



Activiti 5.8

中文用户手册

译 栗建涛

目录

第一章、简介	1
1.1 许可	1
1.2 下载	1
1.3 源码	1
1.4 所需的软件	1
1.4.1 JDK 5+	1
1.4.2 Ant 1.8.1+	1
1.4.3 Eclipse 3.6.2	1
1.5 报告问题	1
1.6 试验性的特性	1
1.7 内部实现类	2
第二章、入门	3
2.1 一分钟版	3
2.2 演示设置	3
2.3 workspace 文件夹下的示例项目	4
2.4 类库依赖	4
2.5 Eclipse 设置	5
2.6 查看数据库	7
2.7 数据库表的命名说明	7
第三章、配置	8
3.1 创建 ProcessEngine	8
3.2 ProcessEngineConfiguration bean	9
3.3 数据库配置	9
3.4 作业执行器的激活	10
3.5 邮件服务器的配置	11
3.6 历史的配置	11
3.7 在表达式、脚本中公布配置的 beans	11
3.8 支持的数据库	11
3.9 创建数据库表	11
3.10 数据库更新	12
第四章、Spring 的集成	13
4.1 ProcessEngineFactoryBean	13
4.2 事务	13
4.3 表达式	16
4.4 自动资源部署	17
4.5 单元测试	17
第五章、API	19
5.1 引擎 API	19
5.2 异常策略	19
5.3 单元测试	20
5.4 调试单元测试	21
5.5 web 应用程序中的工作流引擎	23
5.6 流程虚拟机（PVM）API	24
5.7 表达式	25

第六章、部署	26
6.1 业务归档文件	26
6.1.1 编程式部署	26
6.1.2 使用 ant 部署	26
6.1.3 使用 Activiti Explorer 部署	27
6.2 外部资源	27
6.2.1 Java 类	27
6.2.2 在流程中使用 Spring beans	28
6.2.3 创建独立应用	28
6.3 流程定义的版本	28
6.4 提供流程图	29
6.5 生成流程图	29
第七章、BPMN	31
7.1 BPMN 是什么	31
7.2 示例	31
7.3 定义流程	31
7.4 入门：10 分钟指南	32
7.4.1 先决条件	32
7.4.2 目标	32
7.4.3 用例	33
7.4.4 流程图	33
7.4.5 XML 的描述	33
7.4.6 启动流程实例	34
7.4.7 任务列表	36
7.4.8 认领任务	37
7.4.9 完成任务	38
7.4.10 结束流程	39
7.4.11 代码综述	39
7.4.12 未来改进	41
7.5 BPMN 2.0 结构	41
7.5.1 自定义扩展	41
7.5.2 事件	42
定时器事件的定义	42
7.5.3 启动事件	43
7.5.4 空启动事件	44
描述	44
图形化符号	44
XML 表示	44
7.5.5 定时器启动事件	44
描述	44
图形化符号	44
XML 表示	45
7.5.6 终止事件	45
7.5.7 空终止事件	45
描述	45
图形化符号	45

XML 表示	46
7.5.8 异常终止事件	46
描述	46
图形化符号	46
XML 表示	46
7.5.9 顺序流	47
描述	47
图形化符号	47
XML 表示	47
7.5.10 条件顺序流	47
描述	47
图形化符号	48
XML 表示	48
7.5.11 默认顺序流	49
描述	49
图形化符号	49
XML 表示	49
7.5.12 分支	50
7.5.13 排他分支	50
描述	50
图形化符号	51
XML 表示	51
7.5.14 并行分支	52
描述	52
图形化符号	52
XML 表示	52
7.5.15 包容分支	54
描述	54
图形化符号	55
XML 表示	55
7.5.16 用户任务	56
描述	56
图形化符号	57
XML 表示	57
到期时间	57
用户的分配	57
Activiti 对于任务分配的扩展	58
7.5.17 脚本任务	60
描述	60
图形化符号	60
XML 表示	60
脚本中的变量	61
脚本的结果	61
7.5.18 Java 服务任务	61
描述	61
图形化符号	62

XML 表示	62
实现	63
字段的注入	63
服务任务的结果	65
处理异常	65
7.5.19 WebService 任务	66
描述	66
图形化符号	66
XML 表示	66
WebService 任务的 IO 规范	67
服务任务的数据输入关系	68
服务任务的数据输出关系	68
7.5.20 业务规则任务	69
描述	69
图形化符号	69
XML 表示	69
7.5.21 邮件任务	70
Mail 服务器的配置	70
定义邮件任务	70
用法举例	71
7.5.22 Mule 任务	72
定义 Mule 任务	72
用法举例	72
7.5.23 手动任务	73
描述	73
图形化符号	73
XML 表示	73
7.5.24 Java 接收任务	73
描述	73
图形化符号	74
XML 表示	74
7.5.25 执行监听器	74
执行监听器上的字段注入	76
7.5.26 任务监听器	77
7.5.27 多实例（for each）	78
描述	78
图形化符号	79
XML 表示	79
边界事件与多实例	80
7.5.28 边界事件	81
7.5.29 定时器边界事件	81
描述	81
图形化符号	81
XML 表示	81
使用边界事件的已知问题	82
7.5.30 异常边界事件	83

描述	83
图形化符号	84
XML 表示	84
示例	84
7.5.31 中间捕获事件	85
7.5.32 定时器中间捕获事件	85
描述	85
图形化符号	85
XML 表示	86
7.5.33 子流程	86
描述	86
图形化符号	86
XML 表示	87
7.5.34 调用活动（子过程）	88
描述	88
图形化符号	88
XML 表示	88
传递变量	88
示例	89
7.6 异步的延续	90
第八章、表单	92
8.1 表单属性	92
8.2 外部的表单渲染	95
第九章、JPA	96
9.1 要求	96
9.2 配置	96
9.3 用法	97
9.3.1 简单示例	97
9.3.2 查询 JPA 流程变量	99
9.3.3 使用 Spring beans 和 JPA 的高级示例	99
第十章、历史	102
10.1 查询历史	102
10.1.1 HistoricProcessInstanceQuery	102
10.1.2 HistoricActivityInstanceQuery	102
10.1.3 HistoricDetailQuery	103
10.1.4 HistoricTaskInstanceQuery	104
10.2 历史的配置	104
10.3 审查目的的历史	105
第十一章、Eclipse Designer	106
11.1 安装	106
11.2 Activiti Designer 编辑器的特性	107
11.3 Activiti Designer 的 BPMN 特性	109
11.4 Activiti Designer 的部署特性	113
11.5 扩展 Activiti Designer	114
11.5.1 定制画板	114
11.5.1.1 扩展的设置（Eclipse/Maven）	115

11.5.1.2 将扩展应用到 Activiti Designer	117
11.5.1.3 向画板添加形状	119
11.5.1.4 属性的类型	122
11.5.1.5 禁用画板中默认形状	125
11.5.2 校验图形和导出到自定义的输出格式	127
11.5.2.1 创建 ProcessValidator 扩展	128
11.5.2.2 创建 ExportMarshaller 扩展	129
第十二章、Activiti Explorer	131
12.1 用例概述	131
12.2 用例	132
12.3 启动流程实例	132
12.4 我的实例	132
12.5 管理	133
12.6 修改数据库	135
第十三章、Activiti 的附加组件	136
13.1 Cycle	136
13.2 基于 Signavio 核心组件的 Activiti Modeler	136
第十四章、REST API	137
14.1 仓库	137
14.1.1 上传部署	137
14.1.2 获取部署	138
14.1.3 获取部署资源	138
14.1.4 删除部署	138
14.1.5 删除多个部署	139
14.2 引擎	139
14.2.1 获取流程引擎	139
14.3 流程	140
14.3.1 列出流程定义	140
14.3.2 获得流程定义	140
14.3.3 获得流程定义表单	141
14.3.4 启动流程实例	141
14.3.5 列出流程实例	142
14.3.6 获得流程实例图	143
14.4 任务	143
14.4.1 获取任务概述	143
14.4.2 列出任务	143
14.4.3 获得任务	144
14.4.4 获得任务表单	145
14.4.5 执行任务操作	145
14.4.6 列出表单属性	145
14.5 身份	146
14.5.1 登陆	146
14.5.2 获得用户	146
14.5.3 列出用户的组	146
14.5.4 获取组	147
14.5.5 列出组内的用户	147

14.6 管理	148
14.6.1 列出作业	148
14.6.2 获得作业	149
14.6.3 执行作业	149
14.6.4 执行多个作业	149
14.6.5 列出数据库表	150
14.6.6 获得表的元数据	150
14.6.7 获得表数据	150
第十五章、Cdi 集成	152
15.1 设置 activiti-cdi	152
15.1.1 查找流程引擎	152
15.1.2 配置流程引擎	153
15.1.3 部署流程	154
15.2 存在 Cdi 的上下文相关的流程的执行	154
15.2.1 将会话与流程实例关联	154
15.2.2 声明式地控制流程	155
15.2.3 在流程中引用 Bean	155
15.2.4 使用@BusinessProcessScoped 注解的 bean	156
15.2.5 注入流程变量	156
15.2.6 接收流程事件	156
15.2.7 附加特性	157
15.3 编写测试	157
15.4 已知的局限性	158
附录	159
附录一 认识 ant 构建脚本	159
附录二 认识发布文件结构	160
翻译日程	161
关于文档	162

第一章、简介

1.1 许可

Activiti 是在 [Apache V2 许可](#)下发布的。

1.2 下载

<http://activiti.org/download.html>

1.3 源码

发布中以 jar 文件的形式包含了大部分的源码。要想寻找并构建完整的源码库，请阅读‘[构建发布](#)’的 wiki 页面。

1.4 所需的软件

1.4.1 JDK 5+

Activiti 运行在版本 5 以上的 JDK 上。转到 [Oracle Java SE 下载页面](#)，点击按钮“下载 JDK”。网页中也有安装说明。要核实安装是否成功，在命令行上运行 `java -version`。将打印出安装的 JDK 的版本。

1.4.2 Ant 1.8.1+

从 [Ant 下载页面](#)下载最新稳定版的 Ant。解压文件，确保其 bin 文件夹在操作系统的 path 下。在命令行上运行 `ant -version`来检查 Ant 是否安装成功。将打印安装的 Ant 版本。

1.4.3 Eclipse 3.6.2

从 [Eclipse 的下载页面](#)下载 **Eclipse Classic** 版的 eclipse。解压下载的文件，然后就可以运行 eclipse 路径下的 eclipse 文件了。这个指南的后面，有关于[在 eclipse 中设置 Activiti 示例项目](#)的一节以及关于[安装 eclipse designer 插件](#)的一节。

1.5 报告问题

每个自重的开发者都可能阅读过[如何巧妙的问问题](#)。

读完后，你可以将问题和意见发送到[用户论坛](#)上，在我们 [JIRA 论题追踪器](#)中创建论题。

1.6 试验性的特性

标记了[EXPERIMENTAL]的章节不应该被认为是稳定的。

所有包名中含有`.impl`的类都是内部实现类，不能视为稳定的。然而，如果用户指南中是以配置值提及到那些类的，那么它们是被支持的，并且可以被看做是稳定的。

1.7 内部实现类

jar 文件内，含有`.impl`的包（如，`org.activiti.engine.impl.pvm.delegate`）内的所有类都是实现类，应该被视为是内部的。不能保证实现类里的类及接口的稳定性。

第二章、入门

2.1 一分钟版

从 [Activiti 站点](#) 下载完 Activiti 发布的 zip 文件后,按照此步骤运行默认设置的演示安装程序。你需要运行 [Java 运行时环境](#),并安装 [Ant](#)。

- 解压 Activiti 发布的 zip 文件。
- 打开终端窗体（译注，即命令行窗体），导航到解压文件的 setup 文件夹内。
- 输入 **ant.demo.start**，然后按 enter 键。
- 脚本完成后，会在浏览器内启动所有的 Activiti 的 web 应用。以 kermit/kermit 登录。

就是这样！如果想要了解更多关于上面步骤中实际上发生了什么，参看[更长的版本](#)。

你也可以在[十分钟指南](#)中学习到所有关于 Activiti 和 BPMN 2.0 的内容。

2.2 演示设置

‘演示设置’是 setup 目录下的一个 ant 脚本，它会立即设置 Activiti 环境。

要执行该脚本，需要运行 [Java 运行时环境](#)，并安装 [Ant](#)。并确保正确设置了 `JAVA_HOME` 和 `ANT_HOME` 系统变量。具体操作取决于你的操作系统，[ant 的指南](#)给出了对此的描述。此演示安装程序是使用 Ant 1.7.1+来进行测试的。

做法是，在命令提示行中打开解压过的 Activiti 发布包中的 setup 文件夹，然后输入：

```
ant demo.start
```

此 ant 脚本会按以下步骤执行：

- (*) 构建 web 应用。所有类库都存储在`${activiti.home}/setup/files/dependencies/libs`。不带类库的那些 web 应用都存储在`${activiti.home}/setup/files/webapps`。构建 web 应用指的就是`${activiti.home}/setup/files/webapps`下结合了类库的那些 web 应用程序。
- (*) 在`${activiti.home}/apps/h2`下安装 H2。
- 启动 H2 数据库。
- (*) 在 H2 数据库中创建 Activiti 的表
- (*) 在 Activiti identity 表中插入演示用户和组（如下）
- (*) 将示例流程部署到 Activiti 引擎数据库中
- (*) 如果`${downloads.dir}`没有 Tomcat，就下载 Tomcat。
- (*) 在`${activiti.home}/apps/apache-tomcat-${tomcat.version}`下安装 Tomcat。
- (*) 创建 Activiti 配置的 jar
- (*) 将 REST 接口应用部署到 tomcat 下。
- (*) 将 Activiti web 应用部署到 tomcat 下。
- 启动 tomcat。

(*) 只在第一次运行 demo.start 时被执行。

运行该任务后，H2 和 Tomcat 会在后台运行。运行 `ant demo.stop` 来结束这些进程。

也可以单独调用构建脚本内的其它任务。运行 `ant -p` 来获取详细信息。

存在的演示用户：

表 2.1 演示用户

用户 Id	密码	安全角色
kermit	kermit	管理员
gonzo	gonzo	经理
fuzzid	fuzzid	用户

现在就可以访问下面的 web 应用程序了：

表 2.2 web 应用工具

Web 应用名称	URL	描述
Activiti Explorer	http://localhost:8080/activiti-explorer	流程引擎用户控制台。利用此工具来启动新的流程、分配任务、查看并认领任务等等。此工具也可以用来管理 Activiti 引擎。

注意 Activiti 的演示程序的安装以一种尽量简单而快速的方式来展示 Activiti 能力和功能。然而，这并不意味着 Activiti 只有这一种使用方式。因为 Activiti 仅仅是一个 jar，所以可以嵌入在任何 Java 环境下：swing 或在 Tomcat、JBoss、WebSphere 下，等等。或者你也可以作为典型、独立的 BPM 服务器来运行 Activiti。能使用 Java，就能使用 Activiti。

2.3 workspace 文件夹下的示例项目

发布包中有一个 workspace 目录，里面包含一些 java 例子项目：

- **activiti-engine-examples:** 这套示例展示了 Activiti 最常用的用法：BPMN 流程定义和流程的执行被存储在数据库中，并且示例中使用了持久化 API。
此项目包含 Eclipse 项目文件、ant 的构建文件以及 maven pom 文件。ant 构建文件是独立于 maven pom 的。两者展示了如何单独使用 ant 或 maven 来构建、部署流程。
- **activiti-spring-examples:** 这些示例展示了在 Spring 环境下如何使用 Activiti 引擎。
- **activiti-groovy-examples:** 这些示例展示了 groovy 的类库依赖以及一个使用 groovy 脚本的流程。
- **activiti-jpa-examples:** 这些示例展示了类库依赖以及 Activiti 中如何使用 JPA。
- **activiti-cxf-examples:** 这些示例展示了类库依赖以及在 Activiti 中如何使用 web 服务。

“[Eclipse 的设置](#)”一节介绍了如何设置 eclipse 环境来演示这些示例项目。

作为 deno.start 的一部分，这些示例项目会被添加进来。这意味着所有的 libs 和配置文件都会被放到一个恰当的位置。如果不运行 deno.start，要想将带有 libs 的这些项目添加在一个合适的位置，就要运行 setup 目录下的此命令：

```
ant inflate.examples
```

之后，activiti-engine-examples 和 activiti-spring-examples 就会包含 libs-runtime 路径和 libs-test 路径，它们分别着包含运行时的依赖 jars 和测试期间的依赖 jars。

2.4 类库依赖

为了防止由于重复包含类库而导致发布的文件过大，把所有的类库都组织到了 setup/files/dependencies/libs/文件夹下。

setup.build.xml 文件内的 ant 脚本将利用 libs 内的库来扩充这些示例（inflate.example 任务），当这些 web 应用被构建完成时，它们将包含相应的 libs。

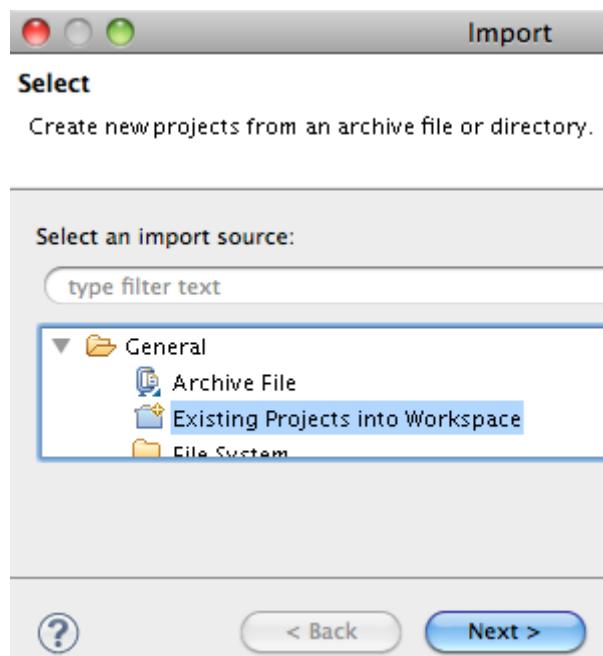
下面这些存在于 setup/files/dependencies 内的文件描述了这些类库的依赖：

- libs.engine.runtime.txt：运行 Activiti 引擎所需的类库。
- libs.engine.runtime.test.txt：连同 libs.engine.runtime.txt 内，测试所需的类库。
- libs.engine.runtime.feature.groovy.txt：连同 libs.engine.runtime.txt 内，使用 groovy 脚本所需的类库。
- libs.engine.runtime.txt：连同 libs.engine.runtime.txt 内，使用 JPA 变量引用能力的类库。
- libs.spring.runtime.txt：在 Spring 环境下运行 Activiti 引擎所有的运行时类库。（包含了 libs.engine.runtime.txt 内的库）
- libs.spring.runtime.test.txt：连同 libs.engine.runtime.txt 内，在 Spring 环境下运行测试所需的类库。

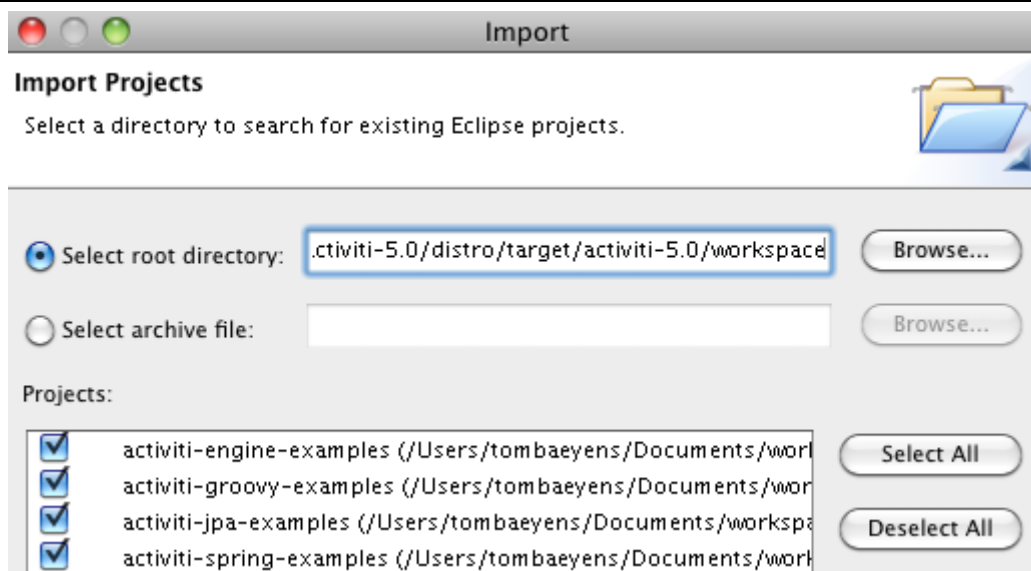
2.5 Eclipse 设置

要想在 eclipse 内运行并演示这些示例，需要进行几步简单操作：

File → Import ...



选择 General → Existing Projects into Workspace，然后点击 Next。

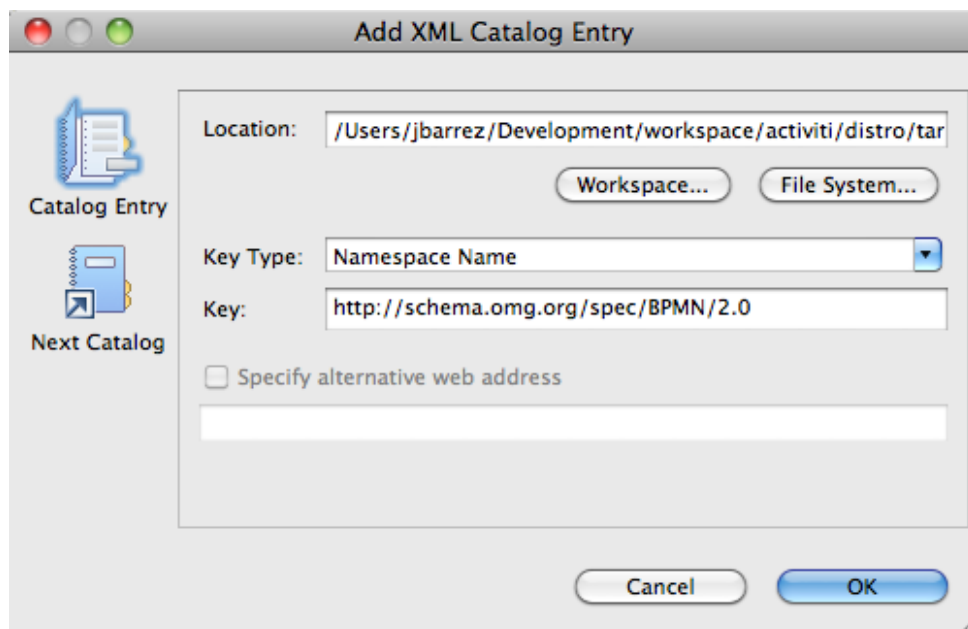


点击'Browse...', 选择目录`${activiti.home}/workspace`, 接下来你会看到示例项目自动被选中了。

然后，点击 Import 对话框内的 Finish 按钮，这样就设置完了。

方便起见，打开 ant 视图（Window → Show View → Ant），将文件 `activiti-engine-examples/build.xml` 拖入 ant 窗口内。这样，你就可以双击运行构建目标了。

要使 BPMN 2.0 XML 在输入时能进行自动补全、校验，你可以将 BPMN 2.0 XML 的模式添加进 XML catalog。进入 Window --> Preferences --> XML --> XML Catalog --> Add，从 file system 内选择文件夹\${activiti.home}/docs/xsd/BPMN20.xsd。重复的操作配置\${activiti.home}/docs/xsd/activiti-bpmn-extensions-5.4.xsd。



参看 11 章. Eclipse Designer 关于如何将 Activiti Designer 插件安装到 Eclipse 的说明。

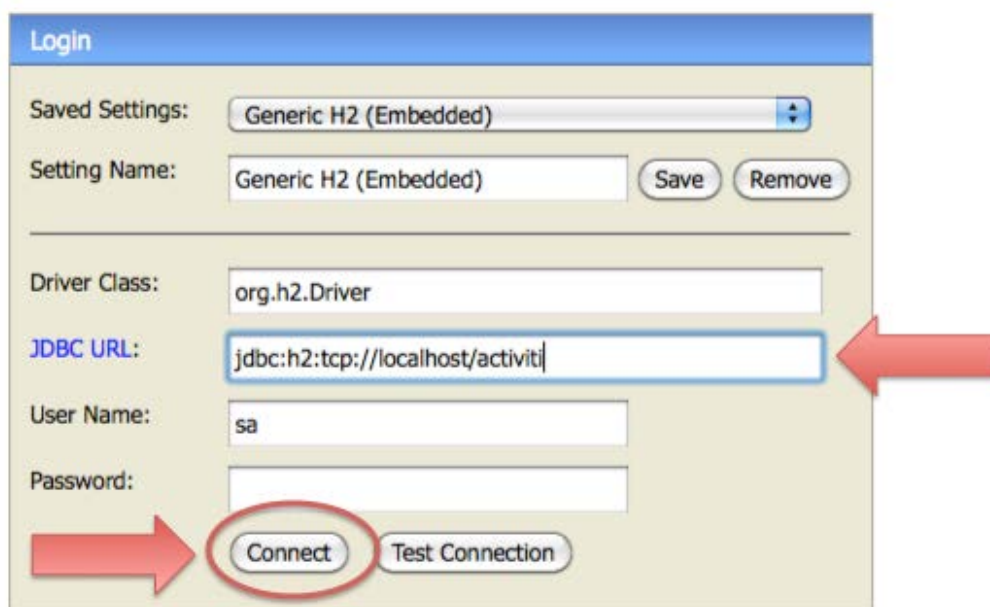
2.6 查看数据库

测试完演示设置后，要想查看数据库，需要运行 `setup` 文件夹下的如下 Ant 目标。

```
ant h2.console.start
```

这将启动 H2 的 web 控制台。注意这个 Ant 目标没有返回，所以需要使用 `CTRL + C` 关闭此控制台。在 JDBC URL 域内输入下面的 URL，然后点击 Connect:

```
jdbc:h2:tcp://localhost/activiti
```



现在就能够浏览 Activiti 的数据库模式并查看这些表的内容了。

要想修改数据库，见“[修改数据库](#)”一节。

2.7 数据库表的命名说明

Activiti 数据库中表的命名都是以 `ACT_` 开头的。第二部分是一个两个字符的表用例的标识。此用例大体与服务 API 是匹配的。

- **ACT_RE_***: 'RE'代表 repository。带此前缀的表包含的是静态信息，如，流程定义、流程的资源（图片、规则，等）。
- **ACT_RU_***: 'RU'代表 runtime。就是这个运行时的表存储着流程变量、用户任务、变量、作业等内的运行时的数据。Activiti 只存储流程实例执行期间的运行时数据，当流程实例结束时，将删除这些记录。这就保持了这些运行时的表的小且快。
- **ACT_ID_***: 'ID'代表 identity。这些表包含着标识信息，如用户、用户组等等。
- **ACT_HI_***: 'HI'代表 history。就是这些表包含着历史的相关数据，如结束的流程实例、变量、任务、等等。
- **ACT_GE_***: 通用数据，各种情况都使用的数据。

第三章、配置

3.1 创建 ProcessEngine

Activiti 流程引擎是通过 `acitiviti.cfg.xml` 文件进行配置的。注意，这不适用于使用 [Spring 构建流程引擎](#) 的情况。

获得 ProcessEngine 最简单的方式是使用 `org.activiti.engine.ProcessEngines` 类：

```
ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine()
```

它会查找类路径下的 `activiti.cfg.xml` 文件，然后根据文件中的配置构建引擎。下面的片段展示了一个示例配置。下一节将对配置属性做详细的介绍。

```
<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="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">

    <property name="jdbcUrl" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000"/>
    <property name="jdbcDriver" value="org.h2.Driver"/>
    <property name="jdbcUsername" value="sa"/>
    <property name="jdbcPassword" value=""/>

    <property name="databaseSchemaUpdate" value="true"/>

    <property name="jobExecutorActivate" value="false"/>

    <property name="mailServerHost" value="mail.my-corp.com"/>
    <property name="mailServerPort" value="5025"/>
  </bean>

</beans>
```

注意该 `xml` 配置文件实际上是一个 `Spring` 配置文件。这并不意味着 **Activiti 只能在 Spring 环境下使用**！我们只是利用 `Spring` 的解析和依赖注入能力来构建引擎。

也可以以编程的方式使用配置文件创建 `ProcessEngineConfiguration` 对象。也可以使用 `bean` 的 `id`（例子，见第 3 行）。

```
ProcessEngineConfiguration.createProcessEngineConfigurationFromResourceDefault();
ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource);
ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource, String beanName);
ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(InputStream inputStream);
ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(InputStream inputStream, String beanName);
```

也可以不使用配置文件，创建默认的配置对象（更多信息，参看[所支持的其它类](#)）。


```
ProcessEngineConfiguration.createStandaloneProcessEngineConfiguration();
ProcessEngineConfiguration.createStandaloneInMemProcessEngineConfiguration();
```

所有这些 `ProcessEngineConfiguration.createXXX()` 方法都会返回 `ProcessEngineConfiguration` 对象，可以进一步对其做必要的调整。调用 `buildProcessEngine()` 方法后，会创建一个 `ProcessEngine`：

```
ProcessEngine processEngine = ProcessEngineConfiguration.createStandaloneInMemProcessEngineConfiguration()
    .setDatabaseSchemaUpdate(ProcessEngineConfiguration.DB_SCHEMA_UPDATE_FALSE)
    .setJdbcUrl("jdbc:h2:mem:my-own-db;DB_CLOSE_DELAY=1000")
    .setJobExecutorActivate(true)
    .buildProcessEngine();
```

3.2 ProcessEngineConfiguration bean

`activiti.cfg.xml` 文件内必须包含一个 id 为 'processEngineConfiguration' 的 bean。

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
```

这个 bean 接下来被用来构造 `ProcessEngine`。存在多个可用于定义 `processEngineConfiguration` 的类。这些类代表着不同的环境，有相应的默认设置。最好选择（最）适合你环境的那个类，这样会尽量减小需要配置流程引擎属性的数量。以下是目前可用的类（将来可能会更多）：

- **org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration**：此流程引擎用于独立环境下。Acitviti 会处理事务。默认，只在该引擎启动时检查数据库（不存在 Activiti 数据库模式或模式版本不合适都会抛出异常）。
- **org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration**：这是一个方便单元测试的类。Acitviti 会处理事务。默认使用 H2 内存数据库。此数据库在流程引擎启动与关闭时进行创建和删除。使用这个类时，可能不需要进行额外的配置（除了在使用作业执行器（job executor）或邮件功能时）。
- **org.activiti.spring.SpringProcessEngineConfiguration**：Spring 环境下使用流程引擎的情况下使用。更多信息，参看 [Spring 集成](#) 一节。
- **org.activiti.engine.impl.cfg.JtaProcessEngineConfiguration**：（[试验性的](#)）使用 JTA 事务，以独立模式运行的流程引擎的环境下使用。

3.3 数据库配置

有两种方式来配置 Activiti 引擎使用的数据库。第一个选择就是定义数据库的 jdbc 属性：

- **jdbcUrl**：数据库的 jdbc url。
- **jdbcDriver**：特定数据库类型驱动的实现。
- **jdbcUsername**：连接数据库的用户名。
- **jdbcPassword**：连接数据库的密码。

根据提供的 jdbc 属性创建的 datasource 使用默认配置的 [MyBatis](#) 连接池。可以选择性地设置以下属性来调整连接池（出自 MyBatis 文档）：

- **jdbcMaxActiveConnections**: 任何时候连接池所包含有效连接的最大个数。默认是 10。
- **jdbcMaxIdleConnections**: 任何时候连接池所包含闲置连接的最大个数。
- **jdbcMaxCheckoutTime**: 连接从连接池‘检出’到真正获取到之间的毫秒数。默认为 20000（20 秒）。
- **jdbcMaxWaitTime**: 这是一个低层次的设置，当一个连接占用了过长时间，它让连接池有机会打印日志，并重新尝试获取连接（以避免如果误配了连接池，发生永久性失败）。默认为 20000（20 秒）。

数据库配置的示例：

```
<propertyname="jdbcUrl" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000"/>
<propertyname="jdbcDriver" value="org.h2.Driver"/>
<propertyname="jdbcUsername" value="sa"/>
<propertyname="jdbcPassword" value=""/>
```

或者，可以使用 javax.sql.DataSource 的实现（例如，[Apache Commons](#) 的 DBCP）：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
<propertyname="driverClassName" value="com.mysql.jdbc.Driver"/>
<propertyname="url" value="jdbc:mysql://localhost:3306/activiti"/>
<propertyname="username" value="activiti"/>
<propertyname="password" value="activiti"/>
<propertyname="defaultAutoCommit" value="false"/>
</bean>

<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">

<propertyname="dataSource" ref="dataSource"/>

...
```

注意，Activiti 中没有包含支持 datasource 的类库。所以你必须确保类路径下存在那些类库（比如，DBCP）。

不管是使用 jdbc 还是 datasource，都可以设置以下属性：

- **databaseType**: 一般不需要指定这个属性，因为引擎会自动从数据库连接元数据分析得到。只在为防止万一自动检测失败时指定。可能的值有：{h2, mysql, oracle, postgrs, mssql, db2}。在不使用默认的 H2 数据库时，这个属性是必须的。该设置决定了使用哪种 create/drop 脚本和查询。（译注，这句话的意思是，使用哪种数据风格的 DML、DCL）见‘[支持的数据库](#)’一节来查看支持哪种数据库。
- **databaseSchemaUpdate**: 允许在流程引擎启动和关闭时设置处理数据库模式的策略。
 - false（默认）：创建流程引擎时检查数据库模式的版本是否与类库匹配，如果版本不匹配就会抛出异常。
 - true：构建流程引擎时，执行检查，如果有必要会更新数据库模式。如果数据库模式不存在，就创建一个。
 - create-drop：创建流程引擎时创建数据库模式，关闭流程引擎时删除数据库模式。

3.4 作业执行器的激活

JobExecutor 是管理触发定时器线程的组件（以及接下来介绍的异步消息）。进行单元测试时，处理多线程是很多余的。因此该 API 允许对作业（jobs）进行查询（ManagementService.createJobQuery）和执行（ManagementService.executeJob），这样一来就可以在单元测试中控制作业的执行了。要想避免该作业执行器接口，可以将其关闭。

默认，JobExecutor 在流程引擎启动时被激活。当不想让 JobExecutor 在流程引擎启动时被激活，指定

```
<propertyname="jobExecutorActivate"value="false"/>
```

3.5 邮件服务器的配置

可选的。Activiti 支持在业务流程内发送电子邮件。要想发送电子邮件，需要配置一个有效的 SMTP 邮件服务器。关于配置选项，见[邮件任务](#)。

3.6 历史的配置

可选的。允许对流程引擎的[history 功能](#)进行设置。详细介绍，见[历史的配置](#)。

```
<propertyname="history"value="audit"/>
```

3.7 在表达式、脚本中公布配置的 beans

默认，activiti.cfg.xml 配置文件以及你自己的 Spring 配置文件中指定的所有 beans 都可以在表达式和脚本内使用。如果想要限制配置文件中 beans 的可见性，可以配置流程引擎配置中的 beans 属性。ProcessEngineConfiguration 的 beans 属性是个 map。当指定了此属性后，只有指定在该 map 中的 beans 对表达式和脚本才是可见的。公布出来的 beans 是以指定在该 map 内的名称进行公布的。

3.8 支持的数据库

以下是 Activiti 用于引用数据库的类型（大小写敏感）。

表 3.1 支持的数据库

Activiti 数据库类型	被测试的版本	jdbc url 示例	注意
h2	1.2.132	jdbc:h2:tcp://localhost/activiti	默认配置的数据库
mysql	5.1.11	jdbc:mysql://localhost:3306/activiti?autoReconnect=true	测试使用的是 mysql-connector-java 数据库驱动
oracle	10.2.0	jdbc:oracle:thin:@localhost:1521:xe	
postgres	8.4	jdbc:postgresql://localhost:5432/activiti	
db2	DB2 9.7 使用 db2jcc4	jdbc:db2://localhost:50000/activiti	[试验性的]
mssql	2008 使用 JDBC jtds-1.2.4	jdbc:jtds:sqlserver://localhost:1433/activiti	[试验性的]

3.9 创建数据库表

为你的数据库创建数据库表的最简单的方式是

- 将 Activiti 引擎的 jars 添加到你的 classpath 下

- 添加适当的数据库驱动
- 添加 Activiti 的配置文件（*activiti.cfg.xml*）到你的 classpath 下，配置文件指向你的数据库（见[数据库配置一节](#)）
- 执行 *DbSchemaCreate* 类的 main 方法

然而，通常只有数据库管理员才能够在数据库上执行 DDL 语句。此 DDL 的 SQL 语句可以在 Activiti 引擎的 jar 文件（*activiti-engine-x.jar*）中的 *org/activiti/db/create* 包（*drop* 文件夹包含了 *dorp* 语句）找到。sql 文件的格式为

```
activiti.{db}.{create|drop}.{type}.sql
```

其中 *db* 是任何所[支持的数据库](#)，*type* 为

- **engine:** 引擎执行所需的表。必需的。
- **identity:** 可选表，在使用随引擎分发的默认 identity 管理时使用。
- **history:** 包含历史和审核信息。可选：当历史级别设置为 *none* 时不需要。

3.10 数据库更新

[\[试验性的\]](#)

对于更新的完整测试我们还没有充分的信心。这就是为什么我们将其标记为试验性的原因。在运行更新前，确保将数据库进行了备份（利用你数据库的备份能力）。

默认，每次创建流程引擎时都会执行版本的检查。这一般发生在你的应用程序或 Activiti web 应用启动的时候。如果 Activiti 的类库发现类库的版本与 Activiti 数据库表的版本不一样，就会抛出异常。

要想更新，必须要在你的 *activiti.cfg.xml* 配置文件配置以下配置属性：

```
<beans... >

<beanid="processEngineConfiguration"
    class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
<!-- ... -->
<propertyname="databaseSchemaUpdate"value="true"/>
<!-- ... -->
</bean>

</beans>
```

同时，将与你的数据库相适应的数据库驱动置于 classpath 下。更新你的应用程序下的 Activiti 类库。或者启动新版本的 Activiti，然后将它指到包含老版本的数据库。databaseSchemaUpdate 为 true 时，Activiti 会在第一次发现其类库与数据库模式不同步时自动将数据库模式更新到一个比较新的版本。

第四章、Spring 的集成

虽然没有 Spring 你也同样可以使用 Activiti，但我们提供了一些非常不错的集成特性，这一章将对其进行介绍。

4.1 ProcessEngineFactoryBean

可以作为普通的 Spring bean 对 ProcessEngine 进行配置。集成的出发点是类 `org.activiti.spring.ProcessEngineFactoryBean`。这个 bean 使用了一个流程引擎配置来创建流程引擎。这意味着[配置一节](#)介绍的方式和所有配置属性对于 Spring 也是完全一样的。

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    ...
</bean>

<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
<property name="processEngineConfiguration" ref="processEngineConfiguration"/>
</bean>
```

一定要注意 `processEngineConfiguration` bean 现在使用的是 `org.activiti.spring.SpringProcessEngineConfiguration` 类。

4.2 事务

我们将一步步地解释发布中的 Spring 示例 `SpringTransactionIntegrationTest`。其中有我们在此例子中使用的 Spring 配置文件（位于 `SpringTransactionIntegrationTest-context.xml`）。引述中包含了 `dataSource`、`transactionManager`、`processEngine` 以及 Activiti 引擎的服务。

将 `DataSource` 传递给 `SpringProcessEngineConfiguration`（使用 `dataSource` 属性）后，Activiti 内部使用 `org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy` 来封装传进来的 `DataSource`。这就确保了从该 `DataSource` 获取的 SQL 连接与 Spring 事务能够完美地结合。这意味着不再需要你在 Spring 配置文件中代理 `dataSource` 了，但同样允许向 `SpringProcessEngineConfiguration` 传递 `TransactionAwareDataSourceProxy`。该例子中不会有多余的包装发生。

在 Spring 配置文件中自己声明 `TransactionAwareDataSourceProxy` 时，务必不要将其应用给已经感知了 Spring 事务的资源。（比如，`DataSourceTransactionManager` 和 `JPATransactionManager` 使用的是非代理的 `dataSource`）。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:tx="http://www.springframework.org/schema/tx"
  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
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-2.5.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx"
```

```

http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

<beanid="dataSource"class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
<propertyname="driverClass"value="org.h2.Driver"/>
<propertyname="url"value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000"/>
<propertyname="username"value="sa"/>
<propertyname="password"value=""/>
</bean>

<beanid="transactionManager"class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
<propertyname="dataSource"ref="dataSource"/>
</bean>

<beanid="processEngineConfiguration"class="org.activiti.spring.SpringProcessEngineConfiguration">
<propertyname="dataSource"ref="dataSource"/>
<propertyname="transactionManager"ref="transactionManager"/>
<propertyname="databaseSchemaUpdate"value="true"/>
<propertyname="jobExecutorActivate"value="false"/>
</bean>

<beanid="processEngine"class="org.activiti.spring.ProcessEngineFactoryBean">
<propertyname="processEngineConfiguration"ref="processEngineConfiguration"/>
</bean>

<beanid="repositoryService"factory-bean="processEngine"factory-method="getRepositoryService"/>
<beanid="runtimeService"factory-bean="processEngine"factory-method="getRuntimeService"/>
<beanid="taskService"factory-bean="processEngine"factory-method="getTaskService"/>
<beanid="historyService"factory-bean="processEngine"factory-method="getHistoryService"/>
<beanid="managementService"factory-bean="processEngine"factory-method="getManagementService"/>

...

```

Spring 配置文件的其余部分是 beans 以及我们将在此示例中使用的配置:

```

<beans>
...
<tx:annotation-driventransaction-manager="transactionManager"/>

<beanid="userBean"class="org.activiti.spring.test.UserBean">
<propertyname="runtimeService"ref="runtimeService"/>
</bean>

<beanid="printer"class="org.activiti.spring.test.Printer"/>

</beans>

```

首先任意使用一种 Spring 创建其应用上下文的方式创建 Spring 应用上下文。此示例中可以利用类路径下的 XML 资源来配

置我们的 Spring 应用上下文：

```
ClassPathXmlApplicationContext applicationContext =
    new ClassPathXmlApplicationContext("org/activiti/examples/spring/SpringTransactionIntegrationTest-context.xml");
```

或者，如果测试的话：

```
@ContextConfiguration("classpath:org/activiti/spring/test/transaction/SpringTransactionIntegrationTest-context.xml")
```

接下来我们就可以获取 Activiti 的服务 beans，然后调用其上的方法。ProcessEngineFactoryBean 给服务添加了另外的拦截器，其在 Activiti 的服务方法上应用的是 Propagation.REQUIRED 事务语义。所以我们可以这样使用 repositoryService 来部署流程。

```
RepositoryService repositoryService = (RepositoryService) applicationContext.getBean("repositoryService");
String deploymentId = repositoryService
    .createDeployment()
    .addClasspathResource("org/activiti/spring/test/hello.bpmn20.xml")
    .deploy()
    .getId();
```

其它相关方式同样可以利用。此例中，Spring 事务是围绕 userBean.hello()方法的，且 Activiti 的服务方法的调用也会加入进该事务。

```
UserBean userBean = (UserBean) applicationContext.getBean("userBean");
userBean.hello();
```

UserBean 看上去是这样的。还记得上文在 Spring bean 的配置中我们向 userBean 注入了 repositoryService。

```
publicclass UserBean {

    /** 由Spring注入 */
    private RuntimeService runtimeService;

    @Transactional
    publicvoid hello() {
        // 这里，我们可以在领域模型内做些事务性的处理
        // 当执行到Activiti RuntimeService的startProcessInstanceByKey方法时，
        // 它会合并到这一事务里
        runtimeService.startProcessInstanceByKey("helloProcess");
    }

    publicvoid setRuntimeService(RuntimeService runtimeService) {
        this.runtimeService = runtimeService;
    }
}
```


4.3 表达式

使用 `ProcessEngineFactoryBean` 时，默认，所有 BPMN 流程中的[表达式](#)都能‘看到’所有 Spring beans。利用一个可配置的 map，可以限制暴露给表达式的 beans，甚至根本不暴露任何 beans。下面的例子公布了一个 bean（printer），可以以关键字“printer”来使用。要想不公布任何 beans，只需给 `SpringProcessEngineConfiguration` 的‘beans’属性传递一个空的列表。‘beans’属性不设置时，上下文内所有 Spring beans 都是可用的。

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
  ...
  <property name="beans">
    <map>
      <entry key="printer" value-ref="printer"/>
    </map>
  </property>
</bean>

<bean id="printer" class="org.activiti.examples.spring.Printer"/>
```

此时被公布的 beans 就可以在表达式中使用了：比如，`SpringTransactionIntegrationTest` 的 `hello.bpmn20.xml` 展示了如何使用 UEL 方法表达式来调用 Spring bean 的方法：

```
<definitions id="definitions" ...>

  <process id="helloProcess">

    <startEvent id="start"/>
    <sequenceFlow id="flow1" sourceRef="start" targetRef="print"/>

    <serviceTask id="print" activiti:expression="#{printer.printMessage()}" />
    <sequenceFlow id="flow2" sourceRef="print" targetRef="end"/>

    <endEvent id="end"/>

  </process>

</definitions>
```

其中 `Printer` 看起来如：

```
public class Printer {

  public void printMessage() {
    System.out.println("hello world");
  }
}
```

并且 Spring bean 的配置（上文也展示了）看上去如：


```

<beans...>
    ...

<bean id="printer" class="org.activiti.examples.spring.Printer"/>

</beans>

```

4.4 自动资源部署

Spring 集成针对部署资源也有其特殊性质。在流程引擎的配置中，可以指定一系列资源。在创建流程引擎时，所有那些资源会被扫描、部署。有时要进行过滤以防止重复部署。只有当资源真的有过修改时，才会向 Activiti 数据库进行新的部署。这对于很多用例，那些经常要重启 Spring 容器的情况（如，为了测试），是很有意义的。

示例

```

<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    ...
<property name="deploymentResources" value="classpath*:/org/activiti/spring/test/autodeployment/autodeploy.*.bpmn20.xml"/>
</bean>

<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
<property name="processEngineConfiguration" ref="processEngineConfiguration"/>
</bean>

```

4.5 单元测试

在集成 Spring 时，利用标准的 [Activiti 测试工具](#)，可以很容易地对业务流程进行测试。下面的例子展示了如何在基于 Spring 的单元测试内测试业务流程：

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:org/activiti/spring/test/junit4/springTypicalUsageTest-context.xml")
public class MyBusinessProcessTest {

    @Autowired
    private RuntimeService runtimeService;

    @Autowired
    private TaskService taskService;

    @Autowired
    @Rule
    public ActivitiRule activitiSpringRule;
}

```

```
@Test
@Deployment
public void simpleProcessTest() {
    runtimeService.startProcessInstanceByKey("simpleProcess");
    Task task = taskService.createTaskQuery().singleResult();
    assertEquals("My Task", task.getName());

    taskService.complete(task.getId());
    assertEquals(0, runtimeService.createProcessInstanceQuery().count());
}
}
```

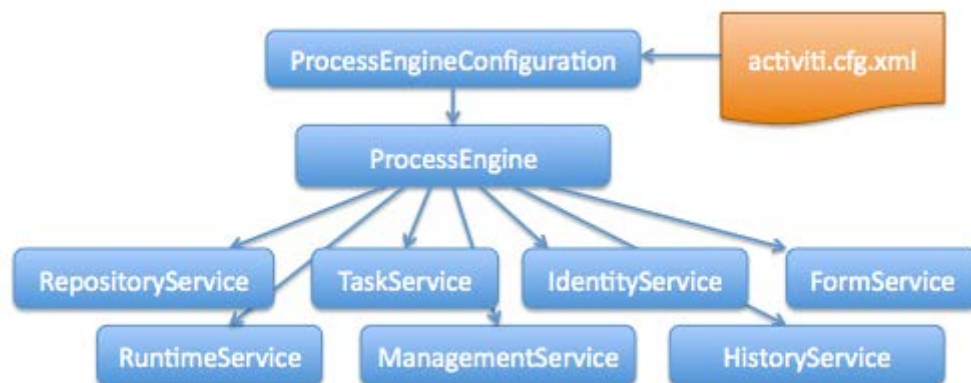
注意要让此运行，需要在 Spring 的配置中定义 *org.activiti.engine.test.ActivitiRulebean*（上文示例中，它是被自动注入的）。

```
<bean id="activitiRule" class="org.activiti.engine.test.ActivitiRule">
<property name="processEngine" ref="processEngine"/>
</bean>
```

第五章、API

5.1 引擎 API

引擎 API 是与 Activiti 交互最常用的方式。主要的出发点是 `ProcessEngine`，可以使用[配置](#)一节中介绍的几种方式对其进行创建。由 `ProcessEngine`，可以获取到含有工作流/BPM 方法的不同服务。`ProcessEngine` 以及那些服务对象都是线程安全的。所以可以为整个服务器保留一份它的引用。



```

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

RuntimeService runtimeService = processEngine.getRuntimeService();
RepositoryService repositoryService = processEngine.getRepositoryService();
TaskService taskService = processEngine.getTaskService();
ManagementService managementService = processEngine.getManagementService();
IdentityService identityService = processEngine.getIdentityService();
HistoryService historyService = processEngine.getHistoryService();
FormService formService = processEngine.getFormService();
  
```

这些服务的名称是相当一目了然的。更多关于服务和引擎 API 的详细信息，见 [javadocs](#)。

`ProcessEngines.getDefaultProcessEngine()` 会在第一次被调用时初始并构建流程引擎，接下来对该方法的调用返回的都是同一个流程引擎。利用 `ProcessEngines.init()`、`ProcessEngines.destroy()` 可以正确创建、关闭流程引擎。

`ProcessEngines` 会浏览所有 `activiti.cfg.xml` 和 `activiti-context.xml` 文件。对于那些 `activiti.cfg.xml` 文件，将以 Activiti 特有的方式来构建流程引擎：

`ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(inputStream).buildProcessEngine()`。对于那些 `activiti-context.xml` 文件，将以 Spring 的方式来构建流程引擎：首先，创建 spring 应用上下文；然后，从该上下文中获取流程引擎。

5.2 异常策略

Activiti 中基本的异常是 `org.activiti.engine.ActivitiException`，是未检查异常。此 API 随时可能会抛出此类异常，但在 [javadocs](#) 中介绍了具体方法内可能的‘预期’异常。比如，取自 `TaskService` 的一段代码：

```
/**
 *在任务成功执行时调用.
 * @param taskId待完成任务的id, 不能为空
 * @throws ActivitiException 当不存在给定id的任务时, 抛出ActivitiException异常
 */
void complete(String taskId);
```

在上面的例子中，如果不存在与传过来的 id 对应的任务（译注，task），就会抛出异常。同样，因为 javadoc 内明确规定了 taskId 不能为 null 值，所以如果传递过来的是 null 值，也会抛出 ActivitiException。

尽管我们试图避免引入庞大的异常层次，但仍然添加了以下在特定类下被抛出的异常子类：

- ActivitiWrongDbException：当 Activiti 引擎发现数据库模式的版本与引擎的版本不匹配时抛出。
- ActivitiOptimisticLockingException：当数据库内由于并发访问同一数据项导致了乐观锁时抛出。
- ActivitiClassLoadingException：当找不到需要加载的类或加载类时出现了错误时抛出（如，JavaDelegates, TaskListeners, ...）。

5.3 单元测试

业务流程是软件项目中必不可少的一部分，且应该按测试普通应用程序逻辑的方式来对其进行测试：使用单元测试。由于 Activiti 是嵌入式的 Java 引擎，所以编写针对业务流程的单元测试就像编写普通单元测试一样那么简单。

Activiti 支持 Junit 3 和 Junit 4 风格的单元测试。在 Junit 3 风格下，要继承 `org.activiti.engine.test.ActivitiTestCase`。这样就可以通过 `protected` 修饰的成员字段来访问流程引擎以及那些服务了（译注，`ActivitiTestCase` 将其作为属性做了封装）。在单元测试中的 `setup()` 方法内，流程引擎默认使用类路径下的资源文件 `activiti.cfg.xml` 进行初始化。要想指定不同的配置文件，要重写 `getConfigurationResource()` 方法。如果单元测试的配置资源文件一样，那么流程引擎将被静态地缓存于多个单元测试之间。

继承 `ActivitiTestCase` 后，就可以使用 `org.activiti.engine.test.Deployment` 来注解测试方法了。运行单元测试前，测试类所在包内的格式为 `testClassName.testMethod.bpmn20.xml` 的资源文件会被部署。测试结束时，会删除该部署，包括所有相关的流程实例、任务，等等。`Deployment` 注解也支持显式地设置资源文件的位置。更多详细信息，见 [javadocs](#)。

总之，Junit 3 风格的单元测试看起来如下：

```
publicclass MyBusinessProcessTest extends ActivitiTestCase {

    @Deployment
    publicvoid testSimpleProcess() {
        runtimeService.startProcessInstanceByKey("simpleProcess");

        Task task = taskService.createTaskQuery().singleResult();
        assertEquals("My Task", task.getName());

        taskService.complete(task.getId());
        assertEquals(0, runtimeService.createProcessInstanceQuery().count());
    }
}
```

要想利用 Junit 4 编写单元测试的风格达到相同的功能，需要使用规则 `org.activiti.engine.test.ActivitiRule`。通过此规则，就可以通过 `getters` 方法来获取流程引擎和那些服务了（译注，`ActivitiRule` 将其作为 `getter` 属性做了封装）。就像使用 `ActivitiTestCase`（见上文），包含此规则后就能使用 `org.activiti.engine.test.Deployment`（见上文介绍的其使用和配置）注解了，并且它会查找类路径下的默认配置文件。如果单元测试的配置资源文件一样，那么流程引擎将被静态地缓存于多个单元测试之间。

以下代码展示了使用 Junit 4 风格的测试以及 `ActivitiRule` 的用法的例子。

```
public class MyBusinessProcessTest {

    @Rule
    public ActivitiRule activitiRule = new ActivitiRule();

    @Test
    @Deployment
    public void ruleUsageExample() {
        RuntimeService runtimeService = activitiRule.getRuntimeService();
        runtimeService.startProcessInstanceByKey("ruleUsage");

        TaskService taskService = activitiRule.getTaskService();
        Task task = taskService.createTaskQuery().singleResult();
        assertEquals("My Task", task.getName());

        taskService.complete(task.getId());
        assertEquals(0, runtimeService.createProcessInstanceQuery().count());
    }
}
```

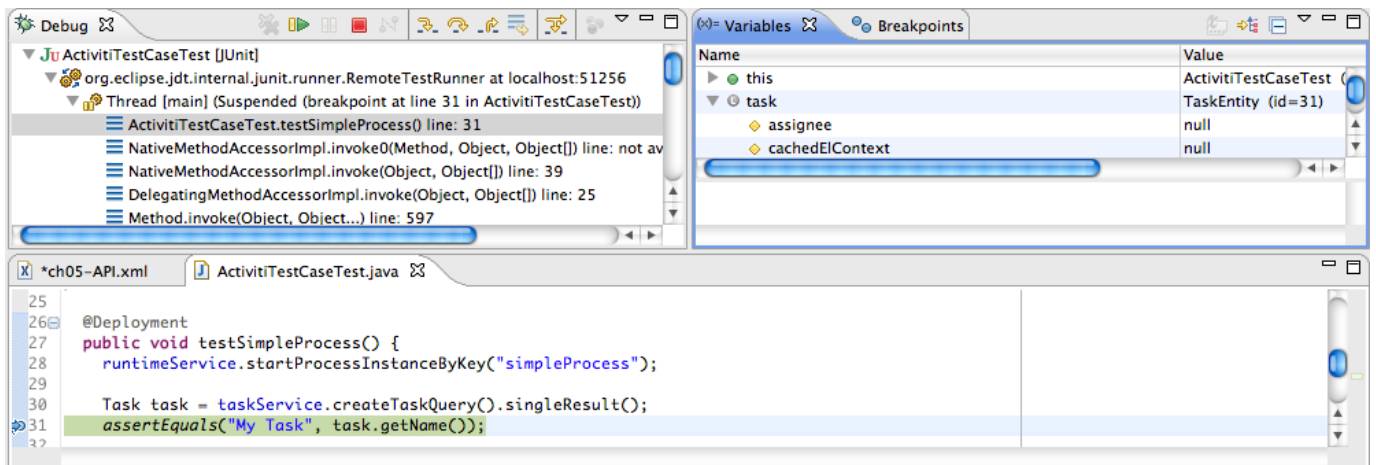
5.4 调试单元测试

在使用内存中的 H2 数据库进行单元测试时，以下说明可以在调试会话内很容易地查看到 `Activiti` 数据库中的数据。这些截图取自 Eclipse，但对于其它的 IDE 而言机制也应该是一样的。

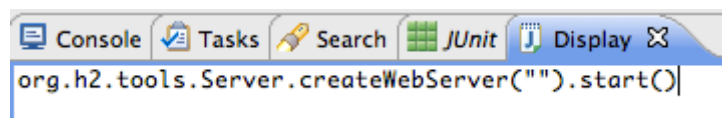
假设我们在单元测试的某处放置了断点。Eclipse 中双击代码旁的左侧边框就能完成：

```
27 public void testSimpleProcess() {
28     runtimeService.startProcessInstanceByKey("simpleProcess");
29
30     Task task = taskService.createTaskQuery().singleResult();
31     assertEquals("My Task", task.getName());
32 }
```

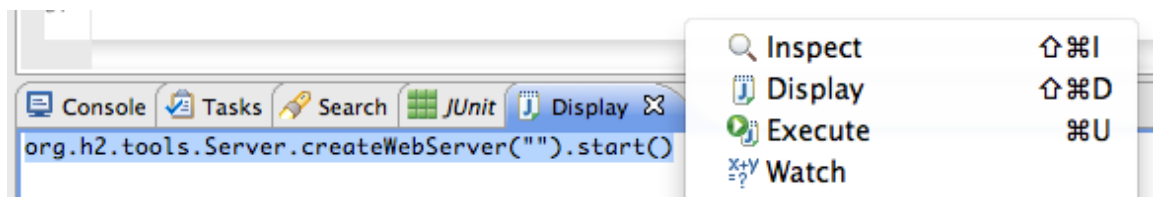
如果现在以 `debug` 模式运行单元测试（测试类中右击，选择‘Run as’，然后选择‘Junit test’），测试的执行路径断在我们的断点处，此时我们可以在右上边的面板中查看我们测试的变量。



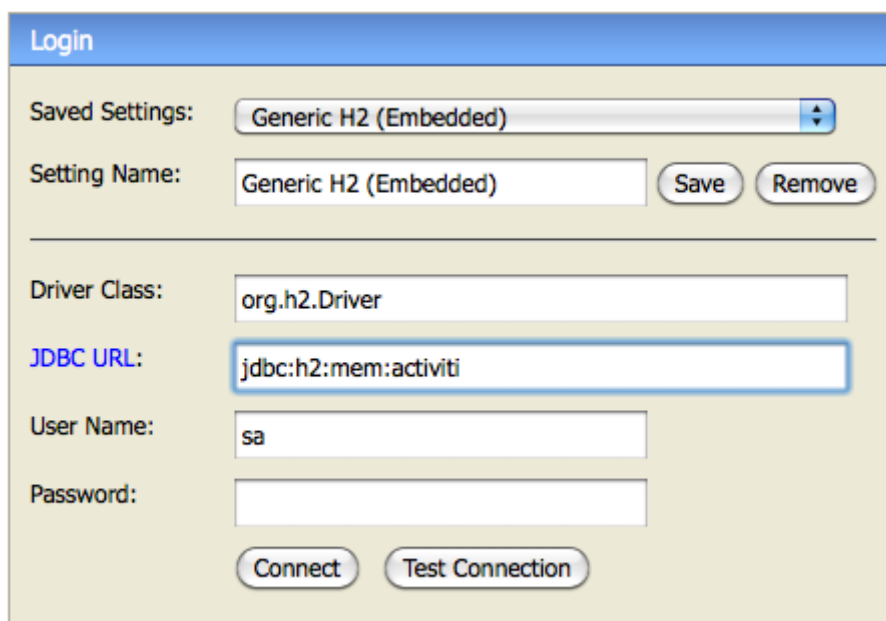
要想查看 Activiti 的数据，需要打开‘Display’窗体（如果没有这个窗体，打开 Window->Show View->Other，选择 *Display*）。然后输入（代码补全可用）`org.h2.tools.Server.createWebServer("").start()`。



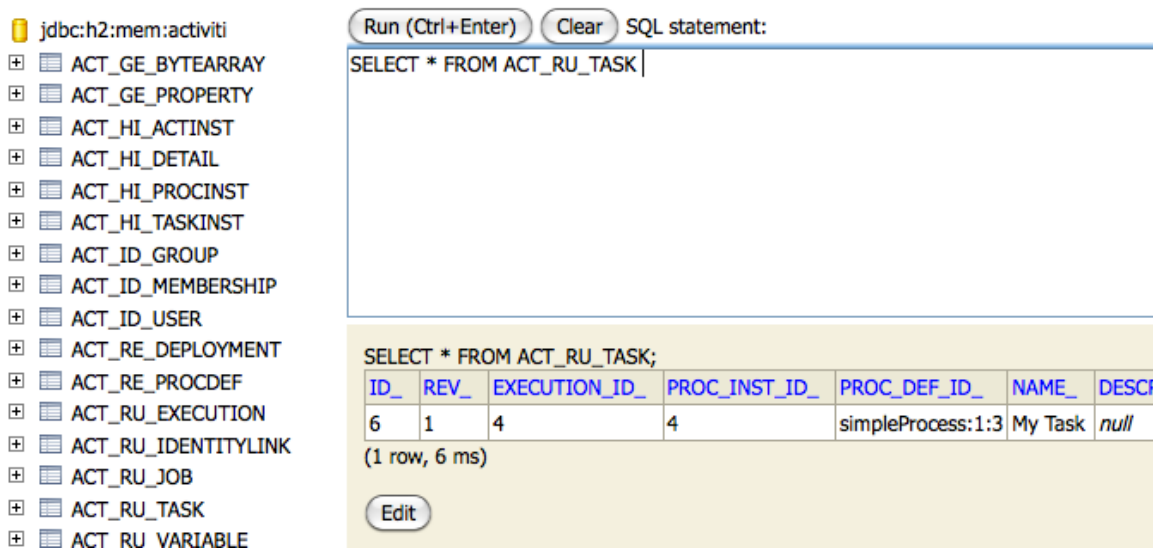
选中你刚才输入的那行代码，然后右击。此时选择‘Display’（或使用快捷键）



此时打开浏览器，转到 <http://localhost:8082>，然后填入内存数据库的 jdbc url（默认为 jdbc:h2:mem:activiti），点击连接按钮。



现在你就可以可到 Activiti 的数据了，利用它来理解单元测试是如何以及为什么以某种方式来执行流程的。



5.5 web 应用程序中的工作流引擎

ProcessEngine 是线程安全的类，可以很容易地在多个线程之间共享。在 web 应用中，这意味着可以在容器启动时创建流程引擎，在容器销毁时关闭流程引擎。

以下代码展示了在纯 Servlet 环境下如何编写一个简单的 ServletContextListener 来初始化以及销毁流程引擎：

```
publicclass ProcessEnginesServletContextListener implements ServletContextListener {

    publicvoid contextInitialized(ServletContextEvent servletContextEvent) {
        ProcessEngines.init();
    }

    publicvoid contextDestroyed(ServletContextEvent servletContextEvent) {
        ProcessEngines.destroy();
    }

}
```

方法 contextInitialized 会委托给 ProcessEngines.init()。它会查找类路径下的那些 activiti.cfg.xml 资源文件，然后根据所给的那些配置（比如，多个 jars 文件中都有那样一个名称的配置文件）创建 ProcessEngine。如果类路径下有多个那样的资源文件，要确保它们有不同的流程引擎名称（译注，配置文件流程引擎的 id）。当需要流程引擎时，可以使用：

```
ProcessEngines.getDefaultProcessEngine();
```

来获得或

```
ProcessEngines.getProcessEngine("myName");
```

当然了，可以使用创建流程引擎的任意变体，如[配置](#)一节中所描述的。

context-listener 内的 contextDestroyed 方法委托给了 ProcessEngines.destroy()。它会妥善地关闭所有初始化的流程引擎。

5.6 流程虚拟机（PVM）API

[\[试验性的\]](#)此 API 可能会在接下来的发布中做修改。

流程虚拟机的 API 公布了 Process Virtual Machine（流程虚拟机）的 POJO 核心。阅读、操作该 API 对于旨在学习为目的来理解 Activiti 内部工作机理是很有意思的。并且该 POJO API 也可以用来构建新的流程语言。

示例：

```
PvmProcessDefinition processDefinition = new ProcessDefinitionBuilder()
    .createActivity("a")
        .initial()
        .behavior(new WaitState())
        .transition("b")
    .endActivity()
    .createActivity("b")
        .behavior(new WaitState())
        .transition("c")
    .endActivity()
    .createActivity("c")
        .behavior(new WaitState())
    .endActivity()
    .buildProcessDefinition();

PvmProcessInstance processInstance = processDefinition.createProcessInstance();
processInstance.start();

PvmExecution activityInstance = processInstance.findExecution("a");
assertNotNull(activityInstance);

activityInstance.signal(null, null);

activityInstance = processInstance.findExecution("b");
assertNotNull(activityInstance);

activityInstance.signal(null, null);

activityInstance = processInstance.findExecution("c");
assertNotNull(activityInstance);
```


5.7 表达式

Activiti 使用 UEL 作为表达式的解决方案。UEL 代表 *Unified Expression Language*（统一表达式语言），它是 JEE6 规范（详细信息，见 [EE6 规范](#)）的一部分。为了支持所有环境下的最新 UEL 规范的所有特性，我们使用改进了的 JUEL。

表达式可以使用在诸如 [Java 服务任务](#)、[执行监听器](#)、[任务监听器](#) 以及 [带条件的顺序流](#)。尽管存在两种类型的表达式，值表达式和方法表达式，但是，Activiti 对此做了抽象，两者都能使用在需要表达式的地方。

- **值表达式**：对值进行解析。默认，可以使用所有的流程变量。同样，所有 Spring 的 bean（如果使用 Spring 的话）也可以使用在表达式中。示例：

```
${myVar}
${myBean.myProperty}
```

- **方法表达式**：调用有参或不带参数的方法。在调用没有参数的方法时，方法名后一定要添上空圆括号。实参可以是字符串值或自解析的表达式。示例：

```
${printer.print()}
${myBean.addNewOrder('orderName')}
${myBean.doSomething(myVar, execution)}
```

注意，表达式支持解析基本数据类型（包括，对它们进行比较）、beans、lists、数组以及 maps。

除所有流程变量以外，还有一些公布了的你可以使用在表达式内的对象：

- **execution**：持有关于进行中执行路径的额外信息的 `DelegateExecution`。
- **task**：持有关于当前任务的额外信息的 `DelegateTask`。**注意**：只在由任务监听器求值的表达式中有效。
- **authenticatedUserId**：当前被认证用户的 id。如果没有用户被认证，该变量无效。

更具体的用法和例子，参考 [Spring 内的表达式](#)、[Java 服务任务](#)、[执行监听器](#)、[任务监听器](#) 以及 [带条件的顺序流](#)。

第六章、部署

6.1 业务归档文件

要部署流程，需要将其包装在业务归档文件中。业务归档文件是向 Activiti 引擎部署的单元。业务归档文件基本上就是个压缩文件。其内可以含有 BPMN2.0 流程、任务表单、规则以及其它类型的文件。通常，业务归档文件包含一些命名资源。

业务归档文件被部署后，会扫描其内以 .bpmn20.xml 为扩展名的 BPMN 文件。每个那样的文件都将被解析，其可能会包含多个流程定义。

注意，业务归档文件中的 Java 类**不会被添加到类路径下**。必须将业务归档文件中流程定义使用的所有自定义类（比如，Java 服务任务或时间监听器的实现）添加到 Activiti 引擎的类路径下才能运行流程。

6.1.1 编程式部署

通过压缩文件来部署业务归档文件可以像这样来完成：

```
String barFileName = "path/to/process-one.bar";
ZipInputStream inputStream = new ZipInputStream(new FileInputStream(barFileName));

repositoryService.createDeployment()
    .name("process-one.bar")
    .addZipInputStream(inputStream)
    .deploy();
```

也可以通过单独的资源进行部署。详细信息见 javadocs。

6.1.2 使用 ant 部署

使用 ant 部署业务归档文件，首先，需要定义任务 deploy-bar。确保配置的 jar、Activiti 的 jar 以及所有依赖都在类路径下。

```
<taskdef name="deploy-bar" classname="org.activiti.engine.impl.ant.DeployBarTask">
  <classpath>
    <fileset dir="...">
      <include name="activiti-cfg.jar"/>
      <include name="your-db-driver.jar"/>
    </fileset>
    <fileset dir="${activiti.home}/lib">
      <include name="activiti-engine-${activiti.version}.jar"/>
      <include name="ibatis-sqlmap-*.jar"/>
    </fileset>
  </classpath>
</taskdef>

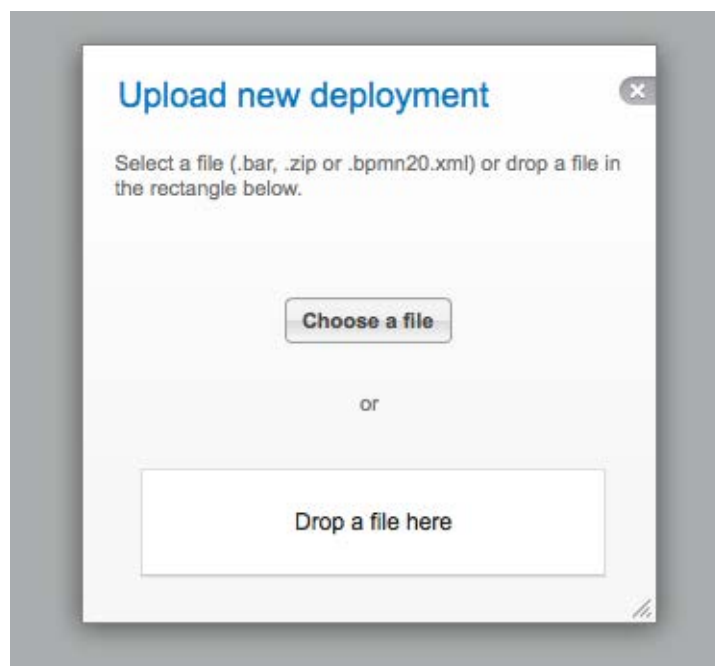
<deploy-bar file=".../yourprocess.bar"/>
```

6.1.3 使用 Activiti Explorer 部署

Activiti Explorer web 应用允许通过 web 应用的用户接口上传 bar 文件（以及单独的 bpmn20.xml 文件）。选择 *Management* 标签，点击 *Deployment*。



一个弹出窗口此时允许从你的计算机选择文件，或者你可以简单的拖拽指定的区域（如果你的浏览器支持）。



6.2 外部资源

流程定义存在于 Activiti 的数据库中。在使用服务任务、执行监听器以及在 Activiti 的配置文件内使用 Spring 的 bean 时，流程定义可以引用到这些代理类。这些类以及 Spring 的配置文件必须对所有流程引擎是可用的。

6.2.1 Java 类

启动流程实例时，流程内使用的所有自定义类（例如，服务任务、事件监听器以及任务监听器,...使用的 Java 代理）必须存在于流程引擎的类路径下。

但在部署业务归档文件的时候，是不要求这些类一定要存在于类路径下的。这意味着在使用 ant 部署一个新的业务归档文件时，不要求代理类一定得在类路径下。

当使用演示设置并想要添加自定义类时，必须将包含有自定义类的 jar 文件添加到 activiti-rest 应用的 lib 文件夹下。不要忘记也要把自定义类的依赖包含进来（如果有的话）。这也是放置 activiti-engine 的 jar 的路径。可以在分发包下 `${activiti.home}/apps/apache-tomcat-6.0.29/webapps/activiti-rest/WEB-INF/lib/` 找到此文件夹。

6.2.2 在流程中使用 Spring beans

当在表达式或脚本内使用 Spring 的 beans 时，这些 beans 对于引擎必须是可用的。如果你正在构建你自己的 web 应用，并且是按 [Spring 集成一节](#) 中描述的那样来配置你上下文中的流程引擎，这样就行了。但要记住，如果使用 Activiti rest 应用，你也要更新该上下文内的 Activiti rest 应用。可以将

`${activiti.home}/apps/apache-tomcat-6.0.29/webapps/activiti-rest/lib/activiti-cfg.jar` 中的文件 `activiti.cfg.xml` 替换成包含有你 Spring 上下文配置的 `activiti-context.xml`。

6.2.3 创建独立应用

你可以考虑把 Activiti rest 应用添加进你自己的 web 应用中，这样一来就只存在一个流程引擎了，就不用再确保是否所有的流程引擎在其类路径下都有其代理类以及是否使用了恰当的 Spring 配置了。

6.3 流程定义的版本

BPMN 没有版本的概念。这样也好，因为可执行的 BPMN 流程文件有可能会作为部署项目的一部分将被存储到 SVN 库中。流程定义的版本是在部署期间创建的。部署时，在流程定义被存储到 Activiti 数据库前，Activiti 会给流程定义分配一个版本号。

对于业务归档文件内的每一个流程定义，都会执行以下步骤来初始化属性 `key`、`version`、`name` 以及 `id`：

- XML 文件中流程定义的 `id` 属性作为流程定义的 `key` 属性。
- XML 文件中流程定义的 `name` 属性作为流程定义的 `name` 属性。如果不指定 `name` 属性，那么 `id` 属性作为 `name`。
- 带有特定 `key` 的流程第一次被部署时，被分配的版本号为 1。同一 `key` 值的流程定义的后续部署，版本号会被设置为比当前最大的部署版本号大 1 的值。`key` 属性用来区分流程定义。
- 流程定义的 `id` 属性被设置为 `{ processDefinitionKey } : { processDefinitionVersion } : { generated-id }`，其中 `generated-id` 是唯一性的数字，用来确保缓存在集群环境下流程定义 `id` 的唯一性。

举个流程的例子

```
<definitions id="myDefinitions">
  <process id="myProcess" name="My important process">
    ...
```

在部署该流程定义时，数据库中流程定义看上去如下：

表 6.1

id	key	name	version
myProcess : 1 : 676	myProcess	My important process	1

假如我们现在部署同一流程的更新版（比如，修改了一些用户任务），但流程定义的 `id` 保持不变。流程定义表现在会包括以下条目：

表 6.2

id	key	name	version
myProcess : 1 : 676	myProcess	My important process	1
myProcess : 2 : 870	myProcess	My important process	2

当调用 `runtimeService.startProcessInstanceByKey("myProcess")` 时，此时将使用版本号为 2 的流程定义，因为这是最新版本号的流程定义。

6.4 提供流程图

可以向部署中添加流程图图片。图片将被存储到 Activiti 库中，并且可以通过该 API 来访问。图片也用来在 Activiti Explorer 中显示流程。

假如类路径下有个流程，`org/activiti/expenseProcess.bpmn20.xml`，流程 key 为 'expense'。遵循使用流程图片的命名规范（按此特定顺序）：

- 如果在部署时存在名称为 BPMN 2.0 xml 文件名后跟流程 key 以及图像后缀的图片资源，那么就使用这个图片作为流程图片。在我们的例子中，为 `org/activiti/expenseProcess.expense.png` (或 .jpg/ .gif)。如果一个 BPMN 2.0 xml 文件中定义了多个图片，这种方法便很有意义了。每个流程图片的文件名中都含有流程 key。
- 如果不存在那样的图片，部署时会寻找与 BPMN 2.0 xml 文件名匹配的图片资源。在我们的例子中，为 `org/activiti/expenseProcess.png`。注意这意味着定义在同一个 BPMN 2.0 文件中的**每个流程定义**都拥有相同的流程图片。如果每个 BPMN 2.0 xml 文件中只有一个流程定义，这当然是不成问题的。

编程式部署的例子：

```
repositoryService.createDeployment()
    .name("expense-process.bar")
    .addClasspathResource("org/activiti/expenseProcess.bpmn20.xml")
    .addClasspathResource("org/activiti/expenseProcess.png")
    .deploy();
```

接下来可以通过 API 来获得此图像资源：

```
ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()
    .processDefinitionKey("expense")
    .singleResult();

String diagramResourceName = processDefinition.getDiagramResourceName();
InputStream imageStream = repositoryService.getResourceAsStream(processDefinition.getDeploymentId(),
    diagramResourceName);
```

6.5 生成流程图

如果部署时不提供图片，如[前一节](#)所描述的，当流程定义中包含有必需的‘图形交互’信息时，Activiti 引擎会生成图像。使用 [Activiti Modeler](#) 便如此（以及在不久的将来使用 Activiti Eclipse Designer 时）。

可以按部署时[提供图像](#)完全相同的方式来获取该资源。



第七章、BPMN

7.1 BPMN 是什么

见我们的 [BPMN 2.0 上的常见问题解答（FAQ）](#)。

7.2 示例

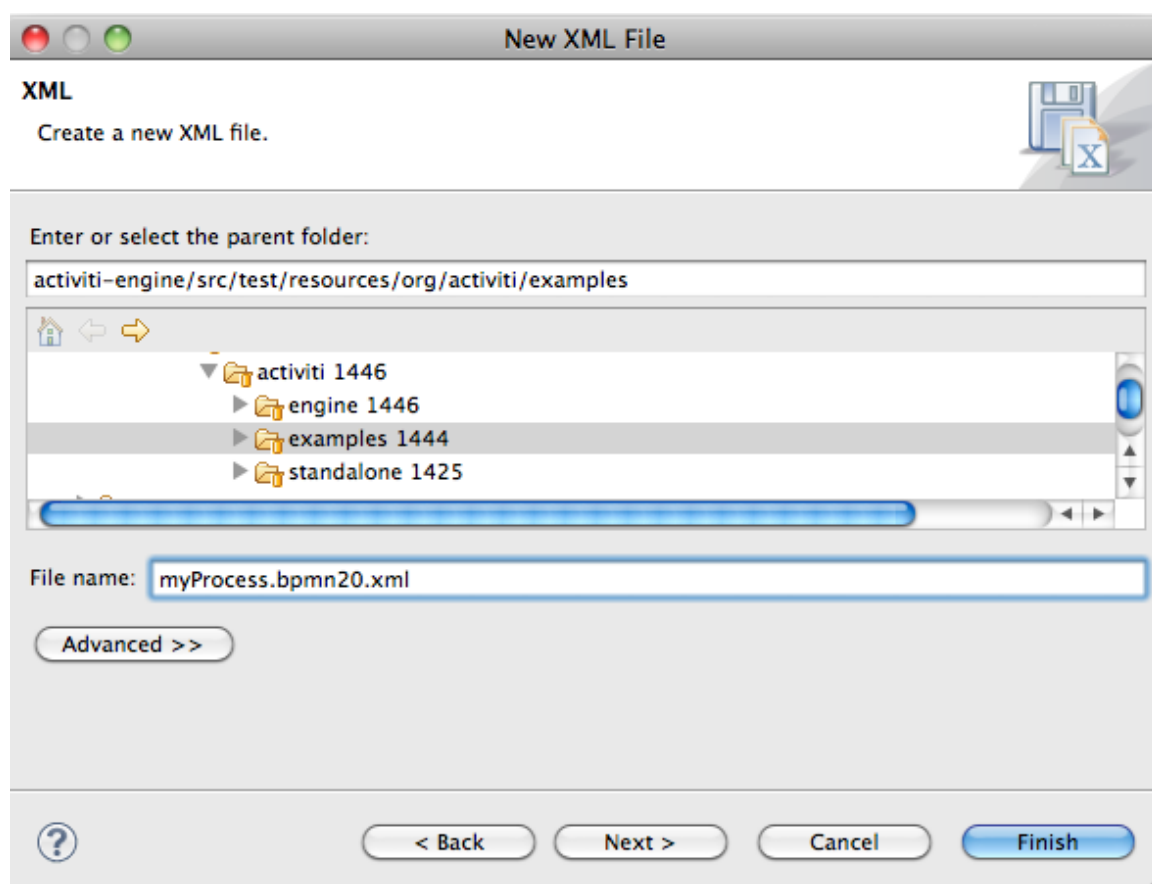
接下来章节中所描述的那些 BPMN 2.0 结构的例子可以在 Activiti 的发布包中的 *workspace/activiti-x-examples* 文件夹下找到。

更多信息参看具体关于 [示例](#) 的章节。

7.3 定义流程

要创建新的 BPMN 2.0 流程定义，最好 [正确设置](#) 好了 Eclipse。

创建一个新 XML 文件（任意项目上右击，然后选择 *New->Other->XML-XML File*），然后为其起个名。务必使该文件以 **.bpmn20.xml** 结尾，否则流程引擎不会选择该文件进行部署。



BPMN 2.0 模式的根元素是 **definition** 元素。在该元素内，可以定义多个流程定义（尽管我们建议每个文件中只包含一个流程定义，因为这会在接下来部署流程时简化维护）。一个空的流程定义看上去如下。注意，最少的定义元素只需要有 *xmlns*

和 `targetNamespace` 的声明。`targetNamespace` 可以任意指定，它对于对流程进行分类是很有用的。

```
<definitions
xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:activiti="http://activiti.org/bpmn"
targetNamespace="Examples">

<process id="myProcess" name="My First Process">
  ..
</process>

</definitions>
```

当然你也可以添加 BPMN 2.0 xsd 模式的在线 schemaLocation，作为在 [Eclipse 中配置 XML catalog](#) 的另一个选择。

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
http://www.omg.org/spec/BPMN/2.0/20100501/BPMN20.xsd"
```

`process` 元素有两个属性：

- **id**：这个属性是**必须的**，映射为 *Activiti* 中 *ProcessDefinition* 对象的 **key** 属性。这个 **id** 接下来可以通过 *RuntimeService* 的 *startProcessInstanceByKey* 方法来启动该流程定义的一个新流程实例。这个方法总是选取流程定义的**最新的部署版本**。

```
ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("myProcess");
```

值得注意的是这与调用 *startProcessInstanceById* 方法是不一样。这个方法期望的字符串 **id** 是由 *Activiti* 引擎在部署期间生成的，并且可以通过调用方法 *processDefinition.getId()* 来获取该 **id**。生成的 **id** 的格式为 '**key:version**'，长度限制在 **64 个字符**。如果得到一个 *ActivitiException* 说生成的 **id** 太长了，那么限制一下该流程中 **key** 字段的文本长度。

- **name**：该属性是**可选的**，映射为 *ProcessDefinition* 对象的 **name** 属性。流程引擎本身不使用这个属性，所以，举个例子，该属性可以用来在用户接口上展示更为友好的名称。

7.4 入门：10 分钟指南

本章我们会包含一个（非常简单的）业务流程用来介绍一些基础的 *Activiti* 概念以及 *Activiti* API。

7.4.1 先决条件

这个指南假设你在运行着 [Activiti 的演示设置](#)。当然了，你也该安装 *Eclipse*，并导入 [Activiti 的示例](#)。

7.4.2 目标

这个指南的目标是学习 *Activiti* 和一些 BPMN2.0 的基本概念。最终的结果将是一个简单的 *Java SE* 程序，该程序部署了流程定义，并且使用了 *Activiti* 引擎 API 与此流程进行交互。我们也会接触有关 *Activiti* 的一些工具。当然了，你在该指南中学到的东西也可以用于构建你自己的涉及业务流程的 **web** 应用。

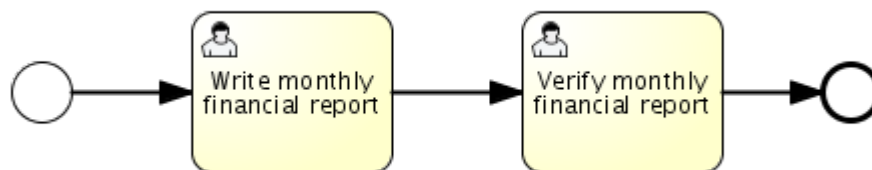
7.4.3 用例

该用例很简单：有个公司，让我们称之为 BPMCorp。BPMCorp 内，每个月都要为公司股东编写财务报表。这是会计部门的职责。报表编制完成后，在将其发送给股东前，上级管理层的一个成员要审核该文件。

下面章节使用的所有的文件以及代码片段都可以在随 Activiti 发布包一块分发的[示例](#)中找到。查找包 `org.activiti.examples.bpmn.usertask`。

7.4.4 流程图

上面描述的业务流程可以使用 [Activiti Modeler](#) 或 [Activiti Designer](#) 来图形式的可视化。然而，对于这个指南，我们会自己键入 XML，因为这让我们能学到更多东西。我们的流程的图形化的 BPMN 2.0 符号看起来像：



我们看到的是一个 [none start 事件](#)（左边的圆圈），跟着两个[用户任务](#)：“编写月度财务报表”和“审核月度财务报表”，结尾是一个 [none end 事件](#)（右边粗边框的圆圈）。

7.4.5 XML 的描述

这个业务流程（`FinancialReportProcess.bpmn20.xml`）的 XML 版本看上去如下所示。很容易地辨认出我们流程的主要的元素（点击链接进入到详细的 [BPMN 2.0](#) 章节）：

- [none start 事件](#)让我们知道了该流程的入口点是什么。
- [用户任务](#)声明是对流程中人为任务的描述。注意，第一个任务被分配给了 `accountancy` 组，而第二个任务被分配给了 `management` 组。更多关于将用户和组如何分配到用户任务的信息，参看关于[用户任务分配](#)的章节。
- 当到达 [none end 事件](#)时，流程结束。
- 元素之间是通过顺序流（[sequence flows](#)）连接的。这些顺序流有 `source` 和 `target`，它们定义了顺序流的方向。

```

<definitionsid="definitions"
targetNamespace="http://activiti.org/bpmn20"
xmlns:activiti="http://activiti.org/bpmn"
xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL">

<processid="financialReport"name="Monthly financial report reminder process">

<startEventid="theStart"/>

<sequenceFlowid="flow1"sourceRef="theStart"targetRef="writeReportTask"/>

<userTaskid="writeReportTask"name="Write monthly financial report">
</userTask>

</process>
</definitions>

```

```

        Write monthly financial report for publication to shareholders.
    </documentation>
    <potentialOwner>
    <resourceAssignmentExpression>
    <formalExpression>accountancy</formalExpression>
    </resourceAssignmentExpression>
    </potentialOwner>
    </userTask>

    <sequenceFlow id='flow2' sourceRef='writeReportTask' targetRef='verifyReportTask' />

    <userTask id="verifyReportTask" name="Verify monthly financial report">
    <documentation>
        Verify monthly financial report composed by the accountancy department.
        This financial report is going to be sent to all the company shareholders.
    </documentation>
    <potentialOwner>
    <resourceAssignmentExpression>
    <formalExpression>management</formalExpression>
    </resourceAssignmentExpression>
    </potentialOwner>
    </userTask>

    <sequenceFlow id='flow3' sourceRef='verifyReportTask' targetRef='theEnd' />

    <endEvent id="theEnd" />

    </process>

</definitions>

```

7.4.6 启动流程实例

我们现在已经创建了我们业务流程的**流程定义**。由那样一个流程定义，我们可以创建**流程实例**。这个例子中，流程实例配有每月财务报表的创建和审核。所有的流程实例共享同一个流程定义。

要能由一个给定流程定义创建流程实例，首先我们必须**部署**这一流程定义。部署流程意味着两件事：

- 流程定义将被存储到为 Activiti 引擎配置好了的持久性的数据库中。因此，通过部署我们的业务流程，就确保了在重启引擎后引擎也能获得该流程定义。
- BPMN 2.0 流程定义文件将被解析成一个内存对象模型，可以通过 Activiti API 对它进行操作。

关于部署的更多信息可以在[关于部署的专门的章节](#)中找到。

正如那一章所描述的，有几种方式进行部署。一种方式是通过 API，如下所示。注意与 Activiti 引擎的所有交互都是通过它的**服务**来实现的。

```
Deployment deployment = repositoryService.createDeployment()
    .addClasspathResource("FinancialReportProcess.bpmn20.xml")
    .deploy();
```

现在我们可以使用在流程定义中定义的 `id` 来启动一个新的流程实例（参看 XML 文件中的流程元素）。注意在 Activiti 的术语中此 `id` 称为 **key**。

```
ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("financialReport");
```

这将创建一个流程实例，并首先通过 `start` 事件。通过 `start` 事件后，流程会沿着 `start` 事件的所有输出流执行（该例子中只有一个流），执行到第一个任务（“编制月度财务报表”）。此时 Activiti 引擎会向持久化数据库中存储一个任务。此时，关联在该任务上的用户或组的分配得以解析，并且也被存储到数据库中。值得注意的是 Activiti 引擎会继续流程的执行步骤直到流程执行进入一种等待状态，比如用户任务。在这样一种等待状态，流程实例的当前状态被存储到数据库中。流程会保持该状态直到有用户决定完成其任务。那时，流程引擎会继续执行流程直到流程进入一个新的等待状态或流程终点。其间，如果遇到流程引擎重启或崩溃的情况，流程状态也是安全的保存在数据库中。

任务创建后，`startProcessInstanceByKey` 方法就会返回，因为用户任务的活动处于等待状态。该例子中，任务分配给了一个组，这意味着该组中的每个成员都是任务执行的**候选者**。

现在，我们可以匆匆将这些凑在一起并创建一个简单的 Java 程序。创建一个新的 Eclipse 项目，然后将 Activiti jars 和依赖添加到类路径下（可以在 `setup/files/dependencies/libs` 下找到）。在能够调用 Activiti 服务前，我们首先要构造一个 `ProcessEngine` 供我们访问服务。这里我们使用 `‘standalone’` 的配置，它构造了一个 `ProcessEngine`，其使用的数据库也是 demo setup 使用的数据库。

你可以在[此](#)下载该流程定义的 XML。该文件含有上面展示的 XML，也包含了 Activiti 工具中可视化流程的一些必要的 BPMN 图形交互信息。

```
public static void main(String[] args) {

    // 创建Activiti流程引擎
    ProcessEngine processEngine = ProcessEngineConfiguration
        .createStandaloneProcessEngineConfiguration()
        .buildProcessEngine();

    // 获得Activiti的服务
    RepositoryService repositoryService = processEngine.getRepositoryService();
    RuntimeService runtimeService = processEngine.getRuntimeService();

    // 部署流程定义
    repositoryService.createDeployment()
        .addClasspathResource("FinancialReportProcess.bpmn20.xml")
        .deploy();

    // 启动流程实例
    runtimeService.startProcessInstanceByKey("financialReport");
}
```

7.4.7 任务列表

现在我们可以使用 `taskService` 通过添加如下的逻辑就能获得该任务：

```
List<Task> tasks = taskService.createTaskQuery().taskCandidateUser("kermit").list();
```

注意，我们传递给该方法的用户必须是 `accountancy` 组的成员，因为这在流程定义中进行了声明。

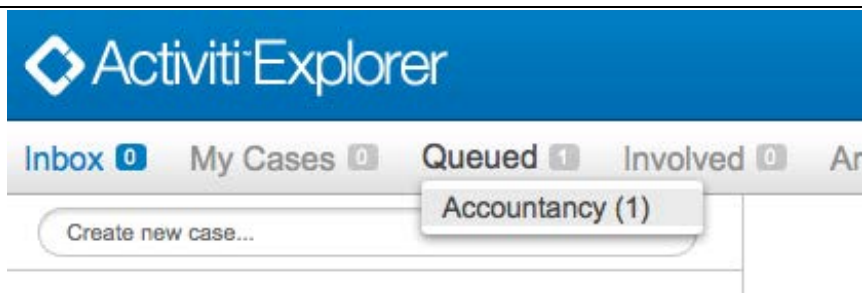
```
<potentialOwner>
<resourceAssignmentExpression>
<formalExpression>accountancy</formalExpression>
</resourceAssignmentExpression>
</potentialOwner>
```

我们也可以利用这个组名使用任务查询的 API 得到同样的结果。此时我们可以向代码中添加如下逻辑：

```
TaskService taskService = processEngine.getTaskService();
List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup("accountancy").list();
```

因为我们配置 `ProcessEngine` 使用的是与 `demo setup` 所使用的相同的数据库（如果你还没有执行过，请运行[演示设置](#)），此时我们就可以登录到 [Activiti Explorer](#)（以 `fizzie/fizzie` 登录），我们可以发现，选择 `Processes` 页面后，接着点击 ‘`Actions`’ 列内相应的 ‘`月度财务报表`’ 流程的 ‘`启动流程`’ 链接就能启动我们的业务流程。

如上所述，流程将执行到第一个用于任务。由于我们是以 `kermit` 登录的，所有在启动了一个流程实例后，我们可以看到他的一个新的候选任务。选择 `任务` 页面来查看这个新任务。注意，即使该流程是由另外某个人启动的，该任务对 `accountancy` 组内的每个成员也是作为候选任务可见的。



7.4.8 认领任务

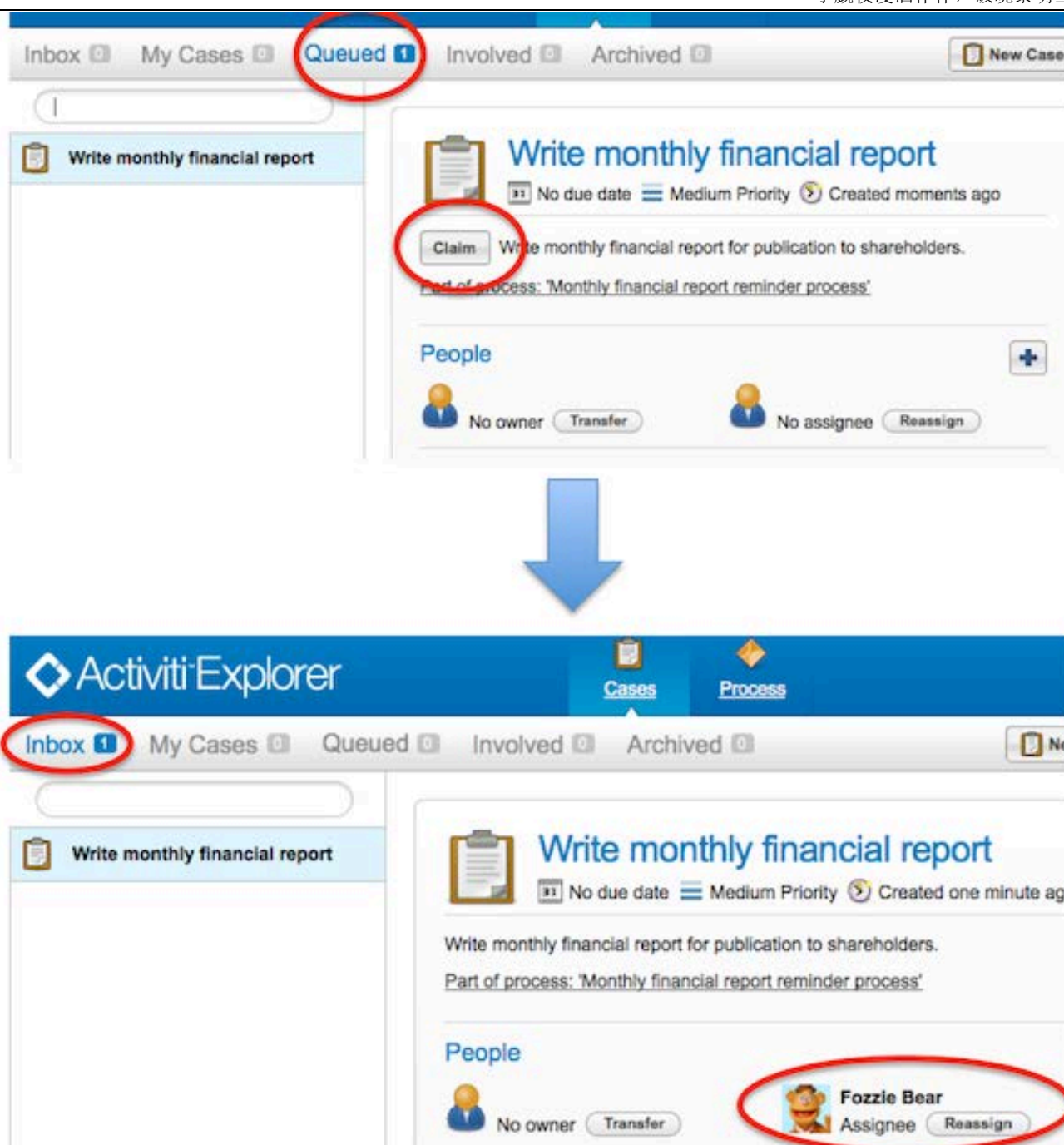
会计现在需要**认领此任务**。通过认领任务，将有专人作为该任务的**代理人**（译注，代理人即为分配到任务的人或责任人），同时该任务会从会计组其他成员的任务列表中消失。程序上完成认领任务如下：

```
taskService.claim(task.getId(), "fizzie");
```

该任务现在存在于**认领该任务的候选者的个人任务列表**中。

```
List<Task> tasks = taskService.createTaskQuery().taskAssignee("fizzie").list();
```

在 Activiti Explorer UI 中，点击 *claim* 按钮会调用同样操作。此时该任务将移到登录用户的个人任务列表内。同时可以看到该任务的责任人改为当前登录用户。



7.4.9 完成任务

会计现在可以开始编制财务报表了。一旦编制完该报表，那么他就可以**完成该项任务**了，这意味着完成了该项任务所要的所有工作。

```
taskService.complete(task.getId());
```

对 Activiti 引擎而言，这是让流程实例继续执行的一个外部信号。任务本身会从运行时的数据中被删除。流程会沿着该任务唯一的输出流迁移，将该执行路径带入第二个任务（‘财务报表审核’）。会经历如第一个任务所描述的同样的过程，不同的是该任务将被分配到 *management* 组。

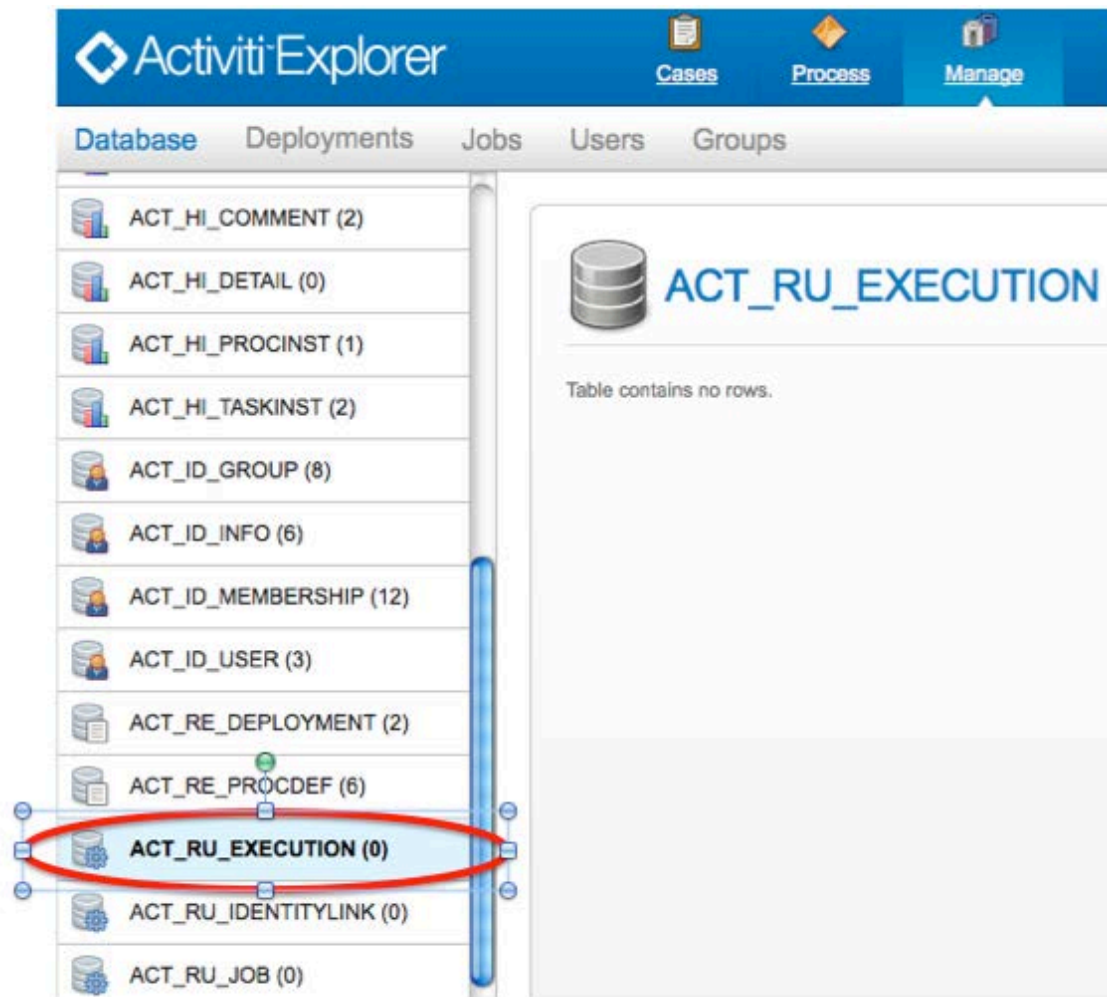
在 demo setup 内，是通过点击任务列表中的 *complete* 按钮来完成任务的。因为 Fozzie 不是会计，我们需要登出 Activiti

Explorer，然后以 *kermi*t（是个管理者）登录。在未分配任务列表中可以看到第二个任务。

7.4.10 结束流程

可以按照之前描述的方式来获取、认领审核任务。完成第二个任务会将流程带到结束该流程实例的结束事件。该流程实例以及所有相关的运行时执行数据都会从数据库中删除。

在登录 Activiti Explorer 后，你可以证实这一点，因为存储流程执行数据的表中找不到任何记录。



程序上，你也可以使用 `historyService` 来检查该流程是否结束了。

```
HistoryService historyService = processEngine.getHistoryService();
HistoricProcessInstance historicProcessInstance =
historyService.createHistoricProcessInstanceQuery().processInstanceId(procId).singleResult();
System.out.println("Process instance end time: " + historicProcessInstance.getEndTime());
```

7.4.11 代码综述

选取上面章节中所有的代码片段，之后你会得到如此内容（该代码考虑到你可能会通过 Activiti Explorer UI 启动几个流程实例。这样的话，它会获取到一系列的任务而不只一个任务，所以此代码总能运行）：

```
publicclass TenMinuteTutorial {

    publicstaticvoid main(String[] args) {

        // 创建 Activiti流程引擎
        ProcessEngine processEngine = ProcessEngineConfiguration
            .createStandaloneProcessEngineConfiguration()
            .buildProcessEngine();

        // 取得 Activiti 服务
        RepositoryService repositoryService = processEngine.getRepositoryService();
        RuntimeService runtimeService = processEngine.getRuntimeService();

        // 部署流程定义
        repositoryService.createDeployment()
            .addClasspathResource("FinancialReportProcess.bpmn20.xml")
            .deploy();

        // 启动流程实例
        String procId = runtimeService.startProcessInstanceByKey("financialReport").getId();

        // 获得第一个任务
        TaskService taskService = processEngine.getTaskService();
        List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup("accountancy").list();
        for (Task task : tasks) {
            System.out.println("Following task is available for accountancy group: " + task.getName());

            // 认领任务
            taskService.claim(task.getId(), "fizzie");
        }

        // 查看Fizzie 现在是否能够获取到该任务
        tasks = taskService.createTaskQuery().taskAssignee("fizzie").list();
        for (Task task : tasks) {
            System.out.println("Task for fizzie: " + task.getName());

            // 完成任务
            taskService.complete(task.getId());
        }

        System.out.println("Number of tasks for fizzie: "
            + taskService.createTaskQuery().taskAssignee("fizzie").count());

        // 获取并认领第二个任务
        tasks = taskService.createTaskQuery().taskCandidateGroup("management").list();
```



```

    for (Task task : tasks) {
        System.out.println("Following task is available for accountancy group: " + task.getName());
        taskService.claim(task.getId(), "kermit");
    }

    //完成第二个任务结束结束流程
    for (Task task : tasks) {
        taskService.complete(task.getId());
    }

    // 核实流程是否结束
    HistoryService historyService = processEngine.getHistoryService();
    HistoricProcessInstance historicProcessInstance =
        historyService.createHistoricProcessInstanceQuery().processInstanceId(procId).singleResult();
    System.out.println("Process instance end time: " + historicProcessInstance.getEndTime());
}
}

```

本代码也可以作为随示例分发的单元测试（是的，你应该对你的流程进行单元测试！在[单元测试](#)一节中可以有关它的内容）。

7.4.12 未来改进

很容易看出，该业务流程过于简单了以至于不能用于现实生活中。然而，因为你马上就要深入研究 Activiti 中的 BPMN 2.0 结构，所以你可以利用以下内容增强该业务流程：

- 定义做为决定的**分支**。这样，管理者可以拒绝财务报表，将任务打回给会计。
- 声明、使用**变量**，这样我们可以对报表进行存储、引用了，以便将其可视化在表单中。
- 在流程结束时定义**服务任务**，将财物报表发送给每个股东。
- 等等

7.5 BPMN 2.0 结构

7.5.1 自定义扩展

BPMN 2.0 标准对于有关各方都是个好东西。终端用户（end-users）不会被锁定于一个提供商所特有的解决方案。框架，特别是一些开源的框架，如 Activiti，可以实现（并且往往实现地会更好）具有与那些大的提供商所实现同样特性的解决方案。有了 BPMN 2.0 标准，可以很容易、很顺利地从一个提供商向 Activiti 迁移。

然而，标准往往是不同公司（以及往往是愿景）之间多次讨论与妥协的结果的这个事实，的确成了它的一个缺陷。当开发人员阅读流程定义 BPMN 2.0 XML 时，有时会感觉到某些结构或处理的方式太过于繁琐。因为 Activiti 把易开发性放到第一位，所以我们引入了一些称为“**Activiti BPMN extensions**”的东西。这些‘扩展’以一种新结构或方式简化了某些结构，但它们不属于 BPMN 2.0 规范。

尽管 BPMN 2.0 规范明确声明它也支持自定义扩展，但是我们保证：

- 这种自定义扩展（译注，指 Activiti 扩展）的先决条件是必须要有向**标准处理方式**的简单转换（译注，即能用自定义

扩展完成的功能，也必须能以标准的方式完成，自定义扩展更多考虑的应该是简化这一话题）。所以当你决定要使用自定义扩展时，不用担心没有后路的问题。

- 使用自定义扩展时，要在新的 XML 元素、属性等内明确地指明 **activiti:**命名前缀。
- 这些扩展的目标是最终将它们推入到 BPMN 规范的下一个版本，或者至少要引起关于修正 BPMN 规范结构的讨论。

因此，不管你想不想使用自定义扩展，这完全由你自己决定。多个因素会影响你的决定（图形化编辑器的使用、公司政策，等等）。我们对此提供是因为我们相信标准中的有些地方是可以做得更简单或更有效率。关于扩展，随时可以给我们反馈（正面的/负面的），或者把你的有关自定义扩展的新思路告诉我们。谁知道，或许有一天你的想法会出现在规范中！

7.5.2 事件

事件用于对发生在流程生命周期的事情进行建模。事件总是被形象成一个圆圈。在 BPMN 2.0 中，存在两种主要的事件类型：*捕获事件*和*抛出事件*。

- **捕获：**流程执行到该事件时，会等待事件触发。事件触发类型由内部图标或 XML 中的类型声明来定义。捕获事件视觉上可以通过里面没有填充的内部图标与抛出事件进行区分（也就是说，图标是白色的）。
- **抛出：**流程执行到该事件时，事件就会被触发。该事件触发的类型由内部图标或 XML 中的类型声明来定义。抛出事件视觉上可以通过内部图标与抛出的事件进行区分，抛出事件的图标使用黑色填充。

定时器事件的定义

定时器事件是被定义的定时器触发的事件。可以作为[启动事件](#)、[中间事件](#)或[边界事件](#)来使用。

定时器的定义只能有以下的一个元素：

- **timeData：**该格式以 [ISO 8601](#) 格式指定了触发事件的确定时间（译注，即，在确定时刻触发定时器事件）。示例：

```
<timerEventDefinition>
<timeDate>2011-03-11T12:13:14</timeDate>
</timerEventDefinition>
```

- **timeDuration：**指定定时器事件在触发前运行多长时间，*timeDuration* 可以作为 *timerEventDuration* 的子元素来指定。使用的格式是 [ISO 8601](#) 格式（这是 BPMN 2.0 规范所要求的）。示例（间隔 10 天）：

```
<timerEventDefinition>
<timeDuration>P10D</timeDuration>
</timerEventDefinition>
```

- **timeCycle：**指定循环的时间间隔（译注，即，每隔多长时间执行一次循环），这对于周期性的启动流程或者给过期的用户任务发送提示是很有帮助的。时间循环元素可以使用两种格式来指定。首先是循环次数的持续的格式，这是 [ISO 8601](#) 所规定的。示例（循环 3 次，每次循环持续 10 小时）：

```
<timerEventDefinition>
<timeCycle>R3/PT10H</timeCycle>
</timerEventDefinition>
```

此外，你也可以使用 cron 表达式来指定循环次数，下面的示例展示了每 5 分钟触发一次，starting at full hour：

```
00/5 * * * ?
```

请查看使用 cron 表达式的[指南](#)。

注意：第一个符号是以秒表示的，不像标准的 Unix cron 是以分钟表示的。

循环持续的次数更适合处理那些在时间上相对于某个特定的时间点（如，用户任务开始时）来计算的相对定时器，然而 cron 表达式可以处理绝对定时器，这对于 [timer start events](#) 是特别有用的。

你可以在定义定时器事件时使用表达式，这样你就可以基于流程变量来影响定时器的定义。流程变量必须包含恰当定时器类型的 ISO 8601 字符串（或循环类型 cron）。

```
<boundaryEventid="escalationTimer"cancelActivity="true"attachedToRef="firstLineSupport">
  <timerEventDefinition>
    <timeDuration>${duration}</timeDuration>
  </timerEventDefinition>
</boundaryEvent>
```

注意：只有在开启作业执行器时，定时器事件才能被触发（即，需要在 `activiti.cfg.xml` 中将 `jobExecutorActivat` 设置为 `true`），因为默认 job executor 是被禁用的。

7.5.3 启动事件

启动事件表示流程的开始。定义了流程如何被启动的启动事件类型（当收到消息、特定的时间间隔、等等，启动流程）是以一个小图标来形象表示事件的。在 XML 表示中，类型是由子元素的声明给出的。

启动事件**总是捕获型的**：从概念上讲，该事件（任何时候）会一直等待直到触发发生。

启动事件中，可以指定以下 Activiti 所特有的属性：

- **formKey:** 指向一个用户必须在启动新流程实例时填写的表单模板。更多信息见[表单一节](#)。示例：

```
<startEventid="request"activiti:formKey="org/activiti/examples/taskforms/request.form"/>
```

- **initiator:** 指明在流程启动时存储被认证的用户 id 的变量名。示例：

```
<startEventid="request"activiti:initiator="initiator"/>
```

被认证的用户必须必须在 `try-finally` 块中使用方法 `IdentityService.setAuthenticatedUserId(String)` 来设置，如下：

```
try {
    identityService.setAuthenticatedUserId("bono");
    runtimeService.startProcessInstanceByKey("someProcessKey");
} finally {
    identityService.setAuthenticatedUserId(null);
}
```

这段代码出自 Activiti Explorer 应用。因此要结合[第八章表单](#)才能运行。

7.5.4 空启动事件

描述

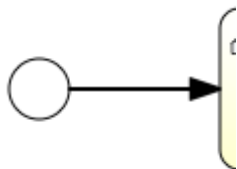
从技术上讲，‘空’启动事件意味着不给启动流程实例指定触发器（译注，流程中事件的发生是需要某种行为来触发的，这种行为我们称之为触发行为或触发器，而空启动事件不需要通过这样的触发行为就能发生）。这意味着流程引擎不能预期什么时候流程实例要被启动。空启动事件使用在通过调用 `startProcessInstanceByXXX` 方法启动流程实例的时候。

```
ProcessInstance processInstance = runtimeService.startProcessInstanceByXXX();
```

注意：子流程总是有一个空启动事件。

图形化符号

空启动事件被形象化成不带内图标 的圆（即，没有触发器类型）。



XML 表示

空启动事件的 XML 表示是不带子元素的普通启动事件声明（其它启动事件类型都有声明类型的子元素）。

```
<startEvent id="start" name="my start event"/>
```

7.5.5 定时器启动事件

描述

定时器启动事件用于在给定的时间点创建流程实例。它可以用在只启动一次的流程中，也可以用在特定时间间隔下启动的流程。

注意：子流程中不能使用定时器启动事件。

图形化符号

定时器启动事件被形象化成带有时钟内图标 的圆。



XML 表示

定时器启动事件的 XML 表示是带有定时器定义子元素的普通启动事件声明。详细配置请参考[定时器定义](#)。

示例：从 2011 年 3 月 11 日 12:13 开始，流程将启动 4 次，每次间隔 5 分钟。

```
<startEventId="theStart">
<timerEventDefinition>
<timeCycle>R4/2011-03-11T12:13/PT5M</timeCycle>
</timerEventDefinition>
</startEvent>
```

示例：流程将在选定的时间上启动一次

```
<startEventId="theStart">
<timerEventDefinition>
<timeDate>2011-03-11T12:13:14</timeDate>
</timerEventDefinition>
</startEvent>
```

7.5.6 终止事件

终止事件表明流程或子流程（的执行路径）的结束。终止事件**总是抛出型的**。这意味着当流程执行到终止事件时，有一个**结果**会被抛出。结果的类型是以事件的内部黑色图标来表示的。XML 表示中，类型是由子元素的声明给出的。

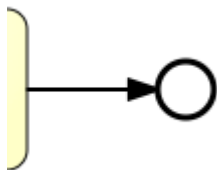
7.5.7 空终止事件

描述

‘空’终止事件意味着不指定当达到该事件时抛出的**结果**。这样，流程引擎除了结束当前的执行路径不会再执行任何其它操作。

图形化符号

空终止事件被形象化成不带内部图标（无结果类型）的粗边框圆。



XML 表示

空终止事件的 XML 表示为没有子元素的普通的终止事件的声明（其它终止事件类型都有声明类型的子元素）。

```
<endEvent id="end" name="my end event"/>
```

7.5.8 异常终止事件

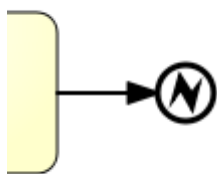
描述

当流程执行到**异常终止事件**时，会结束当前的执行路径，并抛出一个异常。异常可以被与之匹配的[中间边界异常事件](#)捕获。如果没有匹配的边界异常事件，默认会采用[空终止事件](#)对待。

要点：BPMN 异常与 Java 异常是不一样的。事实上，两者没有任何共同点。BPMN 异常事件是对**业务异常**建模的一种方式。Java 异常则是以[它所特有的方式](#)来进行处理。

图形化符号

异常终止事件被形象化成其内有异常图标的特殊终止事件（带粗边框的圆）。异常图标是全黑的，来表示是抛出的语义。



XML 表示

异常终止事件表示为带有 `errorEventDefinition` 子元素的终止事件。

```
<endEvent id="myErrorEndEvent">
  <errorEventDefinition errorRef="myError"/>
</endEvent>
```

`errorRef` 属性可以引用流程之外定义的 `error` 元素。

```
<error id="myError" errorCode="123"/>
...
<process id="myProcess">
...
```

`error` 元素的 `errorCode` 属性将用来查找与之匹配的捕获边界异常事件。如果 `errorRef` 与任何定义的异常都不匹配，那么 `errorRef` 会被当作 `errorCode` 的简写来使用。这是 Activiti 特有的简写。更具体地，在功能上下面的片段

```
<error id="myError" errorCode="error123" />
```

```
...
<processid="myProcess">
...
<endEventid="myErrorEndEvent">
<errorEventDefinitionerrorRef="myError"/>
</endEvent>
...
```

等价于

```
<endEventid="myErrorEndEvent">
  <errorEventDefinitionerrorRef="error123"/>
</endEvent>
```

注意，errorRef 必须符合 BPMN 2.0 模式，并且必须是有效的 QName。

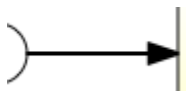
7.5.9 顺序流

描述

顺序流是两个流程元素的连接者。一个元素在流程执行期间被访问后，流程会沿着该元素所有输出顺序流继续执行。这意味着 BPMN 2.0 默认行为是并行的：两个输出顺序流会创建两条独立、并行的执行路径。

图形化符号

顺序流被形象化成由起始元素指向目标元素的箭头。箭头总是指向目标元素。



XML 表示

顺序流要有流程唯一的 id，以及指向现有**起始**元素和**目标**元素的引用。

```
<sequenceFlowid="flow1"sourceRef="theStart"targetRef="theTask"/>
```

7.5.10 条件顺序流

描述

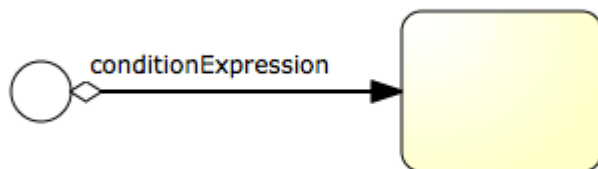
可以在顺序流上定义条件。当顺序流程左侧是 BPMN 2.0 的活动时，默认会计算其输出顺序流上的条件。选取条件成立的输出顺序流来执行。如果选取了多个顺序流，就会创建多个**执行路径**，并且流程会以并行的方式来执行。

注意：以上适用于 BPMN 2.0 的活动（以及事件），但是不适用于分支。根据分支的类型，其会以其特有的方式来处理带

有条件的顺序流。

图形化符号

条件顺序流被形象化成始点为一个小菱形的普通顺序流。条件表达式紧挨着顺序流。



XML 表示

条件顺序流是以 XML 中含有 **conditionExpression** 子元素的普通顺序流来表示的。注意目前仅支持 *tFormatExpressions*，省略 *xsi:type=""* 定义默认也是只支持此类型的表达式。

```
<sequenceFlow id="flow" sourceRef="theStart" targetRef="theTask">
  <conditionExpression xsi:type="tFormatExpression">
    <![CDATA[${order.price > 100 && order.price < 250}]]>
  </conditionExpression>
</sequenceFlow>
```

目前，conditionExpression 只能使用 UEL，关于此的详细信息见[表达式](#)一节。使用的表达式必须解析为布尔类型的值，否则在计算条件时会抛出异常。

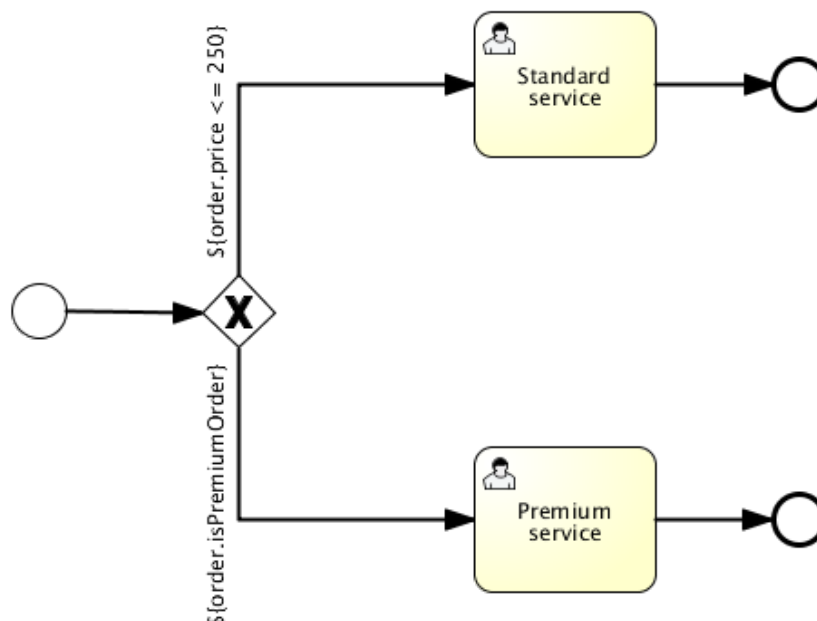
- 下面的示例以典型的 JavaBean 风格通过 getters 来引用流程变量的数据。

```
<conditionExpression xsi:type="tFormatExpression">
  <![CDATA[${order.price > 100 && order.price < 250}]]>
</conditionExpression>
```

- 该示例调用了一个返回布尔类型值的方法。

```
<conditionExpression xsi:type="tFormatExpression">
  <![CDATA[${order.isStandardOrder()}]]>
</conditionExpression>
```

Activiti 的发布包中有下面使用了值表达式和方法表达式的例子（见 *org.activiti.examples.bpmn.expression*）：



7.5.11 默认顺序流

描述

所有 BPMN 2.0 任务以及分支都可以有一个**默认顺序流**。当且仅当没有其它顺序流被选择时，才会选择该顺序流作为活动的输出顺序流。默认顺序流上的条件总是被忽略掉。

图形化符号

默认顺序流被形象化成起点带‘斜线’的普通顺序流。



XML 表示

某个活动的默认顺序流是由那个活动上的 **default** 属性定义的。下面的 XML 片段展示了包含一个默认顺序流 *flow2* 的排他分支。只有当 *conditionA* 和 *conditionB* 都为 *false* 时，才会选取它作为分支的输出顺序流。

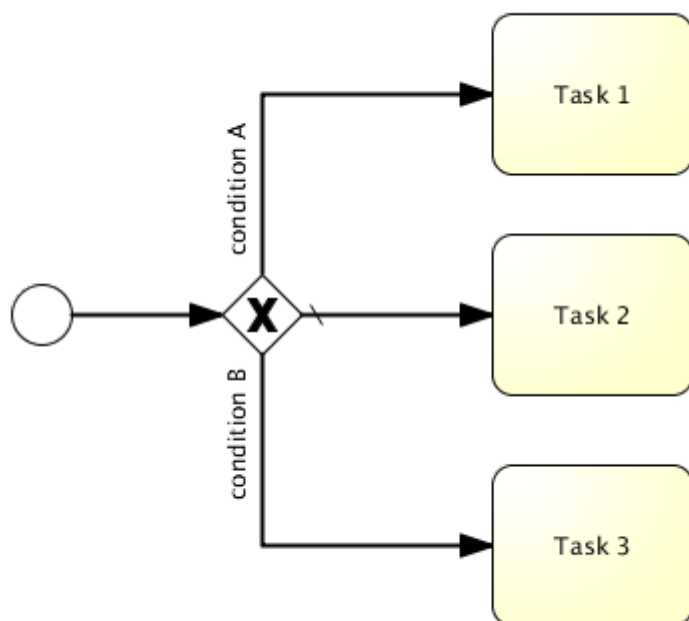
```

<exclusiveGateway id="exclusiveGw" name="Exclusive Gateway" default="flow2"/>
<sequenceFlow id="flow1" sourceRef="exclusiveGw" targetRef="task1">
  <conditionExpression xsi:type="tFormalExpression">${conditionA}</conditionExpression>
</sequenceFlow>
<sequenceFlow id="flow2" sourceRef="exclusiveGw" targetRef="task2"/>
<sequenceFlow id="flow3" sourceRef="exclusiveGw" targetRef="task3">
  <conditionExpression xsi:type="tFormalExpression">${conditionB}</conditionExpression>

```

```
</sequenceFlow>
```

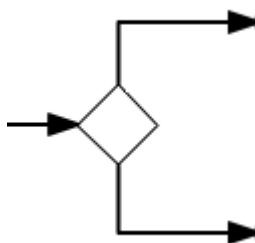
与下图示相符：



7.5.12 分支

gateway 用来控制执行流（或如 BPMN 2.0 描述的，执行令牌）。gateway 可以回收或创建令牌。

gateway 被形象化为里面有图标的菱形。图标说明了 gateway 的类型。



7.5.13 排他分支

描述

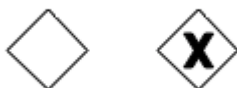
排他分支（也称为 XOR 分支，或更专业点，exclusive data-based gateway）用来对流程中的**决定**进行建模。流程执行到该分支时，按照输出流定义的顺序对它们进行计算。条件为 true 的顺序流（或没有设置条件，从概念上讲顺序流上定义的为‘true’）被选中继续执行流程。

注意此处输出顺序流的含义与 BPMN 2.0 中一般情形下的顺序流是不一样的。虽然通常所有那些条件为 true 的顺序流都

会被选中以并行的方式继续流程的执行，但在使用排他分支时，只能有一个顺序流被选中。在多个顺序流条件为 **true** 的情况下，XML 中最先定义的那个被选中来继续流程的执行（仅有那个会被选中）。如果没有选中任何顺序流，就会抛出异常。

图形化符号

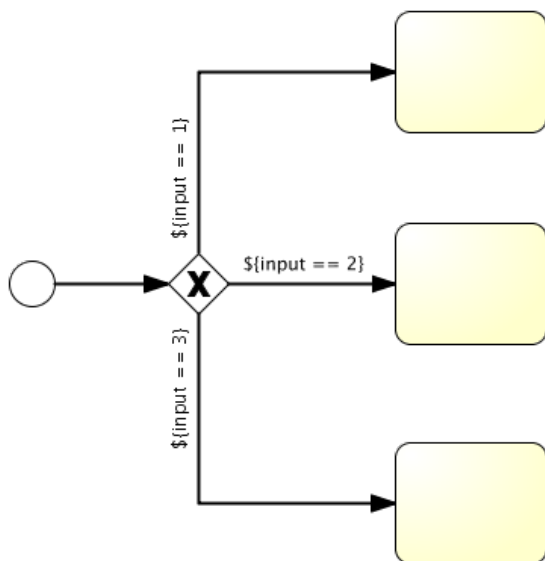
排他分支被形象化为里面有一个‘X’的特殊分支（即一个菱形），指的是 XOR 的语义。注意，不带图标的分支默认是排他分支。BPMN 2.0 规范不允许在同一流程定义中同时使用带 X 和不带 X 的菱形。



XML 表示

排他分支的 XML 表示是很直观的：定义分支的行以及定义在输出顺序流上的条件表达式。参看关于[条件顺序流](#)的一节来查看该表达式有哪些可用的选项。

例如以下模型：



XML 的表示如下：

```
<exclusiveGateway id="exclusiveGw" name="Exclusive Gateway"/>

<sequenceFlow id="flow2" sourceRef="exclusiveGw" targetRef="theTask1">
  <conditionExpression xsi:type="tFormalExpression">${input == 1}</conditionExpression>
</sequenceFlow>

<sequenceFlow id="flow3" sourceRef="exclusiveGw" targetRef="theTask2">
  <conditionExpression xsi:type="tFormalExpression">${input == 2}</conditionExpression>
```

```
</sequenceFlow>
```

```
<sequenceFlow id="flow4" sourceRef="exclusiveGw" targetRef="theTask3">
```

```
<conditionExpression xsi:type="tFormalExpression">${input == 3}</conditionExpression>
```

```
</sequenceFlow>
```

7.5.14 并行分支

描述

分支也可以用来对流程中的并发进行建模。流程模型中引入并行最简单的分支就是**并行分支**，它能拆分出多个执行路径或将多个输入执行路径进行合并。

并行分支的功能取决于输入和输出的顺序流：

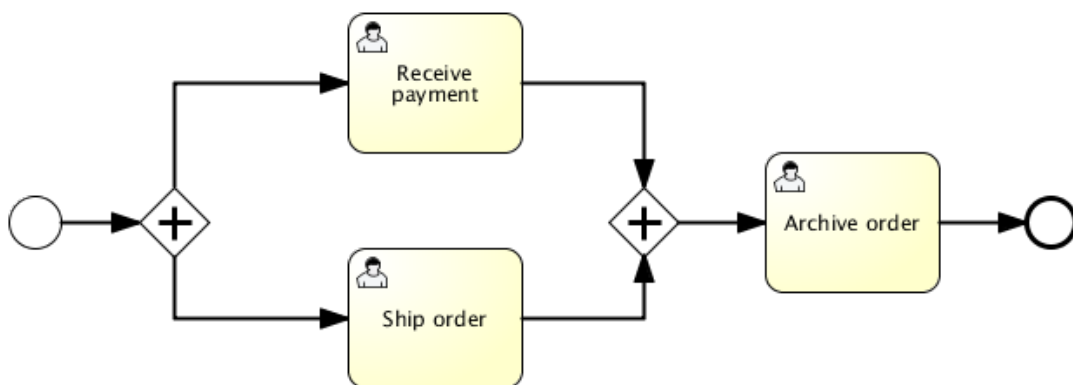
- **拆分 (fork)**：并行执行所有的输出顺序流，为每一个顺序流创建一个并行执行路径。
- **合并 (join)**：所有到达并行分支的并发性的执行路径都等待于此，直到执行路径都执行到它们各自的输入流。然后，流程通过合并分支继续向下执行。

注意：如果并行分支有多个输入流和输出流，那么它可以同时具有 **fork 和 join 行为**。这样，分支会在拆分出多个并发的执行路径前，首先合并所有的输入流。

与其它分支一个重要的不同点是并行分支不会计算条件。如果在连接并行分支的顺序流上定义了条件，那么那些条件会被简单地忽略掉。

图形化符号

并行分支被形象化成里面有‘加号’的分支（菱形），加号表示‘与’的意思。



XML 表示

XML 中需要使用一行这样的代码来定义并行分支：

```
<parallelGatewayid="myParallelGateway"/>
```

实际的行为（fork、join）由连接到并行分支的顺序流来定义。

例如，上面的模型归结为如下的 XML：

```
<startEventid="theStart"/>
<sequenceFlowid="flow1"sourceRef="theStart"targetRef="fork"/>

<parallelGatewayid="fork"/>
<sequenceFlowsourceRef="fork"targetRef="receivePayment"/>
<sequenceFlowsourceRef="fork"targetRef="shipOrder"/>

<userTaskid="receivePayment"name="Receive Payment"/>
<sequenceFlowsourceRef="receivePayment"targetRef="join"/>

<userTaskid="shipOrder"name="Ship Order"/>
<sequenceFlowsourceRef="shipOrder"targetRef="join"/>

<parallelGatewayid="join"/>
<sequenceFlowsourceRef="join"targetRef="archiveOrder"/>

<userTaskid="archiveOrder"name="Archive Order"/>
<sequenceFlowsourceRef="archiveOrder"targetRef="theEnd"/>

<endEventid="theEnd"/>
```

上面的示例中，在启动流程后，将创建两个任务：

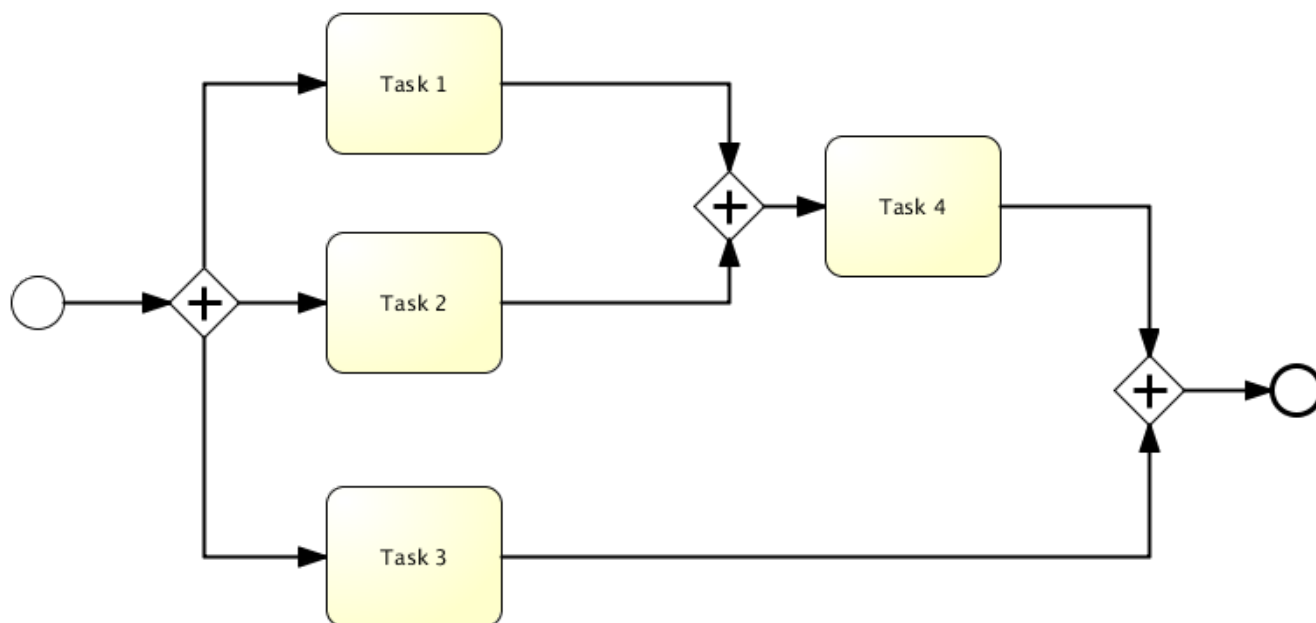
```
ProcessInstance pi = runtimeService.startProcessInstanceByKey("forkJoin");
TaskQuery query = taskService.createTaskQuery()
    .processInstanceId(pi.getId())
    .orderByTaskName()
    .asc();

List<Task> tasks = query.list();
assertEquals(2, tasks.size());

Task task1 = tasks.get(0);
assertEquals("Receive Payment", task1.getName());
Task task2 = tasks.get(1);
assertEquals("Ship Order", task2.getName());
```

当这两个任务完成后，第二个并行分支将合并这两条执行路径，因为只有一条输出流，所以不会创建并发的执行路径，只有 *Archive Order* 任务被创建。

注意，不需要‘均衡’并行分支（即，为并行分支匹配输入/输出顺序流得数量）。并行分支会等所有的输入流都到达，然后为每个输出流分别创建一个并发的执行路径，这不会受流程模型中其他构造的影响。所以，下面的流程是符合 BPMN 2.0 的：



7.5.15 包容分支

描述

包容分支可以被视为是排他分支和并行分支的结合。像排他分支一样，你可以在输入流上定义条件，包容分支会计算这些条件。但主要的不同是包容分支可以像并行分支那样选择不只一个顺序流。

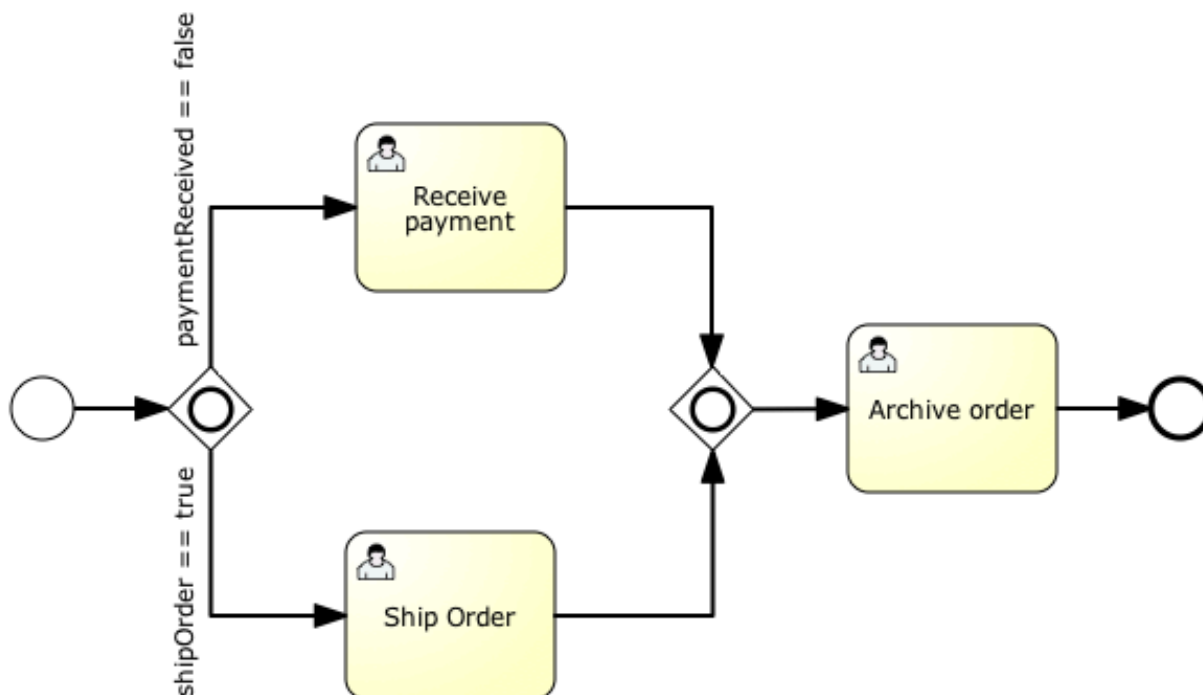
包容分支的功能取决于输入和输出的顺序流：

- **拆分 (fork)：** 计算所有的输出流条件，对于条件为 **true** 的顺序流，会为其每个创建一个并行执行路径而被延续执行。
- **合并 (join)：** 所有到达包容分支的并发的执行路径都等待于此，直到有流程令牌的执行路径都执行到它们各自的输入流（译注，这句话初看很费解，其实此处倒是体现了包容分支的实质：因为排他分支没有 **join** 功能；而并行分支虽然有 **join** 功能，但是它只适用于并行建模，因为它会等待所有指向它的输入流所在的执行路径，这样一来指向它的所有输入流都必须具有执行令牌（即可执行）；如果碰到这样的场景：{条件 a：路径 a}，{条件 b：路径 b}，{条件 c：路径 c}，如果条件 a,b 成立，c 不成立，路径 a,b 会执行，路径 c 不会执行，如果在建模时我们将路径 a,b,c 交汇于并行分支，那么将会再也无法进行，其实这从并行建模的概念上没有准确的理解并行分支）。这是与并行分支主要的不同。因此换句话说，包容分支仅仅等待将来要被执行的输入流。合并后，流程经由合并包容分支继续向下执行。

注意：如果一个包容分支有多个输入流和输出流，那么它可以同时具有 **fork** 和 **join** 行为。这样的话，此分支会在为条件是 **true** 的输出顺序流拆分成多个并发的执行路径前，首先合并所有拥有流程令牌的输入流。

图形化符号

包容分支被形象化成内部有‘圆’标记的分支（菱形）。



XML 表示

定义包容分支要使用一行这样的 XML：

```
<inclusiveGateway id="myInclusiveGateway"/>
```

实际的行为（fork、join）是由连接到包容分支的顺序流来定义。

例如，上面的模型归结为如下的 XML：

```

<startEvent id="theStart"/>
<sequenceFlow id="flow1" sourceRef="theStart" targetRef="fork"/>

<inclusiveGateway id="fork"/>
<sequenceFlow sourceRef="fork" targetRef="receivePayment">
<conditionExpression xsi:type="tFormalExpression">${paymentReceived == false}</conditionExpression>
</sequenceFlow>
<sequenceFlow sourceRef="fork" targetRef="shipOrder">
<conditionExpression xsi:type="tFormalExpression">${shipOrder == true}</conditionExpression>
</sequenceFlow>

<userTask id="receivePayment" name="Receive Payment"/>
  
```

```

<sequenceFlow sourceRef="receivePayment" targetRef="join"/>

<userTask id="shipOrder" name="Ship Order"/>
<sequenceFlow sourceRef="shipOrder" targetRef="join"/>

<inclusiveGateway id="join"/>
<sequenceFlow sourceRef="join" targetRef="archiveOrder"/>

<userTask id="archiveOrder" name="Archive Order"/>
<sequenceFlow sourceRef="archiveOrder" targetRef="theEnd"/>

<endEvent id="theEnd"/>

```

以上例子中，流程启动后，如果流程变量 `paymentReceived == false` 且 `shipOrder == true`，那么两个任务将被创建。如果只有一个流程变量的结果 `true`，那么只有一个任务被创建。如果没有条件结果为 `true`，就会抛出异常。可以通过指定默认的输出顺序流来避免这样的异常发生。一下例子中，会创建一个任务，`ship order` 任务：

```

HashMap<String, Object> variableMap = new HashMap<String, Object>();
variableMap.put("receivedPayment", true);
variableMap.put("shipOrder", true);
ProcessInstance pi = runtimeService.startProcessInstanceByKey("forkJoin");
TaskQuery query = taskService.createTaskQuery()
    .processInstanceId(pi.getId())
    .orderByTaskName()
    .asc();

List<Task> tasks = query.list();
assertEquals(1, tasks.size());

Task task = tasks.get(0);
assertEquals("Ship Order", task.getName());

```

当该任务完成时，第二个包容分支会将这两个执行路径进行合并，因为只有一条输出顺序流，所以不会创建并行的执行路径，并且只有 *Archive Order* 任务会被激活。

注意，不需要对包容分支进行‘均衡’（即，为包容分支匹配输入/输出顺序流得数量）。包容分支会等所有的输入流都到达，然后为每个输出流分别创建一个并发的执行路径，这不会受流程模型中其他构造的影响。

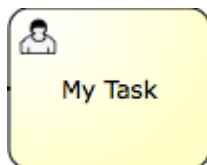
7.5.16 用户任务

描述

‘用户任务’用来对那些需要人参与完成的工作进行建模。当流程执行到这样的用户任务时，会在被分配到该任务的用户或用户组的任务列表中创建新的任务。（译注，即在用户或用户组的任务列表中创建新的任务）

图形化符号

用户任务被形象化成左上有一个小人图标的特殊任务（椭圆）。



XML 表示

XML 中定义用户任务如下。*id* 属性是必须的，*name* 属性是可选的。

```
<userTask id="theTask" name="Important task"/>
```

用户任务中可以包含描述。事实上，任何 BPMN 2.0 中的元素都可以有描述。描述是通过添加 **documentation** 元素来定义。

```
<userTask id="theTask" name="Schedule meeting">
  <documentation>
    Schedule an engineering meeting for next week with the new hire.
  </documentation>
</userTask>
```

描述文本可以以标准 java 的方式从任务中获得：

```
task.getDescription()
```

到期时间

每个任务都含有一个表明该任务到期时间的字段。Query API 可以用来查询在某个时间点的前或后任务是否过期。

有个 Activity 的扩展，允许在任务定义中指定一个表达式来设置在创建任务时任务初始的超期时间。该表达式结果必须是 **java.util.Date** 或 **null**。例如，你可以使用由流程中之前表单输入或在之前 Service Task 计算出来的日期。

```
<userTask id="theTask" name="Important task" activiti:dueDate="${dateVariable}"/>
```

任务的超期时间也可以使用 TaskService 或在任务监听器中利用传过来的 DelegateTask 进行修改。

用户的分配

用户任务可以直接分配给用户。这是通过定义 **humanPerformer** 子元素来完成的。那样一个 *humanPerformer* 定义需要 **resourceAssignmentExpression** 元素，该元素实际上定义了用户。目前，只支持 **formalExpressions**。

```
<process... >
  ...
  <userTask id='theTask' name='important task'>
    <humanPerformer>
      <resourceAssignmentExpression>
```

```
<formalExpression>kermit</formalExpression>
</resourceAssignmentExpression>
</humanPerformer>
</userTask>
```

只能有一个用户作为执行者分配到任务上。在 Activiti 术语中，该用户称为**代理人**（译注，或称为**责任人**）。存在代理人的任务在其他人的任务列表中是不可见的，这些任务存在于所谓的**代理人个人任务列表**中。

直接分配给用户的任务可以通过 `TaskService` 来获取，如下：

```
List<Task> tasks = taskService.createTaskQuery().taskAssignee("kermit").list();
```

也可以把任务放进所谓的人员的**候选任务列表**中。这时，就要利用 **potentialOwner** 了。用法类似于 *humanPerformer*。一定要注意需要对 **formal** 表达式中的每个元素进行定义以指名是用户还是用户组（流程引擎是猜测不到的）。

```
<process... >

...

<userTaskid='theTask' name='important task'>
<potentialOwner>
<resourceAssignmentExpression>
<formalExpression>user(kermit), group(management)</formalExpression>
</resourceAssignmentExpression>
</potentialOwner>
</userTask>
```

使用 *potential owner* 定义的任务可以按照如下方式获取（或类似于在有代理者任务中使用 *TaskQuery*）：

```
List<Task> tasks = taskService.createTaskQuery().taskCandidateUser("kermit");
```

这将获得所有 **kermit** 作为**候选用户**的任务，也就是，**formal** 表达式包含的 *user(kermit)*。这也会获得所有**分配给 kermit 所在组**（例如，*group(management)*，如果 **kermit** 是那个组的成员，并且使用了 *identity* 组件）的任务。用户组是在运行时解析的，并且用户组可以通过 *IdentityService* 管理。

如果不给文本字符串指定是用户还是用户组，流程引擎默认认为是用户组。因此下面这与**声明为 group(accountancy)**效果是一样的。

```
<formalExpression>accountancy</formalExpression>
```

Activiti 对于任务分配的扩展

用户和用户组的分配在那些分配并不复杂的情况下显然是很麻烦的。为了避免这种复杂性，用户任务上的**自定义扩展**就变得可能了。

- **assignee 属性**：这个自定义扩展允许将用户任务直接分配给用户。

```
<userTaskid="theTask" name="my task" activiti:assignee="kermit"/>
```

这与[上面](#)使用 **humanPerformer** 效果是一样的。

- **candidateUsers 属性：**这个自定义扩展可以使用户成为任务的候选者。

```
<userTaskId="theTask"name="my task"activiti:candidateUsers="kermit, gonzo"/>
```

这与上面使用 **potentialOwner** 效果是一样的。注意不要求使用像在 *potential owner* 中使用的 *user(kermit)* 声明，因为该属性只用于用户。

- **candidateGroups 属性：**这个自定义扩展允许为任务定义一组候选者。

```
<userTaskId="theTask"name="my task"activiti:candidateGroups="management, accountancy"/>
```

这与上面使用 **potentialOwner** 效果是一样的。注意不要求使用像在 *potential owner* 中使用的 *group(management)* 声明，因为该属性只用于组。

- **candidateUsers 和 candidateGroups** 可以定义在同一用户任务上。

如果以上方案仍然不够，那么可以使用 **create** 事件上的[任务监听器](#)让自定义的分配逻辑来处理：

```
<userTaskId="task1"name="My task">
<extensionElements>
<activiti:taskListenerevent="create"class="org.activiti.MyAssignmentHandler"/>
</extensionElements>
</userTask>
```

传递给任务监听器实现的 **DelegateTask** 允许设置代理人和候选用户/组：

```
publicclass MyAssignmentHandler implements TaskListener {

publicvoid notify(DelegateTask delegateTask) {
// 在此执行自定义的身份查找

// 接下来，例如调用以下方法：
delegateTask.setAssignee("kermit");
delegateTask.addCandidateUser("fozzie");
delegateTask.addCandidateGroup("management");

...
}

}
```

使用 **Spring** 时可能会使用到上面章节中介绍的自定义分配属性，并利用带[表达式](#)的任务监听器监听 **create** 事件将处理委托给 **Spring** 的 **bean**。下面的例子中，代理人是通过调用 **IdapServiceSpring bean** 中的方法 **findManagerOfEmployee** 来设置的。传递的 **emp** 参数，是个流程变量。

```
<userTaskId="task"name="My Task"activiti:assignee="${IdapService.findManagerForEmployee(emp)}/>
```

这对于候选用户和候选组的情况也是类似的：

```
<userTaskId="task"name="My Task"activiti:candidateUsers="${IdapService.findAllSales()}/>
```

注意只有当被调用的方法返回类型是 **String** 或 **Collection<String>**（对于候选用户和候选组）才有效。

```

public class FakeLdapService {

    public String findManagerForEmployee(String employee) {
        return "Kermit The Frog";
    }

    public List<String> findAllSales() {
        return Arrays.asList("kermit", "gonzo", "fozzie");
    }

}

```

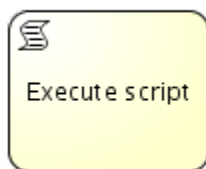
7.5.17 脚本任务

描述

脚本任务是自动的活动。当流程执行到脚本任务时，执行相应的脚本。

图形化符号

脚本任务被形象化成左上角带有‘脚本’图标的特殊的 BPMN 2.0 任务（椭圆）。



XML 表示

通过指定 **script** 和 **scriptFormat** 来定义脚本任务。

```

<scriptTask id="theScriptTask" name="Execute script" scriptFormat="groovy">
  <script>
    sum = 0
    for ( i in inputArray ) {
      sum += i
    }
  </script>
</scriptTask>

```

scriptFormat 属性的值必须是 [JSR-223](#)（Java 平台脚本，scripting for the Java platform）所兼容的名称。默认 Groovy jar 随 Activiti 的发布包被一起分发了。如果你想要使用其他（JSR-223 兼容的）脚本引擎，只要将相应的 jar 添加到类路径下，

然后使用恰当的名称就行了。

脚本中的变量

所有那些进入脚本任务的执行路径能访问到的流程变量都可以在脚本中使用。该例子中，脚本变量'*inputArray*'实际上是个（整形数组类型的）流程变量。

```
<script>
    sum = 0
    for ( i in inputArray ) {
        sum += i
    }
</script>
```

也可以使用赋值语句在脚本中设置流程变量。在上面的例子中，在脚本任务执行完成后'*sum*'变量将作为流程变量存储起来。要避免这种行为，可以使用本地脚本变量。在 Groovy 中，需要使用关键字'*def*': '*def sum = 0*'。那样，流程变量就不会被存储了。

另一种方法是使用当前的 *execution* 来设置变量，它是被称为'*execution*'的保留变量。

```
<script>
    def scriptVar = "test123"
    execution.setVariable("myVar", scriptVar)
</script>
```

注意：以下名称被保留，不能用来做为变量的名称：**out**、**out:print**、**lang:import**、**context**、**elcontext**。

脚本的结果

通过给脚本任务定义的'*activity:resultVariable*'属性指定一个字符串来表示流程变量名，就可以将脚本任务的返回值分配给一个现有的或新的流程变量。流程变量现值会被脚本执行结果值所重写。不指定结果变量名时，会忽略脚本的结果值。

```
<scriptTask id="theScriptTask" name="Execute script" scriptFormat="juel" activity:resultVariable="myVar">
<script>#{echo}</script>
</scriptTask>
```

上面的例子中，在脚本执行完成后，脚本执行的结果（表达式'*#{echo}*'的结果值）被设置到名称为'*myVar*'的流程变量中。

7.5.18 Java 服务任务

描述

Java 服务任务用来调用外部 Java 类。

图形化符号

服务任务被形象化成左上角带有小齿轮图标椭圆。



XML 表示

有 4 种方式来声明如何调用 Java 的逻辑：

- 指定实现了 `JavaDelegate` 或 `ActivitiBehavior` 的类
- 计算结果为代理对象的表达式
- 调用方法表达式
- 计算值表达式

要指定在流程执行期间被调用的类，需要使用 `activity:class` 属性来提供完全限定的类名。

```
<serviceTask id="javaService"
    name="My Java Service Task"
    activiti:class="org.activiti.MyJavaDelegate"/>
```

更多关于如何使用这样的类的细节见[实现](#)一节。

也可以使用解析结果为对象的表达式。这个对象必须遵循与使用 `activiti:class` 属性创建对象时一样的规则（见[下文](#)）。

```
<serviceTask id="serviceTask" activiti:delegateExpression="${delegateExpressionBean}"/>
```

这里，`delegateExpressionBean` 是一个定义在 Spring 容器中的实现了 `JavaDelegate` 接口的 bean。

使用属性 `activiti:expression` 指定一个会被计算的 UEL 方法表达式。

```
<serviceTask id="javaService"
    name="My Java Service Task"
    activiti:expression="#{printer.printMessage()}/>
```

会调用 `printer` 对象上的方法 `printMessage`（不带参数）。

也可以向表达式的方法中传递参数。

```
<serviceTask id="javaService"
    name="My Java Service Task"
    activiti:expression="#{printer.printMessage(execution, myVar)}/>
```

会调用 `printer` 对象上的方法 `printMessage`。传递的第一个参数是 `DelegateException`，其在表达式上下文默认以 `execution` 来使用。传递的第二个参数是当前 `execution` 中名为 `myVar` 变量的值。

使用属性 `activiti:expression` 来指定一个会被计算的 UEL 值表达式。

```
<serviceTask id="javaService"
```

```
name="My Java Service Task"
activiti:expression="#{split.ready}"/>
```

会调用名称为 `split` 的 bean 上属性 `ready` 的 `getter` 方法，`getReady`（不带参数）。命名对象是在流程执行中的流程变量和（如果适用）Spring 上下文中被解析的。

实现

要实现一个可以在流程执行期间中调用的类，该类需要实现 `org.activiti.engine.delegate.JavaDelegate` 接口，在 `execute` 方法中提供必要的逻辑。当流程执行到此步，会执行定义在该方法中的逻辑，然后以 BPMN 2.0 的默认方式离开该活动。

让我们举例来创建一个 Java 类，该类用来将流程变量的字符串改为大写。这个类需要实现

`org.activiti.engine.delegate.JavaDelegate` 接口，该接口需要我们实现 `execute(DelegateExecution)` 方法。就是这个方法会被流程引擎调用到，并且需要包含业务逻辑。可以通过 [DelegateExecution](#) 接口来访问、操作流程实例的信息，如流程变量以及其它。（点击链接了解 Javadoc 中详细操作）

```
public class ToUppercase implements JavaDelegate {

    public void execute(DelegateExecution execution) throws Exception {
        String var = (String) execution.getVariable("input");
        var = var.toUpperCase();
        execution.setVariable("input", var);
    }
}
```

注意：只会创建定义在 `serviceTask` 上的 java 类的一个实例。所有流程实例共享同一个用于调用 `execute(DelegateExecution)` 的类实例。这意味着，该类中一定不要使用成员变量，并且必须是线程安全的，因为可能会在不同的线程中同时执行该方法。这也能影响处理[字段注入](#)的方式。

流程定义中引用的类（即使用 `activiti:class`）在部署时不会被实例化。只有当流程第一次执行到使用到该类的时候，才创建该类的实例。如果找不到该类，会抛出 `ActivitiException`。这是由于部署的环境（特别是类路径）与实际运行的环境往往是不同的。比如，在使用 `ant` 或在 `Activiti Explorer` 中使用业务归档文件来部署流程时，类路径不包含参照的类。

[\[内部的：非公布的实现类\]](#)可能会提供一个实现了 `org.activiti.engine.impl.pvm.delegate.ActivityBehavior` 接口的类。接下来实现类就能够访问更强大的 `ActivityExecution` 了，比如，它可以影响流程的控制流。但要注意这个不是一个很好的做法，应该尽量避免这样做。所以，对于高级的用例，如果你真正知道你要做什么，建议使用接口 `ActivityBehavior`。

字段的注入

可以向代理类的字段注入值。支持以下注入形式：

- 固定字符串值
- 表达式

如果可以的话，是通过遵循 Java Bean 的命名规范的代理类中（例如，字段 `firstName` 的 `setter` 方法是 `setFirstName(...)`）的 `public setter` 方法将值注入的。如果字段不存在可用的 `setter` 方法，将设置代理类的 `private` 成员变量。在一些情况下，`SecurityManagers` 是不允许修改 `private` 字段的，所以给你要进行注入的字段公布 `public setter` 方法会更加安全。不管流程定义中值声明成什么类型，注入目标类上的 `setter` 或 `private` 字段的类型必须是 `org.activiti.engine.delegate.Expression`。

下面的代码片段展示了如何向字段中注入常量。使用‘class’属性进行字段注入。**注意，在实际的字段注入声明的前面，需要声明‘extensionElements’ XML 元素，这是 BPMN 2.0 XML 模式的要求。**

```
<serviceTask id="javaService"
name="Java service invocation"
activiti:class="org.activiti.examples.bpmn.servicetask.ToUpperCaseFieldInjected">
<extensionElements>
<activiti:fieldname="text" stringValue="Hello World"/>
</extensionElements>
</serviceTask>
```

类 ToUpperCaseFieldInjected 有一个类型为 org.activiti.engine.delegate.Expression 的 text 字段。当调用 text.getValue(execution) 时，返回配置的‘Hello world’字符串。

或者，对于长文本（例如，一行 e-mail），可以使用子元素‘activiti:string’：

```
<serviceTask id="javaService"
name="Java service invocation"
activiti:class="org.activiti.examples.bpmn.servicetask.ToUpperCaseFieldInjected">
<extensionElements>
<activiti:fieldname="text">
<activiti:string>
Hello World
</activiti:string>
</activiti:field>
</extensionElements>
</serviceTask>
```

要注入运行时动态解析的值，可以使用表达式。表达式中可以使用流程变量，或 Spring 定义的 bean（如果使用了 Spring）。如[服务任务实现](#)中所描述的，所有流程实例共享一个定义在服务任务中的 Java 类实例。要想达到字段上值的动态注入，可以将值表达式和方法表达式注入到 org.activiti.engine.delegate.Expression，它会使用 execute 方法中传进来的 DelegateExecution 对 org.activiti.engine.delegate.Expression 进行运算/调用。

```
<serviceTask id="javaService" name="Java service invocation"
activiti:class="org.activiti.examples.bpmn.servicetask.ReverseStringsFieldInjected">

<extensionElements>
<activiti:fieldname="text1">
<activiti:expression>${genderBean.getGenderString(gender)}</activiti:expression>
</activiti:field>
<activiti:fieldname="text2">
<activiti:expression>
Hello ${gender == 'male' ? 'Mr.' : 'Mrs.} ${name}
</activiti:expression>
</activiti:field>
</extensionElements>
</serviceTask>
```


下面的示例类使用了被注入的表达式，并使用了当前 `DelegateExecution` 对这些表达式进行解析。完整代码以及测试可以在 `org.activiti.examples.bpmn.servicetask.JavaServiceTaskTest.testExpressionFieldInjection` 中找到。

```
public class ReverseStringsFieldInjected implements JavaDelegate {

    private Expression text1;
    private Expression text2;

    public void execute(DelegateExecution execution) {
        String value1 = (String) text1.getValue(execution);
        execution.setVariable("var1", new StringBuffer(value1).reverse().toString());

        String value2 = (String) text2.getValue(execution);
        execution.setVariable("var2", new StringBuffer(value2).reverse().toString());
    }
}
```

或者，你也可以以属性的方式设置表达式，而不是使用子元素，这样可以使 XML 显得不那么冗长。

```
<activiti:fieldname="text1" expression="${genderBean.getGenderString(gender)}"/>
<activiti:fieldname="text1" expression="Hello ${gender == 'male' ? 'Mr.' : 'Mrs.'} ${name}"/>
```

由于该 `java` 类的实例是可重用的，所以注入只在 `serviceTask` 第一次被调用时发生。一旦在代码中修改过了这些字段，其值不会再被重新注入了，因此你应该把它们看作是不可变的，并且对它们不要做任何的修改。

服务任务的结果

通过为服务任务定义的 `activiti:resultVariable` 属性指定一个字符串表示的流程变量名，可以将服务执行（服务任务仅使用了表达式）的返回值分配给一个现有的或一个新的流程变量。服务执行的返回值会重写流程变量的当前值。如果没有指定结果变量名，服务执行的结果值会被忽略。

```
<serviceTask id="aMethodExpressionServiceTask"
  activiti:expression="#{myService.doSomething()}"
  activiti:resultVariable="myVar"/>
```

上面示例中，服务执行完成后，服务执行的结果（调用流程变量或 `Spring bean` 中名为 `myService` 的对象上方 `doSomething()` 的返回值）被设置到了叫 `myVar` 的流程变量。

处理异常

在执行自定义的逻辑时，常常需要捕获某种异常。一个常见的用例是一旦某条路径上发生异常，将流程导向另一条路径。下面的例子展示了这是如何做到的。

```
<serviceTask id="javaService"
  name="Java service invocation"
  activiti:class="org.activiti.ThrowsExceptionBehavior">
</serviceTask>
```

```
<sequenceFlow id="no-exception" sourceRef="javaService" targetRef="theEnd"/>
<sequenceFlow id="exception" sourceRef="javaService" targetRef="fixException"/>
```

这里，服务任务有两条输出流，分别是 exception 和 no-exception。一旦发生异常，顺序流的 id 属性用来引导顺序流。

```
public class ThrowsExceptionBehavior implements ActivityBehavior {

    public void execute(ActivityExecution execution) throws Exception {
        String var = (String) execution.getVariable("var");

        PvmTransition transition = null;
        try {
            executeLogic(var);
            transition = execution.getActivity().findOutgoingTransition("no-exception");
        } catch (Exception e) {
            transition = execution.getActivity().findOutgoingTransition("exception");
        }
        execution.take(transition);
    }
}
```

7.5.19 WebService 任务

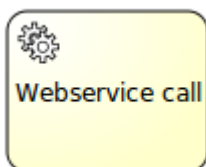
[试验性的]

描述

WebService 任务用于同步调用外部 web 服务。

图形化符号

WebService 任务与 Java 服务任务的表示是一样。



XML 表示

要使用 WebService，我们需要引入其操作方法以及它那些复杂的类型。使用指向该 WebService 的 WSDL 的 import 标签就能自动完成。

```
<import importType="http://schemas.xmlsoap.org/wsdl/"
location="http://localhost:63081/counter?wsdl"/>
```

```
namespace="http://webservice.activiti.org/" />
```

上面的声明是让 Activiti 引入定义，但并不会为你创建项目定义以及消息。如果我们想要调用一个叫“prettyPrint”的方法，那么我们需要为请求和响应的消息创建对应的消息以及项目定义：

```
<messageid="prettyPrintCountRequestMessage"itemRef="tns:prettyPrintCountRequestItem"/>
<messageid="prettyPrintCountResponseMessage"itemRef="tns:prettyPrintCountResponseItem"/>

<itemDefinitionid="prettyPrintCountRequestItem"structureRef="counter:prettyPrintCount"/>
<itemDefinitionid="prettyPrintCountResponseItem"structureRef="counter:prettyPrintCountResponse"/>
```

声明服务任务前，我们必须定义 BPMN 接口以及实际关联到 WebService 方法的操作。基本上，我们只需要定义‘interface’以及所需的‘operations’。对于每个 operation 的传入传出的消息重用了前面定义 messages。例如，下面的声明定义了‘counter’接口和‘prettyPrintCountOperation’操作：

```
<interfacename="Counter Interface"implementationRef="counter:Counter">
<operation id="prettyPrintCountOperation"name="prettyPrintCount Operation"
implementationRef="counter:prettyPrintCount">
<inMessageRef>tns:prettyPrintCountRequestMessage</inMessageRef>
<outMessageRef>tns:prettyPrintCountResponseMessage</outMessageRef>
</operation>
</interface>
```

接下来，通过使用值为##WebService 的 implementation 属性以及指向该 WebService 操作的引用就可以声明 WebService 任务了。

```
<serviceTaskid="webService"
name="Web service invocation"
implementation="##WebService"
operationRef="tns:prettyPrintCountOperation">
```

WebService 任务的 IO 规范

除非我们使用的是针对数据输入、输出关系的简化方案（见下文），否则每个 WebService 任务都需要声明 IO 规范来表述哪些是任务的输入、哪些是任务的输出。这个方案非常简单，但并不是 BPMN 2.0 所支持的，拿 prettyPrint 来说，根据前面声明的 item definitions，我们定义了 input 集和 output 集：

```
<ioSpecification>
  <dataInputitemSubjectRef="tns:prettyPrintCountRequestItem"id="dataInputOfServiceTask"/>
  <dataOutputitemSubjectRef="tns:prettyPrintCountResponseItem"id="dataOutputOfServiceTask"/>
  <inputSet>
    <dataInputRefs>dataInputOfServiceTask</dataInputRefs>
  </inputSet>
  <outputSet>
    <dataOutputRefs>dataOutputOfServiceTask</dataOutputRefs>
  </outputSet>
</ioSpecification>
```

服务任务的数据输入关系

有两种方式指定数据的输入关系：

- 使用表达式
- 使用简化的方案

使用表达式指定数据的输入关系，我们需要定义 **source** 项和 **target** 项，然后对每项的字段进行分配。下面的例子中，我们给项目分配了 **prefix** 字段和 **suffix** 字段：

```
<dataInputAssociation>
<sourceRef>dataInputOfProcess</sourceRef>
<targetRef>dataInputOfServiceTask</targetRef>
<assignment>
<from>${dataInputOfProcess.prefix}</from>
<to>${dataInputOfServiceTask.prefix}</to>
</assignment>
<assignment>
<from>${dataInputOfProcess.suffix}</from>
<to>${dataInputOfServiceTask.suffix}</to>
</assignment>
</dataInputAssociation>
```

另一方面，我们可以使用更简单的简化方案。‘**sourceRef**’元素是 **Activiti** 变量名，‘**targetRef**’元素是项目定义的一个属性。下面的例子中，我们将变量‘**PrefixVariable**’的值分配给了字段‘**prefix**’，将变量‘**SuffixVariable**’的值分配给了为字段“**suffix**”：

```
<dataInputAssociation>
<sourceRef>PrefixVariable</sourceRef>
<targetRef>prefix</targetRef>
</dataInputAssociation>
<dataInputAssociation>
<sourceRef>SuffixVariable</sourceRef>
<targetRef>suffix</targetRef>
</dataInputAssociation>
```

服务任务的数据输出关系

有两种方式指定数据的输出关系：

- 使用表达式
- 使用简化的方案

使用表达式指定数据的输出关系，我们需要定义 **target** 变量，以及 **source** 表达式。这个方案很简单，类似于数据的输入关系：

```
<dataOutputAssociation>
<targetRef>dataOutputOfProcess</targetRef>
<transformation>${dataOutputOfServiceTask.prettyPrint}</transformation>
</dataOutputAssociation>
```

另一方面，我们可以使用更简单的简化方案。'sourceRef'元素是项目定义的一个属性，'targetRef'元素是 Activiti 的一个变量名。方案很简单，类似于数据的输入关系：

```
<dataOutputAssociation>
<sourceRef>prettyPrint</sourceRef>
<targetRef>OutputVariable</targetRef>
</dataOutputAssociation>
```

7.5.20 业务规则任务

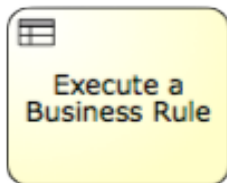
[试验性的]

描述

业务规则任务用于同步执行一个或更多规则。Activiti 使用 Drools Expert 和 Drool 规则引擎来执行业务规则。目前而言，包含有业务规则的.drl 文件必须与定义了业务规则任务的流程定义一同部署才能执行这些规则。这意味着在流程中使用的所有.drl 文件必须像任务表单一样被打包进流程的 BAR 文件中。更多关于使用 Drool Expert 来创建业务规则的信息，请参考 [Jboss Drools](#) 上的 Drools 文档。

图形化符号

业务规则任务是使用表格图标来表示的。



XML 表示

要执行部署在与流程定义所在 BAR 文件中的一个或更多业务规则，我们需要定义输入变量和结果变量。对于输入变量的定义，可以定义一个由逗号分隔的流程变量列表。输出变量的定义可以只包含一个变量名，用来将执行过的业务规则的输出对象存储到一个流程变量中。注意，结果变量将包含一个对象列表。如果没有指定结果的变量名，默认使用 org.activiti.engine.rules.OUTPUT。

下面的业务规则任务会执行所有随流程定义一块部署的规则：

```
<processid="simpleBusinessRuleProcess">
<startEventid="theStart"/>
<sequenceFlowsourceRef="theStart"targetRef="businessRuleTask"/>
<businessRuleTaskid="businessRuleTask"activiti:ruleVariablesInput="{order}"
activiti:resultVariable="rulesOutput"/>
<sequenceFlowsourceRef="businessRuleTask"targetRef="theEnd"/>
```

```
<endEventid="theEnd"/>

</process>
```

也可以配置业务规则任务让它只执行被部署的.drl 文件中定义的一组规则。由逗号分隔开的规则名列表必须像这样来指定：

```
<businessRuleTaskid="businessRuleTask"activiti:ruleVariablesInput="{order}"
activiti:rules="rule1, rule2"/>
```

这个例子中，只有 rule1 和 rule2 会执行。

你也可以定义一个不会被执行的规则列表。

```
<businessRuleTaskid="businessRuleTask"activiti:ruleVariablesInput="{order}"
activiti:rules="rule1, rule2"exclude="true"/>
```

在这个例子中，与流程定义部署在同一个 BAR 文件中，除了 rule1 和 rule2 之外的所有流程都会被执行。

7.5.21 邮件任务

Activiti 允许利用自动的邮件服务任务，给一个或更多收件人发送包括 cc, bcc, html 内容等内容的 e-mail 来增强业务流程。注意，邮件任务并不是 BPMN 2.0 规范的“官方”任务（因而，它并没有专门的图标）。因此，在 Activiti 中邮件任务是作为特有服务任务来实现的。

Mail 服务器的配置

Activiti 引擎使用外部 SMTP 邮件服务器来发送电子邮件。实际发送电子邮件时，引擎需要知道如何连接到该邮件服务器。以下属性可以在文件 *activiti.cfg.xml* 中进行设置：

表 7.1 邮件服务器的配置

属性	是否必须？	描述
mailServerHost	否	邮件服务器的主机名（例如，mail.mycorp.com）。默认是 localhost
mailServerPort	是，如果不是默认端口	SMTP 通信端口。默认是 25
mailServerDefaultFrom	否	在不提供发件人 e-mails 时，发件人的默认 e-mail 地址。默认是 activiti@activiti.org
mailServerUsername	如果适合你的服务器	一些服务器需要证书才能发送 e-mails。默认没有设置。
mailServerPassword	如果适合你的服务器	一些服务器需要证书才能发送 e-mails。默认没有设置。

定义邮件任务

邮件任务是作为特有的[服务任务](#)来实现的，通过将服务任务的类型设置为‘mail’来定义。

```
<serviceTaskid="sendMail"activiti:type="mail">
```

邮件任务是通过[字段注入](#)配置的。所有那些属性的值都可以包含流程执行期间解析的 EL 表达式。可以设置以下属性：

表 7.2 邮件服务器的配置

属性	是否必须?	描述
to	是	e-mail 的收件人。多个收件人由逗号分隔的列表来定义。
form	否	发件人 e-mail 地址。如果没提供，使用地址的 默认配置 。
subject	否	e-mail 的主题。
cc	否	e-mail 的 cc。多个收件人由逗号分隔的列表来定义。
bcc	否	e-mail 的 bcc。多个收件人由逗号分隔的列表来定义。
charset	否	允许改变邮件的字符集，这对于很多非英语的语言是必不可少的。
html	否	e-mail 内容的 HTML 片段。
text	否	e-mail 的内容，如果需要非丰富的 e-mail。对于不支持丰富内容的 e-mail 客户端，可以结合 html 使用。客户端会选择使用纯文本。

用法举例

下面的 XML 片段展示了使用邮件任务的例子。

```
<serviceTaskid="sendMail"activiti:type="mail">
  <extensionElements>
    <activiti:fieldname="from"stringValue="order-shipping@thecompany.com"/>
    <activiti:fieldname="to"expression="{recipient}"/>
    <activiti:fieldname="subject"expression="Your order {orderId} has been shipped"/>
    <activiti:fieldname="html">
      <activiti:expression>
        <![CDATA[
          <html>
            <body>

              Hello ${male ? 'Mr.' : 'Mrs.' } ${recipientName},<br/><br/>

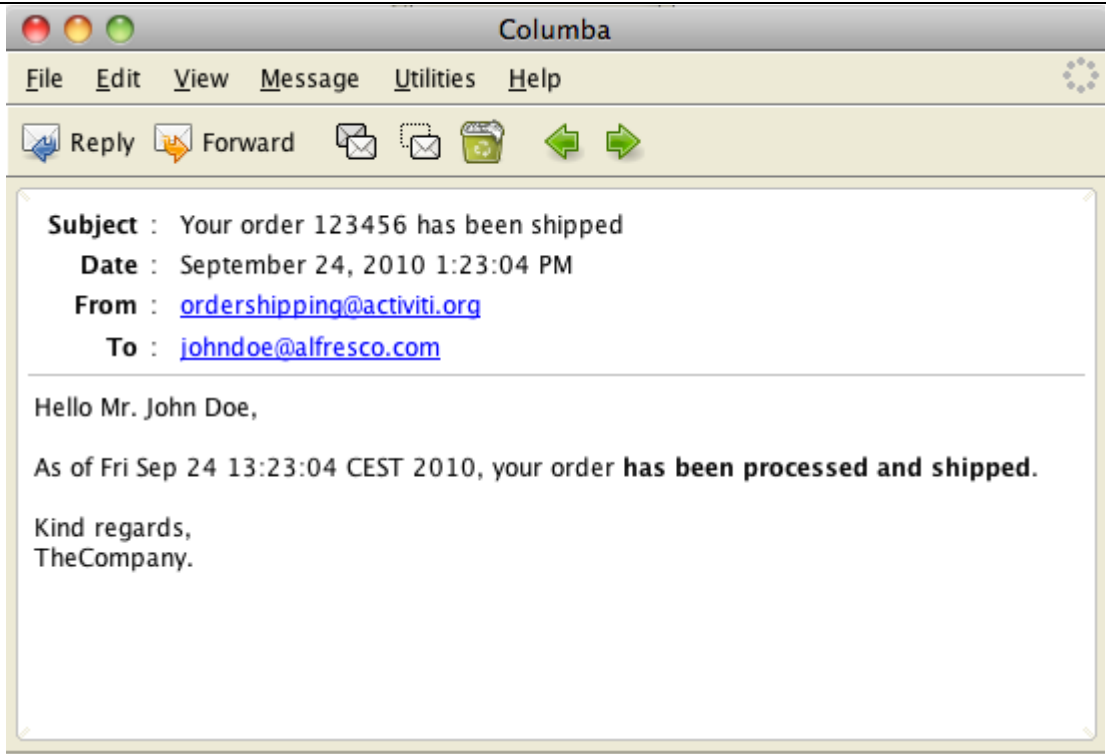
              As of ${now}, your order has been <b>processed and shipped</b>.<br/><br/>

              Kind regards,<br/>

              TheCompany.

            </body>
          </html>
        ]]>
      </activiti:expression>
    </activiti:fieldname>
  </extensionElements>
</serviceTask>
```

结果如下：



7.5.22 Mule 任务

Mule 任务允许向 Mule 发送消息以此增进了 Activiti 的集成特性。注意 Mule 任务并不是 BPMN 2.0 规范的“官方”任务（因而也没有专门的图标）。因此，在 Activiti 中 Mule 任务是作为特有服务任务来实现的。

定义 Mule 任务

Mule 任务是作为特有的服务任务来实现的，通过将服务任务的类型设置为‘mule’来定义。

```
<serviceTask id="sendMule" activiti:type="mule">
```

Mule 任务是通过字段注入配置的。所有那些属性的值都可以包含流程执行期间解析的 EL 表达式。可以设置以下属性：

表 7.3Mule 服务器的配置

属性	是否必须?	描述
endpointUrl	是	你想要调用的 Mule 终端。
language	是	你想要使用来计算 payloadExpression 字段的语言。
payloadExpression	是	An expression that will be the message's payload.
resultVariable	否	存储调用结果的变量名。

用法举例

以下 XML 片段展示了一个使用 Mule 任务的例子。

```
<extensionElements>
```



```

<activiti:fieldname="endpointUrl">
<activiti:string>vm://in</activiti:string>
</activiti:field>
<activiti:fieldname="language">
<activiti:string>juel</activiti:string>
</activiti:field>
<activiti:fieldname="payloadExpression">
<activiti:string>"hi"</activiti:string>
</activiti:field>
<activiti:fieldname="resultVariable">
<activiti:string>theVariable</activiti:string>
</activiti:field>
</extensionElements>

```

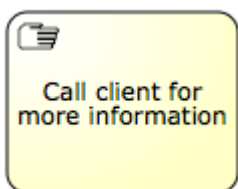
7.5.23 手动任务

描述

手动任务定义了 BPM 引擎之外的任务。用来对那些需要人来完成的工作进行建模，引擎不需要知道他是系统还是 UI 接口。对引擎而言，手动任务是作为**直接通过的活动**处理的，流程执行到此会自动继续流程的执行。

图形化符号

手动任务被形象化成左上角带有小“手”图标椭圆。



XML 表示

```
<manualTask id="myManualTask" name="Call client for more information"/>
```

7.5.24 Java 接收任务

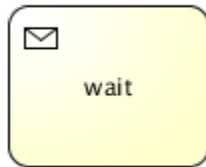
描述

接收任务是一个等待消息到来的简单任务。目前，我们仅实现了该任务的 Java 语义。当流程执行到接收任务时，流程状态被提交到持久化数据库中。这意味着，流程将进入一种等待状态，直到引擎接收到明确的消息，来触发流程通过接收

任务继续执行。

图形化符号

接收任务被形象化成左上角带有消息图标的任务（椭圆）。消息是白色的（黑色的消息图标表示已经发送的意思）。



XML 表示

```
<receiveTask id="waitState" name="wait"/>
```

要继续当前等待在这样一个接收任务的流程实例，需要调用使用了执行到此接收任务的执行路径的 id 的方法 `runtimeService.signal(executionId)`。下面的代码片段展示实际是如何操作的：

```
ProcessInstance pi = runtimeService.startProcessInstanceByKey("receiveTask");
Execution execution = runtimeService.createExecutionQuery()
    .processInstanceId(pi.getId())
    .activityId("wait")
    .singleResult();
assertNotNull(execution);

runtimeService.signal(execution.getId());
```

7.5.25 执行监听器

兼容性提示：5.3 发布后，我们发现执行监听器、任务监听器、以及表达式仍然是非公开的 api。这些类在 `org.activiti.engine.impl...` 的子包（含有 `impl`）内。`org.activiti.engine.impl.pvm.delegate.ExecutionListener`、`org.activiti.engine.impl.pvm.delegate.TaskListener`、以及 `org.activiti.engine.impl.pvm.el.Expression` 已经被废弃了。从现在开始，你应该使用 `org.activiti.engine.delegate.ExecutionListener`、`org.activiti.engine.delegate.TaskListener`、以及 `org.activiti.engine.delegate.Expression` 了。在新公开可用的 API 中，不能再访问 `ExecutionListenerExecution.getEventSource()` 了。除了编译器的警告外，现有的代码应该可以很好地运行。可以考虑转向新公布的 API 接口（包名中不包含 `impl`）。

执行监听器允许你在流程执行期间发生某些事件时执行外部 Java 代码或计算表达式。可捕获的事件如下：

- 流程实例的启动和结束。
- 迁移（Taking a transition 译注，在 UML 中是由一个状态转换为另一种状态这一过程称为‘迁移’，故在此沿用）
- 活动的开始和结束

下面的流程定义包含 3 个执行监听器：

```

<processid="executionListenersProcess">

<extensionElements>
<activiti:executionListenerclass="org.activiti.examples.bpmn.executionlistener.ExampleExecutionListenerOne"event="start"/>
</extensionElements>

<startEventid="theStart"/>
<sequenceFlowsourceRef="theStart"targetRef="firstTask"/>

<userTaskid="firstTask"/>
<sequenceFlowsourceRef="firstTask"targetRef="secondTask">
<extensionElements>
<activiti:executionListenerclass="org.activiti.examples.bpmn.executionlistener.ExampleExecutionListenerTwo"/>
</extensionElements>
</sequenceFlow>

<userTaskid="secondTask">
<extensionElements>
<activiti:executionListenerexpression="{myPojo.myMethod(execution.event)}"event="end"/>
</extensionElements>
</userTask>
<sequenceFlowsourceRef="secondTask"targetRef="thirdTask"/>

<userTaskid="thirdTask"/>
<sequenceFlowsourceRef="thirdTask"targetRef="theEnd"/>

<endEventid="theEnd"/>

</process>

```

第一个执行监听器在流程启动时被通知。监听器是个外部的 Java 类（如 ExampleExecutionListenerOne），且必须实现了 org.activiti.engine.delegate.ExecutionListener 接口。当事件发生时（该例子中是结束事件），调用方法 notify(ExecutionListenerExecution execution)。

```

publicclass ExampleExecutionListenerOne implementsExecutionListener {

    publicvoid notify(ExecutionListenerExecution execution) throws Exception {
        execution.setVariable("variableSetInExecutionListener", "firstValue");
        execution.setVariable("eventReceived", execution.getEventName());
    }
}

```

也可以使用实现了 org.activiti.engine.delegate.JavaDelegate 接口的代理类。这些代理类接下来还可以在其它构造中使用，比如 serviceTask 的代理。

第二个执行监听器在迁移发生时被调用。注意该监听器元素没有定义事件，因为迁移上只触发选择事件。忽略定义在迁

移上的监听器的 **event** 属性值。

最后一个执行监听器在活动 `secondTask` 结束后被调用。监听器声明中没有使用类，而是定义了一个表达式，其在触发事件时会被计算/调用。

```
<activiti:executionListener expression="${myPojo.myMethod(execution.eventName)}" event="end"/>
```

就像其它表达式，在此也可以使用 `execution` 变量。因为 `execution` 实现类的对象有一个表示事件名的属性，所以可以使用 `execution.eventName` 向你的方法中传递事件名。

执行监听器也支持使用 `delegateExpression`，类似[服务任务](#)。

```
<activiti:executionListener event="start" delegateExpression="${myExecutionListenerBean}"/>
```

执行监听器上的字段注入

在使用由 `class` 属性配置的执行监听器时，可以应用字段注入。这与[服务任务的字段注入](#)（其对字段注入做了概述）是同一机制。

下面的片段显示了使用字段注入的执行监听器的简单示例流程。

```
<process id="executionListenersProcess">
  <extensionElements>
    <activiti:executionListener class="org.activiti.examples.bpmn.executionListener.ExampleFieldInjectedExecutionListener" event="start">
      <activiti:field name="fixedValue" stringValue="Yes, I am "/>
      <activiti:field name="dynamicValue" expression="${myVar}"/>
    </activiti:executionListener>
  </extensionElements>

  <startEvent id="theStart"/>
  <sequenceFlow sourceRef="theStart" targetRef="firstTask"/>

  <userTask id="firstTask"/>
  <sequenceFlow sourceRef="firstTask" targetRef="theEnd"/>

  <endEvent id="theEnd"/>
</process>
```

```
public class ExampleFieldInjectedExecutionListener implements ExecutionListener {

    private Expression fixedValue;

    private Expression dynamicValue;

    public void notify(ExecutionListenerExecution execution) throws Exception {
```

```

        execution.setVariable("var", fixedValue.getValue(execution).toString() + dynamicValue.getValue(execution).toString());
    }
}

```

类 `ExampleFieldInjectedExecutionListener` 连结了 2 个被注入的字段（一个是固定的，一个是动态的），并将其存储到流程变量 “var” 中。

```

@Deployment(resources =
{"org/activiti/examples/bpmn/executionListener/ExecutionListenersFieldInjectionProcess.bpmn20.xml"})
public void testExecutionListenerFieldInjection() {
    Map<String, Object> variables = new HashMap<String, Object>();
    variables.put("myVar", "listening!");

    ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("executionListenersProcess", variables);

    Object varSetByListener = runtimeService.getVariable(processInstance.getId(), "var");
    assertNotNull(varSetByListener);
    assertTrue(varSetByListener instanceof String);

    // 结果是由固定的注入字段和注入表达式联合的结果
    assertEquals("Yes, I am listening!", varSetByListener);
}

```

7.5.26 任务监听器

任务监听器用来执行自定义的 Java 逻辑或任务事件相关的表达式。

任务监听器只能作为[用户任务](#)的子元素添加到流程定义中。注意，任务监听器必须作为 *BPMN 2.0* 元素 *extensionElements* 的子元素，并且必须在 *activiti* 的命名空间，因为任务监听器是 *Activiti* 特有的构造。

```

<userTask id="myTask" name="My Task">
  <extensionElements>
    <activiti:taskListener event="create" class="org.activiti.MyTaskCreateListener"/>
  </extensionElements>
</userTask>

```

任务监听器支持以下属性：

- **event**（必须的）：触发调用任务监听器的任务事件的类型。可能的事件有
 - **create**：在任务被创建，且所有的任务属性被设置后发生。
 - **assignment**：在任务分配给某人后发生。注意：当流程执行到 *userTask*，在触发 *create* 事件之前，首先触发 *assignment* 事件。这像是一个反常的顺序，但原因很实际：接收到 *create* 事件，我们往往会要查看包括代理人在内的所有任务的属性。
 - **complete**：在任务完成后，任务从运行时的数据中被删除之前发生。
- **class**：被调用的代理类。该类必须实现 `org.activiti.engine.impl.pvm.delegate.TaskListener` 接口。

```

public class MyTaskCreateListener implements TaskListener {

```

```
public void notify(DelegateTask delegateTask) {
    // 自定义逻辑于此
}
}
```

也可以使用[字段注入](#)向代理类中传递流程变量或 `execution`。注意，代理类的实例是在流程部署时创建的（就像 Activiti 中其它类的代理一样），这意味着所有流程实例的执行共享这一实例。

- **expression**（不能与 `class` 属性一块使用）：指定事件发生时执行的表达式。可以将 `DelegateTask` 对象以及事件名（使用 `task.eventName`）作为参数传递给被调用对象。

```
<activiti:taskListener event="create" expression="${myObject.callMethod(task, task.eventName)}"/>
```

- **delegateExpression**：允许指定一个值为实现了 `TaskListener` 接口的对象的表达式，类似于[服务任务](#)。

```
<activiti:taskListener event="create" delegateExpression="${myTaskListenerBean}"/>
```

7.5.27 多实例（for each）

描述

多实例活动是对业务流程中某个步骤定义重复的一个途径。在程序设计的概念中，多实例是**每一个**意思：它允许对于给定集合中的每一项都**顺序地或并行地**执行某步操作或者甚至执行完整的子流程。

多实例是个常规活动，它定义了些额外的属性（所谓的‘**多实例特性**’）使活动在运行时被执行多次。以下活动可以成为**多实例的活动**：

- [用户任务](#)
- [脚本任务](#)
- [Java 服务任务](#)
- [Web 服务任务](#)
- [业务规则任务](#)
- [邮件任务](#)
- [手动任务](#)
- [接收任务](#)
- [（嵌入的）子流程](#)
- [调用活动](#)

分支、事件不能是多实例的。

按规范要求的，每个被创建出来的执行路径实例的父执行路径都有以下变量：

- **nrOfInstances**：实例总数。
- **nrOfActiveInstances**：当前活跃的，也就是还没完成的，实例的个数。对于顺序的多实例，该值总是 1。
- **nrOfCompleteInstances**：已经完成的实例的个数。

通过调用方法 `execution.getVariable(x)` 可以获得这些值。

此外，每个被创建出来的执行路径都有一个执行路径本地变量（即，对其它执行路径是不可见的，没有存储在流程实例一级）：

- **loopCounter**：表示 for-each 循环中实例的索引。

图形化符号

如果一个活动是多实例的，这可以用活动底部的三条短线来表示。3 条垂直线表示并行地执行这些实例，而 3 条水平线表示顺序地执行这些实例。



XML 表示

要构造一个多实例的活动，活动的 xml 元素需要包含 `multiInstanceLoopCharacteristics` 子元素。

```
<multiInstanceLoopCharacteristics isSequential="false|true">
...
</multiInstanceLoopCharacteristics>
```

isSequential 属性表示该活动的实例是按顺序执行还是按并行执行。

一旦进入该活动，就计算实例的个数。此值有几种配置方式。其一是使用 **loopCardinality** 子元素直接指定个数。

```
<multiInstanceLoopCharacteristics isSequential="false|true">
<loopCardinality>5</loopCardinality>
</multiInstanceLoopCharacteristics>
```

也可以使用结果为正数的表示式。

```
<multiInstanceLoopCharacteristics isSequential="false|true">
<loopCardinality>${nrOfOrders-nrOfCancellations}</loopCardinality>
</multiInstanceLoopCharacteristics>
```

另一种定义实例个数的方式是指定一个流程变量的名称，它是一个使用了 **loopDataInputRef** 子元素的集合。对于集合中的每一项都会有一个实例被创建。作为一个选择，可以使用 **inputDataItem** 子元素将集合中某项设置给相应的执行实例。如下面 XML 例子所展示的：

```
<userTask id="miTasks" name="My Task ${loopCounter}" activiti:assignee="${assignee}">
<multiInstanceLoopCharacteristics isSequential="false">
<loopDataInputRef>assigneeList</loopDataInputRef>
<inputDataItem name="assignee"/>
</multiInstanceLoopCharacteristics>
</userTask>
```

假设变量 `assigneeList` 包含值 `[kermit, gonzo, foziee]`。在上面的片段中，会并行地创建 3 个用户任务。每个执行路径都会拥

有一个叫 **assignee** 的流程变量，它含有集合中的一个值，这个例子中该值用来分配用户任务。

loopDataInputRef 和 inputDataItem 的缺点是：1)名称难记。2)由于 BPMN 2.0 模式的限制，它们不能包含表达式。Activiti 为 multiInstanceCharacteristics 提供了属性 **collection** 以及 **elementVariable** 来解决这个问题：

```
<userTask id="miTasks" name="My Task" activiti:assignee="{assignee}">
  <multiInstanceLoopCharacteristics isSequential="true"
  activiti:collection="{myService.resolveUsersForTask()}" activiti:elementVariable="assignee">
  </multiInstanceLoopCharacteristics>
</userTask>
```

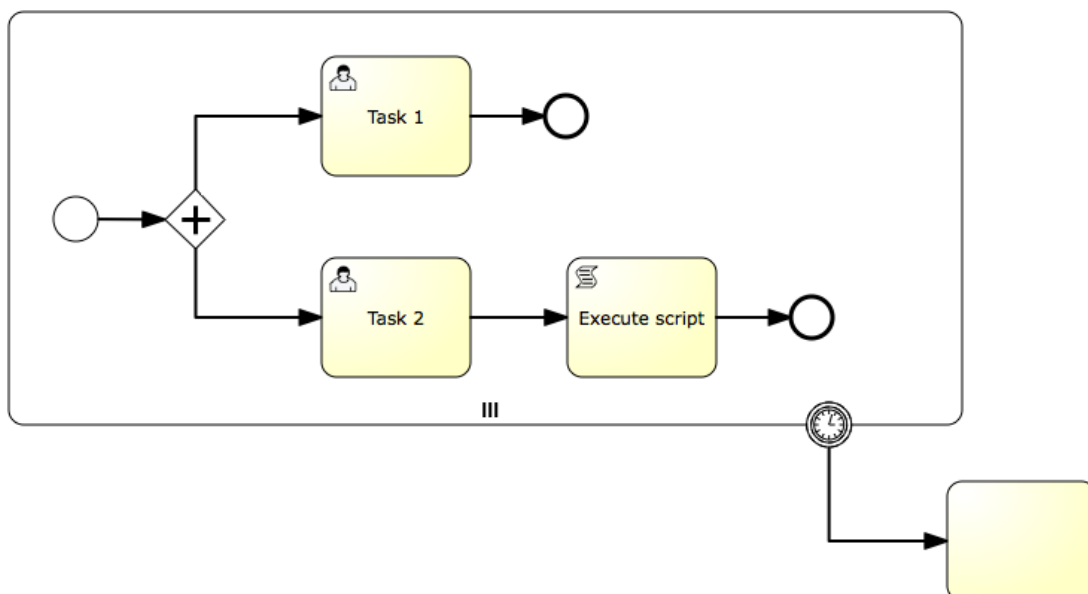
当所有实例完成后，多实例活动就结束了。然而，可以指定一个表达式在每次实例结束时就对其进行计算。当该表达式值为 true 时，就会销毁所有剩余的实例，并结束该多实例活动，向下继续执行流程。该表达式必须定义在 **completionCondition** 子元素内。

```
<userTask id="miTasks" name="My Task" activiti:assignee="{assignee}">
  <multiInstanceLoopCharacteristics isSequential="false"
  activiti:collection="assigneeList" activiti:elementVariable="assignee">
    <completionCondition>${nrOfCompletedInstances/nrOfInstances >= 0.6 }</completionCondition>
  </multiInstanceLoopCharacteristics>
</userTask>
```

在这个示例中，会为集合 assigneeList 中的每个元素创建一个并行的实例。然而，在 60%的任务完成时，会删除其它任务，流程继续向下执行。

边界事件与多实例

因为多实例是普通的活动，所以可以在它的边界定义[边界事件](#)。如果捕获边界事件被中断了，那么所有活跃着的实例都将被销毁。拿以下多实例子流程举例：



这里，当触发定时器时，将销毁所有子流程实例，不管存在多少实例或者是否有哪些内部活动还没完成。

7.5.28 边界事件

边界事件是关联在活动上的 *捕获型* 的事件（边界事件从不抛出）。这意味着在活动运行时，该事件会一直 *监听* 着某一类型的事件触发。当 *捕获* 到事件时，活动被打断，沿事件输出顺序流继续执行。

所有的边界事件都以同样的方式进行定义：

```
<boundaryEvent id="myBoundaryEvent" attachedToRef="theActivity">
  <XXXEventDefinition/>
</boundaryEvent>
```

定义边界事件要有

- 唯一的标识符（流程全局性的）
- 通过 **attachedToRef** 属性所定义的该事件所依附到的活动的引用。注意，边界事件定义的层次与它们关联的活动在同一层（也就是说，活动内部不包含边界事件）。
- 一个 *XXXEventDefinition* 格式的 XML 子元素（如，*TimerEventDefinition*、*ErrorEventDefinition* 等等）定义了边界事件的类型。更多详细介绍，见指定边界事件的类型。

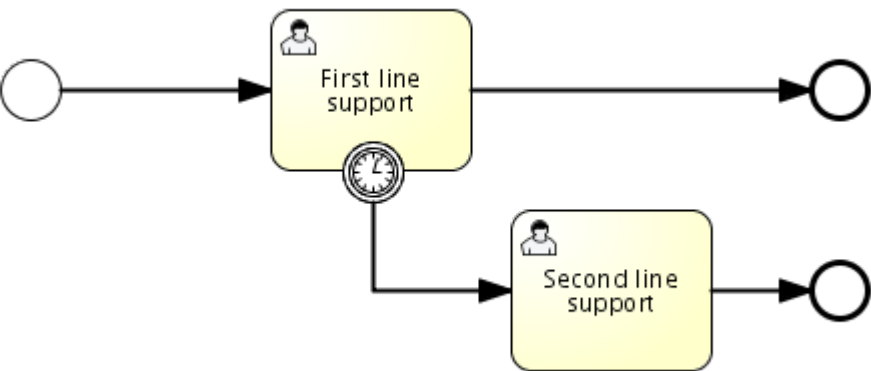
7.5.29 定时器边界事件

描述

定时器事件就像是跑表和警钟。当执行到边界事件关联到的活动时，定时器被启动。当定时器触发时（也就是说，间隔一段时间后），活动被打断，沿着定时器边界事件输出顺序流继续执行。

图形化符号

定时器边界事件被形象化为内部带有定时器跑表图标的典型边界事件（即，边框上的圆形）。



XML 表示

定时器边界事件是作为 [普通边界事件](#) 来定义的。其特有的类型子元素在这个例子中是 **timerEventDefinition** 元素。

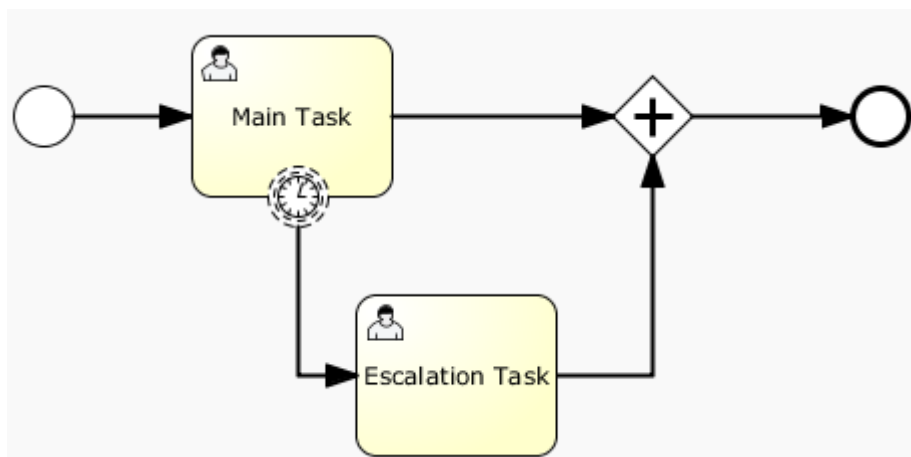
```

<boundaryEventid="escalationTimer"cancelActivity="true"attachedToRef="firstLineSupport">
<timerEventDefinition>
<timeDuration>PT4H</timeDuration>
</timerEventDefinition>
</boundaryEvent>

```

关于定时器更详细的配置信息，请参考[定义定时器事件](#)。

图形表示中，如你在这个例子所看到的，圆的边线是虚线：

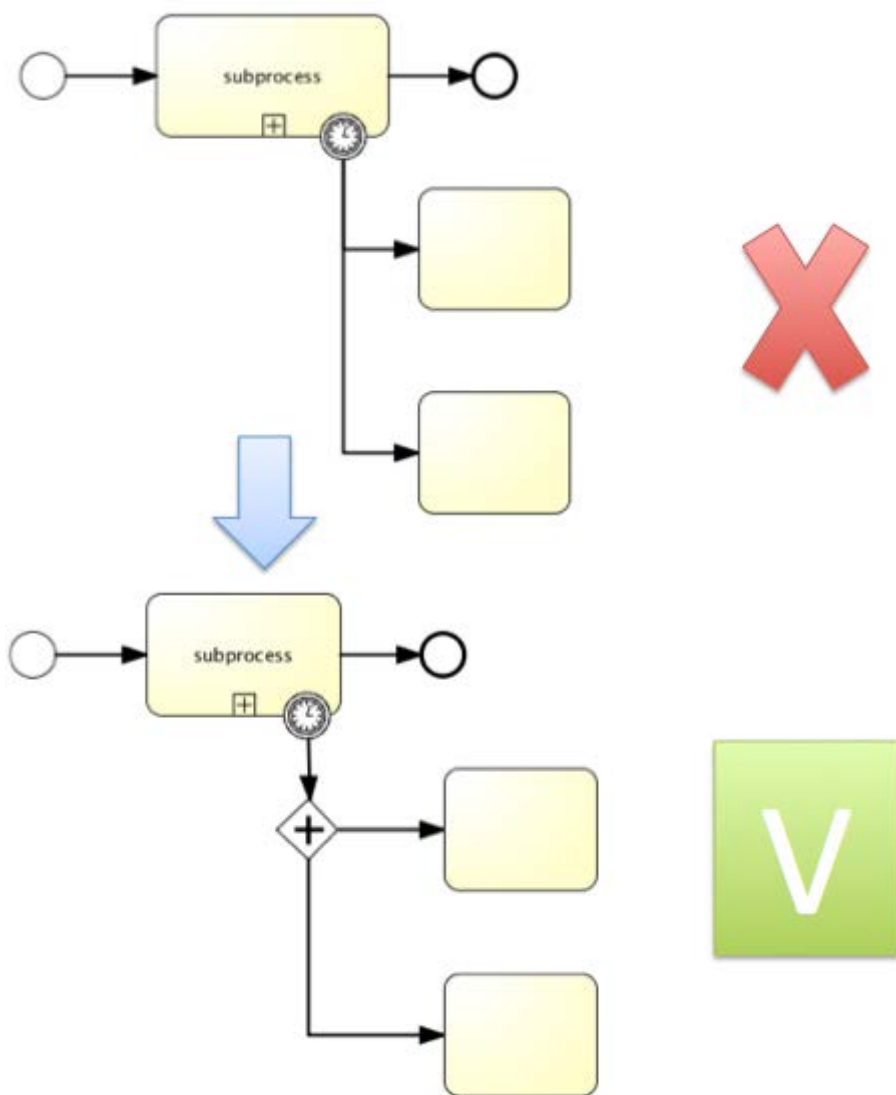


一个典型的用法是另外发送升级的电子邮件，而不是打断正常的流程执行流。

注意：边界定时器事件只有当开启作业执行器时才能触发(即，需要在 `activiti.cfg.xml` 中将 `jobExecutorActivate` 设置为 `true`，因为默认是禁用作业执行器的)。

使用边界事件的已知问题

使用任何类型的边界事件时，都存在并发性的问题。目前，还不能将多个输出顺序流连接到一个边界事件上（见问题[ACT-47](#)）。该问题的解决方法是使用流向并发 gateway 的输出流。



7.5.30 异常边界事件

描述

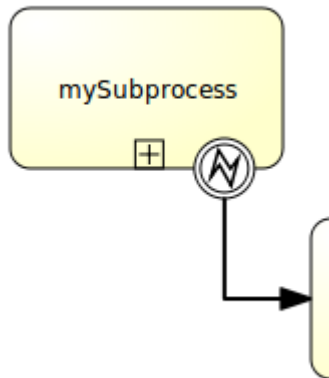
捕获活动边界上异常的媒介，或简称**边界异常事件**，它捕获定义它的那个活动范围内抛出的异常。

在[嵌入的子流程](#)、或[调用活动](#)上定义边界异常是很容易理解的，因为子流程为其内的所有活动创建了一个作用域。异常是由[异常结束事件](#)抛出的。这样的错误信息会一直向上传播给父作用域，直到某个作用域中定义的边界事件与该异常事件定义匹配为止。

当捕获到异常事件后，定义边界事件的活动就会被销毁，同时也销毁其内所有的执行路径（比如，并发的活动、嵌套的子流程，等等）。流程沿着边界事件的输出流继续执行。

图形化符号

边界异常事件被形象化为边沿上内部带有异常图标（圆圈内套小圆圈）。异常图标是白色的，表示捕获的含义。



XML 表示

边界异常事件是作为典型的[边界事件](#)来定义的：

```
<boundaryEvent id="catchError" attachedToRef="mySubProcess">
  <errorEventDefinition errorRef="myError"/>
</boundaryEvent>
```

就像[异常终止事件](#)，`errorRef` 引用了定义在流程元素之外的 `error`。

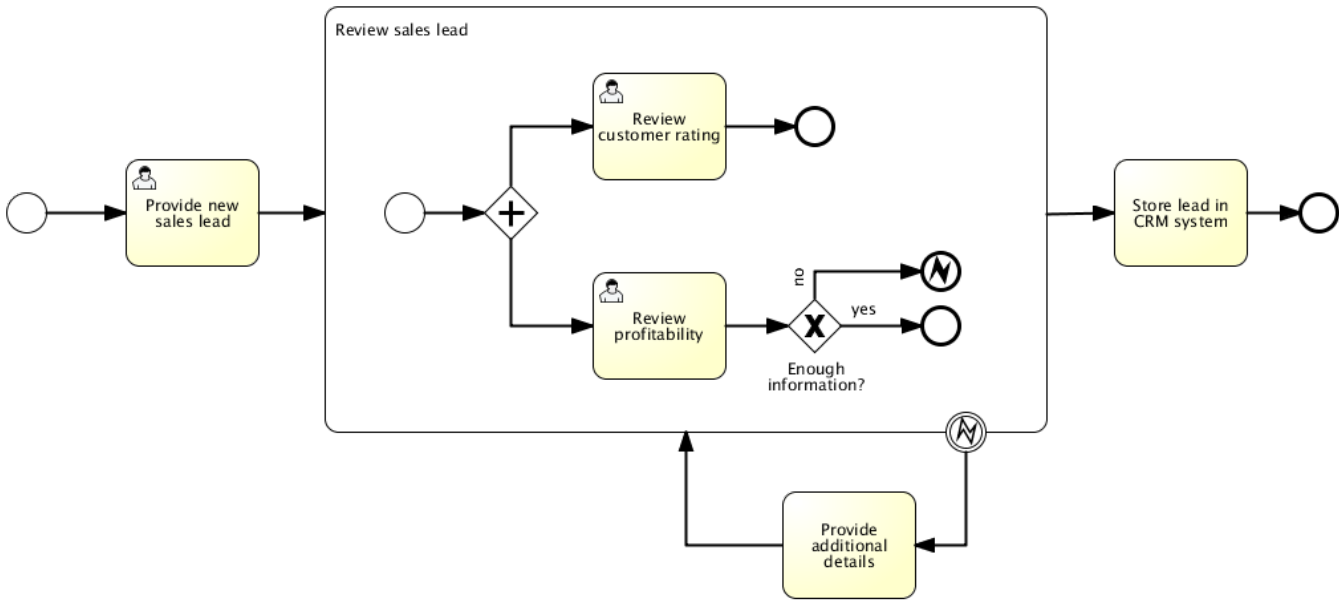
```
<error id="myError" errorCode="123"/>
...
<process id="myProcess">
...
```

`errorCode` 用来匹配捕获到的 `errors`：

- 如果省略 `errorRef`，边界 `error` 事件将捕获任何 `error` 事件，不管该 `error` 的 `errorCode` 是多少。
- 如果提供了 `errorRef`，它指向了一个现有的 `error`，那么此边界事件将只捕获这一出错码的 `errors`。
- 如果提供了 `errorRef`，但没有在 BPMN 2.0 文件中定义任何 `error`，那么 `errorRef` 做为 `errorCode` 来使用（类似于 `error end` 事件）。

示例

下面流程的例子展示了如何使用 `error end` 事件。如果用户任务‘*Review profitability*’是以提供的信息不充分而结束的，就会抛出 `error`。当在子流程的边界上捕获到此 `error`，‘*Review sales lead*’子流程内的所有活跃着的活动都将被销毁（即使‘*Review customer rating*’还没有完成），接着‘*Provide additional details*’用户任务会被创建。



这是 demo setup 中的例子中的一个流程。可以在 `org.activiti.examples.bpmn.event.error` 包中找到流程的 XML 以及单元测试。

7.5.31 中间捕获事件

所有中间捕获事件都是以同样方式来定义的：

```
<intermediateCatchEvent id="myIntermediateCatchEvent">
  <XXXEventDefinition/>
</intermediateCatchEvent>
```

定义中间捕获事件要有

- 一个唯一标识符（流程范围内的）
- 定义了中间捕获事件类型的一个 XXXEventDefinition 格式的 XML 子元素（如，`TimerEventDefinition`，等等）。更多详细介绍，见指定捕获事件的类型。

7.5.32 定时器中间捕获事件

描述

定时器中间事件就像个跑表。流程执行到捕获事件的活动时，定时器被启动。定时器被触发后（也就是说，间隔一段时间后），会沿着定义器中间事件的输出顺序流执行。

图形化符号

定时器中间事件被形象化成内部带有定时器图标的中间捕获事件。



XML 表示

定时器中间媒介事件是作为[中间捕获事件](#)来定义的。其特有的类型子元素是元素 **timerEventDefinition**。

```
<intermediateCatchEvent id="timer">
  <timerEventDefinition>
    <timeDuration>PT5M</timeDuration>
  </timerEventDefinition>
</intermediateCatchEvent>
```

更详细的配置，见[定时器事件的定义](#)

7.5.33 子流程

描述

*子流程*是一个可以包含其它活动、分支、事件，等的活动。它本身的流程作为更大流程的一部分。*子流程*完全定义在父流程内（这就是为什么常常称之为*嵌入的子流程*）。

子流程存在两种主要的用例：

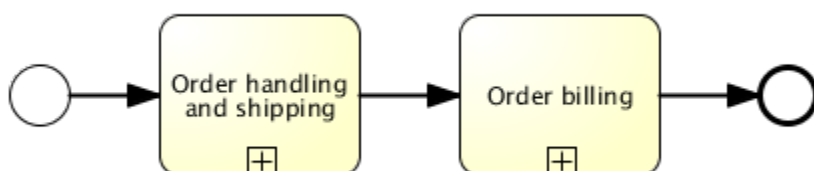
- 子流程允许进行**分层建模**。很多建模工具都允许将子流程进行*折叠*，来隐藏子流程的所有细节，以展示高层次的互相衔接的业务流程概况图。
- 子流程**为事件创建了新的作用域**。子流程执行期间所抛出的事件可以被子流程边界上的[边界事件](#)捕获，因此对于事件所创建的作用域只局限在子流程内。

使用子流程不会强加任何限制：

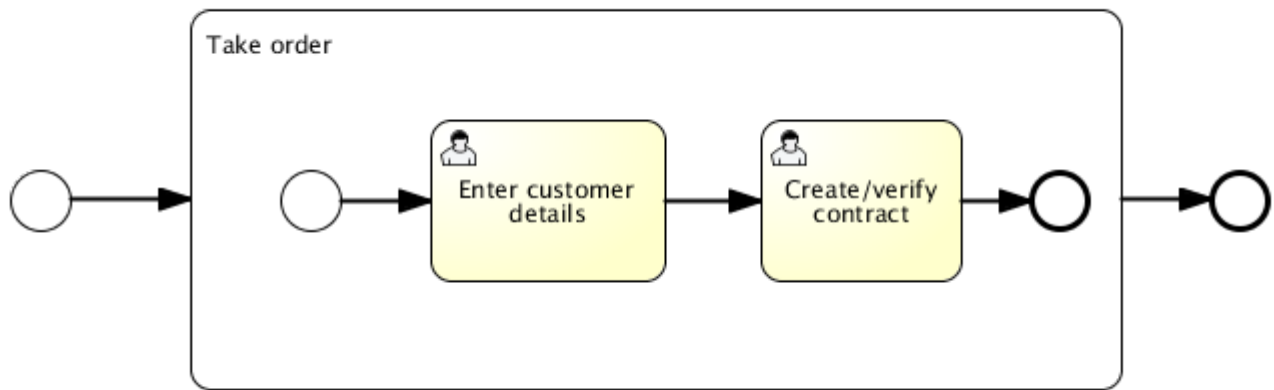
- 子流程只能包含一个 **none start 事件**，不允许有其它的 start event 类型。子流程内**至少有一个 end 事件**。注意，BPMN 2.0 规范允许在子流程中省略启动事件和结束事件，但是目前 Activiti 的实现不支持这样做。
- **顺序流不能跨子流程边界**。

图形化符号

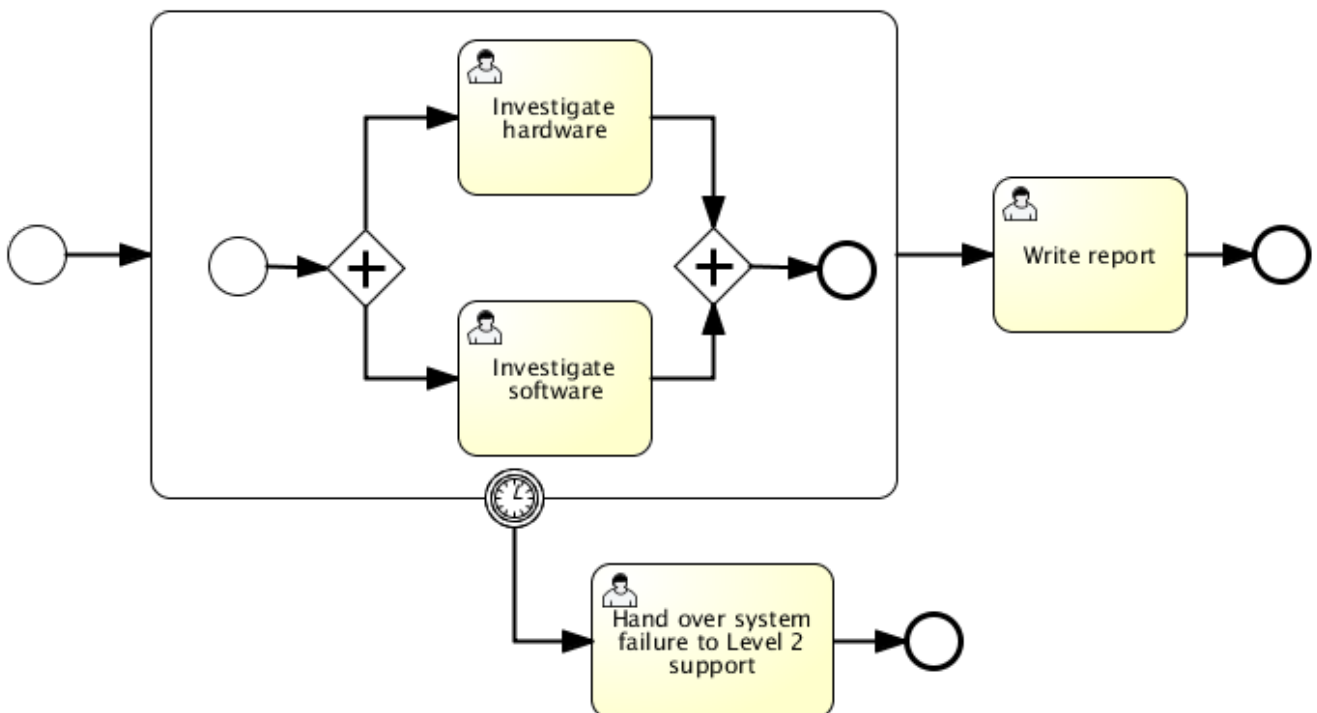
子流程被形象成一个典型的活动，即一个椭圆形。如果子流程被*折叠*起来，那么只显示名称和加号，以提供流程的一个高层次的视图：



如果**展开**子流程，那么子流程的所有步骤都将显示在子流程边界内：



使用子流程的一个主要的原因是为某个事件定义作用域。下面的流程模型显示了：*软件调查/硬件调查*这两个任务需要并行进行，但要在 *Level 2 support* 查阅前，处理完这两个任务。这里，定时器（即，活动必须在此前按时完成）的作用域由子流程来限制。



XML 表示

子流程是由 *subprocess* 元素定义的。所有属于子流程的活动、分支、事件等都要封装在这个元素内。

```
<subprocessid="subProcess">
```

```
<startEventid="subProcessStart"/>
```

... 其它子流程的元素 ...

```
<endEvent id="subProcessEnd"/>

</subProcess>
```

7.5.34 调用活动（子过程）

描述

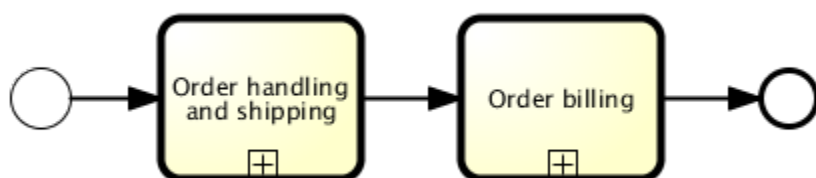
BPMN 2.0 对普通的[子流程](#)（通常称之为嵌入的子流程）和调用活动（两者非常相似）进行了区分。从概念的角度来看，两者都会在流程执行到此活动时调用子过程。

不同的是调用活动引用的是流程定义之外的过程，而[子流程](#)是嵌入在原始流程定义内的。调用活动的主要用例是创建可供其它多个流程定义调用的可重用的过程定义。

当流程执行到[调用活动](#)时，为执行到调用活动的执行路径创建一个新的子执行路径。该子执行路径接下来用于执行这个子过程，就像在普通流程中一样，子过程内可能也会创建并行的子执行路径。高层的执行路径会等待子流程完全结束后，然后接着之前的流程继续向下执行。

图形化符号

调用活动被形象化成[子流程](#)，但带有粗边框（折叠、展开）。取决于建模工具，调用活动也可以被展开，但是默认用折叠的子过程表示。



XML 表示

调用活动是个普通活动，使用的 *calledElement* 属性，它是通过流程定义的 **key** 来引用流程定义。实际上，这指的是 *calledElement* 使用的是流程的 id。

```
<callActivity id="callCheckCreditProcess" name="Check credit" calledElement="checkCreditProcess"/>
```

注意，子过程的流程定义是**在运行时解析的**。这意味着如果需要的话，子过程可以脱离调用过程单独部署。

传递变量

你可以将流程变量传递给子过程，反过来也一样。在子过程启动时，这些数据被拷贝到子过程，当子过程结束时，将这些数据又拷贝回主流程。


```

<callActivityid="callSubProcess"calledElement="checkCreditProcess">
<extensionElements>
<activiti:insource="someVariableInMainProcess"target="nameOfVariableInSubProcess"/>
<activiti:outsources="someVariableInSubProcess"target="nameOfVariableInMainProcess"/>
</extensionElements>
</callActivity>

```

我们使用 **Activiti** 的扩展来简写 BPMN 标准元素中的 *dataInputAssociation* 和 *dataOutputAssociation*，只有按照 BPMN 2.0 的标准形式声明流程变量时它们才能有效。

此处也可以使用表达式：

```

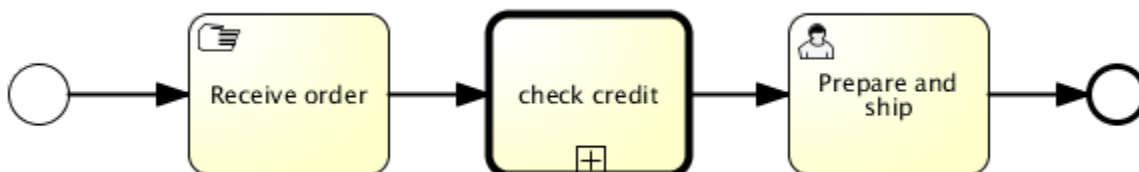
<callActivityid="callSubProcess"calledElement="checkCreditProcess">
<extensionElements>
<activiti:insourceExpression="{x+5}"target="y"/>
<activiti:outsources="{y+5}"target="z"/>
</extensionElements>
</callActivity>

```

结果为 $z = y + 5 = x + 5 + 5$

示例

下面的流程图展示了一个简单的订单处理。由于检查顾客信誉对于很多其它流程可能是公共的，所以 *check credit* 这一步在此被建模成一个调用活动。



流程如下：

```

<startEventid="theStart"/>
<sequenceFlowid="flow1"sourceRef="theStart"targetRef="receiveOrder"/>

<manualTaskid="receiveOrder"name="Receive Order"/>
<sequenceFlowid="flow2"sourceRef="receiveOrder"targetRef="callCheckCreditProcess"/>

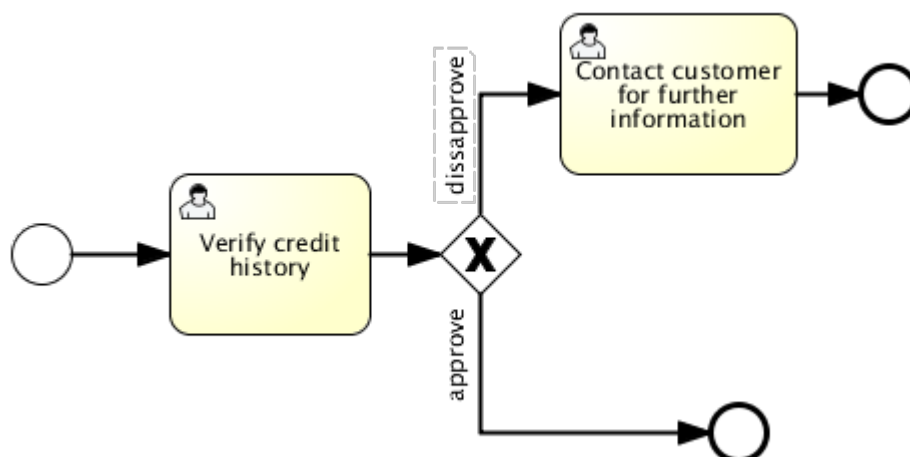
<callActivityid="callCheckCreditProcess"name="Check credit"calledElement="checkCreditProcess"/>
<sequenceFlowid="flow3"sourceRef="callCheckCreditProcess"targetRef="prepareAndShipTask"/>

<userTaskid="prepareAndShipTask"name="Prepare and Ship"/>
<sequenceFlowid="flow4"sourceRef="prepareAndShipTask"targetRef="end"/>

<endEventid="end"/>

```

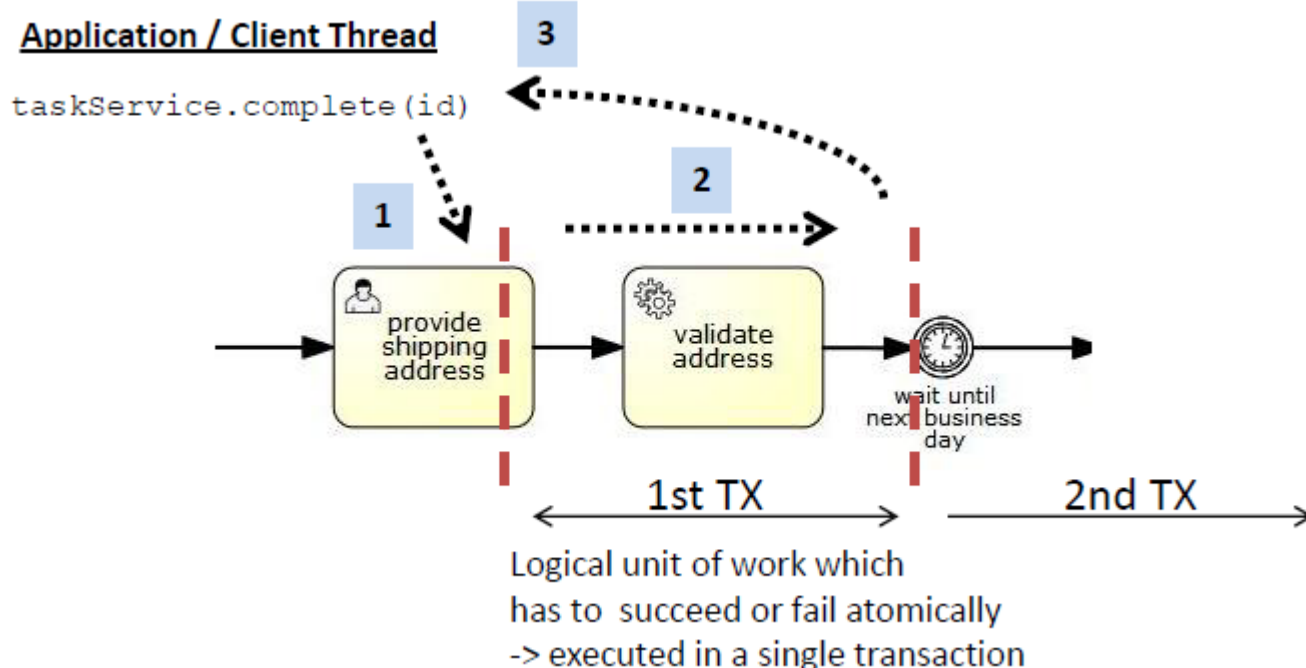
子过程如下：



子过程的流程定义没有什么特别的。它也可以不经流程调用而使用。

7.6 异步的延续

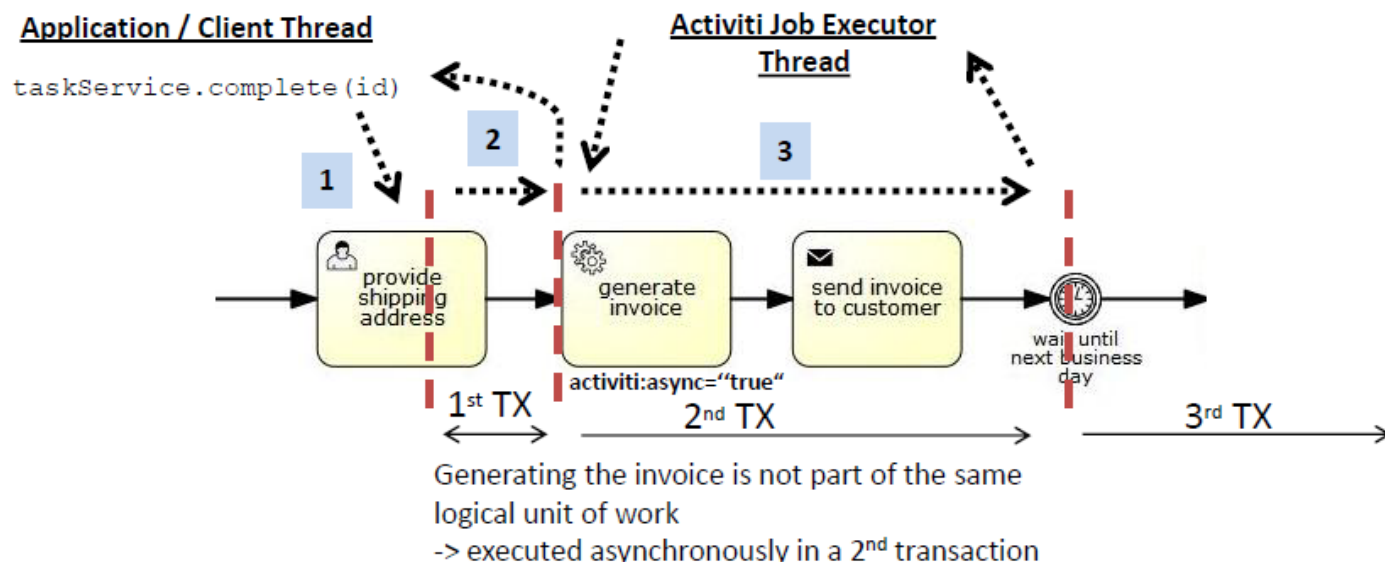
Activiti 以事务的方式执行流程，这种事务方式可以被配置以符合你的需要。让我们首先来看看 Activiti 通常是怎么界定事务的。如果触发 Activiti（例如，启动一个流程、完成一个任务、信号通知执行路径），Activiti 马上就会在流程中推进，直到它到达每条活跃的执行路径的等待状态。更具体地讲，它会在流程图内执行深度优先的搜索，如果它已经到了每条执行路径分支上等待状态就会返回。等待状态是一个“稍后”被执行的任務，这意味着 Activiti 持续了当前执行，等待着再次被触发。触发事件可以来自外部资源，例如我们有一个用户任务或一个接收消息的任务，或者来自 Activiti 自身，例如如果我们有一个定时器事件。图例如下图：



我们看到的是一个有用户任务、服务任务和定时器事件的 BPMN 流程的一部分。完成用户任务和校验地址都是一个工作单元的部分，所以该工作单元要么一次性成功要么失败。这意味着如果服务任务抛出了异常，我们是要回滚当前事务的，这样的话执行将回退到用户任务，并且用户任务将仍然在数据库中存在。这也是 Activiti 的默认行为。在(1)应用程序或客

户端线程完成了任务。在同一线程内 **Activiti** 此时要执行服务，并继续推进直到到达等待状态，此例中为定时器事件(2)。然后将控制返回给潜在提交该事务（如果是由 **Activiti** 启动的）的调用方(3)。

在有些情况下，这并非我们想要的。有时我们需要自定义对流程中事务边界的控制，从而达到对逻辑工作单元进行界定。这时异步延续就起作用了。考虑以下流程（部分）：



这一次我们正在完成用户任务、生成发货单然后将发货单发送给顾客。这一次发货单的生成不是工作单元的一部分了，所以如果生成发货单失败了我们也不想回滚完成了的用户任务。所以我们想要 **Activiti** 做的是完成用户任务(1)、提交事务，然后将控制返回给调用的应用程序。接下来我们想要以后台线程来异步生成发货单。这个后台线程就是 **Activiti** 作业执行器（实际上是个线程池），它会定期轮询数据库以查询作业。所以在这个场景的后面是，当我们到达“generate invoice”任务时，我们为 **Activiti** 能稍后继续该流程创建了一个作业“message”，并将它存入数据库。这项作业接下来会被作业执行器拿到，然后执行。我们也会提示本地作业执行器有新的作业了以提高效率。

要使用这一特性，我们可以使用 `activiti:async="true"` 扩展。所以对于示例来讲，服务任务如此：

```
<serviceTask id="service1" name="Generate Invoice" activiti:class="my.custom.Delegate" activiti:async="true" />
```

`activiti:async` 可以指定在以下 bpmn 的任务类型：任务、服务任务、脚本任务、业务规则任务、发送任务、接收任务、用户任务、子流程以及调用活动。

在用户任务、接收任务或者其它等待状态，异步延续允许我们执行以独立线程/事务来执行启动执行监听器。

第八章、表单

Activiti 提供了既方便又灵活的方式来给业务流程中的手工步骤添加表单。我们提供了两种处理表单的方案：使用表单属性的内置的表单渲染和外部表单渲染。

8.1 表单属性

业务流程相关的所有信息或者是在流程变量自身内，或者是通过流程变量引用的。Activiti 支持存储复杂 Java 对象的流程变量，如序列化的对象、JPA 实体、或字符串表示的 XML 文档。

人员是在启动流程和完成用户任务时参与进流程的。表单需要使用某种 UI 技术被渲染后才能完成与人的通讯。流程定义中可以包含流程变量中复杂 Java 类型的对象到一个**属性**的 Map<String, String>值的转换逻辑（译注，所有属性都被放置在一个 Map 内，Map 中的一个值就代表一个属性）。

接下来，利用 Activiti API 中揭露这一属性信息的方法，UI 技术就可以在这些属性之上构建表单了。属性为流程变量提供了专门（且更局限）的视图。显示表单所需的属性可以在以下例子的 **FormData** 返回值中获得。

```
StartFormData FormService.getStartFormData(String processDefinitionId)
```

或

```
TaskFormdata FormService.getTaskFormData(String taskId)
```

默认，内置的表单引擎‘对待’这些属性就像流程变量一样。所以如果任务的表单属性与流程变量是 1 对 1 的，那么就没有必要声明任务的表单属性了。例如，以下声明：

```
<startEvent id="start"/>
```

当执行到 start 事件时，所有的流程都是可用的，但

```
formService.getStartFormData(String processDefinitionId).getFormProperties()
```

会是空的，因为没有专门的映射被定义

上面的例子中，所有提交的属性都将作为流程变量被存储。这意味着只要在表单中添加一个新的输入域，就可以存储一个新的变量。

属性是由流程变量得来的，但没有必要非得将其以流程变量存储起来。例如，流程变量有可能是个类 Address 的 JPA 实体。由 UI 技术使用的表单属性 StreetName 有可能关联到表达式#{address.street}。

类似的，用户要在表单中提交的属性可以作为流程变量存储或者使用 UEL 值表达式将其以一个流程变量的嵌套属性存储，如，#{address.street}。

类似的，除非 formProperty 特别声明，否则提交的属性默认以流程变量被存储。

同样类型转换也可以作为表单属性和流程变量之间处理的一部分。

例如：

```
<userTaskid="task">
<extensionElements>
<activiti:formPropertyid="room"/>
<activiti:formPropertyid="duration"type="long"/>
<activiti:formPropertyid="speaker"variable="SpeakerName"writable="false"/>
<activiti:formPropertyid="street"expression="#{address.street}"required="true"/>
</extensionElements>
</userTask>
```

- 表单属性 `room` 将映射为 `String` 类型的流程变量 `room`。
- 表单属性 `duration` 将映射为 `java.lang.Long` 类型的流程变量 `duration`。
- 表单属性 `speaker` 将映射为流程变量 `SpeakerName`。其只能使用在 `TaskFormData` 对象中。如果属性 `speaker` 被提交，将抛出 `ActivitiException`。类似的，设置为 `readable="false"` 的属性将从 `FormData` 内被排除掉，但在提交的时候仍然会对其进行处理。
- 表单属性 `street` 将映射为流程变量 `address` 内的类型为 `String` 的 Java bean 属性 `street`。如果设置 `required="true"` 的表单属性没有被提供，那么在提交的时候就会抛出异常。

也可以作为 `FormData` 的一部分来提供类型元数据，`FormData` 可由方法 `StartFormData FormService.getStartFormData(String processDefinitionId)` 和 `TaskFormdata FormService.getTaskFormData(String taskId)` 返回。

我们支持以下表单属性类型：

- `string` (`org.activiti.engine.impl.form.StringFormType`)
- `long` (`org.activiti.engine.impl.form.LongFormType`)
- `enum` (`org.activiti.engine.impl.form.EnumFormType`)
- `date` (`org.activiti.engine.impl.form.DateFormType`)
- `boolean` (`org.activiti.engine.impl.form.BooleanFormType`)

对于声明的每一个表单属性，以下 `FormProperty` 信息可以通过 `List<FormProperty> formService.getStartFormData(String processDefinitionId).getFormProperties()` 和 `List<FormProperty> formService.getTaskFormData(String taskId).getFormProperties()` 得到。

```
publicinterface FormProperty {
/** 此键用于{@link FormService#submitStartFormData(String, java.util.Map)}
* 或{@link FormService#submitTaskFormData(String, java.util.Map)}在提交属性*/
String getId();
/** 显示标记 */
String getName();
/** 定义在此接口中的类型之一，如{@link #TYPE_STRING} */
FormType getType();
/** 可选，用于属性显示*/
String getValue();
/** 这个属性是否可以被读取以在表单中显示，以及是否可以使用方法
* {@link FormService#getStartFormData(String)}和{@link FormService#getTaskFormData(String)}来访问 */
boolean isReadable();
```

```

/** 在用户提交表单时，这个属性是否是必要的？ */
boolean isWritable();
/** 这个属性是否是一个必填的输入域*/
boolean isRequired();
}

```

例如：

```

<startEventid="start">
<extensionElements>
<activiti:formPropertyid="speaker"
name="Speaker"
variable="SpeakerName"
type="string"/>

<activiti:formPropertyid="start"
type="date"
datePattern="dd-MMM-yyyy"/>

<activiti:formPropertyid="direction"type="enum">
<activiti:valueid="left"name="Go Left"/>
<activiti:valueid="right"name="Go Right"/>
<activiti:valueid="up"name="Go Up"/>
<activiti:valueid="down"name="Go Down"/>
</activiti:formProperty>

</extensionElements>
</startEvent>

```

所有这些信息都可以通过 API 进行访问。可以使用 `formProperty.getType().getName()` 来获取类型的名称。甚至使用 `formProperty.getType().getInformation("datePattern")` 可以取得日期模式，使用 `formProperty.getType().getInformation("values")` 可以取得枚举值。

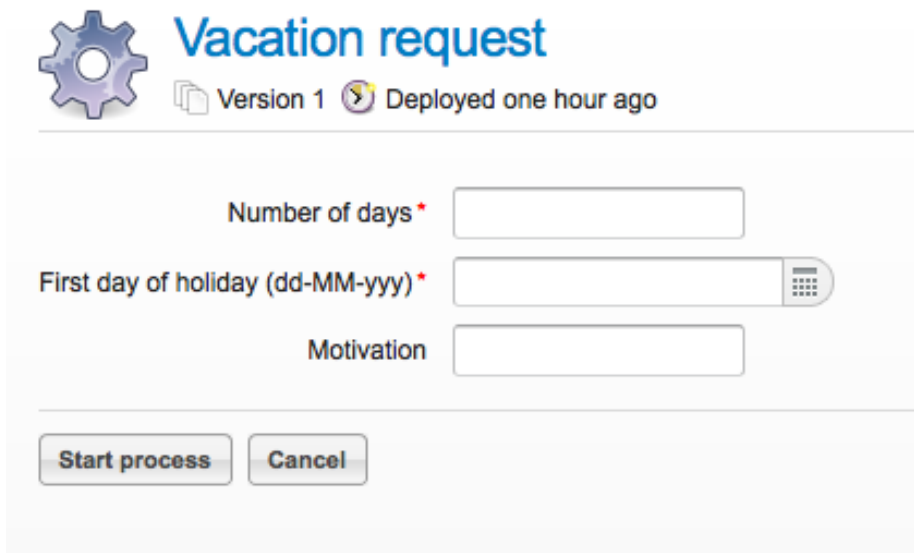
Activity Explorer 支持表单属性，并且会根据表单的定义来渲染表单。以下 xml 片段




```

<startEvent... >
<extensionElements>
<activiti:formPropertyid="numberOfDays"name="Number of days"value="{numberOfDays}"type="long"required="true"/>
<activiti:formPropertyid="startDate"name="First day of holiday (dd-MM-yyy)"value="{startDate}"datePattern="dd-MM-yyyy
hh:mm"type="date"required="true"/>
<activiti:formPropertyid="vacationMotivation"name="Motivation"value="{vacationMotivation}"type="string"/>
</extensionElements>
</userTask>


```

用于 Activity Explorer 时会渲染成流程的启动表单。



 **Vacation request**
 Version 1  Deployed one hour ago

Number of days *

First day of holiday (dd-MM-yyyy) * 

Motivation

8.2 外部的表单渲染

该 API 也允许你来执行 Activiti 引擎以外的你自己的任务表单渲染。这些步骤解释了用于自己渲染任务表单的技术。

本质上，所有渲染表单所需要的数据都是在这两个服务方法其一内进行装配的：`StartFormData`
`FormService.getStartFormData(String processDefinitionId)`和 `TaskFormdata FormService.getTaskFormData(String taskId)`。

可以使用 `ProcessInstance FormService.submitStartFormData(String processDefinitionId, Map<String,String> properties)`和 `void FormService.submitStartFormData(String taskId, Map<String,String> properties)`来提交表单属性。

要想了解表单属性是如何映射到流程变量的，见“[表单属性](#)”一节。

你可以在部署的业务归档文件内放任何表单模板（如果你想要按流程的版本对它们进行存储）。这样它就可以作为部署的资源来使用了，可以这样来获得：先使用 `String ProcessDefinition.getDeploymentId()`，然后使用 `InputStream RepositoryService.getResourceAsStream(String deploymentId, String resourceName)`；这样就能获得你的模板定义文件，用于渲染/显示你自己应用中的表单。

不仅限于任务表单，对于任何其它的目的，你都可以利用这一访问部署的资源的功能。

属性`<userTask activiti:formKey="..."`是通过 API 中的 `String FormService.getStartFormData(String processDefinitionId).getFormKey()`和 `String FormService.getTaskFormData(String taskId).getFormKey()`公布的。你可以使用它来存储部署中的模板的全名（例如，`org/activiti/example/form/my-custom-form.xml`），但这并不是必须的。例如，你也可以将普通的 key 值存储在表单属性中，然后使用一个算法或转换来得到实际中需要使用的模板。这在你想要针对不同用户界面技术渲染不同表单的情况是很方便的，例如，使用在标准屏幕大小下的 web 应用的表单，使用在移动手机屏幕的表单，甚至可以是使用在 IM 表单和 email 表单的模板。

第九章、JPA

可以使用 JPA 实体充当流程变量，允许：

- 基于流程变量更新现有的 JPA 实体，变量可以在用户任务的表单上填入，也可以由服务任务生成。
- 重用已有的领域模型，而无需编写显式的服务来获得实体、更新其值。
- 根据已有实体的属性做出决定（分支）。
- ...

9.1 要求

只支持符合以下要求的实体：

- 实体必须使用 JPA 注解进行配置，我们支持对字段和属性的访问。也可以使用映射的超类。
- 实体中要有使用 @Id 注解的主键，不支持联合主键（@EmbeddedId 和 @IdClass）。Id 字段/属性可以是任何 JPA 规范所支持的类型：基本数据类型以及基本数据类型的包装类型（boolean 除外）、String、BigInteger、BigDecimal、java.util.Date 以及 java.sql.Date。

9.2 配置

要使用 JPA 实体，引擎必须拥有 EntityManagerFactory 的引用。这可以通过配置引用或提供持久化单元名称来完成。会自动检测充当变量的 JPA 实体，并对其做相应的处理。

下面例子中的配置使用了 jpaPersistenceUnitName：

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">

  <!-- Database configurations -->
  <property name="databaseSchemaUpdate" value="true"/>
  <property name="jdbcUrl" value="jdbc:h2:mem:JpaVariableTest;DB_CLOSE_DELAY=1000"/>

  <property name="jpaPersistenceUnitName" value="activiti-jpa-pu"/>
  <property name="jpaHandleTransaction" value="true"/>
  <property name="jpaCloseEntityManager" value="true"/>

  <!-- job executor configurations -->
  <property name="jobExecutorActivate" value="false"/>

  <!-- mail server configurations -->
  <property name="mailServerPort" value="5025"/>
</bean>
```

接下来示例中的配置提供了一个我们自己定义的 EntityManagerFactory（此例中，为 open-jpa 实体管理器）。注意以下代码片段只包含了与本例相关的 beans，省略了其它 beans。使用 open-jpa 实体管理器的完整可行的例子可以在 activiti-spring-examples 下找到（/activiti-spring/src/test/java/org/activiti/spring/test/jpa/JPASpringTest.java）。


```

<beanid="entityManagerFactory"class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
<propertyname="persistenceUnitManager"ref="pum"/>
<propertyname="jpaVendorAdapter">
<beanclass="org.springframework.orm.jpa.vendor.OpenJpaVendorAdapter">
<propertyname="databasePlatform"value="org.apache.openjpa.jdbc.sql.H2Dictionary"/>
</bean>
</property>
</bean>

<beanid="processEngineConfiguration"class="org.activiti.spring.SpringProcessEngineConfiguration">
<propertyname="dataSource"ref="dataSource"/>
<propertyname="transactionManager"ref="transactionManager"/>
<propertyname="databaseSchemaUpdate"value="true"/>
<propertyname="jpaEntityManagerFactory"ref="entityManagerFactory"/>
<propertyname="jpaHandleTransaction"value="true"/>
<propertyname="jpaCloseEntityManager"value="true"/>
<propertyname="jobExecutorActivate"value="false"/>
</bean>

```

同样的配置也可以在编程式构建引擎时完成，例如：

```

ProcessEngine processEngine = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResourceDefault()
    .setJpaPersistenceUnitName("activiti-pu")
    .buildProcessEngine();

```

配置属性：

- **jpaPersistenceUnitName**：使用的持久化单元的名称。（要保证类路径下存在该持久化单元。依据 jpa 规范，默认路径为/META-INF/persistence.xml）。使用 jpaEntityManagerFactory 或者 jpaPersistenceUnitName。
- **jpaEntityManagerFactory**：实现了 javax.persistence.EntityManagerFactory 的 bean 的引用，用于加载实体并刷新更新。使用 jpaEntityManagerFactory 或者 jpaPersistenceUnitName。
- **jpaHandleTransaction**：表示在被使用的 EntityManager 实例上，流程引擎是否需要开启并提交/回滚事务的标志。在使用 Java 事务 API（JTA）时，将其设置为 false。
- **jpaCloseEntityManager**：表示流程引擎是否应该关闭从 EntityManagerFactory 获得的 EntityManager 实例的标志。当 EntityManager 是由容器管理时，要将其设置为 false（例如，在使用不仅局限于单一事务的扩展持久化上下文的时候）。

9.3 用法

9.3.1 简单示例

使用 JPA 变量的例子可以在 JPAVariableTest 中找到。我们会逐步解释 JPAVariableTest.testUpdateJPAEntityValues。

首先，创建基于 META-INF/persistence.xml 的 EntityManagerFactory 作为我们的持久化单元。包括持久化单元所包含的类，以及提供商特有的某些配置。

在测试中我们使用了一个简单的要被持久化的实体，其有一个 `id` 属性和一个 `String` 类型的 `value` 属性。运行测试前，我们会创建一个实体并将其保存起来。

```
@Entity(name = "JPA_ENTITY_FIELD")
publicclass FieldAccessJPAEntity {

    @Id
    @Column(name = "ID_")
    private Long id;

    private String value;

    public FieldAccessJPAEntity() {
        // JPA所需的空的构造方法
    }

    public Long getId() {
        return id;
    }

    publicvoid setId(Long id) {
        this.id = id;
    }

    public String getValue() {
        return value;
    }

    publicvoid setValue(String value) {
        this.value = value;
    }
}
```

启动一个新的流程实例，作为变量添加该实体。至于其它变量，将被存储到引擎的持久化数据库中。下一次该变量被请求时，会根据该类和存储的 `Id` 从 `EntityManager` 加载它。

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

ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("UpdateJPASValuesProcess", variables);
```

在我们的流程定义中第一个节点是一个服务任务，它将调用 `entityToUpdate` 上的方法 `setValue`，就是之前在启动流程实例时设置的 `JPA` 变量，它会被从当前引擎上下文关联的 `EntityManager` 中加载。

```
<serviceTask id='theTask' name='updateJPAEntityTask' activiti:expression='${entityToUpdate.setValue('updatedValue')}' />
```

当服务任务结束时，流程实例将等待在定义在流程定义内的用户任务上，这让我们有机会查看此流程实例。此时，EntityManager 已经被刷新了，并且对实体的修改也被推进了数据库。在获取变量 entityToUpdate 的 value 时，该实体会被再次加载，而且我们获得的实体它的 value 属性为 updateValue。

```
//流程'UpdateJPAValuesProcess'内的服务任务必须已经在entityToUpdate上设置了值。
Object updatedEntity = runtimeService.getVariable(processInstance.getId(), "entityToUpdate");
assertTrue(updatedEntity instanceof FieldAccessJPAEntity);
assertEquals("updatedValue", ((FieldAccessJPAEntity)updatedEntity).getValue());
```

9.3.2 查询 JPA 流程变量

可以查询变量为某一 JPA 实体的 ProcessInstances 和 Executions。**注意**，ProcessInstanceQuery 和 ExecutionQuery 中只有 variableValueEquals(name, entity)支持 JPA 实体。方法 variableValueNotEquals、variableValueGreaterThan、variableValueGreaterThanOrEqual、variableValueLessThan 以及 variableValueLessThanOrEqual 是不支持的，当传递 JPA 实体的值时会抛出 ActivitiException。

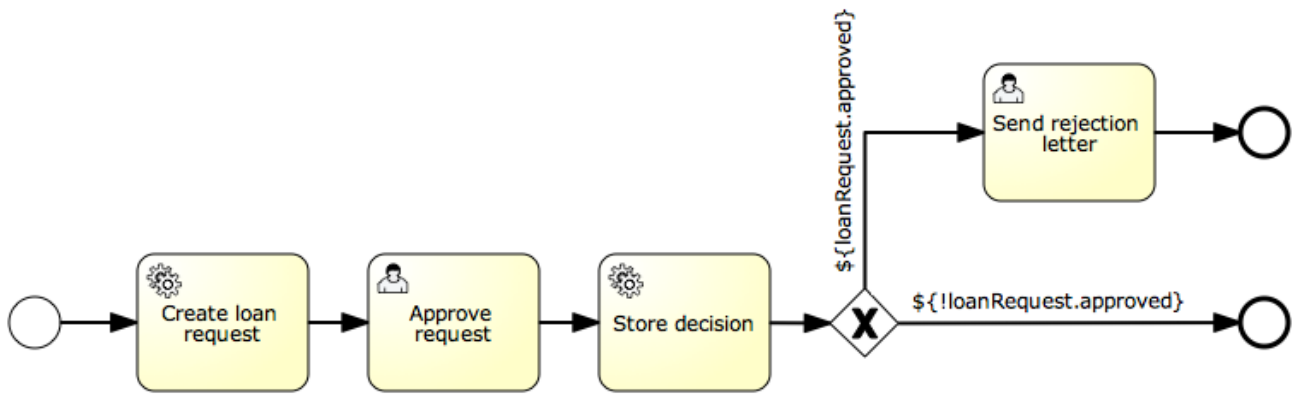
```
ProcessInstance result = runtimeService.createProcessInstanceQuery().variableValueEquals("entityToQuery",
entityToQuery).singleResult();
```

9.3.3 使用 Spring beans 和 JPA 的高级示例

一个更高级的例子，JPASpringTest，可以在 activiti-spring-examples 内找到。其描述了以下简单用例：

- 存在一个使用 JPA 实体的 Spring bean，用于存储贷款请求。
 - 使用 Activiti，我们可以利用那些由现有的 bean 获得的已有的实体，并将其作为变量在流程中使用。
- 流程是按以下步骤进行定义的：
- 服务任务，利用已有的 LoanRequestBean 使用启动流程时接收的变量（比如，可以来源于开始的表单）来创建新的 LoanRequest。使用 activiti:resultVariable（它以一个变量来存储表达式结果）将创建出来的实体以变量进行存储。
 - 用户任务，让经理查看请求，批准/不批准，结果存储到一个 boolean 类型的变量 approvedByManager 上。
 - 服务任务，用来更新贷款请求实体以使该实体与流程同步。
 - 根据实体属性 approved 的值，利用排他分支来决定下一步选择哪条路径：当请求被批准时，流程将结束；否则，会另有一个任务（发送拒绝信），这样就可以由拒绝信手动地通知用户了。

请注意此流程不包含任何表单，所以其只用于单元测试。



```

<?xml version="1.0"encoding="UTF-8"?>
<definitionsid="taskAssigneeExample"
xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:activiti="http://activiti.org/bpmn"
targetNamespace="org.activiti.examples">

<processid="LoanRequestProcess"name="Process creating and handling loan request">
<startEventid='theStart'/>
<sequenceFlowid='flow1'sourceRef='theStart'targetRef='createLoanRequest'/>

<serviceTaskid='createLoanRequest'name='Create loan request'
activiti:expression="${loanRequestBean.newLoanRequest(customerName, amount)}"
activiti:resultVariable="loanRequest"/>
<sequenceFlowid='flow2'sourceRef='createLoanRequest'targetRef='approveTask'/>

<userTaskid="approveTask"name="Approve request"/>
<sequenceFlowid='flow3'sourceRef='approveTask'targetRef='approveOrDissapprove'/>

<serviceTaskid='approveOrDissapprove'name='Store decision'
activiti:expression="${loanRequest.setApproved(approvedByManager)}"/>
<sequenceFlowid='flow4'sourceRef='approveOrDissapprove'targetRef='exclusiveGw'/>

<exclusiveGatewayid="exclusiveGw"name="Exclusive Gateway approval"/>
<sequenceFlowid="endFlow1"sourceRef="exclusiveGw"targetRef="theEnd">
<conditionExpressionxsi:type="tFormalExpression">${loanRequest.approved}</conditionExpression>
</sequenceFlow>
<sequenceFlowid="endFlow2"sourceRef="exclusiveGw"targetRef="sendRejectionLetter">
<conditionExpressionxsi:type="tFormalExpression">${!loanRequest.approved}</conditionExpression>
</sequenceFlow>

<userTaskid="sendRejectionLetter"name="Send rejection letter"/>
<sequenceFlowid='flow5'sourceRef='sendRejectionLetter'targetRef='theOtherEnd'/>

```

```
<endEventid='theEnd'/>
<endEventid='theOtherEnd'/>
</process>

</definitions>
```

虽然上面的例子很简单，但确实展示出了结合 **Spring** 和参数化方法表达式来使用 **JPA** 的强大。此流程根本不需要定制 **java** 代码（当然了，除 **Spring bean** 外），大大加速了部署。

第十章、历史

历史是这样的组件：捕获流程执行期间发生了什么，然后将其永久性的存储起来。与运行时的数据相反，历史的数据会一直保留在数据库中，即便是流程实例已经完成。

存在 4 类历史的实体：

- **HistoricProcessInstances**，包含着有关当前的以及结束了的流程实例的信息。
- **HistoricActivityInstances**，包含着有关活动的一个执行路径的信息。
- **HistoricTaskInstances**，包含着有关当前的以及结束了的（已经完成的和删除了的）任务实例的信息。
- **HistoricDetails**，包含着各种与历史性流程实例、活动实例以及任务实例相关的信息。

因为数据库中保存有已经结束了的以及进行中的实例的历史性实体，所以你可能考虑要查询这些表以减少对运行时流程实例的数据的访问，来保持运行时高性能的执行。

接下来，会在 **Activiti Explore** 中展示该信息。同时，报表也是由此信息生成的。

10.1 查询历史

API 中，是可以查询所有这 4 种历史实体的。**HistoryService** 公布了方法 **createHistoricProcessInstanceQuery()**、**createHistoricActivityInstanceQuery()**、**createHistoricDetailQuery()**和 **createHistoricTaskInstanceQuery()**。

下面几个示例展示了针对历史使用查询 API 的几种可能。所有可能的描述可以在 [javadocs](#) 中 **org.activiti.engine.history** 包找到。

10.1.1 HistoricProcessInstanceQuery

获得已经结束且在流程定义'XXX'中所有已经结束了的流程中花费最多时间来完成（持续时间最长）的 10 个 **HistoricProcessInstances**。

```
historyService.createHistoricProcessInstanceQuery()
    .finished()
    .processDefinitionId("XXX")
    .orderByProcessInstanceDuration().desc()
    .listPage(0, 10);
```

10.1.2 HistoricActivityInstanceQuery

获得类型为'serviceTask'，使用的流程定义 id 为'XXX'，且已经结束了的最后一个 **HistoricActivityInstance**。

```
historyService.createHistoricActivityInstanceQuery()
    .activityType("serviceTask")
    .processDefinitionId("XXX")
```

```
.finished()
.orderByHistoricActivityInstanceEndTime().desc()
.listPage(0, 1);
```

10.1.3 HistoricDetailQuery

紧接着的这个例子，获得 id 为 123 的流程内所完成的所有变量更新。此查询只返回 `HistoricVariableUpdates`。注意某个变量名下有多项 `HistoricVariableUpdate` 是可能的，因为每次流程都要更新变量。可以使用 `orderByTime`（变量更新的时间）或 `orderByVariableRevision`（更新运行时变量的修正）按发生的某一顺序对其进行查看。

```
historyService.createHistoricDetailQuery()
.variableUpdates()
.processInstanceId("123")
.orderByVariableName().asc()
.list()
```

这个例子获得了 id 为“123”的流程中任务提交或启动流程时提交的所有[表单属性](#)。此查询只返回 `HistoricFormProperties`。

```
historyService.createHistoricDetailQuery()
.formProperties()
.processInstanceId("123")
.orderByVariableName().asc()
.list()
```

最后这个例子获得了所有在 id 为“123”的任务上执行的变量更新。返回在该任务上设置过的变量（任务本地的变量）的 `HistoricVariableUpdates`，而非流程实例上变量的更新。

```
historyService.createHistoricDetailQuery()
.variableUpdates()
.taskId("123")
.orderByVariableName().asc()
.list()
```

可以利用 `TaskService` 或在 `TaskListener` 内的 `DelegateTask` 上对任务的本地变量进行设置。

```
taskService.setVariableLocal("123", "myVariable", "Variable value");
```

```
publicvoid notify(DelegateTask delegateTask) {
    delegateTask.setVariableLocal("myVariable", "Variable value");
}
```

10.1.4 HistoricTaskInstanceQuery

获得已经结束且所有任务中花费最多时间来完成的（持续时间最长）的 10 个 HistoricTaskInstances。

```
historyService.createHistoricTaskInstanceQuery()
    .finished()
    .orderByHistoricTaskInstanceDuration().desc()
    .listPage(0, 10);
```

获得删除原因中包含“invalid”的被删除的，且最后分配给用户‘kermit’的 HistoricTaskInstances。

```
historyService.createHistoricTaskInstanceQuery()
    .finished()
    .taskDeleteReasonLike("%invalid%")
    .taskAssignee("kermit")
    .listPage(0, 10);
```

10.2 历史的配置

历史的级别可以使用 ProcessEngineConfiguration 中定义的 HISTORY_* 常量进行编程式的配置：

```
ProcessEngine processEngine = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResourceDefault()
    .setHistory(ProcessEngineConfiguration.HISTORY_AUDIT)
    .buildProcessEngine();
```

级别也可以在 activiti.cfg.xml 或 Spring 上下文内配置：

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">
    <property name="history" value="audit"/>
    ...
</bean>
```

以下历史级别可以被配置：

- **none**：忽略所有历史归档。这时运行时流程的执行效率最高，但却没有任何历史性的信息。
- **activity**：存档所有的流程实例和活动实例。不存档细节。
- **audit**：默认。存档所有的流程实例、活动实例以及提交的表单属性，以便通过表单进行的用户交互可被追溯和查证。
- **full**：这是历史归档的最高级别，因此执行时最慢。这一级别存储了所有在 **audit** 中存储的信息，以及其它所有可能的细节如流程变量的更新。

10.3 审查目的的历史

当[配置](#)的级别最低为 `audit` 时。那么所有由方法 `FormService.submitStartFormData(String processDefinitionId, Map<String, String> properties)`和 `FormService.submitTaskFormData(String taskId, Map<String, String> properties)`提交的属性都将被记录下来。

[已知的限制]目前像 Activiti Explorer 中所实现的表单还不能使用 `submitStartFormData` 和 `submitTaskFormData`。所以在 Activiti Explorer 中使用表单时，表单属性不能进行归档。一个权宜的做法是将历史级别设置为 `full`，然后利用变量更新查看用户任务中所设置的值。见 [ACT-294](#)。

可以像这样使用查询 API 获取表单属性：

```
historyService
    .createHistoricDetailQuery()
    .onlyFormProperties()
    ...
    .list();
```

例子中，只返回 `HistoricFormProperty` 类型的历史明细。

如果在调用提交方法前使用 `IdentityService.setAuthenticatedUserId(String)`设置了认证用户，那么提交表单的那个认证用户就可以使用针对启动表单的 `HistoricProcessInstance.getStartUserId()`及针对任务表单的 `HistoricActivityInstance.getAssignee()`来访问历史。

第十一章、Eclipse Designer

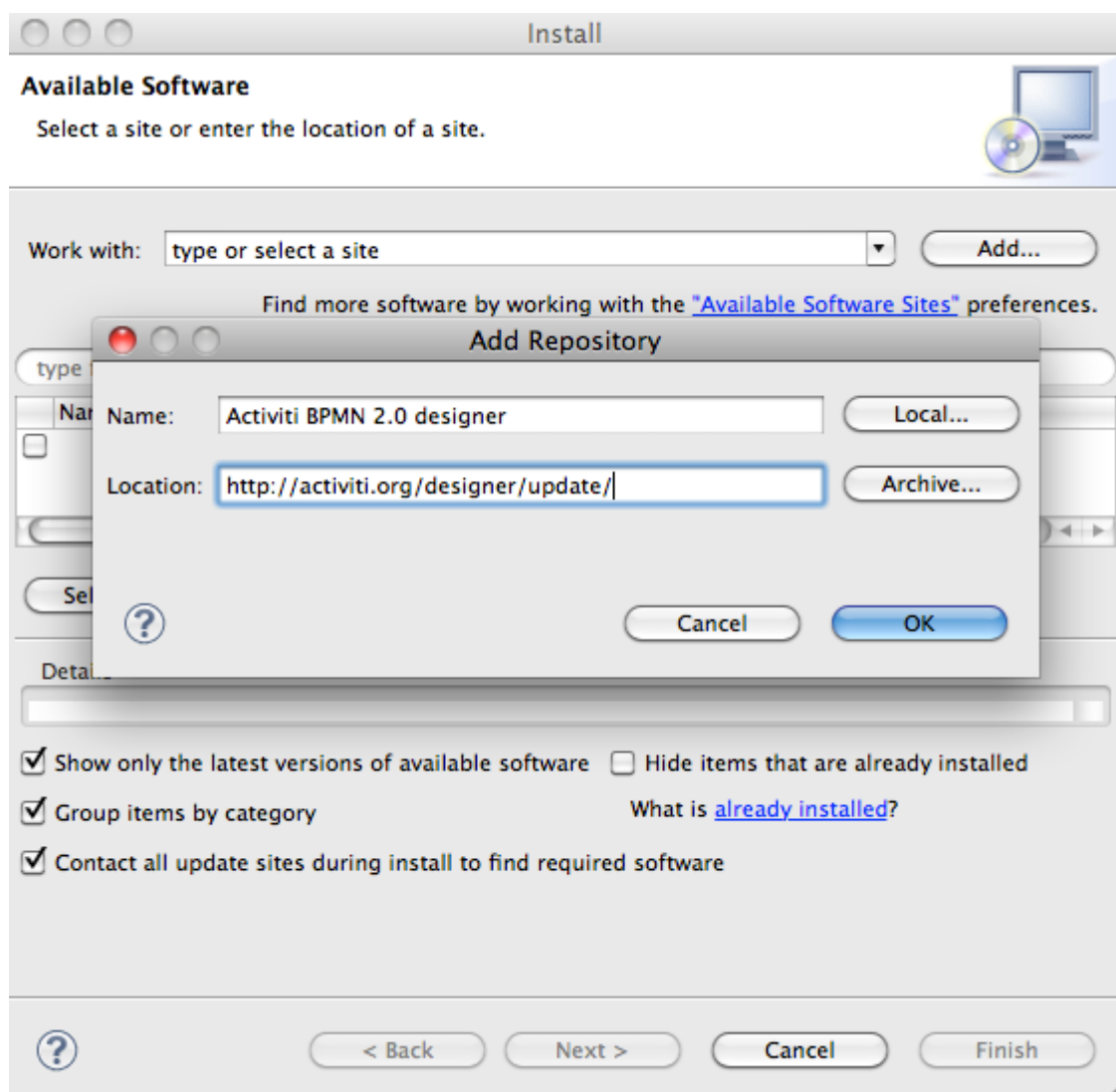
Activiti 同时还有个 Eclipse 插件，Activiti Eclipse Designer，可用于图形化建模、测试、部署 BPMN 2.0 的流程。Activiti 工具库中提供了 [Actiti Modeler](#) 和 Activiti Designer 这两个建模/设计工具。你当然可以按照自己的方式来使用这些工具，但常用的做法是高层的建模时使用 Activiti Modeler。使用 Activiti Modeler 对流程定义进行建模不需要有任何技术性知识。然后可以使用 Activiti Designer 添加一些必要的技术细节，如 Java 服务任务、执行监听器等。由于 Activiti Designer 功能的引入，使得工作流得到了很好的支持。

11.1 安装

以下安装指南在 [Eclipse Indigo](#) 和 [Helios](#) 上都验证了。

打开 **Help**→**Install New Software**。在如下面板中，点击 **Add** 按钮，然后填入下列字段：

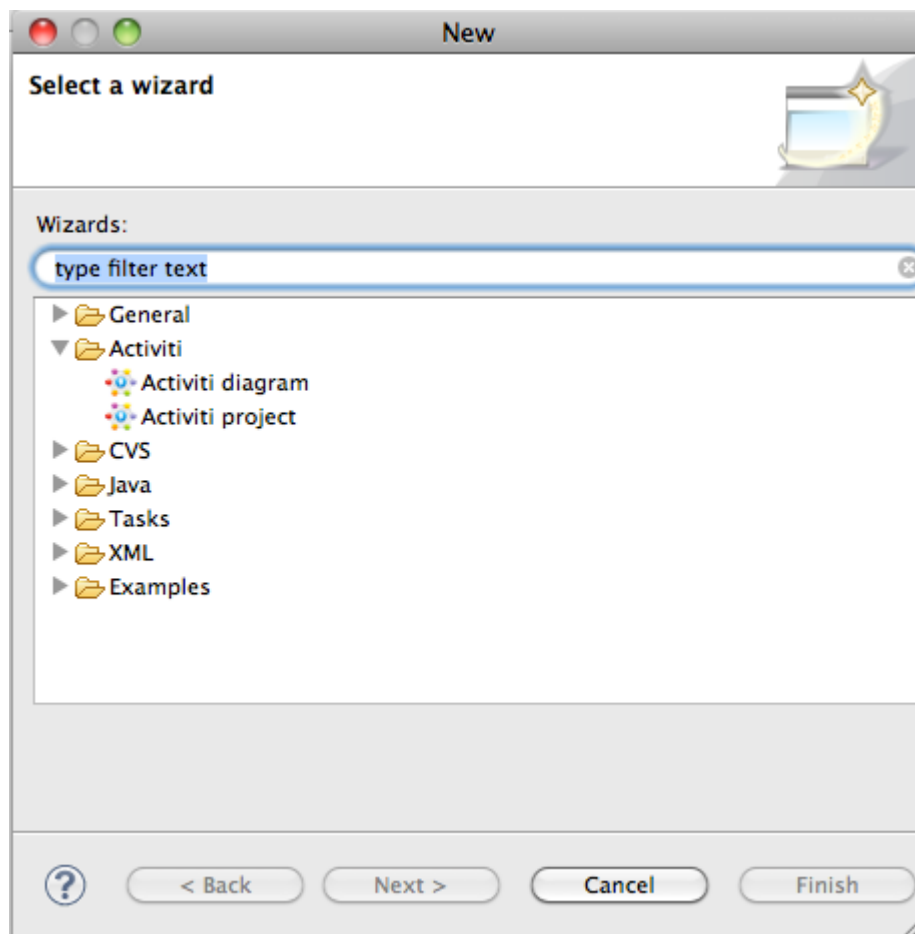
- **Name:** Activiti BPMN 2.0 designer
- **Location:** <http://activiti.org/designer/update/>



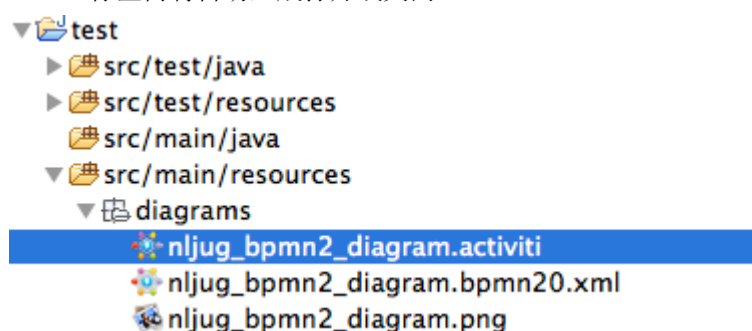
务必要选中“Contact all updates sites..”，因为所有所需的插件接下来都可以被 Eclipse 下载。

11.2 Activiti Designer 编辑器的特性

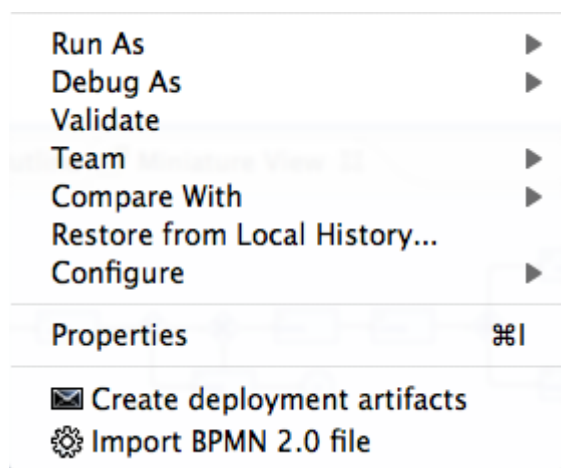
- 创建 Activiti 项目和图形



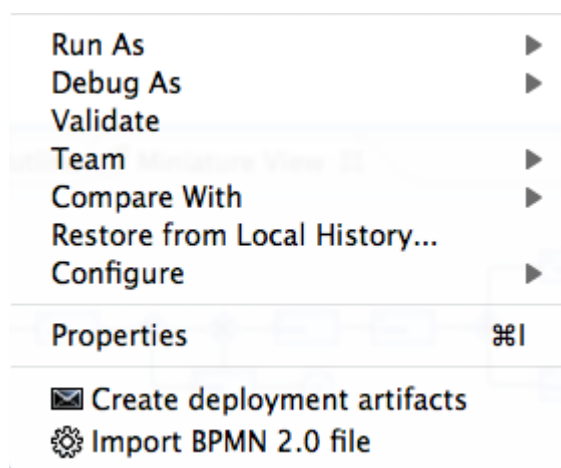
- 每次保存 Activiti 图形后都会自动生成一个 BPMN 2.0 XML 文件和一张流程图片（可以在 Eclipse preference 中的 Activiti 标签内将自动生成打开或关闭）。



- BPMN 2.0 的 XML 文件可以被导入进 Activiti Designer，流程图也会被创建。支持两种方式来引入 BPMN 2.0 的 XML 文件。一是右击 package explorer 中的 Activiti 项目，选择弹出菜单底部的 *Import BPMN 2.0 file* 选项。接下来就可以通过文件选择框选择 BPMN 2.0 的 XML 文件并创建图形了。第二个选择是将 BPMN 2.0 的 XML 文件导入到项目下的 src/main/resources/diagrams 文件夹内（文件名必须以 bpmn20.xml 结尾）。接下来在打开此 BPMN 2.0 的 XML 文件时，图形也被创建了。注意 Activiti Designer 可以从 BPMN 2.0 的 XML 文件中读取 BPMN DI 信息，但在这个版本中，图形只是通过解析、分析 BPMN 2.0 的 XML 来创建的，因为这已经是最好的结果了。这意味着对于即使不带 BPMN DI 信息的 BPMN 2.0 的 XML 文件的导入也是可行的。



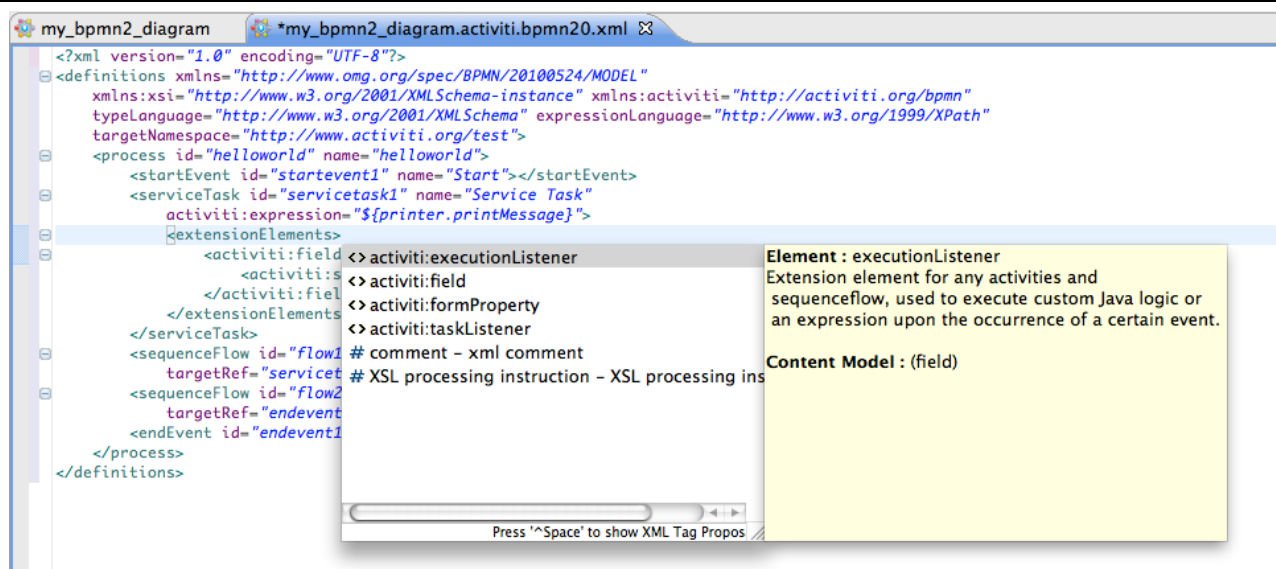
- 对于部署的 BAR 文件以及可选的 JAR 文件，可以通过右击 `package explorer` 视图中的 `Activiti` 项目，然后选择弹出菜单底部的 `Create deployment artifacts` 选项由 `Activiti Designer` 来创建。更多关于 `Designer` 的部署功能，见[部署](#)一节。



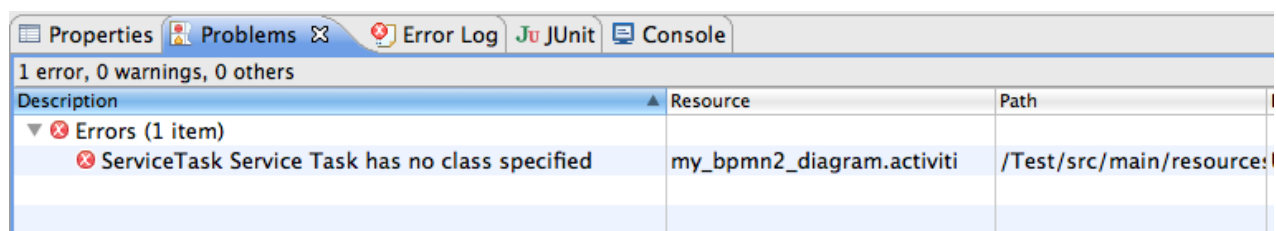
- 生成单元测试（在 `package explorer` 视图下右击 `BPMN 2.0` 的 `XML` 文件，然后选择 `generate unit test`）。生成的单元测试是配置为运行在 `H2` 数据库上的 `Activiti` 配置。你现在就可以运行单元测试来测试你的流程定义了。



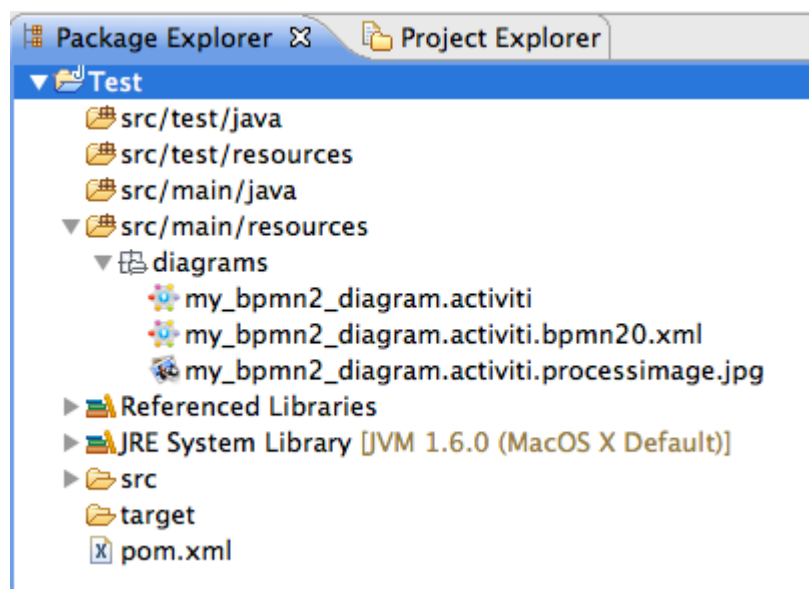
- `BPMN 2.0` 的 `XML` 文件在 `Activiti XML` 的编辑器中打开，它提供了内容提示。注意有 2 个主要的 `XSDs` 配置，`BPMN 2.0` 规范的 `XSD` 和 `Activiti` 扩展 `XSD`。这两个 `XSDs` 还没很好地统一。



- 每次 Activiti 图形保存时都会执行基本的校验，可能的 errors 会在 Eclipse 的 problem 视图内显示。

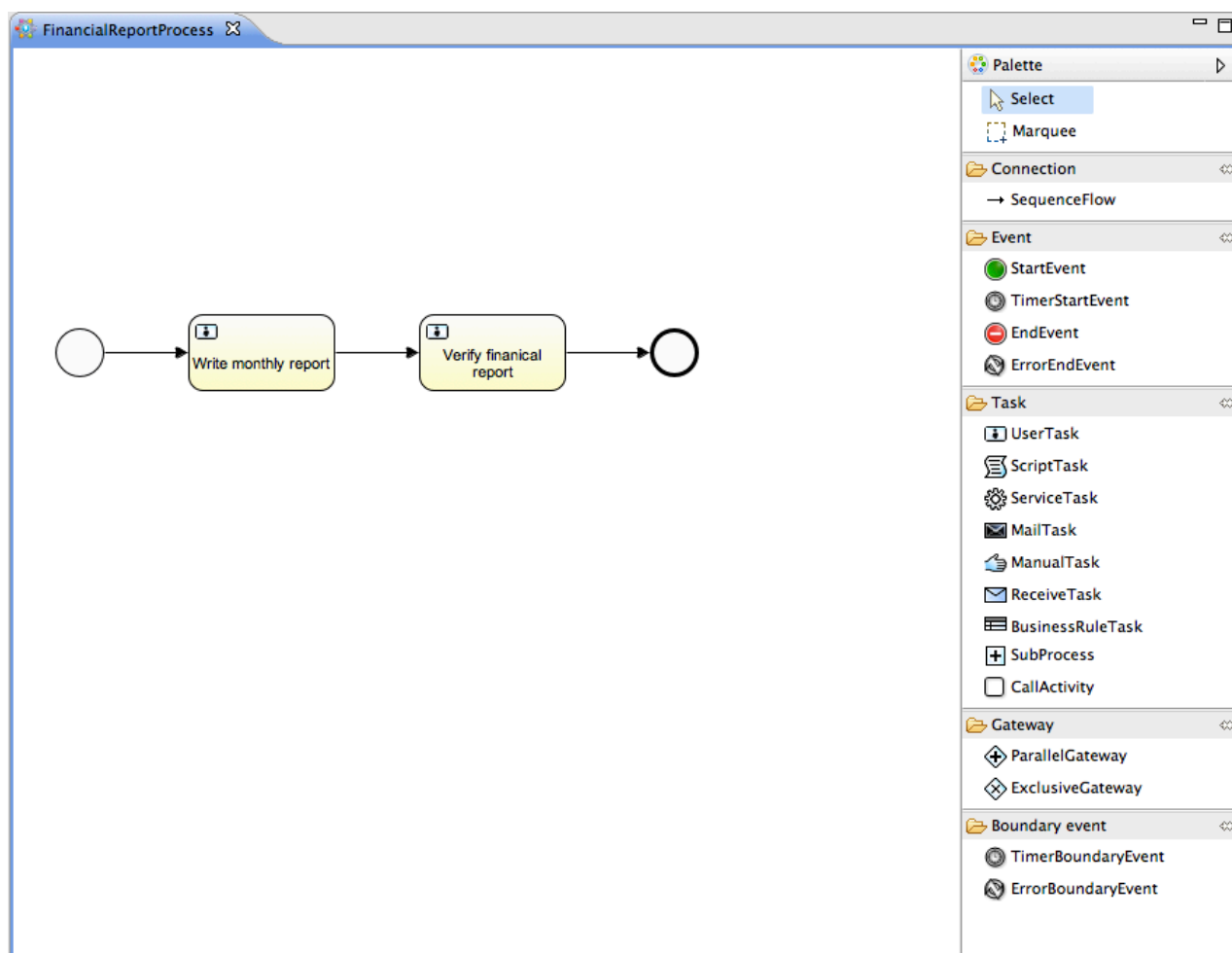


- Activiti 项目可作为 Maven 项目来生成。要配置依赖，需要运行 `mvn eclipse:eclipse`，这样 Maven 的依赖才像预期的那样被配置。注意对流程设计来讲是不需要那些 Maven 依赖的。只是在运行单元测试时需要。

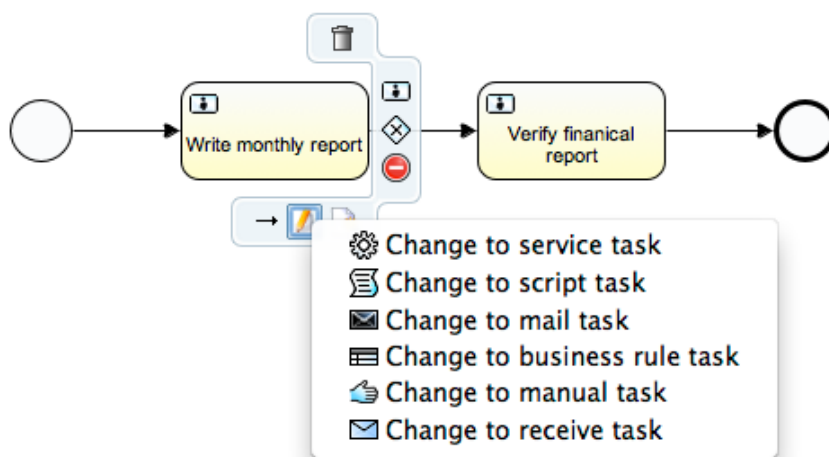


11.3 Activiti Designer 的 BPMN 特性

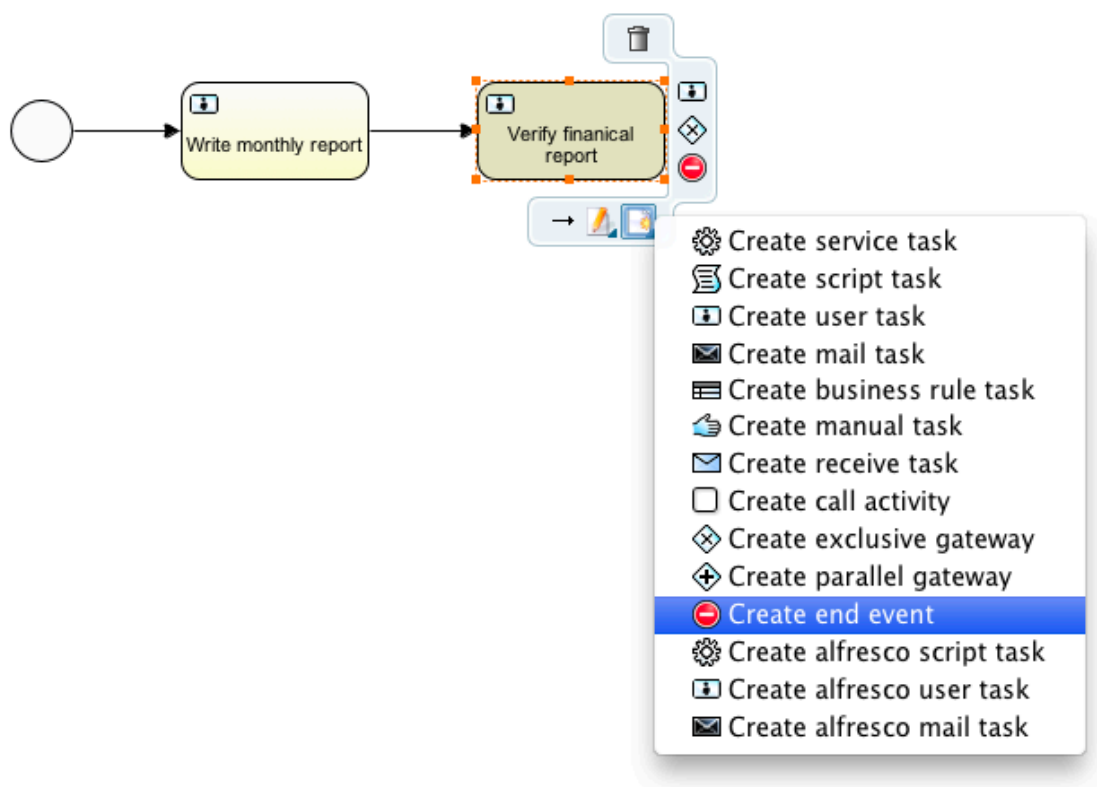
- 支持 start 事件、end 事件、顺序流、并行分支、排他分支、嵌入子过程、调用活动、脚本任务、用户任务、服务任务、邮件任务、手工任务、定时器边界事件以及 error 边界事件。



- 可以很快地改变一个任务的类型，只需鼠标悬停在该元素上，然后选择新的任务类型。



- 可以快速地添加一个新元素，只需鼠标悬停某个元素上，然后选择一个新的元素类型。



- 支持对 Java 服务任务的 Java 类、表达式以及代理表达式的配置。此外，也可以配置字段扩展。

General

Main config

Listeners

Type:

☐ Java class

☒ Expression

☐ Delegate expression

Expression:

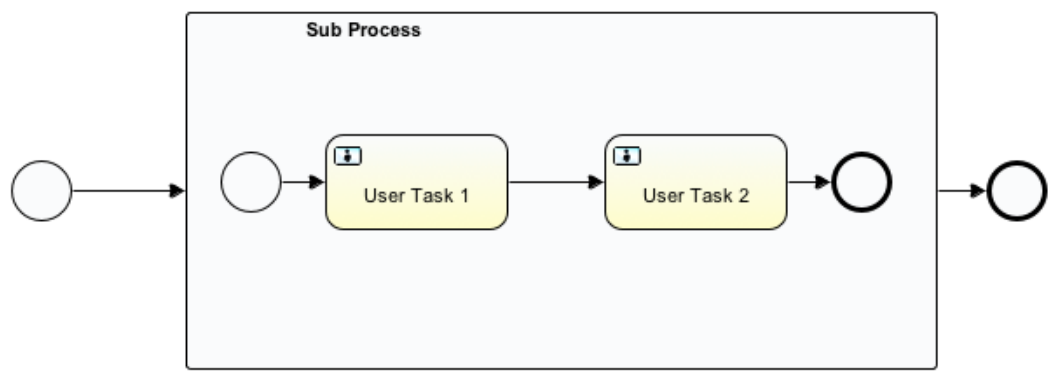
#{printer.printMessage}

Result variable:

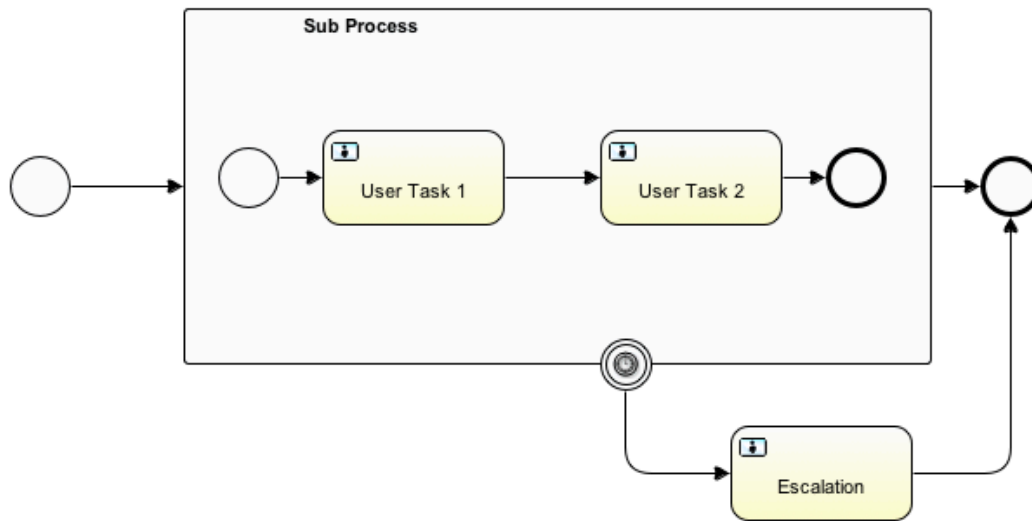
Fields:

Field name	String value / Expression
message	hello

- 支持扩展的嵌入子过程。此版本中不支持嵌入子过程分层。这意味着不能再将嵌入子过程添加进另一个嵌入子过程。



- 支持任务和嵌入子过程上的定时器边界事件。尽管，定时器边界事件最大的意义在于可以在 Activiti Designer 中的用户任务或嵌入子过程上使用。

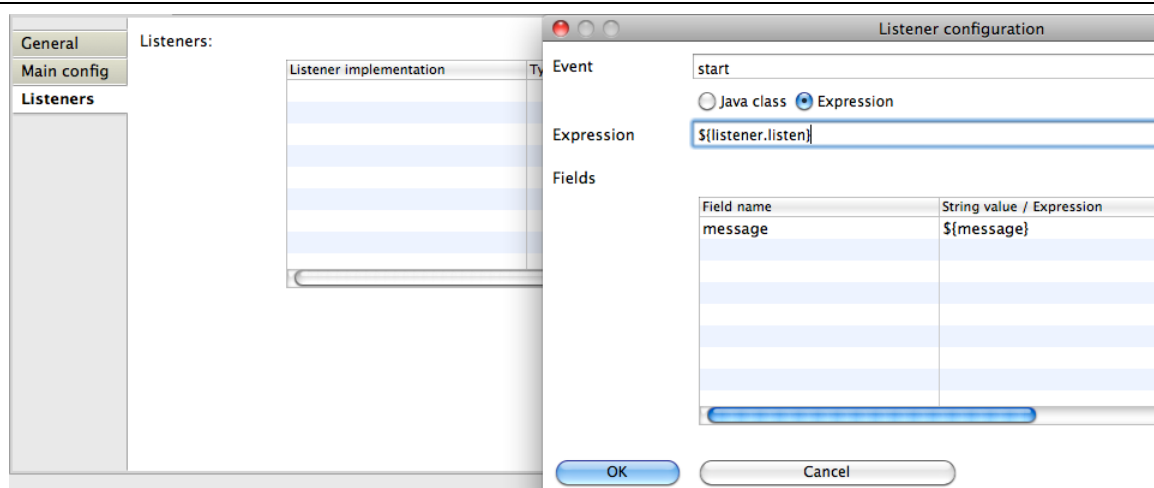


- 支持附加的 Activiti 的扩展，如邮件任务、用户任务候选者配置以及脚本任务的配置。

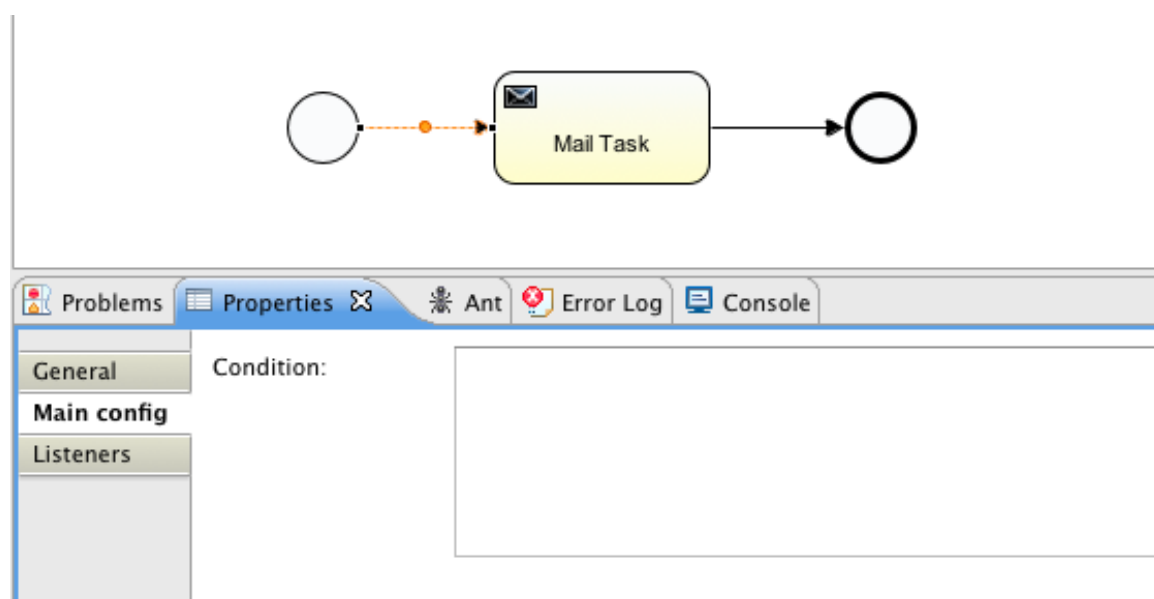
The screenshot shows the Activiti Designer interface. At the top, a 'Mail Task' (rounded rectangle with an envelope icon) is displayed. Below it, the 'Properties' panel is open, showing the 'Main config' tab. The configuration fields are as follows:

Field	Value
To:	
From:	
Subject:	
Cc:	
Bcc:	
Html text:	
Non-Html text:	

- 支持 Activiti 执行监听和任务监听。可以给执行监听添加字段扩展。

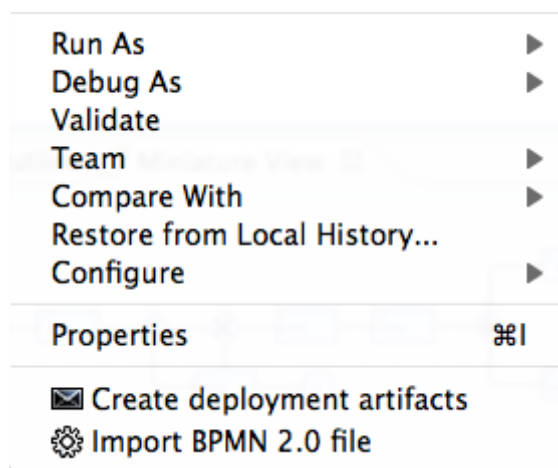


- 支持顺序流上的条件。

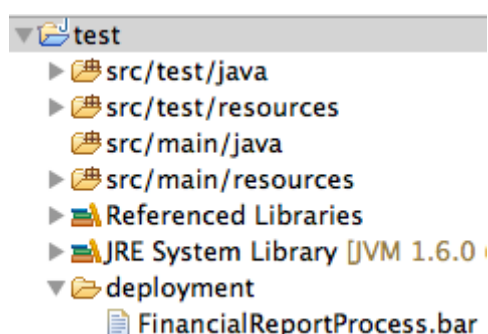


11.4 Activiti Designer 的部署特性

将流程定义和任务表单部署到 Activiti 引擎并不难。要准备一个 BAR 文件，其内含有流程定义的 BPMN 2.0 的 XML 文件，还可以有任务表单以及可以在 Activiti Explorer 中浏览的流程图片。在 Activiti Designer 中创建 BAR 文件是非常简单的。完成流程实现后，在 package explorer 视图下右击你的 Activiti 项目，然后选择弹出菜单底部的 **Create deployment artifacts** 选项。



接下来，deployment 目录会被创建，其内包含有 BAR 文件，你的 Activiti 项目中的 Java 类的 JAR 文件也可以在此。



现在就可以使用 Activiti Explorer 中的部署标签将这个文件部署到 Activiti 引擎上了，相信你已经准备好了。

当你的项目中含有 Java 类时，部署要稍微麻烦一点。这种情况下，Activiti Designer 中的 **Create deployment artifacts** 同时会生成包含编译了的类的 JAR 文件。必须将这个 JAR 文件部署到你的 Activiti Tomcat 安装路径下的 activiti-XXX/WEB-INF/lib 路径下。以使得这些类在 Activiti 引擎的类路径下是可用的。

11.5 扩展 Activiti Designer

[试验性的]可以对 Activiti Designer 提供的默认功能进行扩展。这一节介绍有哪些扩展可用，如何使用这些扩展，并提供了一些使用的例子。当默认的功能不能满足要求，需要额外功能时，或者业务流程建模时有领域所特有的需求时，扩展 Activiti Designer 就变得很有用了。Activiti Designer 的扩展分为两种不同的类型，扩展画板和扩展输出格式。每种形式都有其特有的方式和不同的专业技术。

注意

扩展 Activiti Designer 需要有技术知识以及专业的 Java 编程的知识。根据你要创建的扩展类型，你可能也要熟悉 Maven、Eclipse、OSGI、Eclipse 扩展以及 SWT。

11.5.1 定制画板

流程建模时，可以定制显示给用户的画板。画板是那些可在流程图形的画布上拖拽的形状的集合，它显示在画布右边。正如你在默认画板所看到的，默认的形状被分组到了事件、分支等区隔（称为“抽屉”）中。Activiti Designer 中有两种内置的选择来定制画板中的抽屉和形状：

- 将你自己的形状/节点添加到现有的或新的抽屉内。
- 禁用 Activiti Designer 提供的任一或全部 BPMN 2.0 的形状，除了连接工具和选择工具。

要想定制画板，需要创建一个 JAR 文件，并将其添加到 Activiti Designer 特定的安装目录下（后面有更多关于[如何操作](#)）。这样的 JAR 文件称为扩展。通过编写包含在扩展中的类，Activiti Designer 就能知道你想要的定制。为了使其运行，类必须实现某些接口。你需要将一个集成了这些接口以及一些继承用的基础类的类库添加到你项目的类路径下。

你可以在 Activiti Designer 管理的源码下找到下面列出的代码实例。查看 Activiti 源码 `projects/designer` 目录下 `examples/money-tasks` 目录。

注意

可以使用你所偏爱的工具来设置你的项目，然后使用你选择的构建工具来构建 JAR 文件。下面的说明，设置假设使用的是 Eclipse Helios，Maven（3.x）作为构建工具，但任何其它的设置也都能让你创建同样的结果。

11.5.1.1 扩展的设置（Eclipse/Maven）

下载并解压 [Eclipse](#)（Galileo 或 Helios 都能运行）和最新版本（3.x）的 [Apache Maven](#)。如果使用 2.x 版本的 Maven，在构建项目时会运行出错，所以确保你的版本是最新的。我们假设你能熟练使用基本特性和 Eclipse 的 Java 编辑器。由你决定是使用 Eclipse 的 Maven 特性还是在命令行上运行 Maven 命令。

在 Eclipse 下创建一个新的项目。这是一个普通项目类型。在项目的根路径下创建 `pom.xml` 文件用来包含 Maven 项目的设置。同时创建 `src/main/java` 和 `src/main/resources` 文件夹，这是 Maven 对于 Java 源文件和资源的约定。打开 `pom.xml` 文件，添加以下代码：

```
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.acme</groupId>
<artifactId>money-tasks</artifactId>
<version>1.0.0</version>
<packaging>jar</packaging>
<name>Acme Corporation Money Tasks</name>
...
</pom>
```

正如你所看到的，这只是一个定义了项目 `groupId`、`artifactId` 和 `version` 的基本的 `pom.xml` 文件。我们将创建一个定制，其中包含我们 money 业务中一个的自定义节点。

通过在 `pom.xml` 文件内引入依赖，就可以将集成的类库添加到你项目的依赖中。

```
<dependencies>
```

```

<dependency>
<groupId>org.activiti</groupId>
<artifactId>activiti-designer-integration</artifactId>
<version>0.7.0</version><!-- Current Activiti Designer Version -->
<scope>compile</scope>
</dependency>
</dependencies>

...

<repositories>
<repository>
<id>Activiti</id>
<url>http://maven.alfresco.com/nexus/content/repositories/activiti/</url>
</repository>
</repositories>

```

最后，在 `pom.xml` 文件中添加对 `maven-compiler-plugin` 的配置，Java 源代码级别最低是 1.5（看下面的代码片段）。这在使用注解时需要。也可以包含让 Maven 生成 JAR 的 `MANIFEST.MF` 文件的命令。这不是必需的，但这样你可以使用清单文件的一个特定的属性作为扩展的名称（这个名称可以显示在 Designer 的某处，主要是为以后如果 Designer 中有几个扩展的时候使用）。如果你想这样做，将以下代码片段包含进 `pom.xml` 文件：

```

<build>
<plugins>
<plugin>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
<source>1.5</source>
<target>1.5</target>
<showDeprecation>true</showDeprecation>
<showWarnings>true</showWarnings>
<optimize>true</optimize>
</configuration>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-jar-plugin</artifactId>
<version>2.3.1</version>
<configuration>
<archive>
<index>true</index>
<manifest>
<addClasspath>false</addClasspath>
<addDefaultImplementationEntries>true</addDefaultImplementationEntries>
</manifest>
<manifestEntries>
<ActivitiDesigner-Extension-Name>Acme Money</ActivitiDesigner-Extension-Name>
</manifestEntries>

```

```

</archive>
</configuration>
</plugin>
</plugins>
</build>

```

扩展的名称是由 `ActivitiDesigner-Extension-Name` 属性描述的。现在剩下唯一要做的就是让 Eclipse 根据 `pom.xml` 文件中的指令来设置项目了。所以打开命令窗口，转到 Eclipse 工作空间下你的项目的根文件夹。然后执行以下 Maven 命令：

```
mvn eclipse:eclipse
```

等待构建完成。刷新项目（使用项目上下文菜单（右击），选择 **Refresh**）。此时，`src/main/java` 和 `src/main/resources` 文件夹应该已经是 Eclipse 项目下的资源文件夹了。

注意

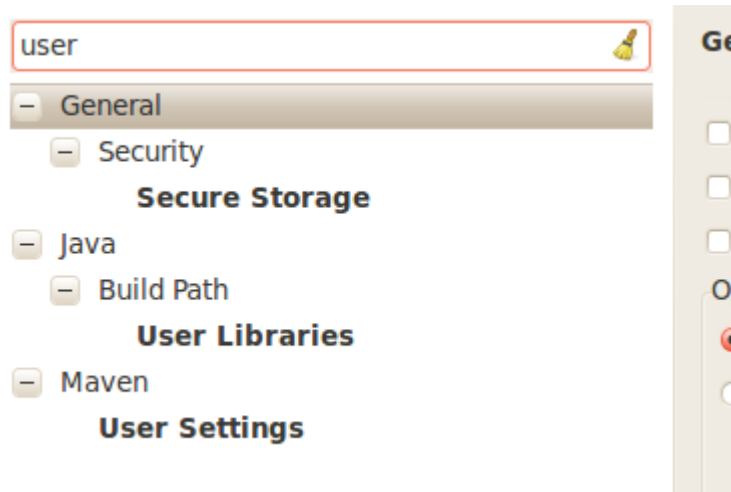
当然你也可以使用 [m2eclipse](#) 插件，只要从项目的上下文菜单（右击）就能启用 Maven 依赖管理。然后在项目上下文菜单中选择 **Maven**→**Update project configuration**。这也能设置源文件夹。

设置就这样。现在就可以开始创建 Activiti Designer 的定制了！

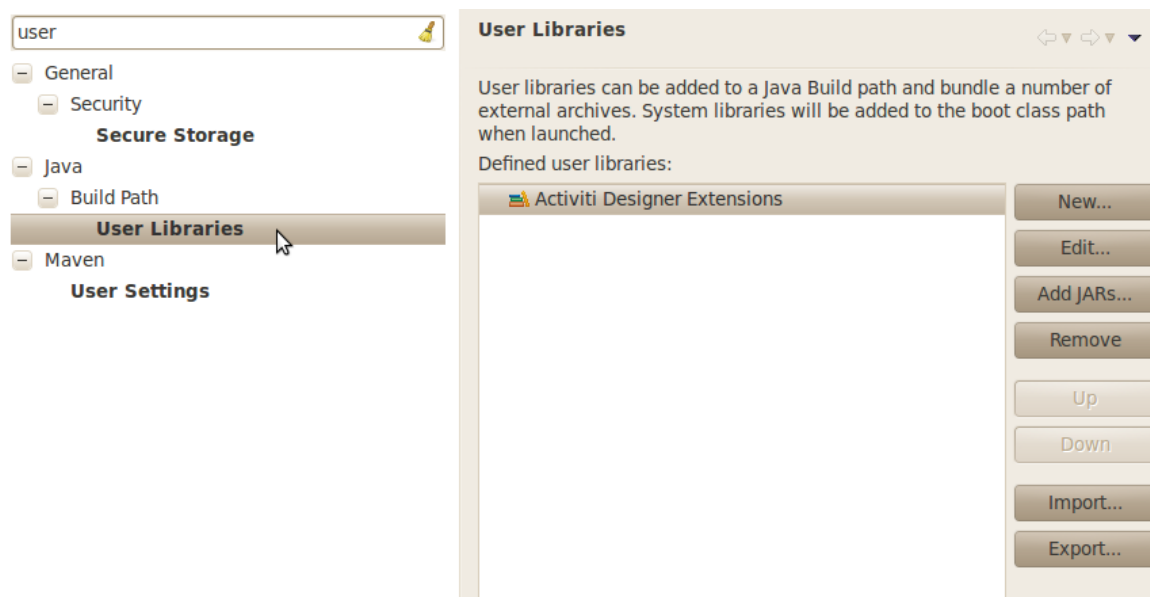
11.5.1.2 将扩展应用到 Activiti Designer

你可能想要知道怎样才能将扩展添加到 Activiti Designer 以使定制被应用。这就是操作步骤：

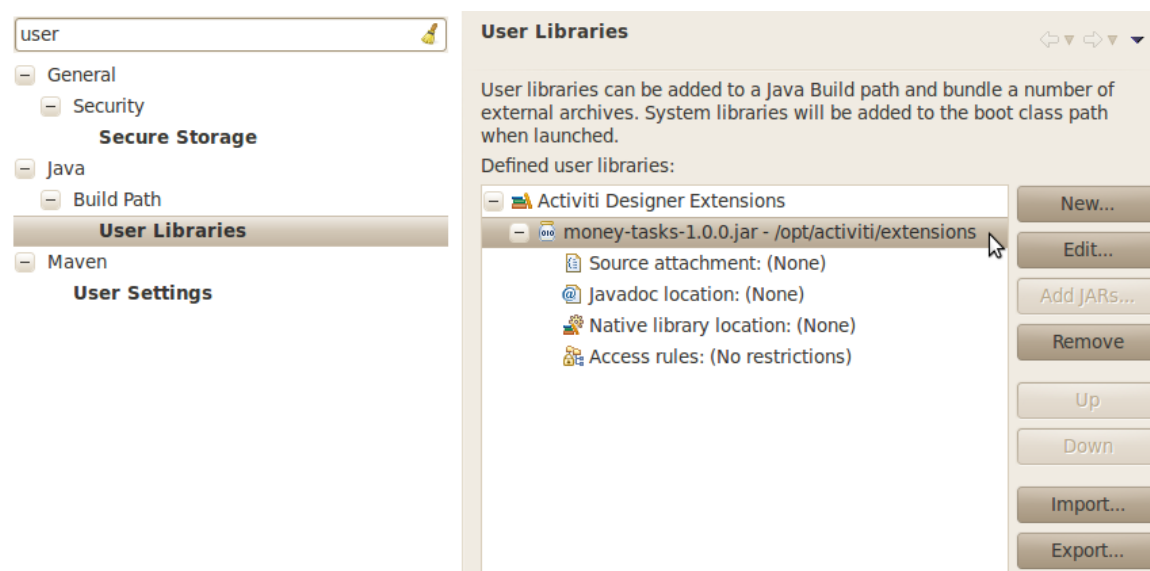
- 一旦创建完扩展的 JAR（例如，利用 Maven 执行项目内的 mvn 的安装程序进行构建），需要将其放在计算机中安装 Activiti Designer 的地方；
- 将扩展存储到硬盘中一个可以的地方，记住这个地方；
- 启动 Activiti Designer，选择菜单栏中的 **Window** → **Preferences**；
- 在 preferences 窗口中，输入 `user` 关键字。在 Java 部分中你应该能看到访问 Eclipse 中的 **User Libraries** 一项。



- 选择 **User Libraries**，在右侧显示的树形视图中你可以添加类库。你会看到用来向 Activiti Designer 添加扩展的默认分组（根据安装的 Eclipse，可能也会看到另外其它的分组）。



- 选择 Activiti Designer Extensions 分组，点击 Add JARs...按钮。导航到存储扩展的文件夹，选择你想要添加的扩展文件。完成后，这个扩展就作为 Activiti Designer Extensions 分组的一部分显示在 preference 窗口内，如下显示。



- 点击 OK 按钮保存并关闭 preferences 对话框。Activiti Designer Extensions 分组自动添加到了你新创建的 Activiti 项目。在 Navigator 或 Package Explorer 视图中你可以看到此用户类库作为一项出现在项目树中。如果工作空间内已经存在 Activiti 的项目了，你也会看到新扩展显示在该分组内。例子如下所示。



此时在打开图形的画板中将有来自新扩展的形状（或禁掉的形状，取决于扩展内的定制）。如果已经打开了图形，将其关闭后重新打开看看画板中有什么变化。

11.5.1.3 向画板添加形状

根据你的项目的设置，现在你就能够很容易在画板中添加形状了。每个要添加的形状都是由 JAR 文件中的类来描述的。注意这些类并不是 Activiti 引擎在运行时使用的类。在扩展内描述的那些属性可以设置给每个 Activiti Designer 内的形状。形状内所参照的运行时的类是要被引擎使用的。就像 Activiti 中所有服务任务一样，这个类必须实现 `JavaDelegate`。

一个形状类就是添加了一些注解的普通 Java 类。这个类必须实现 `CustomServiceTask` 接口，但你不必自己实现这个接口。只需继承 `AbstractCustomServiceTask` 这个基类（目前，必须是直接继承这个类，中间不要有抽象类）。在这个类的 Javadoc 中你可以找到关于它所提供的默认设置以及何时需要重写它所实现的方法的说明。重写让你做一些诸如为画板和画布上的形状提供图标（可以是不同的）、指定你想让节点（活动、事件、分支）拥有的基本形状的事情。

```
/**
 * @author John Doe
 * @version 1
 * @since 1.0.0
 */
public class AcmeMoneyTask extends AbstractCustomServiceTask {
    ...
}
```

需要实现 `getName()` 方法来决定画板中节点使用的名称。也可以将节点放进一个属于它们自己的抽屉，并为其提供一个图标。重写 `AbstractCustomServiceTask` 中恰当的方法。如果你想要提供一个图标，确保图标在 JAR 中 `src/main/resources` 包内，大小为 16x16 像素，格式为 JPEG 或 PNG。提供的路径是相对于那个文件夹的。

给图形添加属性是通过向此类添加成员变量，并使用 `@Property` 注解对其进行注解来完成的，如下：

```
@Property(type = PropertyType.TEXT, displayName = "Account Number")
@Help(displayHelpShort = "Provide an account number", displayHelpLong = HELP_ACCOUNT_NUMBER_LONG)
private String accountNumber;
```

有几个 `PropertyType` 值你可以使用，在[这一节](#)做详细介绍。通过将 `required` 属性设置为 `true` 可以使字段成为必填项。如果用户不填写该字段，就会有消息和红色背景显示。

如果要确保类中这些属性是它们在 `property` 窗口出现的顺序，需要指定 `@Property` 注解的 `order` 属性。

正如你所看到了，有个 `@Help` 注解用来在填写字段时给用户提供一些引导。也可以将 `@Help` 注解用于类本身——这个信息会显示在呈现给用户的属性表的上方。

下面列出了对 `MoneyTask` 的详细阐述。添加了一个 `comments` 字段，你会看到该节点包含了一个图标。

```
/**
```

```

* @author John Doe
* @version 1
* @since 1.0.0
*/
@Runtime(delegationClass = "org.acme.runtime.AcmeMoneyJavaDelegation")
@Help(displayHelpShort = "Creates a new account", displayHelpLong = "Creates a new account using the account number
specified")
public class AcmeMoneyTask extends AbstractCustomServiceTask {

    private static final String HELP_ACCOUNT_NUMBER_LONG = "Provide a number that is suitable as an account number.";

    @Property(type = PropertyType.TEXT, displayName = "Account Number", required = true)
    @Help(displayHelpShort = "Provide an account number", displayHelpLong = HELP_ACCOUNT_NUMBER_LONG)
    private String accountNumber;

    @Property(type = PropertyType.MULTILINE_TEXT, displayName = "Comments")
    @Help(displayHelpShort = "Provide comments", displayHelpLong = "You can add comments to the node to provide a brief
description.")
    private String comments;

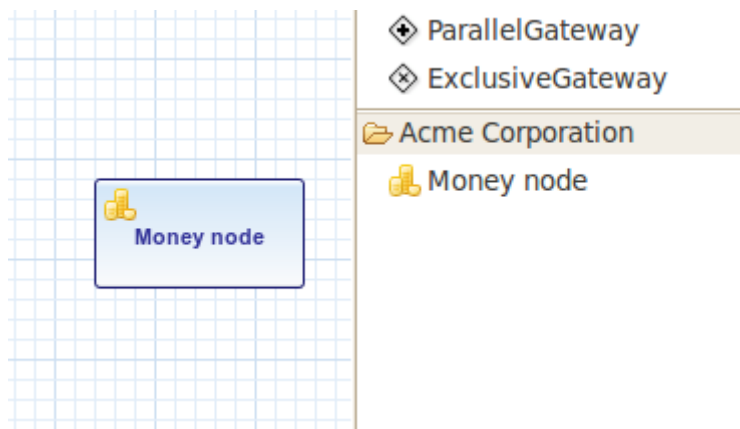
    /**
     * (non-Javadoc)
     *
     * @see org.activiti.designer.integration.servicetask.AbstractCustomServiceTask #contributeToPaletteDrawer()
     */
    @Override
    public String contributeToPaletteDrawer() {
        return "Acme Corporation";
    }

    @Override
    public String getName() {
        return "Money node";
    }

    /**
     * (non-Javadoc)
     *
     * @see org.activiti.designer.integration.servicetask.AbstractCustomServiceTask #getSmallIconPath()
     */
    @Override
    public String getSmallIconPath() {
        return "icons/coins.png";
    }
}

```


如果拿这个形状来扩展 Activiti Designer，画板和对应的节点看起来会如此：



money 任务的属性窗口显示如下。注意对于 accountNumber 字段的必填信息。

每个属性右边的按钮给出了对于字段的帮助。点击按钮显示弹出框，如下显示的。

形状的最后一步是指定当流程实例执行到你的节点时由 Activiti 引擎实例化的类。使用 @Runtime 注解来完成。

delegationClass 属性返回的值必须是运行时类的标准名称。注意该运行时的类不要放在扩展 JAR 文件内，因为它是依赖于 Activiti 类库的。

```
@Runtime(delegationClass = "org.acme.runtime.AcmeMoneyJavaDelegation")
```

11.5.1.4 属性的类型

这一节描述属性的类型，通过将自定义服务任务的 `type` 属性设置为一个 `PropertyType` 值，可以将其用于自定义服务任务。

PropertyType.TEXT

如下所示，创建一个单行文本域。可以是必填项，其验证信息以提示框的方式显示。通过将域的背景颜色改为淡红色来显示验证失败。

Account Number (*): ?

Account Number (*): ?
This field is required

PropertyType.MULTILINE_TEXT

如下所示，创建多行文本域（高度固定为 80 个像素）。可以是必填项，其验证信息以提示框的方式显示。通过将域的背景颜色改为淡红色来显示验证失败。

Comments (*): ?

Comments (*): ?
This field is required

PropertyType.PERIOD

创建结构化的编辑器，使用微调控件编辑每个单位数量来确定一段时间。结果如下所示。可以是必填项（理解为不是所有值都为 0，时间至少要有 1 部分是非零值），其验证信息以提示框的方式显示。通过将域的背景颜色改为淡红色来显示验证失败。该字段的值是以格式为 `1y 2mo 3w 4d 5h 6m 7s` 的字符串进行存储的，表示 1 年，2 个月，3 个星期，4 天，5 小时，6 分钟，7 秒。整个字符串都将被存储，即使有的部分是 0。

Processing Time (*): y, mo, w, d, h, m, s ?

Processing Time (*): y, mo, w, d, h, m, s ?
This field is required

PropertyType.BOOLEAN_CHOICE

创建单个复选框来控制布尔逻辑或切换选择。注意你可以指定 `Property` 注解的 `required` 属性，但它不会被计算，因为那会使用户不得选择，不管是不是选中了选择框。图形存储的值是 `java.lang.Boolean.toString(boolean)`，其结果是 `"true"` 或 `"false"`。

VIP Customer: ☒ ?

PropertyType.RADIO_CHOICE

如下所示，创建一组单选按钮。选择任一单选按钮与选择任一其它单选按钮都是排斥的（即，只允许选择一个）。可以是必填项，其验证信息以提示框的方式显示。通过将这一组的背景颜色改为淡红色来显示验证失败。

这个属性类型要求注解的类的成员还要有@PropertyItems注解（例子，如下）。利用这个额外的注解，你可以指定以字符串数组来提供的项目列表。通过为每个项目添加两个数组项来制定这些项目：第一个是要显示的标题；第二个是要存储的值。

```
@Property(type = PropertyType.RADIO_CHOICE, displayName = "Withdrawl limit", required = true)
@Help(displayHelpShort = "The maximum daily withdrawl amount ", displayHelpLong = "Choose the maximum daily amount that can be withdrawn from the account.")
@PropertyItems({ LIMIT_LOW_LABEL, LIMIT_LOW_VALUE, LIMIT_MEDIUM_LABEL, LIMIT_MEDIUM_VALUE, LIMIT_HIGH_LABEL, LIMIT_HIGH_VALUE })
private String withdrawlLimit;
```

Withdrawl limit (*): ☐ Low (250) ☒ High (2500) ☐ Medium (1000) 

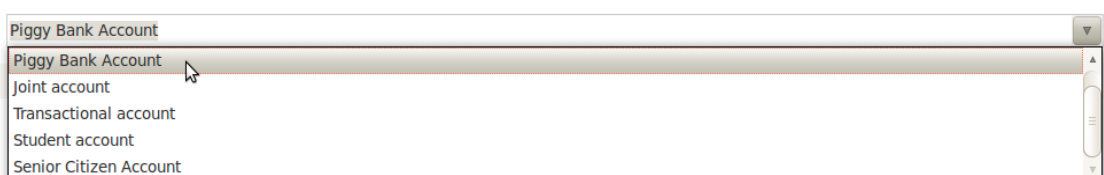

Withdrawl limit (*): ☐ Low (250) ☐ High (2500) ☐ Medium (1000)  This field is required

PropertyType.COMBOBOX_CHOICE

如下所示，创建带有固定选项的下拉列表框。可以是必填项，其验证信息以提示框的方式显示。通过将下拉列表框的背景颜色改为淡红色来显示验证失败。

这个属性类型要求被注解的类的成员还要有@PropertyItems注解（例子，如下）。利用这个额外的注解，你可以指定以字符串数组来提供的项目列表。通过为每个项目添加两个数组项来制定这些项目：第一个是要显示的标题；第二个是要存储的值。

```
@Property(type = PropertyType.COMBOBOX_CHOICE, displayName = "Account type", required = true)
@Help(displayHelpShort = "The type of account", displayHelpLong = "Choose a type of account from the list of options")
@PropertyItems({ ACCOUNT_TYPE_SAVINGS_LABEL, ACCOUNT_TYPE_SAVINGS_VALUE, ACCOUNT_TYPE_JUNIOR_LABEL, ACCOUNT_TYPE_JUNIOR_VALUE, ACCOUNT_TYPE_JOINT_LABEL, ACCOUNT_TYPE_JOINT_VALUE, ACCOUNT_TYPE_TRANSACTIONAL_LABEL, ACCOUNT_TYPE_TRANSACTIONAL_VALUE, ACCOUNT_TYPE_STUDENT_LABEL, ACCOUNT_TYPE_STUDENT_VALUE, ACCOUNT_TYPE_SENIOR_LABEL, ACCOUNT_TYPE_SENIOR_VALUE })
private String accountType;
```

Account type (*):  

Account type (*):

This field is required

PropertyType.DATE_PICKER

如下所示，创建日期选择控件。可以是必填项，其验证信息以提示框的方式显示（注意，使用的控件自动设置为选中系统日期，所以该值很少为空）。通过将此控件的背景颜色改为淡红色来显示验证失败。

这个属性类型要求注解的类成员还要有 `@DatePickerProperty` 注解（示例，见下文）。利用这个额外的注解，你可以指定图形中用于存储日期时间的模式，以及你想要显示的日期选择器的类型。这两个属性都是可选的，并且如果不指定它们都会使用默认值（它们是 `DatePickerProperty` 注解中的静态变量）。`dateTimePattern` 属性用来向 `SimpleDateFormat` 类提供模式。在使用 `swtStyle` 属性时，必须指定 SWT 的 `DateTime` 控件所支持的一个整数，因为正是使用这个控件来渲染这个属性类型的。

```
@Property(type = PropertyType.DATE_PICKER, displayName = "Expiry date", required = true)
@Help(displayHelpShort = "The date the account expires ", displayHelpLong = "Choose the date when the account will expire if no extended before the date.")
@DatePickerProperty(dateTimePattern = "MM-dd-yyyy", swtStyle = 32)
private String expiryDate;
```

Expiry date (*):

◀ December ▶

◀ 2012 ▶

Sun	Mon	Tue	Wed	Thu	Fri	Sat
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

PropertyType.DATA_GRID

如下所示，创建数据表格控件。数据表格允许用户输入任意多行数据，并为每一行的一组固定列输入值（每个行和列交叉处称为单元格）。由用户来决定对行的添加、删除。

这个属性类型要求注解的类成员还要有 `@DataGridProperty` 注解（示例，见下文）。利用这个附加注解，可以指定一些数据网格所特有的属性。需要使用 `itemClass` 属性引用一个类来决定哪些列要加入到表格。`Activiti Designer` 希望成员变量是 `List` 类型。习惯上，可以将 `itemClass` 属性表示的类作为泛型类型来使用。例如，如果你要在表格中编辑一个购物单，你会在 `GroceryListItem` 类上定义表格的列。在你的 `CustomServiceTask` 内，会想要它是像这样：

```
@Property(type = PropertyType.DATA_GRID, displayName = "Grocery List")
@DataGridProperty(itemClass = GroceryListItem.class)
private List<GroceryListItem> groceryList;
```

除了使用了数据表格，“itemClass”类使用的注解与你用来指定 `CustomServiceTask` 的域的注解是一样的。具体的，目前支持 `TEXT`、`MULTILINE_TEXT` 以及 `PERIOD`。你会注意到表格会为每个域创建一个单行的文本控件，不管其 `PropertyType` 是什么类型。这是为了保持表格的图形吸引力和可读性。例如，如果你想要以正常的显示模式显示 `PERIOD` 类型的 `PropertyType`，你能想象的到在不搞乱屏幕的情况下它是永远也不会适合于单元格的（译注，言外之意，必然会搞乱屏幕）。对于

MULTILINE_TEXT 和 PERIOD，添加在每个域上的双击机制都会弹出一个大的 PropertyType 编辑器。用户点击 OK 后，值被存储到域内，因此它能在表格中被看到。

必填属性的处理类似于处理 TEXT 类型的普通域，整个表格在任意域失去焦点时被校验。如果校验失败，数据表格中特定单元格的文本控件的背景颜色会变为浅红。

默认，该组件允许用户添加行，但不允许决定这些行的顺序。如果想要对此允许，必须将 orderable 属性设置为 true，这就能让每行末尾的按钮在表格内上下移动。

注意

此刻，这个属性类型还没被正确注入在运行时的类中。

Account managers:

	First name ?	Last name ?	Authorization period ?	
1.	John	Doe	0y 2mo 0w 0d 0h 0m 0s	↓ ×
2.	Felix			↑ ×

+ Add item

This field is required

11.5.1.5 禁用画板中默认形状

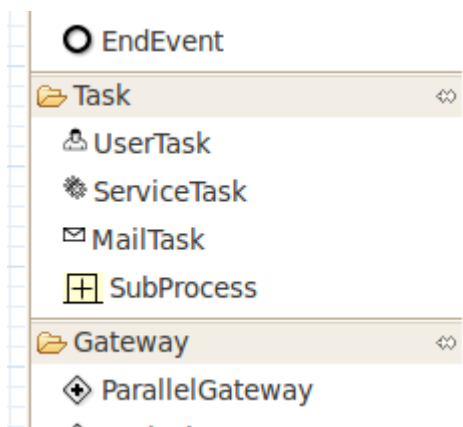
这个定制需要在扩展中包含一个实现了 DefaultPaletteCustomizer 接口的类。不要直接实现这个接口，需要创建 AbstractDefaultPaletteCustomizer 这个基类的子类。目前，这个类没提供任何功能，但 DefaultPaletteCustomizer 接口今后的版本会提供更多的能力让这个基类拥有一些合理的默认行为，所以最好继承，这样你的扩展会兼容将来的发布。

继承 AbstractDefaultPaletteCustomizer 类需要你实现方法 disablePaletteEntries()，此方法必须返回一个 PaletteEntry 的列表。对于每个默认的形状，通过将它对应的 PaletteEntry 值添加到你的列表中可以将其禁掉。注意如果你移除了默认集合中的形状，抽屉内没有剩余的形状了，那么那个抽屉整个就会从画板中被移除。如果你希望禁掉所有默认形状，只需将 PaletteEntry.ALL 添加到你的结果中。如例子，以下代码禁掉了画板中的手工任务和脚本任务形状。

```
public class MyPaletteCustomizer extends AbstractDefaultPaletteCustomizer {

    /**
     * (non-Javadoc)
     *
     * @see org.activiti.designer.integration.palette.DefaultPaletteCustomizer#disablePaletteEntries()
     */
    @Override
    public List<PaletteEntry> disablePaletteEntries() {
        List<PaletteEntry> result = new ArrayList<PaletteEntry>();
        result.add(PaletteEntry.MANUAL_TASK);
        result.add(PaletteEntry.SCRIPT_TASK);
        return result;
    }
}
```

应用这个扩展的结果显示为如下图片。正如你看到的，手工任务和脚本任务形状不再在任务抽屉中了。

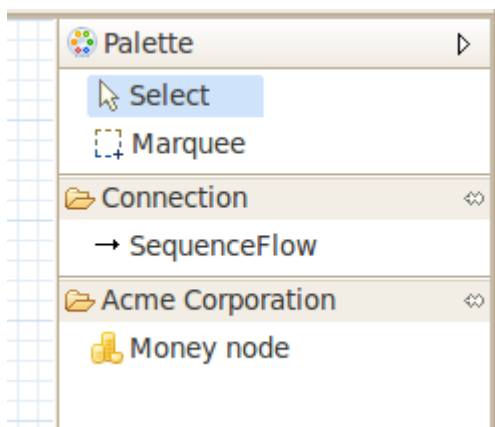


要想禁掉所有默认的形状，可以使用类似于如下的代码：

```
public class MyPaletteCustomizer extends AbstractDefaultPaletteCustomizer {

    /**
     * (non-Javadoc)
     *
     * @see org.activiti.designer.integration.palette.DefaultPaletteCustomizer#disablePaletteEntries()
     */
    @Override
    public List<PaletteEntry> disablePaletteEntries() {
        List<PaletteEntry> result = new ArrayList<PaletteEntry>();
        result.add(PaletteEntry.ALL);
        return result;
    }
}
```

结果看起来就像这样（注意默认形状的那些抽屉不再显示在画板中了）：



11.5.2 校验图形和导出到自定义的输出格式

除了定制画板，你还可以给 **Activiti Designer** 创建能执行校验、将来自图形的信息保存到 **Eclipse** 工作空间内的自定义资源的扩展。对此有内置扩展点，本节说明如何使用这些扩展点。

Activiti Designer 允许编写校验图形的扩展。默认工具中已经存在 **BPMN** 构造的校验了，但如果你想要对更多项进行校验，比如建模约定或 **CustomServiceTasks** 属性中的值，这时你就可以添加自己的校验了。这样的扩展被称为流程校验器。

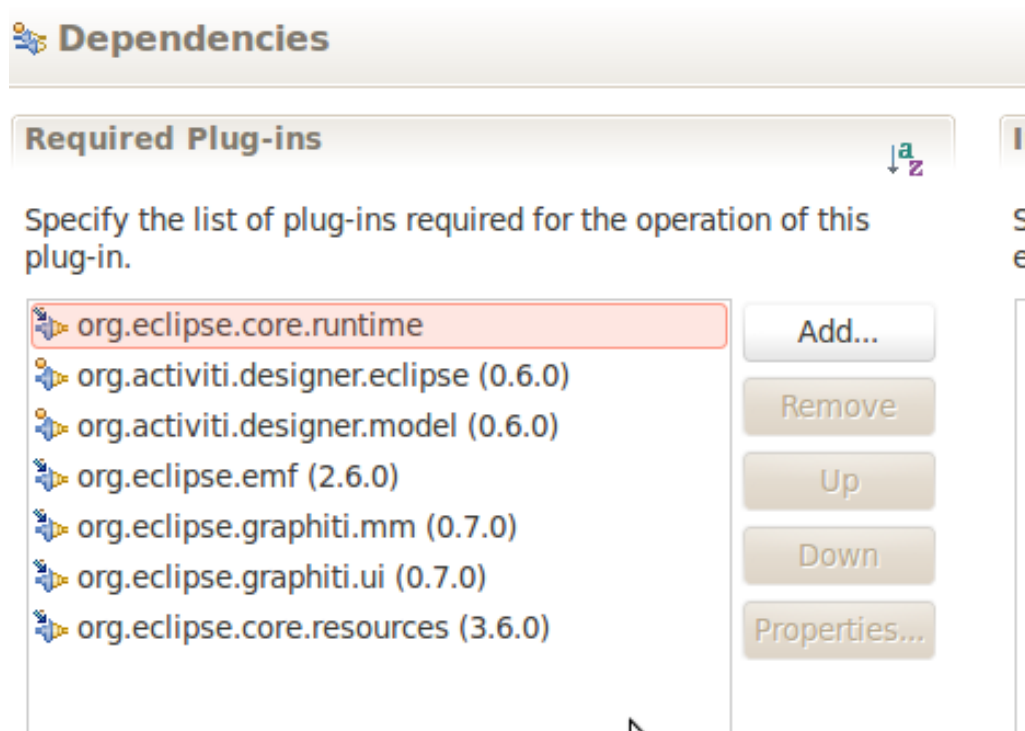
你也可以让 **Activiti Designer** 在保存图形时公布一些额外的格式。这样的扩展被称为输出装配器，每次用户保存时都会由 **Activiti Designer** 自动调用。通过在 **Eclipse** 设置对话框中对保存格式进行设置来启用或禁用这一个行为。

你可以将这两种扩展与 **BPMN 2.0** 的校验、**BPMN 2.0** 的输出以及 **Activiti Designer** 默认在保存时保存的流程图片进行比较。事实上，这些功能使用了相同的扩展特性，你可以将其用于你自己的格式的保存中。

经常，会想将 **ProcessValidator** 和 **ExportMarshaller** 进行联合。假如你有几个 **CustomServiceTask** 在用，其属性你想要在生成的流程中使用。然而，在流程产生出来之前，你想首先校验其中一些值。结合 **ProcessValidator** 和 **ExportMarshaller** 是完成这的最好方法，**Activiti Designer** 能够将你的扩展无缝隙地插入到这一工具内。

要创建 **ProcessValidator** 或 **ExportMarshaller**，需要创建与扩展画板所不同的扩展。原因很简单：在你的代码中，需要访问更多由集成类库提供的 **API**。特别是，你会使用 **Eclipse** 本身中的类。所以要开始，必须先创建一个 **Eclipse** 插件（利用 **Eclipse** 对 **PDE** 的支持来完成），然后将它打包在一个自定义的 **Eclipse** 产品或特性中。解释开发 **Eclipse** 插件的所涉及的所有详细信息超出了本用户指南的范畴，所以以下说明只局限于扩展 **Activiti Designer** 功能。

你的模块必须依赖以下类库：



ProcessValidators 和 **ExportMarshallers** 都是通过继承基础类来创建的。这些基础类从其超类 **AbstractDiagramWorker** 那里继承了一些很有用的方法。利用这些方法可以创建显示在 **Eclipse** 中 **problems** 视图内帮助用户找出问题或要点的消息、警告

以及错误标记。也可以利用 `AbstractDiagramWorker` 中的这些方法通过 `Resources` 和 `InputStreams` 访问图形的内容。

把调用 `clearMarkers()` 作为在 `ProcessValidators` 或 `ExportMarshallers` 内的第一件要做的事可能会是个好主意；这将清理掉所有之前操作人员的标记（标记自动链接到操作人员，清除一个操作人员的标记不会影响到其它标记）。例如：

```
// 首先为图形清除标记
clearMarkers(getResource(diagram.eResource().getURI()));
```

你还应该利用进度监控将进度情况报告给用户，因为校验和/或装配行为都会占用一些时间，在此期间用户被迫等待。报告进度的情况需要一些关于如何使用 Eclipse 特性的知识。仔细看一下[这篇](#)深入解释了概念和用法文章。

11.5.2.1 创建 `ProcessValidator` 扩展

在 `plugin.xml` 文件内给 `org.activiti.designer.eclipse.extension.validation.ProcessValidator` 扩展点创建扩展。需要为此扩展点创建 `AbstractProcessValidator` 类的子类。

```
<?eclipse version="3.6"?>
<plugin>
<extension
point="org.activiti.designer.eclipse.extension.validation.ProcessValidator">
<ProcessValidator
class="org.acme.validation.AcmeProcessValidator">
</ProcessValidator>
</extension>
</plugin>
```

```
publicclass AcmeProcessValidator extends AbstractProcessValidator {
}
```

必须实现几个方法。最重要的是，实现 `getValidatorId()`，为你的校验器返回一个全局唯一的 ID。这让你能在 `ExportMarshaller` 内对其进行调用，甚至可以让其他人在他们的 `ExportMarshaller` 内调用你的校验器。实现 `getValidatorName()`，返回校验器的逻辑名。这个名称在对话框中显示给用户。在 `getFormatName()` 中，可以返回校验器通常校验的图形的类型。

校验本身是在 `validateDiagram()` 方法内进行的。以此来看，根据你的具体功能在这里放什么样的代码。但通常在开始你会取得图形的流程节点，这样你就可以循环遍历它们，收集、对比并校验数据。本代码片段展示给你如何进行这些操作：

```
final EList<EObject> contents = getResourceForDiagram(diagram).getContents();
for (final EObject object : contents) {
if (object instanceof StartEvent) {
// 对StartEvents执行某些校验
}
// 其它的节点类型和校验
}
```

在你完成校验后，不要忘记调用 `addProblemToDiagram()` 和/或 `addWarningToDiagram()`，等等。务必最后返回正确的布尔类

型的结果值来表明你考虑的校验是成功还是失败。这个结果可以被调用 `ExportMarshaller` 使用来决定下一步操作。

11.5.2.2 创建 `ExportMarshaller` 扩展

在 `plugin.xml` 文件内给 `org.activiti.designer.eclipse.extension.export.ExportMarshaller` 扩展点创建扩展。需要为此扩展点创建 `AbstractExportMarshaller` 类的子类。这个抽象基类提供了在编组到你自己的格式时所使用的几个有用的方法，但最重要的是它允许你将资源保存到工作空间内，还允许调用校验器。

```
<?eclipse version="3.6"?>
<plugin>
<extensionpoint="org.activiti.designer.eclipse.extension.export.ExportMarshaller">
<ExportMarshallerclass="org.acme.export.AcmeExportMarshaller">
</ExportMarshaller>
</extension>
</plugin>
```

```
publicclass AcmeExportMarshaller extends AbstractExportMarshaller {
}
```

需要你来实现某些方法，如 `getMarshallerName()` 及 `getFormatName()`。这些方法用来向用户显示选项以及在进度对话框中显示信息，因此要确保你所描述的能表达出你实现的功能。

大部分你的工作是在 `marshallDiagram(Diagram diagram, IProgressMonitor monitor)` 方法内进行的。给你提供了 `diagram` 对象，它包含着有关图形（BPMN 的构造）和图示中对象的所有信息。

如果想先执行某个校验，可以直接在你的装配器内调用校验器。从校验器接收到一个布尔类型的结果值，这样你就能知道校验是否成功。大多数情况下，如果校验无效是不会对图形进行编组的，但你也可以选择继续，甚至校验失败后，创建一个不同的资源。比如：

```
finalboolean validDiagram = invokeValidator(AcmeConstants.ACME_VALIDATOR_ID, diagram, monitor);
if (!validDiagram) {
    addProblemToDiagram(diagram, "Marshalling to " + getFormatName() + " format was skipped because validation of the
diagram failed.", null);
} else {
    //进行编组
}
```

如你所见，如果校验器（这里以常量标识）返回的结果为 `false`，在这里我们选择取消编组。同时也向图形添加了额外的标记，这样用户就能看到文件为什么没被创建的解释了。这并不是必须的，但这似乎对用户是很有用的，并且这展示了在 `ProcessValidators` 和 `ExportMarshallers` 中如何使用这些工具。

一旦获得了所有你所需要的数据，就可以调用 `saveResource()` 方法来创建包含着你的数据的文件了。可以在单个 `ExportMarshaller` 内多次调用 `saveResource()`，所以装配器可用来创建多个输出文件。

利用 `AbstractDiagramWorker` 类中一些的方法可以给输出的资源构造文件名。有几个你已经解析过了的有用的变量，允许

你来创建像<original-filename>_<my-format-name>.xml 的文件名。Javadocs 内有对这些变量的描述，这是一个使用了其中一个变量的例子：

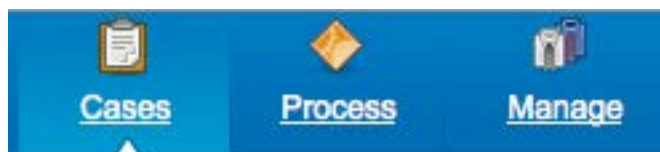
```
private static final String FILENAME_PATTERN = ExportMarshaller.PLACEHOLDER_ORIGINAL_FILENAME + ".acme.xml";  
...  
saveResource(getRelativeURIForDiagram(diagram, FILENAME_PATTERN), bais, this.monitor);
```

这里所发生的是：使用了静态的成员变量来描述文件名模式（这只是最佳方式，你当然可以按照任何你喜欢的方式来指定这个字符串了），并且模式使用了 `ExportMarshaller.PLACEHOLDER_ORIGINAL_FILENAME` 常量给原始文件名插入了一个变量。接下来在 `marshallDiagram()` 方法内，调用了 `getRelativeURIForDiagram()`，它会针对任何变量来解析文件名并替换这些变量（译注，这句话的意思是说如果该方法的参数中含有变量，那么会将变量替换为其表示的实际值，而后再做解析，这是很显然的）。给 `saveResource()` 提供一个到你数据的 `InputStream`，该方法会将数据保存到相对路径为此原始图形的资源中。

当然，你同样应该利用进度监控将进度情况报告给用户。[这篇文章](#)描述了如何来完成。

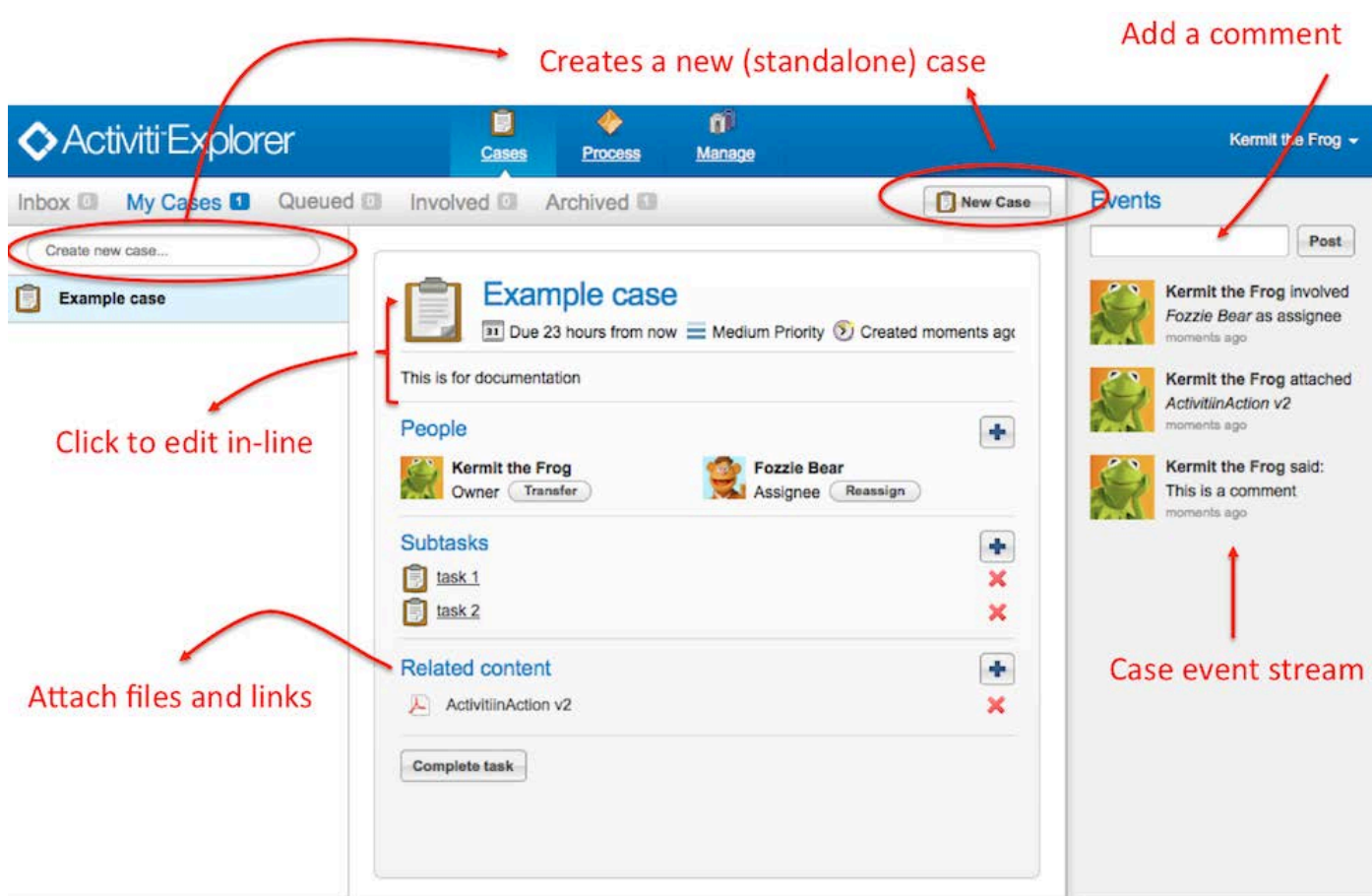
第十二章、Activiti Explorer

Activiti Explorer 是一个 web 应用程序，是在[演示设置](#)中安装的。登进系统后，你会看到显示主要功能的 3 个大图标。



- **Cases:** 用例和任务的管理功能。在此你可以看到运行流程中分配到你的用户任务，或者看到的是你可以认领的组任务。注意，在 Explorer 中我们讲到用例（cases）而不是任务（tasks）。Explorer 允许关联内容、将工作划分成子任务、牵涉不同角色的人员，等等，所以术语“用例”更能涵盖这些可能。Activiti Explorer 就其本身而言也可以用于创建不与流程关联的用例（如，独立的用例）。
- **Process:** 显示部署了的流程定义，允许启动新的流程实例。
- **Manage:** 只对登录的拥有管理员权限的用户可见。允许管理 Activiti 引擎：管理用户和组、执行并查看停住的作业、查看数据库以及部署新的流程定义。

12.1 用例概述



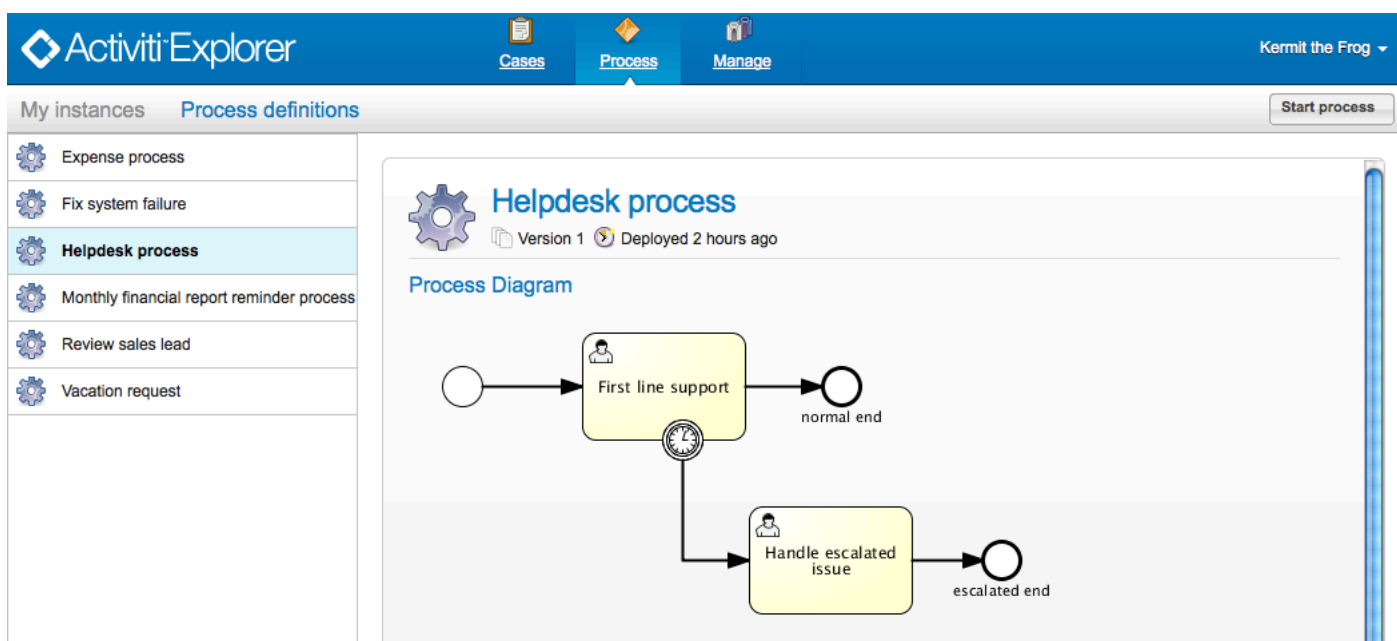
12.2 用例



- **Inbox:** 显示登录用户为责任人的用例。
- **My Cases:** 显示登录用户为所有者的用例。当你创建一个独立的用例，你自动会成为该用例的所有者。
- **Queued:** 显示了你是其中一员的那些组。用例在完成之前，必须在此先要被认领。
- **Involved:** 显示登录用户所牵涉到的用例（如，非责任人或所有者）。
- **Archived:** 含有过去的（有历史记录的）用例。

12.3 启动流程实例

Process definition 标签允许查看所有部署到 Activiti 引擎的流程定义。你可以使用右上角的按钮启动新的流程实例。如果流程定义有启动[表单](#)，那么表单会在启动流程实例之前显示。



12.4 我的实例

My instances 标签显示了你当前所拥有的未完成的用户任务所在的流程实例。它也形象的显示出了流程实例当前的活动以及存储的流程变量。

My instances

Process definitions

Helpdesk process (139)

Expense process (144)

Review sales lead (148)

Vacation request (168)

Expense process (144)

Started 2 minutes ago

Process Diagram

Request expense ref...

Handle expense requ...

Tasks

	NAME	PRIORITY	ASSIGNEE	DUE DATE	CREATED	COMPLETED
	Request expense refund	50	Kermit the Frog		2 minutes ago	

Variables

NAME	VALUE
initiator	kermit

12.5 管理

管理功能只有当登录的用户是安全组 *admin* 中的一员时可见。当点击 *Manager* 图标时，以下标签可见：

- **Database:** 显示了数据库的内容。在开发流程或排查问题时非常有用。

Database

Deployments

Jobs

Users

Groups

ACT_HI_DETAIL (8)

ACT_HI_PROCINST (5)

ACT_HI_TASKINST (12)

ACT_ID_GROUP (8)

ACT_ID_INFO (6)

ACT_ID_MEMBERSHIP (12)

ACT_ID_USER (3)

ACT_RE_DEPLOYMENT (2)

ACT_RE_PROCDEF (6)

ACT_RU_EXECUTION (8)

ACT_RU_IDENTITYLINK (2)

ACT_RU_JOB (1)

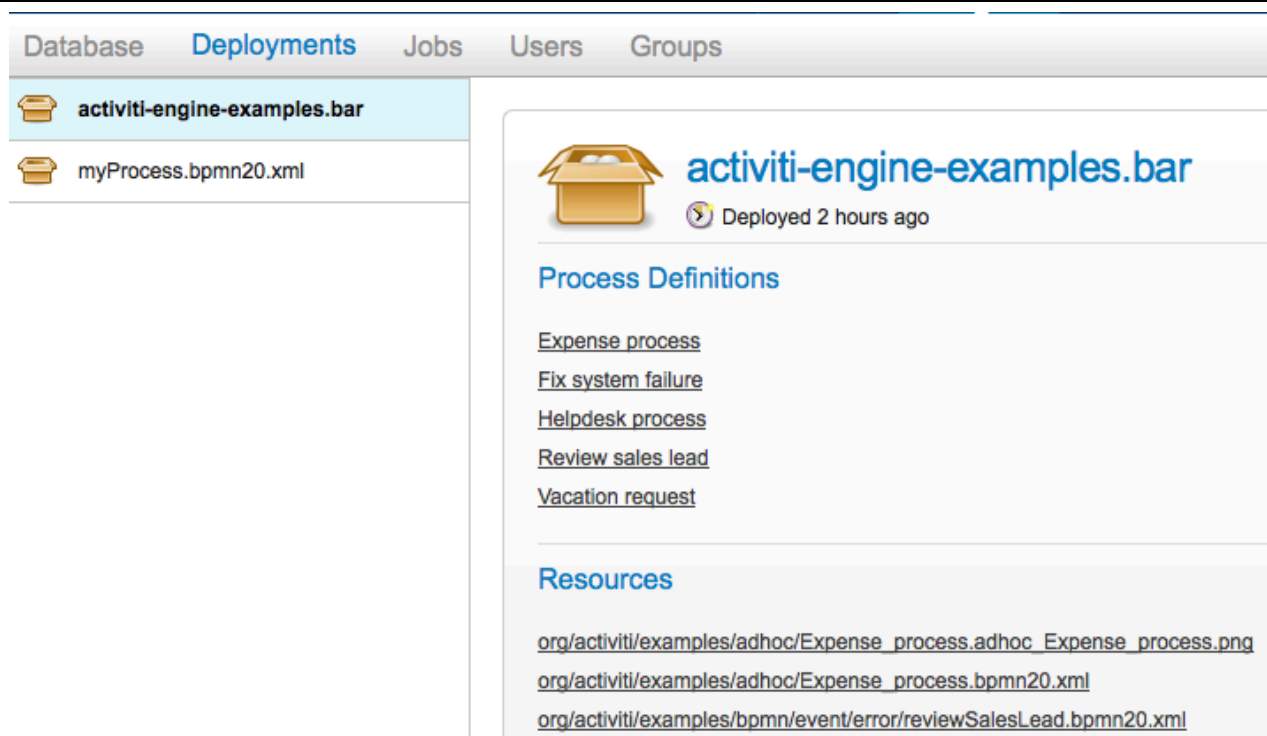
ACT_RU_TASK (8)

ACT_RU_VARIABLE (11)

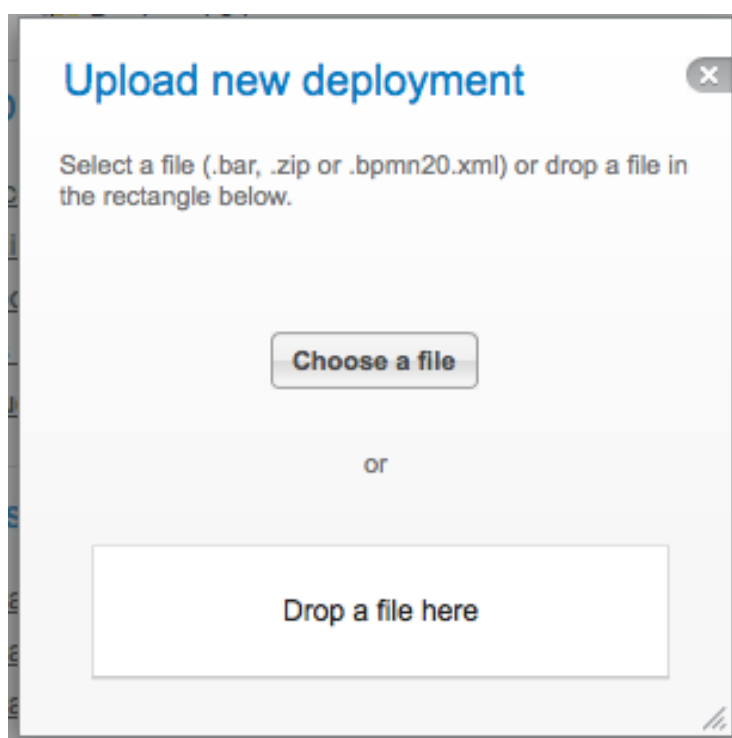
ACT_RU_VARIABLE

ID_	REV_	TYPE_	NAME_	EXECUTION_ID_	PROC_INST_ID_
145	1	string	initiator	144	144
149	1	string	initiator	148	148
155	1	string	customerName	148	148
156	1	null	potentialProfit	148	148
157	1	string	details	148	148
169	1	string	employeeName	168	168
173	1	long	numberOfDays	168	168

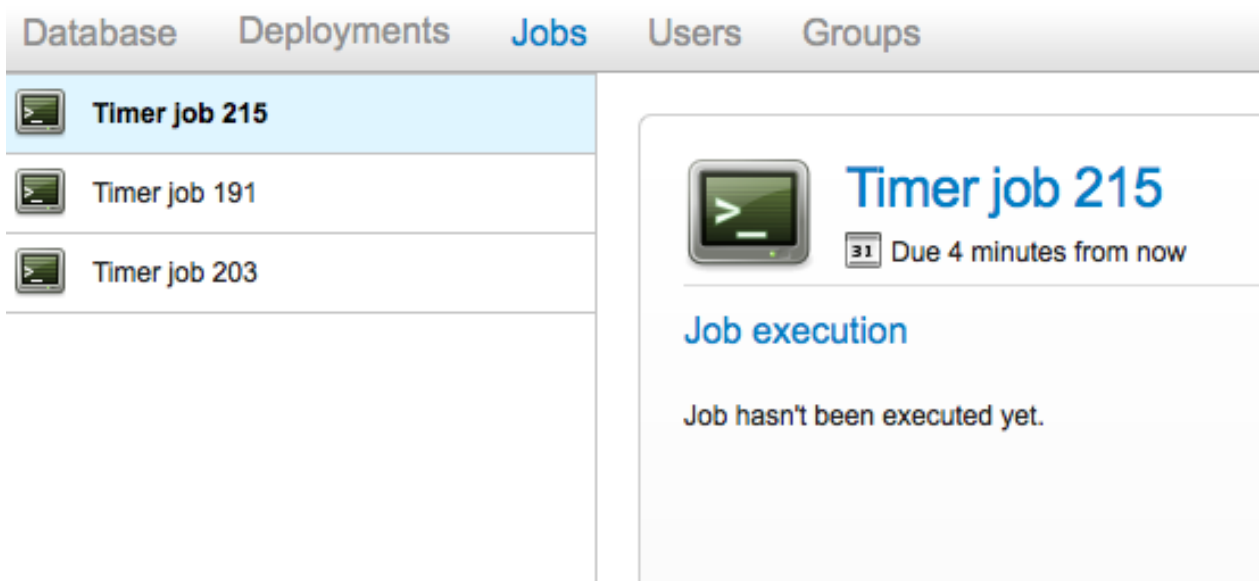
- **Deployments:** 显示了引擎当前的部署，可以看到部署的内容（流程定义、图片、业务规则，等等）。



点击 deployment 标签，可以上传新的部署。在计算机选择业务归档文件或 bpmn20.xml 文件，或者简单地拖拽到指定的区域来部署该新业务流程。



- **Jobs:** 在左侧显示了当前的作业（定时器，等），可以手动执行（例如，机制时间前触发定时器）。如果作业执行失败（例如，邮件服务器不能连接），也会显示出异常。



- **User and Groups:** 管理用户和组：创建、编辑以及删除用户和组。将用户与组进行关联，这样他们就会有更多的权限或者他们可以看到分配给特定组的用例。



12.6 修改数据库

要修改 Explorer 使用的演示设置中的数据库，需要修改属性文件 `apps/apache-timcat-6.x/webapps/activiti-explorer/WEB-INF/classes/db.properties`。同时，也要把匹配的数据库驱动放在类路径下（Tomcat 共享类库或 `apps/apache-timcat-6.x/webapps/activiti-explorer/WEB-INF/lib/`）。

第十三章、Activiti 的附加组件

13.1 Cycle

始于 Activiti Cycle，现在已经移入 [camunda fox BPM 平台](#)，[Cycle](#) 是个 web 应用程序，它为 BPM 项目的相关各方（企业、分析人员、开发人员、管理者、IT 运营...）提供了一个协作平台。它将不同的数据源，如 Activiti Modeler 仓库、Subversion 或者你本地的文件系统，结合到了一个单一的视图，这就使得浏览所有那些项目中涉及的事物（流程模型、部署项目、需求...）变得容易了。另外你可以保持它们之间的关系，并且 Cycle 提供了一些内置的操作，如两个库之间移动事物、下载不同格式的流程模型。提供了一个基础结构来连结自己的库、操作或功能。

Cycle 的概念，尤其是 Business-IT-Alignment 学问是新的。你最好拿它与 Application Lifecycle Management (ALM)工具进行一下比较。Cycle 是由 [camunda](#) 开发的，稳当可以在 [camunda fox 文档](#)内找到。

13.2 基于 Signavio 核心组件的 Activiti Modeler

Signavio 有个基于 BPMN 流程的 web 建模器。有一个为方便后台使用文件进行部署的改进版本。可以在 Google 的 [Signavio core components](#) 项目内找到。

在 wiki [How to build Activiti Modeler from Signavio](#) 中我们已经阐述了一些关于如何构建以及如何将其用于 Activiti 的用法。

第十四章、REST API

<试验性的>...整个 REST 接口还是试验性的。

Activiti 中包含了操作引擎的 REST API，它是在运行设置脚本时被部署到你的服务器内的。REST API 采用了 JSON 格式（<http://www.json.org>），建立在 Spring Webscript（<http://www.springsurf.org>）之上。

每个 REST API 的调用都有其各自的认证级别，必须以用户登录才能调用 REST API（除了登录的服务）。认证使用的是基本的 HTTP 认证，所以如果是管理员（比如 `kermit`）登录来浏览该 REST API，如上所述，就可以执行所有下面所描述的调用。

该 API 遵从常规的 REST API 约定，GET 用于读操作，POST 用于创建对象，PUT 用于已创建对象上的更新和执行操作，还有最后的 DELETE 用于删除对象。当执行的调用影响到了多个对象时，为保持一致性并确保没有数量限制的对象可能被使用到，POST 会被用在所有这些操作上。使用 POST 的原因是 HTTP 的 DELETE 方法暗含是不允许请求体的，因此，理论上，使用 DELETE 的调用可能会使代理将请求体剥离掉。所以按照一致性考虑，为确保此不发生，我们使用了 POST，即使是在可能已经使用了 PUT 来更新多个对象。

其余所有的调用都采用“application/json”的内容类型（除了使用“multipart/form-data”的上传请求）。

用于调用 REST 的基准 URL 是 <http://localhost:8080/activiti-rest/service/>。比如，要想列出引擎中的流程定义，将浏览器的指向 <http://localhost:8080/activiti-rest/service/process-engine>。

请查看以下看看目前有哪些 REST API 的调用可用。请将该“API”的各节视作是对用于实现 REST API 调用的核心 API 功能的“一行提示”。

14.1 仓库

14.1.1 上传部署

使用普通的“html 表单上传”（`enctype=multipart/form-data`）来上传并安装格式为 `.bpmn20.xml`、`.bar` 或 `.zip` 的部署，换句话说，并不是 json 请求。为了能让图形用户界面在上传完成时做出反应，可以以参数的形式提交 `success/failure` 回掉。即，发送“`success=alert`”会导致字符串“`alert`”被放到结尾追加了“`();`”的 js 脚本块中，也就是“`alert();`”。发送“`failure=alert`”将导致脚本块中的“`alert('Some error message if any');`”。实际的部署文件是放在参数“`deployment`”中的。

- 请求：POST /deployment

```
success={success}&failure={success}&deployment={file}
```

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().createDeployment().name(fileName).deploymentBuilder.deploy()
```

- 响应:

```
<html>
<script type="text/javascript">
    alert();
```

```
</script>
</html>
```

14.1.2 获取部署

返回部署的分页列表，其中可按“id”、“name”或“deploymentTime”来排序。

- **请求：** GET /deployments?start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}
- **API：**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().createDeploymentQuery().listPage()
- **响应：**

```
{
  "data": [
    {
      "id": "10",
      "name": "activiti-examples.bar",
      "deploymentTime": "2010-10-13T14:54:26.750+02:00"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

14.1.3 获取部署资源

返回部署中的资源。例如：/deployment/10/resource/org/activiti/examples/bpmn/usertask/FinancialReportProcess.bpmn20.xml。

- **请求：** GET /deployment/{deploymentId}/resource/{resourceName}
- **API：**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().getResourceAsStream(deploymentId, resourceName)
- **响应：**

即，.bpmn20.xml 文件，图片或部署资源所包含的任何类型的文件。

14.1.4 删除部署

删除部署。

- **请求：** DELETE /deployment/{deploymentId}?cascade={cascade?}
- **API：**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().deleteDeployment(deploymentId)

entId)

- 响应:

```
{
  "success": true
}
```

14.1.5 删除多个部署

删除多个部署。

- 请求: POST /deployments?cascade={cascade?}

```
{
  "deploymentIds": [ "10", "11" ]
}
```

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().deleteDeployment(deploymentId)
```

- 响应:

```
{
  "success": true
}
```

14.2 引擎

14.2.1 获取流程引擎

返回流程引擎初始的详细信息。如果启动时有错，出错的详细信息将在相应的“exception”属性中给出。

- 请求: GET /process-engine
- API: ProcessEngines.getProcessEngine(configuredProcessEngineName)
- 响应:

```
{
  "name": "default",
  "resourceUrl": "jar:file:V/<path-to-deployment>/activiti-cfg.jar!/activiti.properties",
  "exception": null,
  "version": "5.8"
}
```

14.3 流程

14.3.1 列出流程定义

返回有关部署的流程定义的详细信息，可按“id”、“name”、“version”或“deploymentTime”来排序。BPMN2.0 XML 流程图的名称是在“resourceName”属性给出的，结合“deploymentId”属性可以由上文中的获取部署资源的 REST API 调用得到这一名称。如果流程中有启动的表单，那么它是由“startFormResourceKey”属性给出的。流程的启动表单可以从获取启动流程表单的 REST API 调用来获取。

- **分页请求：** GET /process-definitions?start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}
- **API：**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getProcessService().createProcessDefinitionQuery().listPage()
- **分页响应：**

```
{
  "data": [
    {
      "id": "financialReport:1",
      "key": "financialReport",
      "version": 1,
      "name": "Monthly financial report",
      "resourceName": "org/activiti/examples/bpmn/usertask/FinancialReportProcess.bpmn20.xml",
      "deploymentId": "10",
      "startFormResourceKey": null
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

14.3.2 获得流程定义

返回关于部署的流程定义的详细信息。

- **请求：** GET /process-definition/{processDefinitionId}
- **API：**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getProcessService().createProcessDefinitionQuery().processDefinitionId(processDefinitionId)
- **响应：**

```
{
  "id": "financialReport:1",
  "key": "financialReport",
```

```

"version": 1,
"name": "Monthly financial report",
"resourceName": "org/activiti/examples/bpmn/usertask/FinancialReportProcess.bpmn20.xml",
"deploymentId": "10",
"startFormResourceKey": null
}

```

14.3.3 获得流程定义表单

返回流程定义的表单。

- **请求:** GET /process-definition/{processDefinitionId}/form[?format=html|json]
- **API:**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().getRenderedStartFormById(processDefinitionId)
- **响应:**

```
<user-defined-response>
```

14.3.4 启动流程实例

基于流程定义创建流程实例，返回关于新创建流程实例的详细信息。附加的变量（来自于表单）可以使用 **body** 对象来传递。换句话说，紧挨着“processDefinitionId”属性添加属性。

这些附加变量也可以使用“元数据字段”来描述。使用一个名称为“numberOfDays_type”设置为“Integer”的额外变量可以将值为“2”名称为“numberOfDays”的变量描述成整形；要想将其描述成必须的变量，需要使用一个名称为“numberOfDays_required”设置为“true”的额外变量。如果不使用到类型描述符，只要是被“”字符包围，其值就会被作为字符串来处理。也可以将类型设置为“Boolean”。

注意如果值提交的是 **true**（而不是“true”），它将被视为布尔类型的值，即便没有使用任何描述符。对于数值也同样有效，即，**123** 将作为整数，而“123”将作为字符串（除非定义描述符）。注意名称中包含“_”的变量是不会被保存的，它们只是被当作元数据的字段。

使用这些元数据字段的原因是为了可以利用标准的 **HTML** 表单来提交值（由于 **HTML** 表单都是以字符串来提交的，所以不可能在 **JSON** 中区分值的类型）。在不久的将来将会支持 **HTML** 的提交。这当然不是客户端向服务器发送关于哪些变量是必须的以及它们是什么类型的最佳方案，但确实是一个能处理简单表单的临时解决方案。目前，我们正在为表单寻找更为恰当的解决方案来包含那些可能会在服务器端使用到的真正的元数据模型，以避免像上文那样使用元数据字段。随时可以在 **Activiti** 论坛里给出建议或技巧。

- **请求:** POST /process-instance

```

{
  "processDefinitionId": "financialReport:1:1700",
  "businessKey": "order-4711"
}

```

- **API:**

ProcessEngines.getProcessEngine(configuredProcessEngineName).getRuntimeService().startProcessInstanceById(processDefinitionId[, businessKey][, variables])

- **响应:**

```
{
  "id": "217",
  "processDefinitionId": "financialReport:1:1700",
  "activityNames": ["writeReportTask"],
  "ended": true
}
```

- 提供 **processDefinitionKey** 而不是 **Id** 的请求: POST /process-instance

```
{
  "processDefinitionKey": "financialReport:1",
  "businessKey": "order-4711"
}
```

14.3.5 列出流程实例

返回关于活跃的流程实例的详细信息，可按“id”、“startTime”、“businessKey”或“processDefinition”进行排序。可以通过“processDefinitionId”或“businessKey”对实例进行过滤。

- **分页请求:** GET

/process-instances?start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}&businessKey={businessKey}&processDefinitionId={processDefinitionId}

- **API:**

ProcessEngines.getProcessEngine(configuredProcessEngineName).getHistoryService().createHistoricProcessInstanceQuery().xxx.listPages()

- **分页响应:**

```
{
  "data": [
    {
      "id": 2,
      "processDefinitionId": "financialReport:1",
      "businessKey": 55,
      "startTime": "12-03-2011",
      "startUserId": "kermit"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

14.3.6 获得流程实例图

返回存在显著活动执行的流程的 png 示意图。如果流程定义中没有 DI 信息就会返回 404。

- **请求:** GET /processInstance/{processInstanceId}/diagram
- **API:** ProcessDiagramGenerator.generateDiagram(pde, "png", getRuntimeService().getActiveActivityIds(processInstanceId));
- **请求:**

存在显著活动执行的流程的 png 示意图

14.4 任务

14.4.1 获取任务概述

返回特定用户的任务概述：分配给该用户任务的个数，用户可以认领多少未分配的任务，以及该用户所在的每个组所拥有多少未分配任务。

- **请求:** GET /tasks-summary?user={userId}
- **API:** ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().createTaskQuery().xxx().count()
- **响应:**

```
{
  "assigned": {
    "total": 0
  },
  "unassigned": {
    "total": 1,
    "groups": {
      "accountancy": 1,
      "sales": 0,
      "engineering": 0,
      "management": 0
    }
  }
}
```

14.4.2 列出任务

返回任务的分页列表，其中可以按"id", "name", "description", "priority", "assignee", "executionId"或"processInstanceId"进行排序。该列表必须基于特定角色的用户：代理人（列出了分配给用户的任务）或候选者（列出了用户可以认领的任务）

或候选组（列出了组中成员可以认领的任务）。如果任务中存在表单，那么是在"formResourceKey"属性给出的。任务的表单可以从获取任务表单的 REST API 调用获得。

- **分页请求：** GET
/tasks?[assignee={userId}]candidate={userId}|candidate-group={groupId}]&start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}
- **API:**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().createTaskQuery().xxx().listPage()
- **分页响应：**

```
{
  "data": [
    {
      "id": 127,
      "name": "Handle vacation request",
      "description": "Vacation request by Kermit",
      "priority": 50,
      "assignee": null,
      "executionId": 118,
      "formResourceKey": "org/activiti/examples/taskforms/approve.form"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

14.4.3 获得任务

返回关于特定任务 id 对应任务的详细信息。

- **请求：** GET /task/{taskId}
- **API:**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().createTaskQuery().taskId(taskId).singleResult()
- **响应：**

```
{
  "id": 127,
  "name": "Handle vacation request",
  "description": "Vacation request by Kermit",
  "priority": 50,
  "assignee": null,
  "executionId": 118,
  "formResourceKey": "org/activiti/examples/taskforms/approve.form"
}
```



```
}
```

14.4.4 获得任务表单

返回任务的表单。

- **请求:** GET /task/{taskId}/form
- **API:** ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().getRenderedTaskForm(taskId)
- **响应:**

```
<user-defined-response>
```

14.4.5 执行任务操作

执行任务上的操作（认领或完成）。对于“完成”操作，体内可以传递附加的变量（表单中）。要想阅读更多关于表单中附加变量的内容，请访问启动流程实例一节。

- **请求:** PUT /task/{taskId}/[claim|complete]

```
{}
```

- **API:** ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().xxx(taskId ...)
- **响应:**

```
{
  "success": true
}
```

14.4.6 列出表单属性

返回流程内定义的运行着的任务的表单的属性列表。

- **请求:** GET /form/{taskId}/properties
- **API:**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getFormService().getTaskFormData(taskId).getFormProperties()
- **响应:**

```
{
  "data": [
    {
      "id": "userName",
      "name": "User",
      "value": "foobar",
      "type": "string",
      "required": "true",
      "readable": "true",
      "writable": "true"
    }
  ]
}
```

```
}  
}
```

14.5 身份

14.5.1 登陆

对用户进行认证。如果用户和密码与请求不匹配，就会返回 403 状态码。如果认证成功，会返回状态码 200 的响应。

- **请求:** POST /login

```
{  
  "userId": "kermit",  
  "password": "kermit"  
}
```

- **API:** `ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().checkPassword(userId, password)`
- **响应:**

```
{  
  "success": true  
}
```

14.5.2 获得用户

返回关于用户的详细信息。

- **请求:** GET /user/{userId}
- **API:**
`ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().createUserQuery().userId(userId).singleResult();`
- **响应:**

```
{  
  "id": "kermit",  
  "firstName": "Kermit",  
  "lastName": "the Frog",  
  "email": "kermit@server.com"  
}
```

14.5.3 列出用户的组

返回用户所在组的分页列表，其中可按" id", "name"或" type"进行排序。要想只获得某类型的组，需要使用" type"参数。

- **分页请求:** GET
`/user/{userId}/groups[?type=groupType]?start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}`

- **API:** `ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().xxx(userId[, groupId])`
- 分页响应:

```
{
  data: [
    {
      "id": "admin",
      "name": "System administrator",
      "type": "security-role"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

14.5.4 获取组

返回组的详细信息。

- 请求: `GET /group/{groupId}`
- **API:**
`ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().createGroupQuery().groupId(groupId).singleResult();`
- 响应:

```
{
  "id": "admin",
  "name": "System administrator",
  "type": "security-role"
}
```

14.5.5 列出组内的用户

返回组中用户的详细信息，可按"id", "firstName", "lastName"或"email"进行排序。

- 分页请求: `GET /groups/{groupId}/users`
- **API:**
`ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().createUserQuery().memberOfGroup(groupId).list()`
- 分页响应:

```
{
  data: [
    {
      "id": "kermit",
```

```

"firstName": "Kermit",
"lastName": "the Frog",
"email": "kermit@server.com"
  }
],
"total": 1,
"start": 0,
"sort": "id",
"order": "asc",
"size": 1
}

```

14.6 管理

14.6.1 列出作业

返回作业的分页列表，可按"id", "process-instance-id", "execution-id", "due-date", "retries"或者任何某个自定义属性的 id 对其进行排序。该列表也可以通过流程实例的 id 进行过滤，或者如果作业被重操作过，或它们是可执行的或者它们只有消息或定时器，这时也可以通过到期时间对其进行过滤。

- **分页请求：** GET
/management/jobs?process-instance={processInstanceId?}&with-retries-left={withRetriesLeft=false}&executable={executable=false}&only-timers={onlyTimers=false}&only-messages={onlyMessage=false}&duedate-lt={iso8601Date}&duedate-ltoe={iso8601Date}&duedate-ht={iso8601Date}&duedate-htoe={iso8601Date}&start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}
- **API：** ProcessEngines.getProcessEngine(configuredProcessEngineName).createJobQuery().xxx().listPage()
- **分页响应：**

```

{
  "data": [
    {
      "id": "212",
      "executionId": "211",
      "retries": -1,
      "processInstanceId": "210",
      "dueDate": null,
      "assignee": null,
      "exceptionMessage": "Can't find scripting engine for `groovy`"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}

```

```
}
```

14.6.2 获得作业

返回作业的详细信息。

- **请求:** GET /management/job/{jobId}
- **API:** ProcessEngines.getProcessEngine(configuredProcessEngineName).createJobQuery().id(jobId).singleResult()
- **响应:**

```
{
  "id": "212",
  "executionId": "211",
  "retries": -1,
  "processInstanceId": "210",
  "dueDate": null,
  "assignee": null,
  "exceptionMessage": "Can't find scripting engine for 'groovy'",
  "stacktrace": "org.activiti.engine.ActivitiException: Can't find scripting engine for 'groovy'\n\tat ..."
```

14.6.3 执行作业

执行作业。

- **请求:** PUT /management/job/{jobId}/execute
- **API:** ProcessEngines.getProcessEngine(configuredProcessEngineName).getManagementService().executeJob(jobId)
- **请求:**

```
{
  "success": true
}
```

14.6.4 执行多个作业

执行多个作业。

- **请求:** POST /management/jobs/execute

```
{
  "jobIds": [ "212" ]
}
```

- **API:**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getManagementService().executeJob(jobId)
- **响应:**

```
{
```

```
"success": true
}
```

14.6.5 列出数据库表

返回引擎内所有数据库表的元数据信息。

- **请求:** GET /management/tables
- **API:** ProcessEngines.getProcessEngine(configuredProcessEngineName).getManagementService().getTableCount()
- **响应:**

```
{
  "data": [
    {
      "tableName": "ACT_GE_PROPERTY",
      "noOfResults": 2
    }
  ]
}
```

14.6.6 获得表的元数据

返回数据库表的元数据。

- **请求:** GET /management/table/{tableName}
- **API:**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getManagementService().getTableMetaData(tableName)
- **响应:**

```
{
  "tableName": "ACT_GE_PROPERTY",
  "columnNames": ["REV_", "NAME_", "VALUE_"],
  "columnTypes": ["class java.lang.Integer", "class java.lang.String", "class java.lang.String"]
}
```

14.6.7 获得表数据

返回数据库表数据的分页列表。

- **分页请求:** GET /management/table/{tableName}/data
- **API:**
ProcessEngines.getProcessEngine(configuredProcessEngineName).getManagementService().createTablePageQuery().tableName(tableName).start(start).size(size).orderXXX(sort).singleResult();
- **分页响应:**

```
{
```

```
"data": [  
  {  
    "NAME_": "schema.version",  
    "REV_": "1",  
    "VALUE_": "5.4"  
  },  
  {  
    "NAME_": "next.dbid",  
    "REV_": "4",  
    "VALUE_": "310"  
  }  
],  
"total": 2,  
"start": 0,  
"sort": "NAME_",  
"order": "asc",  
"size": 2  
}
```

整个 REST 接口还是试验性的...[<试验性的>](#)。

第十五章、Cdi 集成

Activiti-cdi 利用了 Activiti 的可配置性以及 cdi 的可扩展性。Activiti-cdi 最突出的特性是：

- 支持@BusinessProcessScoped 注解的 beans（Cdi beans 的生命周期是与流程实例绑定的），
- 用来解析流程中 Cdi beans（包括 EJBs）的自定义 EI 解析器，
- 利用注解对流程实例进行声明式地控制，
- Activiti 是与 cdi 事件流紧密联系的，
- 可以运行在 Java EE 和 Java SE 下，可以运行于 Spring 下，
- 支持单元测试。

请注意 Activiti-cdi 模块被认为是 Activiti 的<试验性的>特性。要将 Activiti-cdi 包含进 maven 项目，我们需要添加如下 maven 的依赖：

```
<dependency>
<groupId>org.activiti</groupId>
<artifactId>activiti-cdi</artifactId>
<version>5.x</version>
</dependency>
```

将‘x’替换为你所使用的 Activiti 版本（>=5.6）。这样就会引入 Activiti 引擎以及 Spring。

15.1 设置 activiti-cdi

Activiti cdi 可以在不同的环境下设置。这一节我们会简略地过一下配置项。

15.1.1 查找流程引擎

Cdi 扩展需要访问 ProcessEngine。对这个扩展来说，会在运行时查找接口 org.activiti.cdi.impl.ProcessEngineLookup 的实现。

Cdi 模块还有以下@Alternative 的实现，需要在 beans.xml 中启用其中的一个：

- org.activiti.cdi.impl.LocalProcessEngineLookup：这个实现利用 ProcessEngines 工具类来查找 ProcessEngine。在默认配置中，ProcessEngines.NAME_DEFAULT 用来查找 ProcessEngine。这个类可以被继承以设置自定义名称。如果我们想要在应用中构建并管理我们自己的 ProcessEngine，可以启用这个实现。注意：类路径下要有 activiti.cfg.xml 配置。
- org.activiti.cdi.impl.JndiProcessEngineLookup：利用 InitialContext，在 Jndi 中查找 ProcessEngine。默认使用字符串 "activiti/default"来查找 ProcessEngine。这个类可以被继承用来设置自定义的 jndi 名称。
- org.activiti.cdi.test.ProcessEngineLookupForTestsuite：为测试用例构建 ProcessEngine，其对 Cdi 的 beans 是可用的。

"beans.xml"中 LocalProcessEngineLookup 可以按如下方式启用：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
<alternatives>
<class>org.activiti.cdi.impl.LocalProcessEngineLookup</class>
</alternatives>
```



```
</beans>
```

15.1.2 配置流程引擎

配置取决于所选的 `ProcessEngine` 查找策略（转上一节）。在此，我们会把重点放在结合 `LocalProcessEngineLookup` 可用的配置选项，这要求我们在类路径下提供 Spring 的 `activiti.cfg.xml` 文件。

Activiti 提供了几个不同的 `ProcessEngineConfiguration` 的实现，大部分是依赖底层事务管理策略的。Activiti-cdi 模块并不关心事务，这意味着潜在地可以使用任何事务管理策略（甚至是 Spring 事务的抽象）。为了方便起见，Cdi 模块提供了两个自定义的 `ProcessEngineConfiguration` 实现：

- `org.activiti.cdi.CdiJtaProcessEngineConfiguration`：Activiti 中 `JtaProcessEngineConfiguration` 的子类，如果 Activiti 需要使用 Jta 管理的事务，那么可以使用这个类。
- `org.activiti.cdi.CdiStandaloneProcessEngineConfiguration`：Activiti 中 `StandaloneProcessEngineConfiguration` 的子类，如果 activiti 需要使用 Jdbc 事务，那么可以使用这个类。

下面例子中的 `activiti.cfg.xml` 文件使用了外部管理的 Jta 事务，查找默认的 Jboss 数据源：

```
<?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">

  <!-- 查找JTA事务管理器 -->
  <bean id="transactionManager" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:/TransactionManager"></property>
    <property name="resourceRef" value="true"/>
  </bean>

  <!-- 流程引擎配置 -->
  <bean id="processEngineConfiguration" class="org.activiti.cdi.CdiJtaProcessEngineConfiguration">
    <!-- lookup the default Jboss datasource -->
    <property name="dataSourceJndiName" value="java:/DefaultDS"/>
    <property name="databaseType" value="h2"/>
    <property name="transactionManager" ref="transactionManager"/>
    <!-- using externally managed transactions -->
    <property name="transactionsExternallyManaged" value="true"/>
    <property name="databaseSchemaUpdate" value="true"/>
  </bean>
</beans>
```

（注意以上配置需要“spring-context”模块。）Java SE 环境下的配置与[创建流程引擎](#)一节中所提供的例子看起来是完全一样的，只不过是使用“`CdiStandaloneProcessEngineConfiguration`”替换了“`StandaloneProcessEngineConfiguration`”。

15.1.3 部署流程

可以使用标准 Activiti api(RepositoryService)来部署流程。另外，Activiti-cdi 提供了对位于顶层类路径下名称为 processes.xml 文件内所列出的流程进行自动部署的可能：

```
<?xml version="1.0" encoding="utf-8"?>
<!-- 列出部署的流程-->
<processes>
    <process resource="diagrams/myProcess.bpmn20.xml"/>
    <process resource="diagrams/myOtherProcess.bpmn20.xml"/>
</processes>
```

15.2 存在 Cdi 的上下文相关的流程的执行

在这节我们会简略地看一下 Activiti cdi 扩展所使用的上下文相关的流程执行模型。bpmn 业务流程通常会是一个由用户和系统任务构成的长时间运行的交互。运行时，流程被拆分为一系列的由用户和应用逻辑来执行的独立工作单元。Activiti-cdi 中，流程实例可以关联到 cdi 的作用域，这种关联表示的是工作单元。这是极其有用的，如果个工作单元很复杂，比如如果用户任务的实现是由各种不同表单组成的一个复杂序列，且“非流程作用域的”状态需要在交互期间进行维护。默认的配置中，流程实例所关联的是一个“很宽”的活跃作用域，起于会话，如果会话上下文不再活跃了，会撤回该请求。

15.2.1 将会话与流程实例关联

当解析 @BusinessProcessScoped 注解的 beans 或注入流程变量时，我们依赖了 Activiti cdi 作用域与流程实例已有的关联。Activiti-cdi 提供了 org.activiti.cdi.BusinessProcess bean 来控制这一关联，特别是：

- startProcessBy*(...)方法，映射 Activiti RuntimeService 公布的各个方法，允许启动以及随后关联业务流程；
- resumeProcessById(String processInstanceId)，允许与给定 id 的流程实例进行关联；
- resumeTaskById(String taskId)，允许与给定 id 的任务进行关联（引申起来就是相应的流程实例）。

一旦工作单元（例如用户任务）完成，就可以调用 completeTask()方法来取消会话/请求与流程实例的关联。这就会通知 Activiti 当前任务已经完成，使流程实例继续向下执行。

注意 BusinessProcess bean 是 @Named 注解的 bean，这意味着公布的方法可以使用表达式语言进行调用，如在 Jsf 页面中。以下的 Jsf2 片段开启了一个新会话，并将其与流程实例进行了关联，流程实例的 id 是作为请求参数传递的（例如 pageName.jsf?processId=XX）：

```
<f:viewParam name="processId"/>
<f:event type="preRenderView" listener="#{javax.enterprise.context.conversation.begin()}" />
<f:event type="preRenderView" listener="#{businessProcess.resumeProcessById(processId)}" />
```

15.2.2 声明式地控制流程

Activiti-cdi 允许使用注解来声明式地启动流程实例和完成任务。`@org.activiti.cdi.annotation.StartProcess` 注解允许通过“key”或“name”来启动流程实例。注意流程实例是在被注解的方法返回后启动的。

举例：

```
@StartProcess("authorizeBusinessTripRequest")
public String submitRequest(BusinessTripRequest request) {
    // 做些工作
    return "success";
}
```

取决于 Activiti 的配置，被注解的方法的代码与流程实例的启动将合并到同一个事务中。

`@org.activiti.cdi.annotation.CompleteTask` 注解以同样的方式工作：

```
@CompleteTask(endConversation = false)
public String authorizeBusinessTrip() {
    // 做些工作
    return "success";
}
```

`@CompleteTask` 注解提供了结束当前会话的可能。默认的行为是在 Activiti 调用返回后才结束会话。结束会话的行为是可以取消的，如上面例子中展示的。

15.2.3 在流程中引用 Bean

Activiti-cdi 使用了自定义的解析器对 Activiti EI 进行公布 cdi 的 beans。这就使流程中可以引用 bean 了：

```
<userTask id="authorizeBusinessTrip" name="Authorize Business Trip"
    activiti:assignee="#{authorizingManager.account.username}"/>
```

其中“authorizingManager”也可以由一个生产者方法来提供。

```
@Inject @ProcessVariable Object businessTripRequesterUsername;

@Produces
@Named
public Employee authorizingManager() {
    TypedQuery<Employee> query = entityManager.createQuery("SELECT e FROM Employee e WHERE e.account.username="
        + businessTripRequesterUsername + "'", Employee.class);
    Employee employee = query.getSingleResult();
    return employee.getManager();
}
```

我们可以利用同样的特性使用 `activiti:expression="myEjb.method()"` 扩展在服务任务中调用 EJB 的业务方法。注意，MyEjb 类上要有 `@Named` 注解。

15.2.4 使用@BusinessProcessScoped 注解的 bean

利用 Activiti-cdi，bean 的生命周期可以绑定到流程实例上。对于这个扩展，提供了一个自定义的上下文，即 BusinessProcessContext。BusinessProcessScoped beans 的实例作为流程变量存储在当前流程实例中。*BusinessProcessScoped beans 必须具有钝化能力（如可序列化）*。以下是一个流程作用域内 bean 例子：

```
@Named
@BusinessProcessScoped
publicclass BusinessTripRequest implements Serializable {
    privatestaticfinallongserialVersionUID = 1L;
    private String startDate;
    private String endDate;
    // ...
}
```

有时，在没有关联流程实例的情况下，我们有想要使用流程作用域内的 beans，比如在启动流程以前。如果当前没有活跃的流程实例，BusinessProcessScoped beans 的实例会暂时被储存在本地作用域内（即会话或请求，取决于上下文）。如果该作用域之后关联上业务流程实例，那么这个 bean 的实例会被刷新到该流程实例。

15.2.5 注入流程变量

流程变量也可以用于注入。Activiti-cdi 支持

- @BusinessProcessScoped 注解的 bean 的类型安全的注入，使用@Inject [附加限定符] 类型 字段名
- 其它流程变量使用@ProcessVariable(name?) 限定符进行不安全的注入：

```
@Inject @ProcessVariable Object accountNumber;
@Inject @ProcessVariable("accountNumber") Object account
```

要想使用 EL 来引用流程变量，我们也有类似的选项：

- @Named @BusinessProcessScoped 注解的 beans 可以直接被引用，
- 其它流程变量可以利用 ProcessVariables bean 来引用：

```
#{processVariables.get('variableName')}
```

15.2.6 接收流程事件

Activiti 可以被连接到 cdi 事件流上。这样，使用标准的 cdi 事件机制就能够告知我们流程事件了。要启用 cdi 事件对 Activiti 的支持，需要在配置中启用相应的解析监听器：

```
<propertyname="customPostBPMNParseListeners">
    <list>
        <beanclass="org.activiti.cdi.impl.event.CdiEventSupportBpmnParseListener"/>
    </list>
</property>
```

此时，Activiti 被配置为使用 cdi 的事件流来公布事件。下面给出了在 cdi beans 中如何获得流程事件的概述。在 cdi 中，我们可以使用 `@Observes` 注解声明式地指定事件监测器。事件通知是类型安全的。流程事件的类型为 `org.activiti.cdi.BusinessProcessEvent`。以下是一个简单的事件监测方法的例子：

```
public void onProcessEvent(@Observes BusinessProcessEvent businessProcessEvent) {
    // 处理事件
}
```

这个监测器会被所有事件所通知。如果我们想要限制监测器所接收的事件，可以添加限定的注解：

- `@BusinessProcess`：将事件限制到某个流程定义。例子：`@Observes @BusinessProcess("billingProcess") BusinessProcessEvent evt`
- `@StartActivity`：将事件限制到某个活动的事件。例如：`@Observes @StartActivity("shipGoods") BusinessProcessEvent evt`，会在进入 id 为 "shipGoods" 的活动时调用
- `@EndActivity`：将事件限制到某个活动的事件。例如：`@Observes @EndActivity("shipGoods") BusinessProcessEvent evt`，会在离开 id 为 "shipGoods" 的活动时调用
- `@TakeTransition`：将事件限制到某个迁移上的事件。

以上指定的限定符可以自由组合。例如，要想接收 "shipmentProcess" 流程中离开 "shipGoods" 活动时发生的事件，我们可以编写如下监测方法：

```
public void beforeShippingGoods(
    @Observes @BusinessProcess("shippingProcess") @EndActivity("shipGoods") BusinessProcessEvent evt) {
    // 处理事件
}
```

默认的配置中，事件监听器是同步地在同一个事务上下文中调用的。cdi 的事务性监测器（仅在结合了 JavaEE/EJB 时可用）允许控制事件被传递给监测器方法的时间。利用事务性监测器，我们可以确保只有触发事件所在的事务执行成功才通知监测器：

```
public void onShipmentSucceeded(
    @Observes(during = TransactionPhase.AFTER_SUCCESS) @BusinessProcess("shippingProcess")
    @EndActivity("shipGoods") BusinessProcessEvent evt) {
    // 给客户发送邮件
}
```

15.2.7 附加特性

- `ProcessEngine` 以及服务都可以用于注入：`@Inject ProcessEngine, RepositoryService, TaskService, ...`
- 当前流程实例以及任务可以被注入：`@Inject ProcessInstance, Task,`
- 当前业务的 key 可以被注入：`@Inject @BusinessKey String businessKey,`
- 当前流程实例的 id 可以被注入：`@Inject @ProcessInstanceId String pid,`

15.3 编写测试

Activiti-cdi 模块可以使用 `org.activiti.cdi.test.CdiActivitiTestCase`（它是 `PluggableActivitiTestCase` 的扩展）来编写测试用例。

这个基础类负责启动 Weld SE 来充当测试容器，并且负责在测试之间进行内部处理。要想设置 maven 项目的 Activiti-cdi 测试，需要以下附加配置：

- 作为 maven 依赖引入 Weld-SE（通常在 test 范围内，要确保将其放在<dependencies ... />列表中任何 javaee-api 引用的前面）。
- 在 src/test/resources/META-INF 下，要有一个 beans.xml 文件，启用 ProcessEngineLookupForTestsuite 这个 alternative bean 类。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <alternatives>
    <class>org.activiti.cdi.test.ProcessEngineLookupForTestsuite</class>
  </alternatives>
</beans>
```

以下是一个 cdi 感知的 Activiti 测试的例子：

```
public class BusinessProcessBeanTest extends CdiActivitiTestCase {

    @Deployment
    public void test() throws Exception {
        // ... 进行的测试
    }
}
```

注意，beans 可以使用工具方法 `getBeanInstance(Class clazz)` 和 `getBeanInstance(String name)` 进行程式化地解析。会话和请求可以使用 `begin/endRequest()` 以及 `begin/endConversation()` 方法来启用/结束。

15.4 已知的局限性

Activiti-cdi 模块整体还是<试验性的>，还未被视为稳定的。另外，存在以下已知的局限：

- Activiti cdi 模块目前只在 Java SE 环境下进行了测试，连续整合还未测试测试过 Java EE 特性
- 使用 EL 从流程中引用 cdi 还未被支持，如果工作单元是由作业（即定时器事件。）
- 虽然 Activiti cdi 是按 SPI 实现的，并且被设计成一个“可移植的扩展”，但它仅使用 Weld 进行了测试。

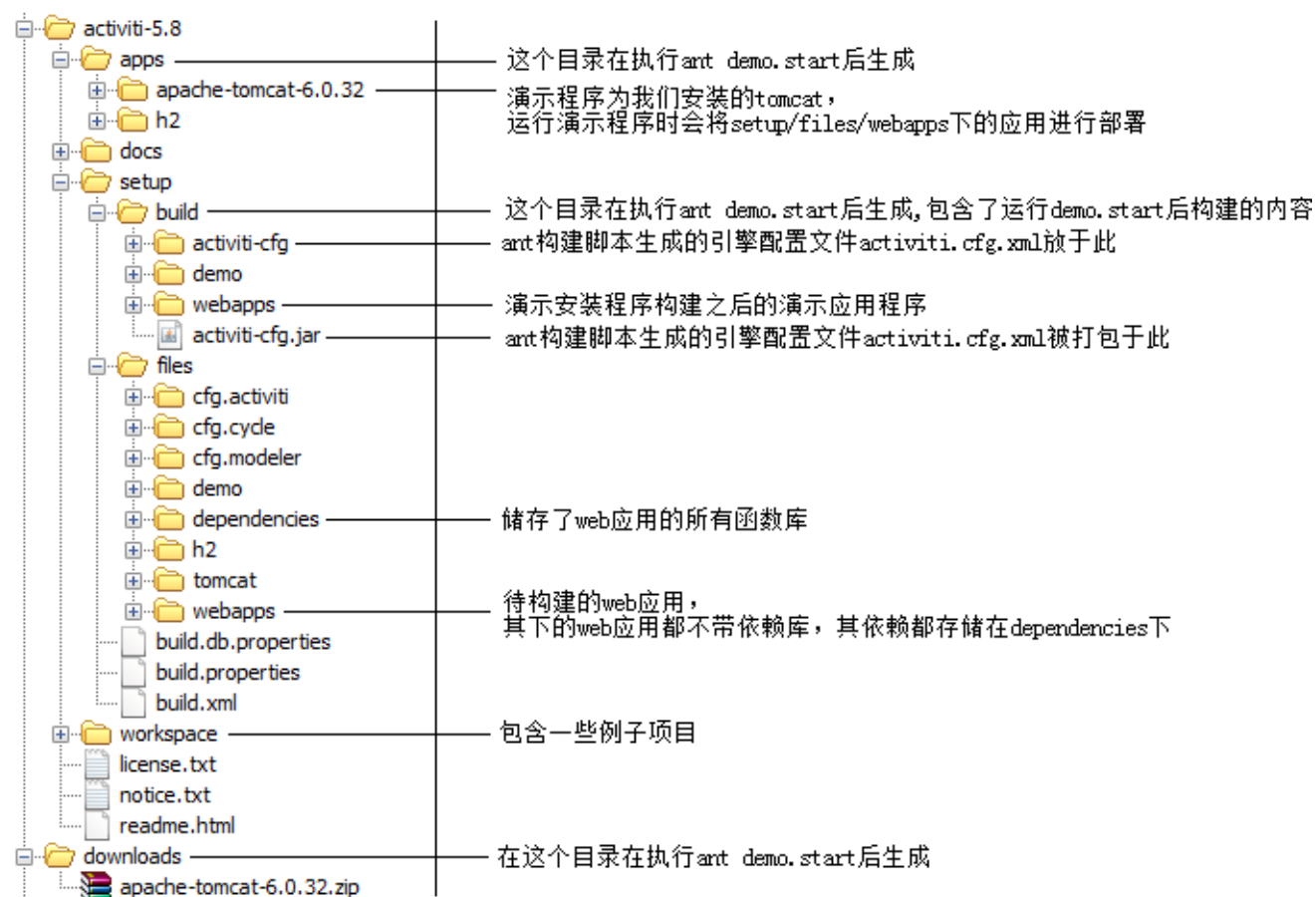
附录

附录一 认识 ant 构建脚本

1. ant demo.start
2. ant cfg.create: 基于在 build.properties 文件中指定的属性来创建配置文件。生成的配置文件可以在 setup/build/activiti-cfg。方便起见，会生成一个 activiti-cfg.jar 的文件，它里面也包含了配置文件。
3. ant demo.stop
4. ant demo.stop demo.clean demo.start
5. ant demo.clean demo.start
6. ant inflate.examples:
7. ant h2.console.start: 启动 H2 的 web 控制台。
8. ant.cfg.crea: 根据 setup 文件夹下的文件 build*.properties 指定的属性来创建数据库的配置文件。

(注: 此列表并不包括发布包内相关构建文件中所有的可执行的 ant 任务, 只是列出了常用的几个。)

附录二 认识发布文件结构



(注：此文件结构可能会依你安装而有所不同)

翻译日程

1. 略

关于文档

关于文档

《Activiti 5.4 用户指南（中文版）》发布以来得到了很多朋友的肯定与支持，由于精力、能力所限，5.4 版本的文档确实在翻译上存在诸多瑕疵，对此笔者表示十分抱歉，并请大家给予谅解。5.8 版本的翻译由于个人工作的原因推迟了很长时间，对此也希望各位朋友能谅解。在此期间，鉴于能力以及精力所限，笔者曾多次考虑不再跟进对 Activiti 的翻译，直到将其发布之日，笔者仍然犹豫是否要将其公布，每想到做事要有始有终，最后还是坚持了下来。

能力有限，本文档中难免不周之处，愿接受广大朋友的批评与指正，对您表示衷心的感谢。相关信息可访问 http://blog.163.com/java_123@yeah/。

同时，笔者也希望能借此结交更多有理想的朋友。

时，2012 年 4 月 9 日

声明

拙文一篇，但实乃笔者数月辛勤劳作之成果，在此特别做出声明：未经笔者同意不得以任何获利为目的的方式对此文进行传播，违者笔者将追究其法律责任。