代码清单 6-11 演示了如何使用 **GroupQuery** 的部分查询方法。

代码清单 6-11: codes\06\6.2\group-query\src\org\crazyit\activiti\GroupQuery.java

```

public static void main(String[] args) {
    // 创建流程引擎
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 得到身份服务组件实例
    IdentityService identityService = engine.getIdentityService();
    // 写入 5 条用户组数据
    String ald = UUID.randomUUID().toString();

```



```

createGroup(identityService, ald, "GroupA", "typeA");
createGroup(identityService, UUID.randomUUID().toString(), "GroupB", "typeB");
createGroup(identityService, UUID.randomUUID().toString(), "GroupC", "typeC");
createGroup(identityService, UUID.randomUUID().toString(), "GroupD", "typeD");
createGroup(identityService, UUID.randomUUID().toString(), "GroupE", "typeE");
// groupId 方法
Group groupA = identityService.createGroupQuery().groupId(ald).singleResult();
System.out.println("groupId method: " + groupA.getId());
// groupName 方法
Group groupB = identityService.createGroupQuery().groupName("GroupB").singleResult();
System.out.println("groupName method: " + groupB.getName());
// groupType 方法
Group groupC = identityService.createGroupQuery().groupType("typeC").singleResult();
System.out.println("groupType method: " + groupC.getName());
// groupNameLike 方法
List<Group> groups = identityService.createGroupQuery().groupNameLike("%group%").list();
System.out.println("groupNameLike method: " + groups.size());
}

// 将用户组数据保存到数据库中
static void createGroup(IdentityService identityService, String id,
    String name, String type) {
    // 调用 newGroup 方法创建 Group 实例
    Group group = identityService.newGroup(id);
    group.setName(name);
    group.setType(type);
    identityService.saveGroup(group);
}

```

代码清单 6-11 调用了 GroupQuery 的 4 个查询方法，输出结果如下：

```

1
userB
userC
5

```

注：GroupQuery 的设置排序条件方法，在 6.2.5 至 6.2.7 小节中已经体现，本小节不再赘述。另外 groupMember 和 potentialStarter 方法，将在用户组与用户关系、流程定义章节中描述。

9.6 原生 SQL 查询

各个服务组件中，提供了 createNativeXXXQuery 的方法，返回 NativeXXXQuery 的实例，这些对象均是 NativeQuery 的子接口。使用 NativeQuery 的方法，可以传入原生的 SQL 进行数据查询，主要使用 sql 方法传入 SQL 语句，使用 parameter 方法设置查询参数，代码清单 6-12 中使用了原生 SQL 查询用户组数据。

代码清单 6-12: codes\06\6.2\native-query\src\org\crazyit\activiti\NativeQueryTest.java

```

public static void main(String[] args) {
    // 创建流程引擎
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 得到身份服务组件实例
}

```

```

IdentityService identityService = engine.getIdentityService();
// 写入 5 条用户组数据
createGroup(identityService, UUID.randomUUID().toString(), "GroupA",
    "typeA");
createGroup(identityService, UUID.randomUUID().toString(), "GroupB",
    "typeB");
createGroup(identityService, UUID.randomUUID().toString(), "GroupC",
    "typeC");
createGroup(identityService, UUID.randomUUID().toString(), "GroupD",
    "typeD");
createGroup(identityService, UUID.randomUUID().toString(), "GroupE",
    "typeE");
// 使用原生 SQL 查询全部数据
List<Group> groups = identityService.createNativeGroupQuery()
    .sql("select * from ACT_ID_GROUP").list();
System.out.println("查询全部数据: " + groups.size());
// 使用原生 SQL 按条件查询, 并设立参数, 只查到一条数据
groups = identityService.createNativeGroupQuery()
    .sql("select * from ACT_ID_GROUP where NAME_ = 'GroupC'")
    .list();
System.out.println("按条件查询: " + groups.get(0).getName());
// 使用 parameter 方法设置查询参数
groups = identityService.createNativeGroupQuery()
    .sql("select * from ACT_ID_GROUP where NAME_ = #{name}")
    .parameter("name", "GroupD").list();
System.out.println("使用 parameter 方法按条件查询: " + groups.get(0).getName());
}

// 将用户组数据保存到数据库中
static void createGroup(IdentityService identityService, String id,
    String name, String type) {
    // 调用 newGroup 方法创建 Group 实例
    Group group = identityService.newGroup(id);
    group.setName(name);
    group.setType(type);
    identityService.saveGroup(group);
}

```

代码清单 6-12 中粗体字代码, 进行了三次原生 SQL 查询, 第二次与第三次查询设置了参数, 第三次参数使用了 `parameter` 设置参数。由于最终调用查询的是 MyBatis 的 `SqlSession`, 因此写 SQL 时, 需要使用 `#{}`。除了用户组数据外, 其他的数据, 都可以使用原生 SQL 查询。运行代码清单 6-12, 输出结果如下:

```

查询全部数据: 5
按条件查询: GroupC
使用 parameter 方法按条件查询: GroupD

```

使用原生 SQL 查询较为灵活, 可以满足大部分的业务需求, 但笔者还是建议尽量少使用原生 SQL 查询, 这样做增强了代码与数据库结构的耦合性。

10 特别子流程

本文要点

➤ 特别子流程

本来还不会更新到子流程的相关知识，但今天有朋友问到 **Activiti6.0** 新支持的特别子流程（**AdHocSubProcess**），博主今天先发特别子流程的内容发了。

特别子流程

Activiti6.0 增加了对特别子流程的支持，在特别子流程的容器中可以存放多个流程节点，这些节点在运行前不存在流程顺序，流程的顺序和执行，由执行时决定。笔者成书时，**Activiti** 尚未提供特别子流程的 **API**，并且 **Eclipse** 的流程设计器也不支持显示特别子流程，本例暂时使用普通的子流程代替。图 13-10 为本例的特别子流程。

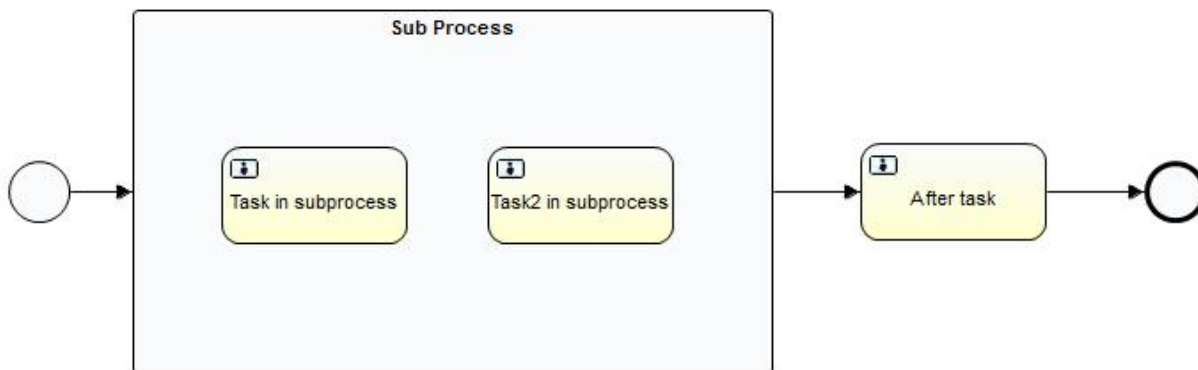


图 13-10 特别子流程

如图 13-10 所示，特别子流程中有两个用户任务，在定义流程时，并没有设定流程走向，当子流程完成后，就会到达“After task”。图 13-10 对应的 BPMN 文件内容，如代码清单 13-12 所示。

代 码 清 单 13-12 :

codes\13\13.1\embeded-subprocess\resource\bpmn\AdHocProcess.bpmn

```
<process id="simpleSubProcess">
  <startEvent id="theStart" />
  <sequenceFlow id="flow1" sourceRef="theStart" targetRef="adhocSubProcess" />
  <adHocSubProcess id="adhocSubProcess" ordering="Sequential">
    <userTask id="subProcessTask" name="Task in subprocess" />
    <userTask id="subProcessTask2" name="Task2 in subprocess" />
  </adHocSubProcess>
  <sequenceFlow id="flow2" sourceRef="adhocSubProcess"
    targetRef="afterTask" />
  <userTask id="afterTask" name="After task" />
  <sequenceFlow id="flow3" sourceRef="afterTask" targetRef="theEnd" />
  <endEvent id="theEnd" />
</process>
```

使用 **adHocSubProcess** 元素来配置特别子流程，其中该元素的 **ordering** 属性，声明特

别子流程中的节点，是会按顺序执行还是会并行，可配置为 **Parallel** 或 **Sequential**。设计完流程后，编写客户端代码部署并执行流程，如代码清单 13-13 所示。

代码清单 13-13 :
codes\13\13.1\embedded-subprocess\src\org\crazyit\activiti\AdHocProcess.java

```
// 创建流程引擎
ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
// 得到流程存储服务组件
RepositoryService repositoryService = engine.getRepositoryService();
// 得到运行时服务组件
RuntimeService runtimeService = engine.getRuntimeService();
TaskService taskService = engine.getTaskService();
// 部署流程文件
repositoryService.createDeployment()
    .addClasspathResource("bpmn/AdHocProcess.bpmn").deploy();
// 启动流程
ProcessInstance pi = runtimeService
    .startProcessInstanceByKey("simpleSubProcess");
System.out.println("开始流程后，执行流数量： "
    + runtimeService.createExecutionQuery()
        .processInstanceId(pi.getId()).count());
// 查询子流程的执行流
Execution exe = runtimeService.createExecutionQuery()
    .processInstanceId(pi.getId()).activityId("adhocSubProcess")
    .singleResult();
// 让执行流到达第二个任务
runtimeService.executeActivityInAdhocSubProcess(exe.getId(),
    ①
    "subProcessTask2");
// 查询执行流数量
System.out.println("让执行流到达第二个任务后，执行流数量： "
    + runtimeService.createExecutionQuery()
        .processInstanceId(pi.getId()).count());
// 完成第二个任务
Task subProcessTask2 = taskService.createTaskQuery()
    .processInstanceId(pi.getId())
    .taskDefinitionKey("subProcessTask2").singleResult();
taskService.complete(subProcessTask2.getId());
// 查询执行流数量
System.out.println("完成子流程的第二任务后，执行流数量： "
    + runtimeService.createExecutionQuery()
        .processInstanceId(pi.getId()).count());
// 完成特别子流程
runtimeService.completeAdhocSubProcess(exe.getId());
    ②
// 查询数量
System.out.println("完成整个特别子流程后，当前任务名称： "
    + taskService.createTaskQuery().processInstanceId(pi.getId())
        .singleResult().getName());
```

代码清单 13-13 中的①，使用 `runtimeService` 的 `executeActivityInAdhocSubProcess` 方法让流程执行特别子流程中的第二个用户任务，②则使用 `completeAdhocSubProcess` 方法完成特别子流程。运行代码清单 13-13，输出如下：

开始流程后，执行流数量： 2

让执行流到达第二个任务后，执行流数量：3
完成子流程的第二任务后，执行流数量：2
完成整个特别子流程后，当前任务名称：After task

根据输出结果可知，在特别子流程中，流程的走向完全由运行时，调用不同的 API 来决定。

11 流程控制逻辑

本小节将以一个简单的例子，讲述 Activiti 关于流程处理的逻辑。

11.1 概述

在 Activiti5 以及 jBPM4，对流程的控制使用的是流程虚拟机这套 API，英文为 Process Virtual Machine，简称 PVM。PVM 将流程中的各种元素抽象出来，形成了一套 Java API。

新发布的 Activiti6.0 版本中，PVM 及相关的 API 已经被移除，取而代之的是一套全新的逻辑，本小节将以一个例子，讲述这套全新逻辑，是如何进行流程控制的，本小节的案例，目的是为了读者了解新版本 Activiti 是如何进行流程控制的。

11.2 设计流程对象

基于 BPMN 规范，Activiti 创建了对应的模型，由于 BPMN 规范过于庞杂，为了简单起见，在本例中，我们也先创建自己的规范。代码清单 18-1 为一份定义我们自己流程的 XML 文档。

代码清单 18-1: codes\18\18.1\my-bpmn\resource\myBpmn.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<process id="testProcess">
  <start id="start" />
  <flows>
    <flow id="flow1" source="start" target="task" />
    <flow id="flow2" source="task" target="end" />
  </flows>
  <nodes>
    <task id="task" />
  </nodes>
  <end id="end" />
</process>
```

代码清单 18-1 是一份自定义的 XML 文档，process 元素下有一个 start 节点、end 节点、nodes 节点以及 flows 节点，设计对应的 Java 对象来表示这些节点，代码清单 18-2 为这些 Java 类的代码。

代码清单 18-2: codes\18\18.1\my-bpmn\src\org\crazyit\activiti\xml

```

public class BaseElement {

    // XML 元素的 ID
    private String id;
    ...省略 setter 和 getter 方法
}

public class FlowElement extends BaseElement {

}

public class FlowNode extends FlowElement {

    // 流程出口
    private SequenceFlow outgoFlow;

    // 流程入口
    private SequenceFlow incomeFlow;

    // 流程节点的行为
    private BehaviorInterface behavior;
    ...省略 getter 和 setter 方法
}

public class Start extends FlowNode {

}

public class End extends FlowNode {

}

public class Task extends FlowNode {

}

public class SequenceFlow extends FlowElement {

    private String source;

    private String target;
    ...省略 setter 和 getter 方法
}

```

代码清单 18-2 中的几个类，分别对应流程 XML 文件中的几个节点，实际上，Activiti 也有一套类似的模型，用于表示 BPMN 规范中的 XML 元素。注意 **FlowNode** 类中维护一个节点行为的对象，该对象在 11.3 小节讲述。除了以上的几个类外，还要创建表示流程的类与表示执行流的类，如代码清单 18-3 所示。

代码清单 18-3:

```

codes\18\18.1\my-bpmn\src\org\crazyit\activiti\xml\MyProcess.java
codes\18\18.1\my-bpmn\src\org\crazyit\activiti\MyExecution.java
public class MyProcess extends BaseElement {

```

```

// 开始节点
private Start start;

// 结束节点
private End end;

// 多个顺序流节点
private List<SequenceFlow> flows;

// 多个节点
private List<FlowNode> nodes;
...省略 setter 和 getter 方法
}

public class MyExecution {

    // 执行流的当前节点
    private FlowNode currentNode;

    private MyProcess process;

    ...省略 getter 和 setter 方法
}

```

在 `MyExecution` 类中，维护一个当前执行流的节点对象，表示当前执行流所到达的节点。我们定义的流程规范中，只允许有一个开始事件和一个结束事件，允许出现多个顺序流节点和多个流程节点。接下来，为这些流程节点创建它们的行为类。

11.3 创建流程节点行为

新建一个行为接口，表示流程节点所需要执行的行为，本例中只有两个行为实现：流程开始行为与任务行为，源文件如代码清单 18-4 所示。

代码清单 18-4: codes\18\18.1\my-bpmn\src\org\crazyit\activiti\behavior

```

public interface BehaviorInterface {

    /**
     * 行为执行方法
     */
    void execute(MyExecution exe);
}

// 开始行为
public class StartBehavior implements BehaviorInterface {

    public void execute(MyExecution exe) {
        System.out.println("执行开始节点");
        // 获取当前节点
        FlowNode currentNode = exe.getCurrentNode();
        // 获取顺序流
        SequenceFlow outgoFlow = currentNode.getOutgoFlow();
    }
}

```

```

        // 设置下一节点
        FlowNode nextNode = exe.getProcess().getNode(outgoFlow.getTarget());
        exe.setCurrentNode(nextNode);
    }
}

// 任务行为
public class TaskBehavior implements BehaviorInterface {

    public void execute(MyExecution exe) {
        System.out.println("执行任务节点");
        // 获取当前节点
        FlowNode currentNode = exe.getCurrentNode();
        // 获取顺序流
        SequenceFlow outgoFlow = currentNode.getOutgoFlow();
        // 获取下一个节点
        FlowNode targetNode = exe.getProcess().getNode(outgoFlow.getTarget());
        // 设置当前节点
        exe.setCurrentNode(targetNode);
    }
}

```

代码清单中的 **StartBehavior**，在流程节点执行时，会自动将当前节点设置为下一个节点。注意获取下一个节点，是通过 **SequenceFlow** 对象进行的。在模型中，顺序流是连接两个流程节点的桥梁，因此 **SequenceFlow** 知道流程将要往哪里走。

11.4 编写业务处理类

业务处理类类似 **Activiti** 的服务组件，本例的服务组件只提供启动流程、完成任务这两个业务方法，用于观察流程走向。代码清单 18-5 为本例的服务组件。

代 码 清 单 18-5 :

codes\18\18.1\my-bpmn\src\org\crazyit\activiti\service\MyRuntimeService.java

```

public class MyRuntimeService {

    /**
     * 启动流程的方法
     */
    public MyExecution startProcess(MyProcess process) {
        // 创建执行流
        MyExecution exe = new MyExecution();
        exe.setProcess(process);
        Start startNode = process.getStart();
        // 设置流程当前节点
        exe.setCurrentNode(startNode);
        // 让流程往前进行
        startNode.getBehavior().execute(exe);
        return exe;
    }

    /**
     * 完成任务

```



```

    */
    public void completeTask(MyExecution exe) {
        // 获取当前的流程节点
        FlowNode current = exe.getCurrentNode();
        // 执行节点的行为
        current.getBehavior().execute(exe);
    }
}

```

开始流程的方法，会直接创建执行流，然后获取开始节点并执行其行为。完成任务的方法，获取当前节点，再执行其行为。前面定义的两个节点行为，均是获取下一个节点，作为执行流的当前节点，即让流程向“前”执行。接下来，需要编写 XML 解析类，解析定义的 XML 文档并转换为流程对象。

11.5 流程 XML 转换为 Java 对象

本例为了简单起见，使用了 XStream 作为工具，将读取的 XML 文件转换为 Java 对象，代码清单 18-6 为 XStream 工具类。

代码清单 18-6: codes\18\18.1\my-bpmn\src\org\crazyit\activiti\xml\XStreamUtil.java

```

public class XStreamUtil {

    private static XStream xstream = new XStream();

    static {
        // 配置 XStream
        xstream.alias("process", MyProcess.class);
        xstream.alias("flow", SequenceFlow.class);
        xstream.alias("task", Task.class);
        xstream.alias("start", Start.class);
        xstream.alias("end", End.class);
        xstream.useAttributeFor(BaseElement.class, "id");
        xstream.useAttributeFor(SequenceFlow.class, "source");
        xstream.useAttributeFor(SequenceFlow.class, "target");
    }

    // 将 XML 文件转换为 Process 实例
    public static MyProcess toObject(File file) {
        try {
            FileInputStream fis = new FileInputStream(file);
            MyProcess p = (MyProcess)xstream.fromXML(fis);
            // 初始化行为与各节点的顺序流
            p.initBehavior();
            p.initSequenceFlow();
            fis.close();
            return p;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}

```

在工具类的 **static** 语句块中，定义了哪个节点转换为哪个 **Java** 类。在 **toObject** 方法中，将 **XML** 转换得到 **MyProcess** 实例后，再调用 **MyProcess** 类的 **initBehavior** 和 **initSequenceFlow** 方法，其中 **initBehavior** 方法用于初始化节点的行为，**initSequenceFlow** 用于设置流程的执行顺序，两个方法如代码清单 18-7 所示。

代码清单 18-7: codes\18\18.1\my-bpmn\src\org\crazyit\activiti\xml\MyProcess.java

```
/**
 * 为各节点设置出入的顺序注
 */
public void initSequenceFlow() {
    // 开始事件的顺序流（设置出口）
    this.start.setOutgoFlow(getSequenceFlowBySource(this.start.getId()));
    // 结束事件顺序流（设置入口）
    this.end.setIncomeFlow(getSequenceFlowByTarget(this.end.getId()));
    // 设置其余节点的顺序流
    for(FlowNode node : nodes) {
        for(SequenceFlow flow : flows) {
            if(flow.getSource().equals(node.getId())) {
                node.setOutgoFlow(flow);
            }
            if(flow.getTarget().equals(node.getId())) {
                node.setIncomeFlow(flow);
            }
        }
    }
}

/**
 * 初始化节点行为
 */
public void initBehavior() {
    // 开始与结束节点
    this.start.setBehavior(new StartBehavior());
    for(FlowNode node : nodes) {
        if(node instanceof Task) {
            node.setBehavior(new TaskBehavior());
        }
    }
}
```

在 **initSequenceFlow** 方法中，主要设置各个节点的出入顺序流，开始事件只有出口，不存在上一个节点，结束事件只有入口，不存在下一个节点。最后遍历流程中的其他节点，根据 **XML** 中的顺序流来设定顺序流在节点中的出入口。

11.6 编写客户端代码

与我们前面使用 **Activiti** 一样，编写客户端代码，加载我们定义的流程文件，启动流程并且完成任务，如代码清单 18-8 所示。

代码清单 18-8: codes\18\18.1\my-bpmn\src\org\crazyit\activiti\TestMain.java

```
String path = TestMain.class.getResource("/").toString();
File xmlFile = new File(new URI(path + "/myBpmn.xml"));
```

```
// 解析流程文件
MyProcess process = XStreamUtil.toObject(xmlFile);
// 启动流程
MyRuntimeService runtimeService = new MyRuntimeService();
MyExecution exe = runtimeService.startProcess(process);
// 查询流程当前节点
System.out.println("当前流程节点: " + exe.getCurrentNode().getId());
// 完成任务
runtimeService.completeTask(exe);
System.out.println("当前流程节点: " + exe.getCurrentNode().getId());
```

运行代码清单 18-8，输出结果如下：

```
执行开始节点
当前流程节点: task
执行任务节点
当前流程节点: end
```

根据结果可知，开始流程后，当前流程节点到达 **task**，调用服务方法完成任务后，流程到达 **end**。

本例使用了一个迷你版的流程引擎来讲述 **Activiti** 对于流程的控制逻辑，实际中，**Activiti** 在这部分的设计更为复杂。在 **Activiti5** 时代，流程虚拟机（PVM）用于定义流程走向，而在 **Activiti6**，流程控制都交由流程元素本身决定。

12 DMN 规范概述

在第 14 章，我们讲解了 **Activiti** 与规则引擎的整合使用，确切来说，是 **Activiti** 与 **Drools** 规则引擎的整合。在 **Activiti6** 版本发布后，**Activiti** 开始实现 **DMN** 规范，换言之，**Activiti** 正在实现自己的规则引擎，虽然尚未完成，但已具雏形。本章将讲述 **DMN** 规范以及初步实现的 **Activiti** 规则引擎。

笔者成书时，**Activiti** 的规则引擎并没有正式发布，官方文档、API 中没有找到相关的资料，本章内容为笔者参考 **Activiti** 规则引擎模块的源代码编写而成，在以后的 **Activiti** 版本中，规则引擎的实现及发布的文档，有可能与本书所描述的内容有所冲突，望读者了解该情况。

DMN 的出现背景

DMN 是英文 **Decision Model and Notation** 的缩写，直译意为决策模型与图形。根据前章节可知，**BPMN** 是 **OMG** 公司发布的工作流规范，而 **DMN** 同样是 **OMG** 公司发布规范，该规范主要用于定义业务决策的模型和图形，1.0 版本发布于 2015 年，目前最新的是 1.1 版本，发布于 2016 年。

BPMN 主要用于规范业务流程，业务决策的逻辑由 **PMML** 等规范来定义，例如在某些业务流程中，需要由多个决策来决定流程走向，而每个决策都要根据自身的规则来决定，并且每个决策之间可能存在关联，此时在 **BPMN** 与 **PMML** 之间出现了空白，**DMN** 规范出现前，决策者无法参与到业务中。为了填补模型上的空白，新增了 **DMN** 规范，定义决策的规范以及图形，**DMN** 规范相当于业务流程模型与决策逻辑模型之间的桥梁。

虽然 DMN 只作为工作流与决策逻辑的桥梁，但实际上，规范中也包含决策逻辑部分，同时也兼容 PMML 规范所定义的表达式语言。换言之，实现 DMN 规范的框架，同时也会具有业务规则的处理能力。

Activiti 与 Drools

Activiti 作为一个工作流引擎，与规则引擎 Drools 本来没有可比之处，它们之间更像互补关系，但是目前 Activiti 正在实现 DMN 规范，Drools 则实现了 PMML 规范，这就意味着，Activiti 的工作引擎完成后，也包含了规则引擎的功能，根据 DMN 规范可知，DMN 规范的实现者，也会对 PMML 提供支持。如此一来，Activiti 的规则引擎与 Drools 将产生竞争关系。

JBoss 旗下有工作流引擎 jBPM，有规则引擎 Drools，Activiti 本身就是工作流引擎，再加上此次更新所加入的规则引擎，估计在不久的将来，Activiti 在工作流引擎以及规则引擎领域，能与 JBoss 分庭抗礼。

DMN 的 XML 样例

DMN 主要定义决策模型，与 BPMN 规范类似，OMG 发布的 DMN 规范含有对应的 XML 约束。当前版本的 Activiti 实现了 decision 部分，因此本章只讲述 DMN 中的 decision 部分。DMN 的 XML 文档，一般情况下文件名后缀为 dmnn。代码清单 15-1 是一份简单的 DMN 文档。

代码清单 15-1: codes\15\15.1\sample.dmn

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/DMN/20151130"
  id="simple" name="Simple" namespace="http://activiti.org/dmn">
  <decision id="decision1" name="Simple decision">
    <decisionTable id="decisionTable">
      <input id="input1">
        <inputExpression id="inputExpression1" typeRef="string">
          <text>input1</text>
        </inputExpression>
      </input>
      <output id="output1" label="Output 1" name="output1" typeRef="string" />
      <rule>
        <inputEntry id="inputEntry1">
          <text><![CDATA[.startsWith('Angus')]]></text>
        </inputEntry>
        <outputEntry id="outputEntry1">
          <text>'Hello, man!'</text>
        </outputEntry>
      </rule>
      <rule>
        <inputEntry id="inputEntry2">
          <text><![CDATA[.startsWith('Paris')]]></text>
        </inputEntry>
```

```

        <outputEntry id="outputEntry2">
            <text>'Hello, baby!</text>
        </outputEntry>
    </rule>
</decisionTable>
</decision>
</definitions>

```

代码清单 15-1 中的 XML 文档，定义了一个 **decision** 节点，该节点中含有一个输入参数、一个输出结果和两个规则。注意代码清单的粗体字代码，使用了 **startsWith** 方法，定义了如果参数字符串以“Angus”开头，则触发第一个规则，如果参数字符以“Paris”开头，则触发第二个规则。关于 XML 文档中各个元素的描述，将在后面章节中讲述。

13 DMN 的 XML 规范

DMN 规范的官方网址为：<http://www.omg.org/spec/DMN/>，在官方网站上可以获取到 DMN 的规范文档、DMN 的 XML Schema 文档和样例文档。笔者已经将以上三份文档下载，并保存到代码目录，以下为这三份文档的代码路径：

- ❑ 规范文档：codes\15\15.2\DMN 规范.pdf
- ❑ XML Schema：codes\15\15.2\dmn.xsd
- ❑ 样例文档：codes\15\15.2\example.xml

决策

在 DMN 规范中，根节点为 **definitions**，该节点下可以出现 **import**、**itemDefinition**、**drgElement** 等元素，其中 **drgElement** 是一个抽象元素，**decision** 元素继承于 **drgElement**。一个 **decision** 表示一次决策，可以为它设置 **name**、**id**、**label** 属性，按照 DMN 规范，**name** 属性是必需的，而其他属性则是可选的，但作为 **decision** 的唯一标识，建议 **id** 也需要设置。一个 **definitions** 下可以定义 0 个或多个 **decision**。代码清单 15-2 定义了一个 **decision** 元素。

代码清单 15-2：codes\15\15.2\decision.dmn

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/DMN/20151130"
    id="simple" name="Simple" namespace="http://activiti.org/dmn">
    <decision id="decision1" name="Simple decision">
        <decision>
    </decision>
</definitions>

```

一个 **decision** 元素由 **question**、**allowedAnswers**、**expression** 等元素组成，其中 **expression** 元素表示决策逻辑，**expression** 是一个抽象元素，DMN 规范中，**decision** 元素下的 **expression** 可出现 0 次或 1 次。

决策表

一个 `decisionTable` 元素表示一个决策表，`decisionTable` 继承于 `expression` 元素，因此在此 `decision` 元素下，`decisionTable` 只允许出现 0 次或 1 次，这是 DMN 定义规范，在 Activiti 的实现中，`decision` 元素下如果不提供 `decisionTable`，会报出异常。代码清单 15-3 定义了一个 `decisionTable` 元素以及它的几个子元素。

代码清单 15-3: codes\15\15.2\decisionTable.dmn

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/DMN/20151130"
  id="simple" name="Simple" namespace="http://activiti.org/dmn">
  <decision id="decision1" name="Simple decision">
    <decisionTable id="decisionTable">
      <input/>
      <output/>
      <rule></rule>
    </decisionTable>
  </decision>
</definitions>
```

根据 DMN 规范，一个 `decision` 下最多只有一个 `decisionTable`，虽然规范中定义可以出现 0 次，但这样做就失去了意义，如果不提供 `decisionTable`，Activiti 的规则引擎会报出异常，信息为：`java.lang.IllegalArgumentException: no decision table present in decision`。

决策表元素有四个属性：`hitPolicy`、`aggregation`、`preferredOrientation` 和 `outputLabel`，其中 `hitPolicy` 属性用于定义规则冲突策略。

元素 `decisionTable` 下的 `input`、`output` 和 `rule` 子元素，可以出现多次，其中规范中定义了 `output` 元素最少出现一次，也就是意味在规范中，一次业务决策必须产生一个结果。在 Activiti 的初步实现中，`decisionTable` 元素下，`input`、`output` 和 `rule` 三个子元素都必须出现一次。

输入参数

在 `decisionTable` 元素下，可以添加多个 `input` 元素来声明输入参数，为 `input` 元素增加 `inputExpression` 元素来声明输入参数的类型以及名称等信息，代码清单 15-4 中定义了输出参数。

代码清单 15-4: codes\15\15.2\input.dmn

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/DMN/20151130"
  id="simple" name="Simple" namespace="http://activiti.org/dmn">
  <decision id="decision1" name="Simple decision">
    <decisionTable id="decisionTable">
      <input id="inputId">
        <inputExpression id="inputExpressionId" typeRef="string">
          <text>personName</text>
        </inputExpression>
      </input>
      <output/>
    </decisionTable>
  </decision>
</definitions>
```

```

        <rule></rule>
      </decisionTable>
    </decision>
  </definitions>

```

代码清单 15-4 的粗体部分，定义了一个参数名称为“**personName**”的输入参数，类型字符串。需要注意的，参数名是 **personName**，而不是 **inputId**。

输出结果

每个决策表至少有一个输出结果，使用 **output** 元素定义输出参数的名称以及数据类型。代码清单 15-5 定义了输出结果。

代码清单 15-5: codes\15\15.2\output.dmn

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/DMN/20151130"
  id="simple" name="Simple" namespace="http://activiti.org/dmn">
  <decision id="decision1" name="Simple decision">
    <decisionTable id="decisionTable">
      <input id="inputId">
        <inputExpression id="inputExpressionId" typeRef="string">
          <text>personName</text>
        </inputExpression>
      </input>
      <output id="resultId" label="Output 1" name="resultName" typeRef="string"
    </output>
    <rule></rule>
  </decisionTable>
</decision>
</definitions>

```

代码清单 15-5 的粗体字代码，定义了一个名称为“**resultName**”的输出结果，类型为字符串类型。需要注意的是，获取结果时，要根据 **name** 属性来获取，而不是 **id** 属性。

规则

在决策表元素下可以定义多个 **rule** 元素，**rule** 元素下可以添加 **inputEntry** 与 **outputEntry** 元素，其中 **inputEntry** 元素下支持使用 MVEL 表达式来实现业务规则的判断逻辑，而 **outputEntry** 元素则表示规则结果的输出，一个 **rule** 下可以出现 0 个或多个 **inputEntry**，而 **outputEntry** 最少出现 1 次。代码清单 15-6 中定义两个 **rule**。

代码清单 15-6: codes\15\15.2\rule.dmn

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/DMN/20151130"
  id="simple" name="Simple" namespace="http://activiti.org/dmn">
  <decision id="decision1" name="Simple decision">
    <decisionTable id="decisionTable" hitPolicy="UNIQUE">
      <input id="input1">
        <inputExpression id="inputExpression1" typeRef="number">

```

```

        <text>personAge</text>
      </inputExpression>
    </input>
    <input id="input2">
      <inputExpression id="inputExpression2" typeRef="string">
        <text>personName</text>
      </inputExpression>
    </input>
    <output id="outputId" label="Output 1" name="myResult" typeRef="string" />
    <rule>
      <inputEntry id="inputEntry2">
        <text><![CDATA[ > 18 ]]></text>
      </inputEntry>
      <inputEntry id="inputEntry2_2">
        <text><![CDATA[ .equals('Angus') ]]></text>
      </inputEntry>
      <outputEntry id="outputEntry2">
        <text>'Man Angus'</text>
      </outputEntry>
    </rule>
    <rule>
      <inputEntry id="inputEntry1">
        <text><![CDATA[ <= 18 ]]></text>
      </inputEntry>
      <outputEntry id="outputEntry1">
        <text>'Child'</text>
      </outputEntry>
    </rule>
  </decisionTable>
</decision>
</definitions>

```

代码清单 15-6 中，定义了两个输入参数、一个输出结果以及两个规则，第一规则触发条件为第 1 个参数值大于 18、第 2 个参数值等于“Angus”，两个条件都符合时，返回“Man Angus”字符串。第二个规则触发条件则是参数 1 的值小于等于 18。

目前 Activiti 还没有完全实现 DMN 规范，只是有一个大概的轮廓，本书的主要内容为 Activiti，下面将开始介绍 Activiti 关于 DMN 规范的 API。

14 Activiti 运行第一个 DMN 应用

前面对 DMN 规范作了一个简单的讲解，本小节将带领大家开发第一个 Activiti 的规则项目，目的让大家对 Activiti 的规则引擎有一个初步了解，在成功运行第一个规则项目后，对 DMN 规范以及 Activiti 的 DMN 实现就不会感觉神秘。

建立项目

与本书前面章节的项目一样，新建一个普通的 Java 项目，后缀为.dmn 的文件存放在 resource/dmn 目录，同样依赖 common-lib/lib 目录（不包括子目录）下的 jar 包。除了依赖 Activiti 的 jar 包外，由于规则引擎使用了 liqui、mvel 等项目，因此还要导入这些项目的包，项目结构以及所使用的 jar 包如图 15-1 所示。

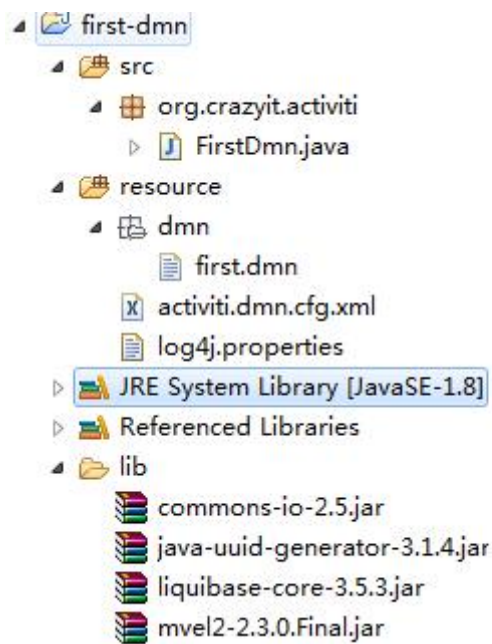


图 15-1 项目结构

需要注意的是，在导入 common-lib/lib 的包时，不要把源代码的包也导入到项目中，例如把规则引擎的源代码包（activiti-dmn-engine-6.0.0-sources.jar）导入到环境中，在运行时，会出现以下异常：org.activiti.dmn.engine.ActivitiDmnException: Error initialising dmn data model。

图 15-1 中的 resource 目录，有一份 activiti.dmn.cfg.xml 的配置文件，该文件包含规则引擎的基础配置，我们将在后面章节中讲述。

规则引擎配置文件

在默认情况下，规则引擎会读取 ClassPath 下的 activiti.dmn.cfg.xml，对于该文件，大家可能觉得比较熟悉，这个文件名，就是流程引擎配置文件的名称中间加入了 dmn 字母。而相对于配置文件的内容，几乎也是与流程引擎一样。代码清单 15-7 是本例中所使用的配置文件。

代码清单 15-7: codes\15\15.3\first-dmn\resource\activiti.dmn.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```

<bean id="dmnEngineConfiguration"
      class="org.activiti.dmn.engine.impl.cfg.StandaloneDmnEngineConfiguration">
  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/act" />
  <property name="jdbcDriver" value="com.mysql.jdbc.Driver" />
  <property name="jdbcUsername" value="root" />
  <property name="jdbcPassword" value="123456" />
</bean>

</beans>

```

规则引擎的配置文件，几乎与流程引擎的配置文件一样，配置一个 `dmnEngineConfiguration` 的 bean，为该 bean 设置 JDBC 的连接属性。规则引擎有哪些配置，将在下面章节中讲述。

编写 DMN 文件

本例中定义一个最简单的规则，当传入的年龄参数大于等于 18 时，就返回“成年人”字符串，如果年龄参数小于 18，就返回“小孩”字符串。代码清单 15-8 为本例的规则文件。

代码清单 15-8: codes\15\15.3\first-dmn\resource\dmn\first.dmn

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/DMN/20151130"
  id="simple" name="Simple" namespace="http://activiti.org/dmn">
  <decision id="decision1" name="Simple decision">
    <decisionTable id="decisionTable">
      <input id="input1">
        <inputExpression id="inputExpression1" typeRef="number">
          <text>personAge</text>
        </inputExpression>
      </input>
      <output id="outputId" label="Output 1" name="myResult" typeRef="string" />
      <rule>
        <inputEntry id="inputEntry2">
          <text><![CDATA[ >= 18 ]]></text>
        </inputEntry>
        <outputEntry id="outputEntry2">
          <text>'成年人'</text>
        </outputEntry>
      </rule>
      <rule>
        <inputEntry id="inputEntry1">
          <text><![CDATA[ < 18 ]]></text>
        </inputEntry>
        <outputEntry id="outputEntry1">
          <text>'小孩'</text>
        </outputEntry>
      </rule>
    </decisionTable>
  </decision>
</definitions>

```

规则文件中，定义了一个输入参数、一个输出结果和两个规则，在前面章节已经对相关

的 DMN 元素作了讲解，在此不再赘述。

加载与运行 DMN 文件

两个引擎不仅仅在配置上类似，连 API 的使用也非常相似。如果在本书前面的章节中，熟练掌握了 Activiti 工作流引擎的 API，那么在学习使用规则引擎的 API 也不会太难。代码清单 15-9 中为规则的运行代码。

代码清单 15-9: codes\15\15.3\first-dmn\src\org\crazyit\activiti\FirstDmn.java

```
public class FirstDmn {

    public static void main(String[] args) {
        // 根据默认配置创建引擎的配置实例
        DmnEngineConfiguration config = DmnEngineConfiguration
            .createDmnEngineConfigurationFromResourceDefault();
        // 创建规则引擎
        DmnEngine engine = config.buildDmnEngine();
        // 获取规则的存储服务组件
        DmnRepositoryService rService = engine.getDmnRepositoryService();
        // 获取规则服务组件
        DmnRuleService ruleService = engine.getDmnRuleService();
        // 进行规则 部署
        DmnDeployment dep = rService.createDeployment()
            .addClasspathResource("dmn/first.dmn").deploy();
        // 进行数据查询
        DmnDecisionTable dt = rService.createDecisionTableQuery()
            .deploymentId(dep.getId()).singleResult();
        // 初始化参数
        Map<String, Object> params = new HashMap<String, Object>();
        params.put("personAge", 19);
        // 传入参数执行决策，并返回结果
        RuleEngineExecutionResult result = ruleService.executeDecisionByKey(
            dt.getKey(), params);
        // 控制台输出结果
        System.out.println(result.getResultVariables().get("myResult"));
        // 重新设置参数
        params.put("personAge", 5);
        // 重新执行决策
        result = ruleService.executeDecisionByKey(dt.getKey(), params);
        // 控制台重新输出结果
        System.out.println(result.getResultVariables().get("myResult"));
    }
}
```

如代码清单 15-9 所示，先读取默认的配置文件来创建 DmnEngineConfiguration 实例，以该实例获取规则引擎 DmnEngine 实例，再以 DmnEngine 为基础，获取两个服务组件：DmnRepositoryService 和 DmnRuleService。DmnRepositoryService 主要负责引擎资源的部署，DmnRuleService 则提供规则的相关服务，例如可以执行规则、查询规则等。

代码清单 15-9 中，使用了 DmnRepositoryService 将 first.dmn 规则文件部署到引擎中，再根据部署的 id 去查询 DmnDecisionTable 实例。代码清单 15-9 中的粗体字代码，使用 DmnRuleService 来执行决策并返回结果，由于我们在 DMN 文件中配置了，需要有一个名

称为 **personAge** 的输入参数，因此要新建一个 **Map** 实例来保存该参数。

在以上例子中，第一次使用 **DmnRuleService** 来执行决策时，传入的“**personAge**”参数值为 19，第二次执行决策时，传入的参数值为 5，运行代码清单 15-9，输出如下：

成年人
小孩

到此，**Activiti** 的第一个应用已经成功运行，根据本小节可知，规则引擎与流程非常相似，配置的读取、规则引擎的创建、服务组件的获取方式、数据查询以及运行，与 **Activiti** 流程引擎如出一辙。如果熟悉 **Activiti** 流程引擎的话，规则引擎的 **API** 将很快掌握。

京东购买地址：<https://item.jd.com/12246565.html>

扫描关注作者公众号，免费获取多种技术的教学视频以及配套代码：



扫码获取视频、电子书