Activiti 5.4

中文用户手册

译者 栗建涛

目 录

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，然后点击“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、开始

2.1 一分钟版

从 Activiti 站点下载完 Activiti 发布的 zip 文件后，按照这些步骤运行默认设置的演示安装程序。你要运行 Java 运行环境，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 登陆。

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。并要确保正确设置了 *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。

如果不提供任何参数，此演示设置将使用默认的设置来启动。要想针对你的环境来配置该演示设置，需要修改属性文件 build.properits 以及 builder.{your-database}.properties。请查看这些文件来了解更多关于配置属性以及可用值。

最简单的启动方式是在命令行中打开 setup 文件夹，然后输入：

```
ant demo.start
```

如果你不修改默认的设置，该 ant 任务将启动 tomcat 以及 H2 数据库。第一次执行该脚本时，它将按如下进行安装：

- （*）构建 web 应用。所有函数库都存储在${actviti.home}/setup/files/dependencies/libs。不带函数库的那些 web 应用都存储在${actviti.home}/setup/files/webapps。构建的 web 应用指的就是${actviti.home}/setup/files/webapps 下结合了函数库的那些 web 应用程序。
- （*）在${activiti.home}/apps/h2 下安装 H2。此步骤只有使用 h2 数据库时才进行。H2 是默认的数据库。
- 启动 H2 数据库。同样，此步骤只有使用 h2 数据库时才进行。如果你使用的是别的数据库，那么假定它已被安

装好并在运行。

- （*）在数据库中创建 Activiti 的表。
- （*）在 Activiti identity 表中插入演示用户和组。
- （*）将示例流程部署到 Activiti 引擎数据库中。
- （*）如果没有 Tomcat，就将 Tomcat 下载到${downloads.dir}。
- （*）在${activiti.home}/apps/apache-tomcat-${tomcat.version}下安装 Tomcat。
- （*）创建 Activiti 配置的 jar
- （*）将 REST 接口应用部署到 tomcat 下。
- （*）将 Activiti Modeler web 应用下载到${activiti.home}/webapps。Activiti Modeler 的许可在 MIT。
- （*）将 Probe、Explorer、以及 Modeler 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，它们会考虑到配置的属性。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
fozzid	fozzid	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 Probe	http://localhost:8080/activiti-probe	管理员管理控制台。利用此工具可以查看配置的流程引擎是否被正确初始化了，以及数据库表的内容。
Activiti Explorer	http://localhost:8080/activiti-explorer	流程引擎用户控制台。利用此工具可以浏览个人以及候选任务列表，然后完成任务。
Activiti Cycle	http://localhost:8080/activiti-cycle	Activiti 协作工具。利用此工具可以浏览仓库及在模型格式间执行转换。
Activiti Modeler powered by Signavio	http://localhost:8080/activiti-modeler	基于 web 的流程设计工具。利用此工具进行以图形化的方式编写符合 BPMN 2.0 的流程定义文件。
Activiti KickStart	http://localhost:8080/activiti-kickstart	允许以临时的方式来快速而高效地制定流程。使用 KickStart 可以快速创建简单流程、快速原型以及临时的工作流。
Activiti Administrator	http://localhost:8080/activiti-administrator	用于管理用户和用户组的 web 应用程序。目前它只是被当作独立的应用程序，但我们打算根据许可将有些 web 应用程序统一成一个 web 应用程序。

注意 Activiti 的演示程序的安装是展示 Activiti 能力和功能的既简单又快速的一种方式。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 Tomcat、JBoss、WebSphere，等等。或者你也可以作为典型而独立的 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

发布中含有一个 workspace 目录，里面包含一些 java 例子项目：

- **activiti-engine-examples**：该套示例展示了 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 的依赖库以及一个使用 groovy 脚本的流程。
- **activiti-jpa-examples**：这些示例展示了依赖库以及 Activiti 中如何使用 JPA。
- **activiti-cxf-examples**：这些示例展示了依赖库以及在 Activiti 中如何使用 web 服务。
- **activiti-cycle-examples**：此项目内含有一个关于 Activiti Cycle 的演示示例。
- **activiti-modeler-examples**：在演示程序安装内 Activiti Modeler 配置的模型库文件。

“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 和配置文件会都会被放到一个恰当的位置。如果不运行 demo.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 将包含 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 依赖函数库

为了防止由于重复包含函数库而导致发布的文件过大，就把所有的函数库都组织到了一个单独的目录 setup/files 下。

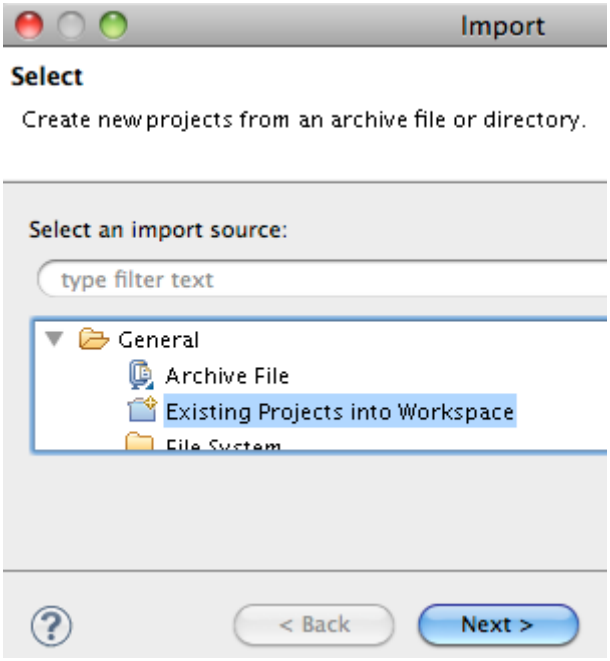
setup.build.xml 文件内的 ant 脚本将利用 libs 内的库来扩充这些示例（inflate.example 任务），当这些 web 应用被构建完成时，它们将包含相应的 libs。

所有的函数库都存在于目录 setup/files/dependencies/libs 内，且下面位于 setup/files/dependencies 内的文件描述了这些函数库：

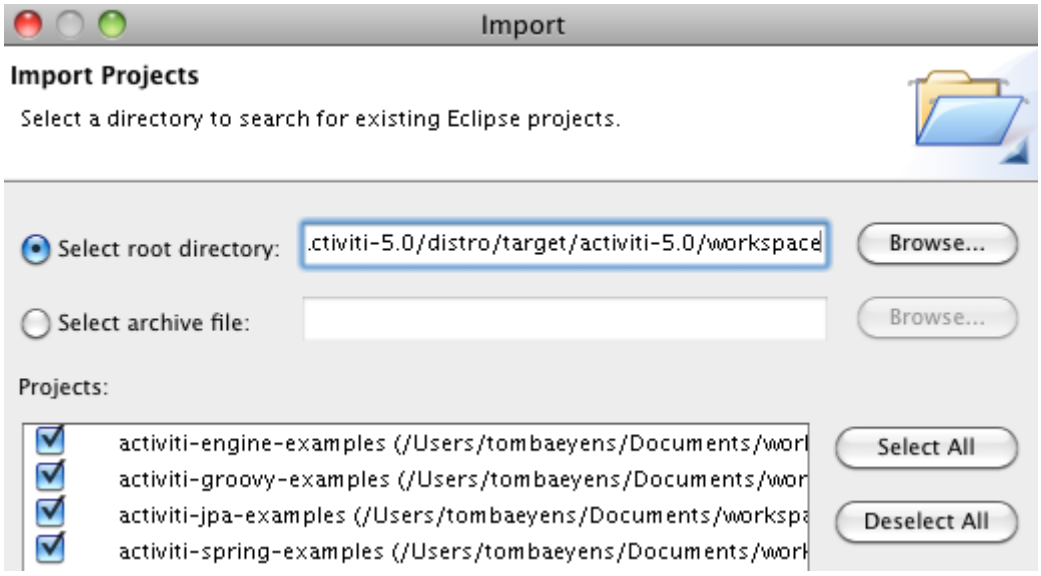
- libs.engine.runtime.txt：运行 Activiti 引擎所需的依赖库。
- libs.engine.runtime.test.txt：添加在 libs.engine.runtime.txt 内，测试所需的依赖库。
- libs.engine.runtime.feature.groovy.txt：添加在 libs.engine.runtime.txt 内，使用 groovy 脚本所需的依赖库。
- libs.engine.runtime.txt：添加在 libs.engine.runtime.txt 内，使用 JPA 变量引用能力的依赖库。
- libs.spring.runtime.txt：在 Spring 环境下运行 Activiti 引擎所有的依赖库。（包含了 libs.engine.runtime.txt 内的库）
- libs.spring.runtime.test.txt：添加在 libs.engine.runtime.txt 内，在 Spring 环境下运行测试所需的依赖库。
- libs.cycle.runtime.test.txt、libs.webapp.rest.txt 和 libs.webapp.ui.txt：分别针对 cycle、rest web 应用以及 UI web 应用，如 Activiti Explorer、Activiti Probe 和 Activiti Cycle，的完整依赖库的列表。

2.5 Eclipse 设置

要想在 eclipse 内运行、演示这些示例，需要进行如下操作：
File → Import …



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

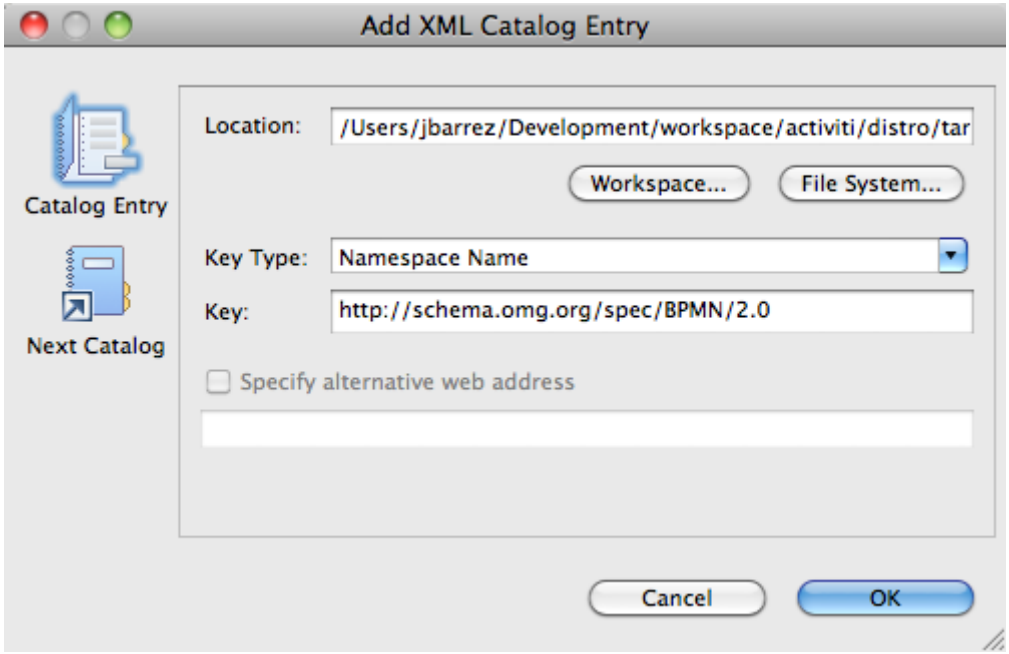


点击'Browse...'，选择目录${activiti.home}/workspace，接下来你会看到示例项目自动被选中了。

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 模式添加进 XML catalog。进入 Preferences --> XML --> XML Catalog --> Add，从 file system 内选择 Activiti 文件夹下的 docs/xsd/BPMN20.xsd。



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

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用例表的标识。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，等中的运行时的数据。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。这就使这些运行时的表保持的小且快。
- **ACT_ID_*：**’ID’代表 identity。这些表包含着标识的信息，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_*：**普通数据，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 只能使用在 Spring 环境下！**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 PrcessEngineConfiguration 对象。也可以使用 bean 的 id（如，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，参看所支持的不同类）。

```
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设置。调用 buildProcessEngine()方法会创建一个 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">
```

接下来使用这个 bean 构造 ProcessEngine。存在多个用于定义 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。如下是目前可用的类（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。如果使用这个类，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 MyBatic 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：

```
<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="" />
```

或者，可以利用 javax.sql.DataSource 的实现（例如，Apache Commons 的 DBCP）：

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

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

    <property name="dataSource" ref="dataSource" />
    ...
```

注意，Activiti 中没有包含支持 datasource 的函数库。所以你必须要确保类路径下存在那些函数库（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，因为引擎自动会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函数要求的匹配，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（Job executor）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

```
<property name="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 e-mail。要想发送 e-mail，需要配置一个有效的 SMTP 邮件服务器。关于配置选项，见 e-mail 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。

```
<property name="history" value="audit" />
```

3.7 在表达式、脚本中公布配置的 beans

默认，在 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 Activti 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	注意
h2	1.2.132	默认配置的数据库
mysql	5.1.11	
oracle	10.2.0	
postpres	8.4	
db2	DB2 9.7 使用 db2jcc4	[试验性的]
mssql	2008 使用 JDBC jtds-1.2.4	[试验性的]

3.9 修改数据库

一件你可能想要做的事就是配置 Activiti 使用别的数据库。要将演示程序设置配置到不同的数据库，或生成不同的数据库的配置文件，按如下步骤：

- 编辑 setup/build.propertyies，将 db 参数修改成你的数据库类型{oracle | mysql | postgres | h2 | db2 | mssql}。

- 编辑 setup/build.${db}.propertyies，将 JDBC 连接参数修改成你安装的数据库的参数。

要想根据你在 build.*.properties 文件指定的属性来创建数据的配置文件，请在 setup 文件内运行（译注，指的是将命令行窗体导航到此文件夹）

```
ant cfg.create
```

可以在 setup/build/activti.cfg 内找到生成的配置文件。同时，方便起见，可以在 setup/build 下找到包含了配置文件的 jar 文件 activiti-cfg.jar。

如果想要在另一个数据库上运行演示设置，首先要使用

```
ant demo.stop demo.clean demo.start
```

来结束演示设置。然后使用

```
ant demo.clean demo.start
```

将演示设置进行清理、重启。

3.10 下载 Oracle 驱动

当你要运行使用 oracle 作为数据源的演示设置时，在调用 ant 任务 demo.start 前，要进行额外的一步操作。

- 许可的原因，我们没有再发布 Oracle JDBC 的驱动，你需要自行下载：http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html。确保你下载的是 ojdbc5.jar（我们是在 10g ojdbc 上使用 11.2.0.1 进行的测试）。
- 将下载的 ojdbc5.jar 复制到 setup/files/dependencies/libs/。确保其文件名为‘ojdbc5.jar’以便被 Activiti 的演示安装程序找到（例如，从 maven repo 获取该驱动程序）。

3.11 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。这一般发生在应用程序或 Activiti web 应用启动的时候。如果 Activiti 的函数库发现库的版本与 Activiti 数据库表的版本不一样，就会抛出异常。

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

```
<beans ... >

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

```
    <property name="databaseSchemaUpdate" value="true" />
    <!-- ... -->
  </bean>

</beans>
```

这样就会更新应用程序中的 Activiti 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 会在第一次发现其函数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。此 bean 使用流程引擎的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 Srping 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/spring-tx-3.0.xsd">

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

  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>

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

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

  <bean id="repositoryService" factory-bean="processEngine" factory-method="getRepositoryService" />
  <bean id="runtimeService" factory-bean="processEngine" factory-method="getRuntimeService" />
  <bean id="taskService" factory-bean="processEngine" factory-method="getTaskService" />
  <bean id="historyService" factory-bean="processEngine" factory-method="getHistoryService" />
  <bean id="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-driven transaction-manager="transactionManager"/>

  <bean id="userBean" class="org.activiti.spring.test.UserBean">
    <property name="runtimeService" ref="runtimeService" />
  </bean>

  <bean id="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。

```
public class UserBean {

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

  @Transactional
  public void hello() {
    // here you can do transactional stuff in your domain model
    // and it will be combined in the same transaction as
    // the startProcessInstanceByKey to the Activiti RuntimeService
    runtimeService.startProcessInstanceByKey("helloProcess");
  }

  public void 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。可以限制暴露给表达式的 beans，甚至根本不暴露任何 beans（译注，此处与文档不太稍有出入）。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" />
```

现在被公布的 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 看起来像：

```
public class Printer {

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

并且 Spring bean 的配置（上文也展示了）看上去像：

```
<beans ...>
   ...

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

</beans>
```

4.4 自动资源部署

Spring 集成对于部署资源也有其特性。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());

  }
}
```

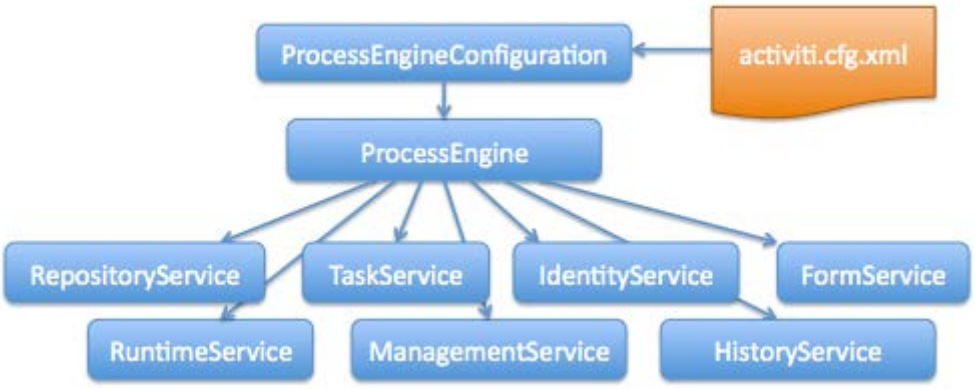
注意要让此运行，需要在 Spring 的配置中定义 *org.activiti.engine.test.ActivitiRule* bean（上文示例中，它是被自动注入的）。

```
<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 process engine，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。该 API 随时都可能抛出此类异常，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 带完成任务的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，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 Juint 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，Juint 3 风格的单元测试看起来如下：

```
public class MyBusinessProcessTest extends ActivitiTestCase {

  @Deployment
  public void 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 Juint 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.Deploymen*（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。

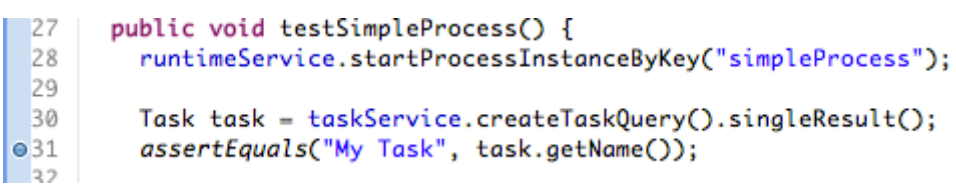
以下代码展示了使用 Junit 4 风格的测试以及 ActivitiRule 的用法的例子。

```
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());
  }
}
```

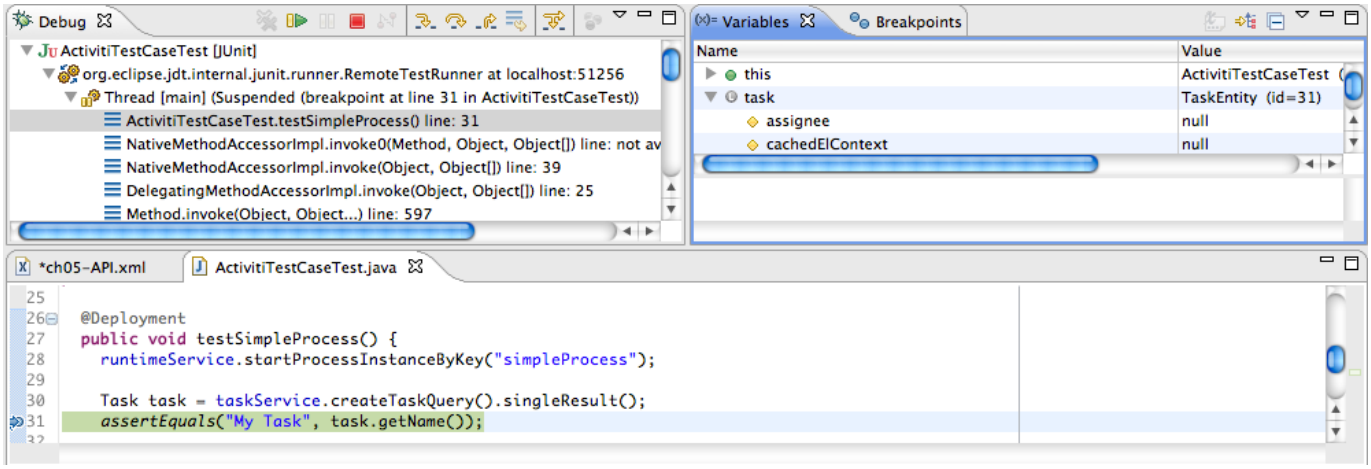
5.4 调试单元测试

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。

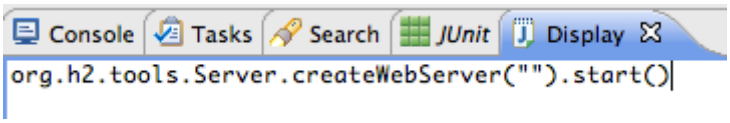
假设我们在单元测试的某处放置了断点。Eclipse 中双击代码旁的左侧边框就能完成：



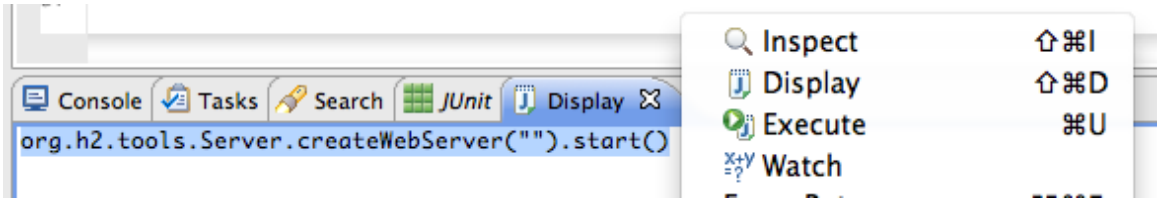
如果现在以 debug 模式运行单元测试（测试类中右击，选择'Run as'，然后选择'Juint test'），测试的执行路径断在我们的断点处，此时我们可以在右上边的面板中查看我们测试的变量。



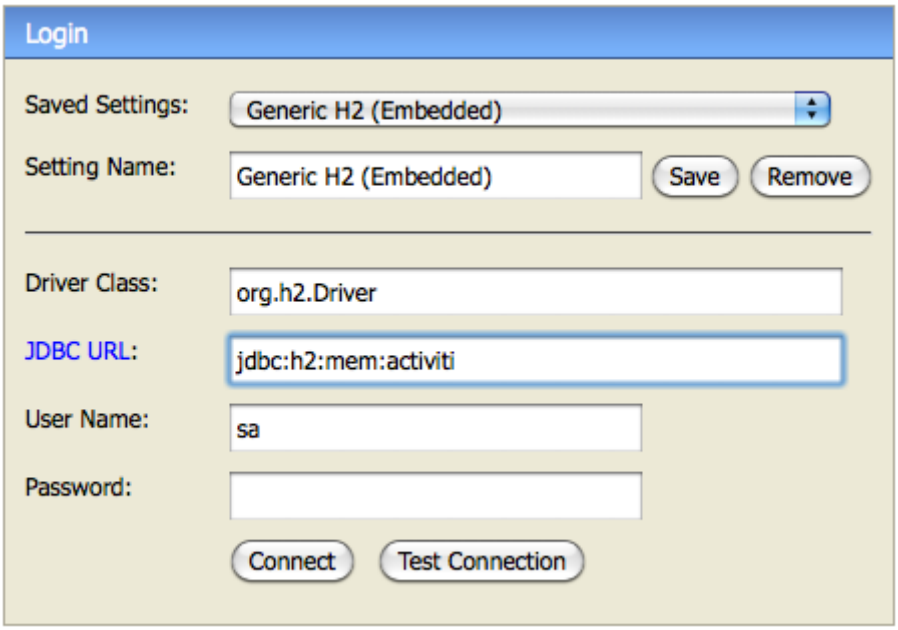
要想查看 Activiti 的数据，需要打开'Display'窗体（如果没有这个窗体，打开 Window->Show View->Other，选择 *Display*）。然后输入（代码补全可用）org.h2.tools.Server.createWebServer("").start()。



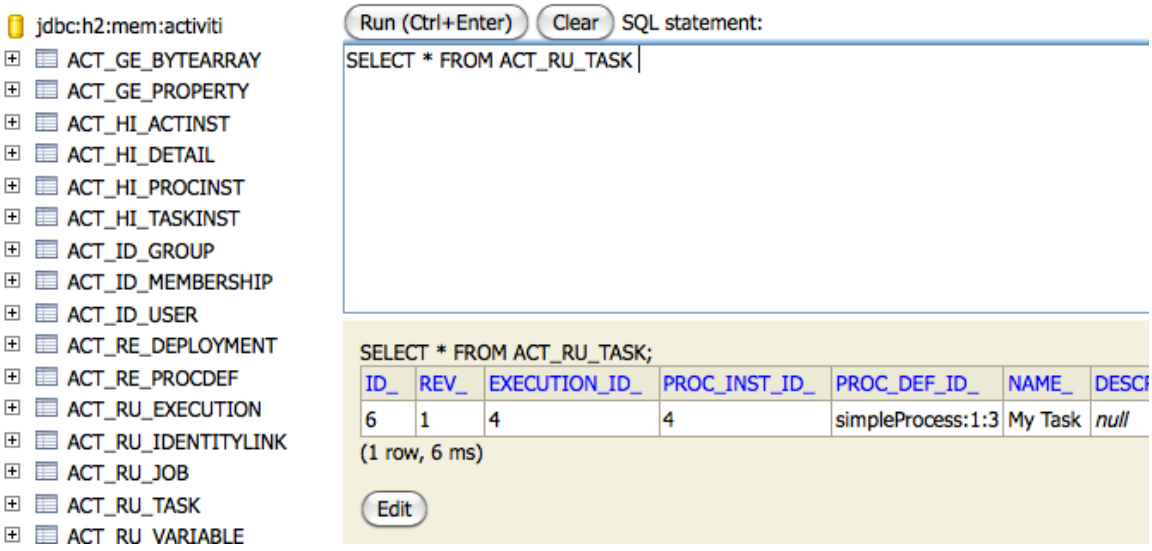
选中你刚才输入的那行代码，然后右击。此时选择'Display'（或使用快捷键）



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、销毁流程引擎：

```
public class ProcessEnginesServletContextListener implements ServletContextListener {

  public void contextInitialized(ServletContextEvent servletContextEvent) {
    ProcessEngines.init();
  }
```

```
  public void 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，然后根据所给的那么配置（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，要保证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。此外，表达式上下文中也可以使用 DelegateExecution，且可以使用名称 execution 对其进行访问。**由于执行路径以 execution 公布了，所以所有名称为 execution 的变量以及 Spring 的 beans 都将被掩盖掉，不能在表达式中使用。**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的圆括号。**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。

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 BPMN 2.0 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 使用 Activiti Probe 部署

在 Activiti Probe 的‘部署’一页，你可以上传业务归档文件将其部署到 Activiti 引擎。

6.1.2 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。

6.1.3 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" />
```

6.1.4 使用 Activiti Probe 部署

也可以通过 probe 部署流程定义，在 Activiti Prove - 部署内有描述。

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 ${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，就不用再确保所有流程引擎的类路径下都有它们的代理类以及应用恰当的 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（按该顺序）：

- 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 BMPN 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（以及不久应用 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

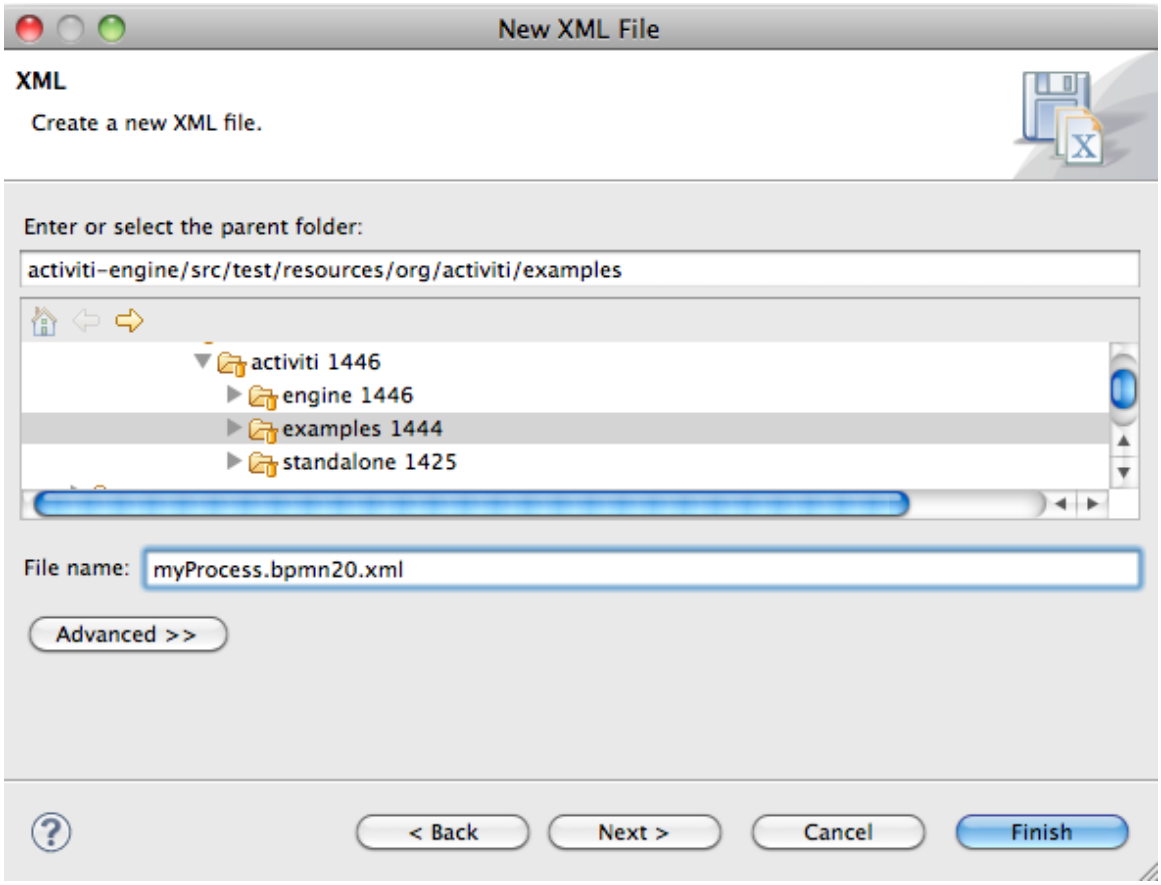
接下来章节中所描述的那些 BPMN 2.0 结构的例子可以在发布的 Activiti 中的 *workspace/activiti-x-examples* 文件夹下找到。

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，可以定义多个流程的定义（尽管我们建议每个文件中只有一个流程的定义，因为这会在接下来部署流程时简化维护）。一个空流程定义看起来如下。注意，该元素最少只需要有 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 Activti 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 Activiti demo setup。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，该程序将部署流程定义，并且使用 Activiti 引擎 API 与该流程进行交互。我们也会接触关于 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。

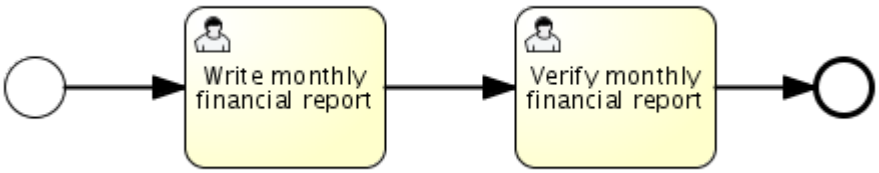
7.4.3 用例

该用例是很简单的：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发布的 Activiti 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 event（左边的圆圈），跟着两个 user tasks：“编写月度财务报表”和“审核月度财务报表”，结尾是一个 none end event（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 event 让我们学习到了该流程的入口点是什么。
- user tasks 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。跟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 event 时，流程结束。
- 元素之间是通过顺序流（sequence flows）连接的。这些顺序流拥有 *source* 和 *target*，它们定义了该顺序流的方向。

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

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

      <startEvent id="theStart" />

      <sequenceFlow id='flow1' sourceRef='theStart' targetRef='writeReportTask' />

      <userTask id="writeReportTask" name="Write monthly financial report" >
        <documentation>
```

```
            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**该流程定义。部署流程意味着两件事：

- 流程定义将存储到为 Activiti 引擎配置好了的持久化数据仓库中。因此，通过我们的部署业务流程，就确保了在重启引擎后引擎也能获得该流程定义。
- BPMN 2.0 流程定义文件会被解析到一个内存对象模型，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用章节中找到。

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。

你可以在这里下载该流程定义的 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();
```

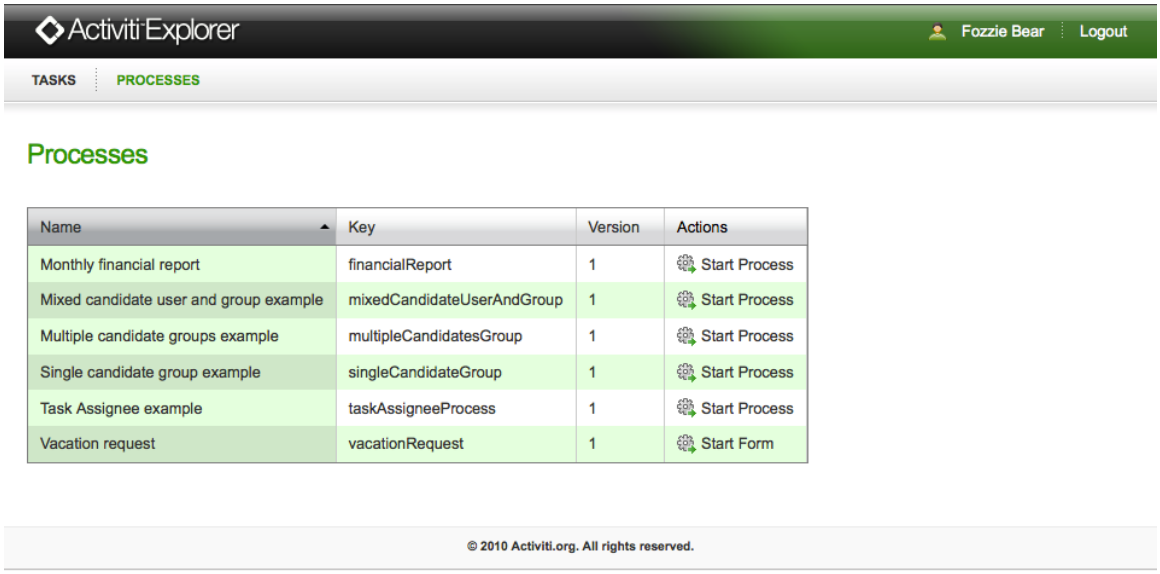
注意，我们传递给该方法的用户必须是 accountancy 组的成员，因为这在流程定义中进行了声明。

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

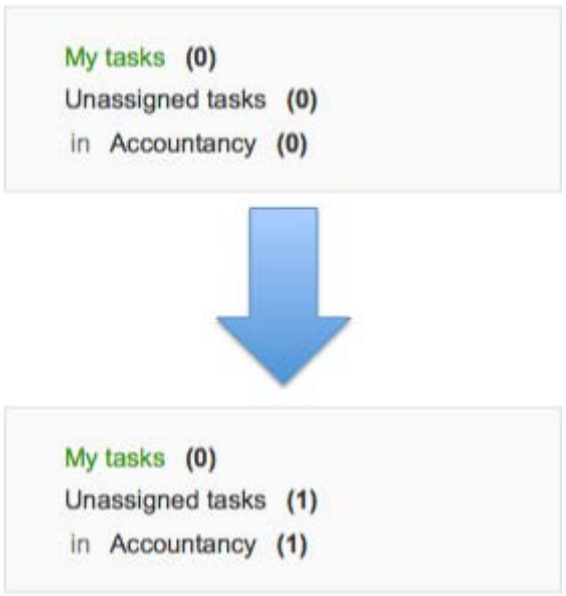
我们也可以使用这个组名利用任务查询 API 达到相同的效果。现在我们可以向代码中添加如下逻辑：

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

既然我们已经配置了 ProcessEngine 使用 demo setup 所有使用的同一个数据库，那么现在我们就可以登录到 Activiti Explorer（以 fozzie/fozzie 登录），然后，选择 *Processes* 页面，接着点击 *'Actions'* 列内对应 *'月度财务报表'* 流程的 '启动流程' 链接。



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，所有当启动了一个流程实例后，我们可以看到一个他所拥有的新候选任务。选择 Tasks 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，该任务对于 accountancy 组中每个成员也是作为候选任务可见的。



7.4.8 认领任务

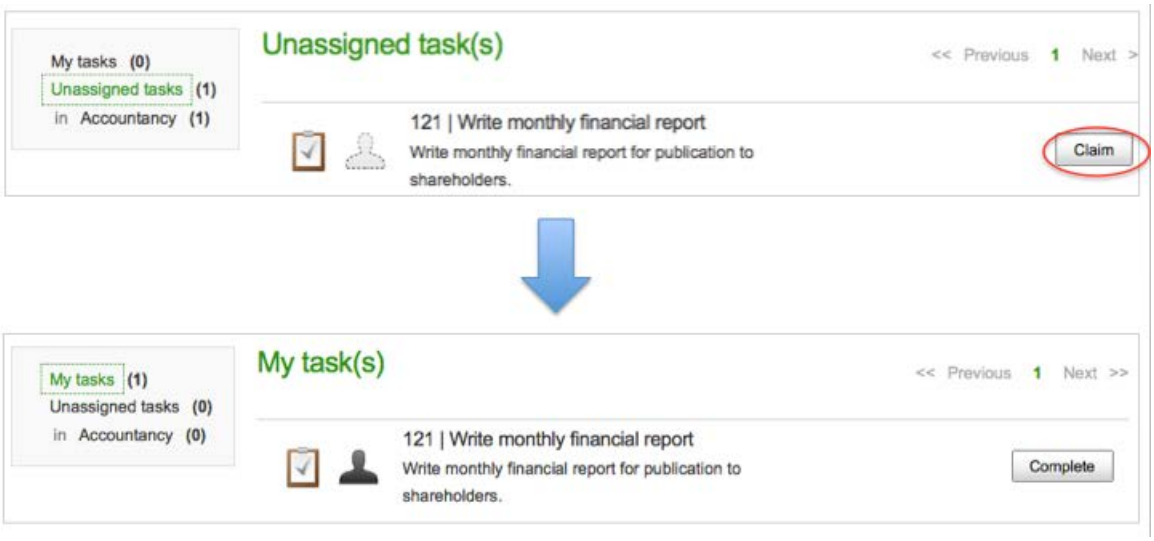
会计现在需要认领该任务。通过认领任务，将有专人作为该任务的代理人，同时该任务会从会计组其他成员的任务列表中消失。程序上完成认领任务如下：

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

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("fozzie").list();
```

在 Activiti Explorer UI 中，点击 claim 按钮将调用同一操作。现在该任务将移到登录用户的个人任务列表内。



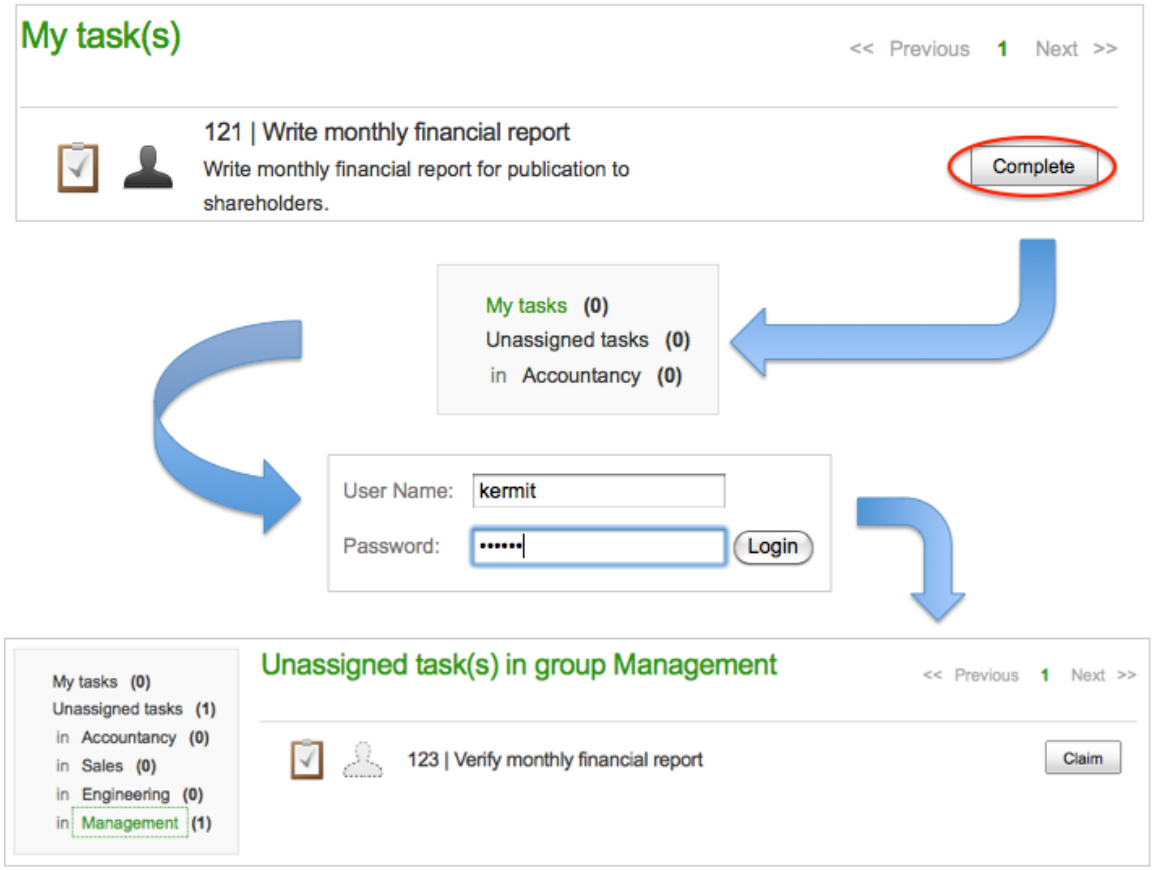
7.4.9 完成任务

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移除。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'*）。发生上述第一个任务描述的机制，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 kermit（是个管理者）登录。在 unassigned 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 Probe 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记录。

程序上，你也可以使用 historySerivce 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，所以改代码总能运行）：

```
public class TenMinuteTutorial {

    public static void 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(), "fozzie");
}

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

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

System.out.println("Number of tasks for fozzie: "
                + taskService.createTaskQuery().taskAssignee("fozzie").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());
    }
}
```

该代码也可以作为随示例发送的单元测试（是的，你可以进行单元测试）。（译注，这句话考虑了好长时间也不知道怎么翻译，看原文也看不懂作者到底在表达什么，可能是自己的理解力真的有问题吧）

7.4.12 以后扩展

很容易看出，该业务流程过于简单了以至于不能用于现实现实生活中。然而，因为你马上要深入研究 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：

- 定义做为决策的 gateway。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好事。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）具有那些大的提供商所实现的特性的解决方案。由于 BPMN 2.0 标准，可以很容易、很顺利地从一个大提供商向 Activiti 迁移。

然而，标准往往是不同公司（常常是愿景）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，只 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，那完全由你自己决定。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

Timer 事件是被定义的定时器触发的事件。可以作为 start 事件、internediate 事件或 boundary 事件来使用。

定时器的定义只能有一个下面的元素：

- **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 *timeEventDuration* 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

```
0 0/5 * * * ?
```

  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的定时器的 ISO 8601 字符串（或循环类型 cron）。

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

注意：只有开启 job executor 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 Start 事件

start 事件表示流程的开始。定义了流程如何被启动的 start 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。

start 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。

start 事件中，可以指定下面的 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。表单一节中可以找到更多信息。示例：

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

- **initiator**：标识流程启动时，存储认证用户 id 的变量名。示例：

```
<startEvent id="request" activiti:initiator="initiator" />
```

必须在 tye-finally 块中，使用方法 IdentityService.setAuthenticatedUserId(String)来设置认证的用户，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 None start 事件

描述

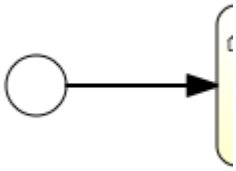
从技术上讲，‘none’启动事件意味着没有为启动流程实例指定触发器（译注，流程中事件是需要某种行为来触发的，这种行为我们称之为触发行为或触发器，而 none start 事件不需要通过这样的触发行为就能发生）。这意味着流程引擎不能预期什么时候流程实例要被启动。none start 事件使用在通过调用 *startProcessInstanceByXXX* 方法启动流程实例的时候。

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

注意：子流程总是使用 none start 事件。

图形化符号

none start 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

none start 事件的 XML 表示是不带子元素的普通 start 事件声明（其它 start 事件的类型都用声明类型的子元素）。

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

7.5.5 Timer start 事件

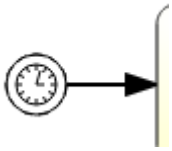
描述

timer start 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：子流程不能使用 timer start 事件。

图形化符号

Timer start 事件被形象化成带有表的内图标的圆。



XML 表示

Timer start 事件的 XML 表示是带有定时器定义子元素的普通 start 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。

```
<startEvent id="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

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

7.5.6 End 事件

结束事件表明流程或子流程（执行路径）的结束。结束事件**总是抛出型的**。这意味着当流程执行到结束事件时，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。

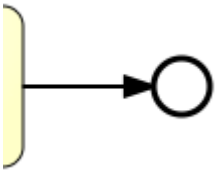
7.5.7 None end 事件

描述

'none' end 事件意味着没有指定在进入该事件时抛出的*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

none end 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

none end 事件的 XML 表示为没有子元素的普通的 end 事件的声明（其它 end 事件的类型通过子元素来声明类型的）。

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

7.5.8 Error end 事件

描述

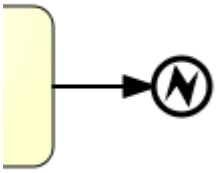
当流程执行到 **error end 事件**时，会结束当前的执行路径，并抛出 error。Error 可以被与之匹配的中间边界 error 事件捕获。如果没有找到匹配的边界 error 事件，默认会使用 none end 事件。

要点：BPMN error 与 Java 异常是不一样的。事实上，两者没有任何共同点。BPMN error 事件是对*业务异常（business exceptions）*

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

error end 事件被形象化成其内有 error 图标的特殊 end 事件（带粗边框的圆）。error 图标是全黑的以表示是抛出的语义。



XML 表示

error end 事件表示为带有 *errorEventDefinition* 子元素的的 end 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 的 **errorCode** 属性将用来查找与之匹配的捕获 boundary error 事件。如果 *errorRef* 与任何定义的 error 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" />
...
<process id="myProcess">
...
   <endEvent id="myErrorEndEvent">
      <errorEventDefinition errorRef="myError" />
   </endEvent>
...
```

等价于

```
<endEvent id="myErrorEndEvent">
      <errorEventDefinition errorRef="error123" />
</endEvent>
```

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

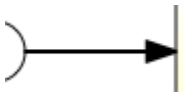
7.5.9 顺序流（sequence flow）

描述

顺序流是两个流程元素的连接器。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的顺序流继续执行。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

顺序流被形象化成从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。

```
<sequenceFlow id="flow1" sourceRef="theStart" targetRef="theTask" />
```

7.5.10 带条件的顺序流（conditional sequence flow）

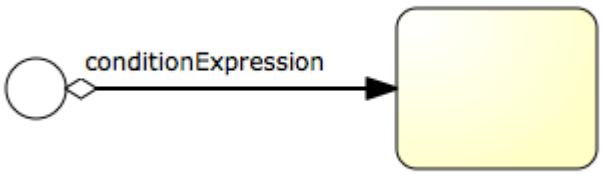
描述

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，就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 gateways。根据 gateway 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

带条件的顺序流被形象化成始点为菱形的常规顺序流。条件表达式紧挨着顺序流。



XML 表示

带条件的顺序流是以 XML 中含有 **conditionExpressio** 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=" "默认也是只支持此类型的表达式。

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

目前，conditionExpression **只能使用 UEL**，在表达式一节可以找到关于此的详细信息。在此使用的表达式的解析的结果必须是布尔类型的值，否则在计算条件时会抛出异常。

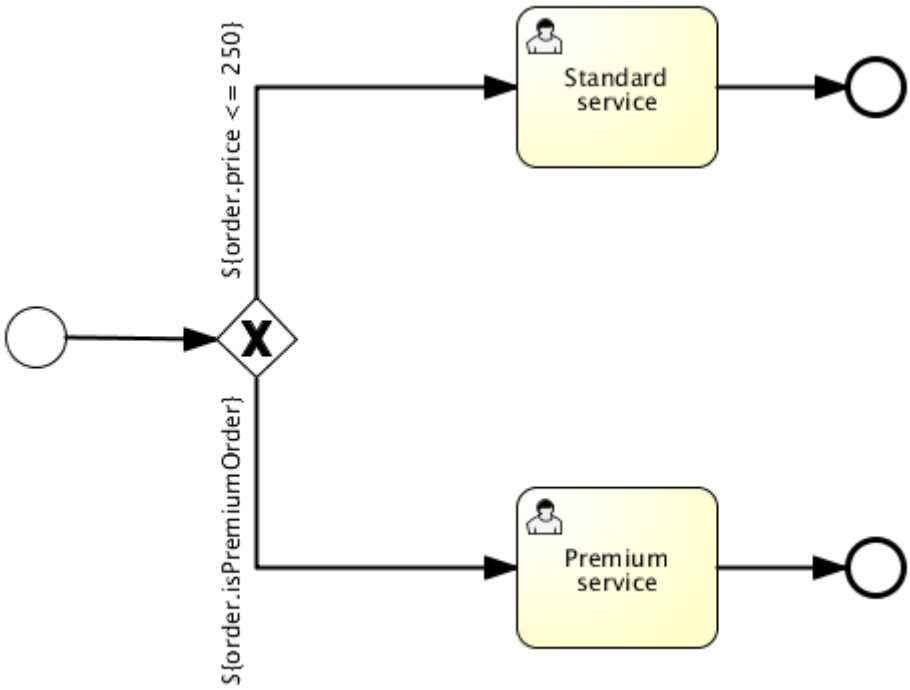
- 下面的示例以典型的 JavaBean 风格的 getters 来引用流程变量的数据。

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

Activiti 的发布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 gateways 都可以有一个**默认的顺序流**。只有当没有其它顺序流被选取的情况下，才选取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

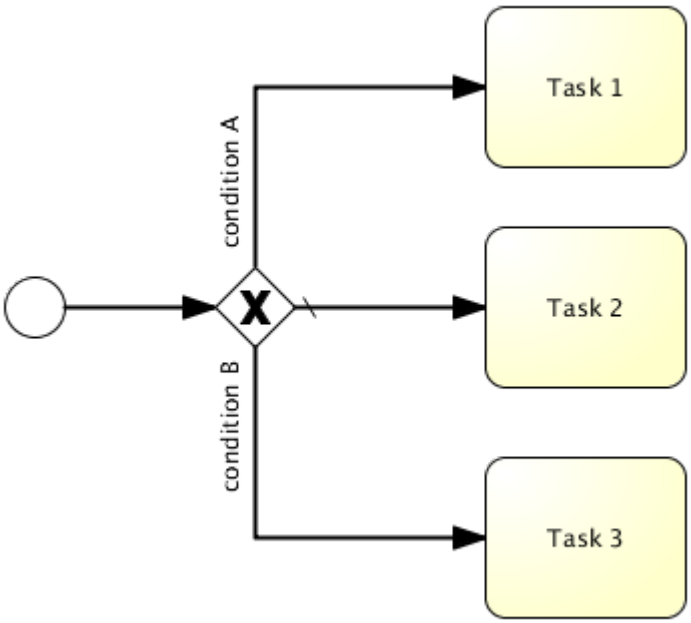
默认顺序流被形象化成起点带‘斜线’的普通顺序流。

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 片段展示了含有默认顺序流 *flow2* 的单一分支（译注，exclusive gateway，对于这个词我一直没能找到一个更为合适的词汇进行翻译）。只有当 *conditionA* 和 *conditionB* 都为 false 时，才会选取它作为 gateway 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>
```

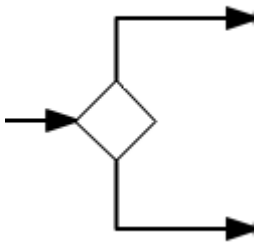
与下图示相符：



7.5.12 分支（Gateways）

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 单一分支（Exclusive Gateway）

描述

exclusive gateway（也称为 XOR gateway，或更专业点，exclusive data-based gateway）用来对流程中的决定进行建模。流程执行到这种 gateway 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，概念上顺序流上定义为'true'）被选取继续执行流程。

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 的顺序流都会被选取以并行的方式继续流程的执行，但在使用 exclusive gateway 时，只有一个顺序流被选取。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取来继续流程的执行（仅有那个会被选中）。如果没有选取到任何顺序流，就

会抛出异常。

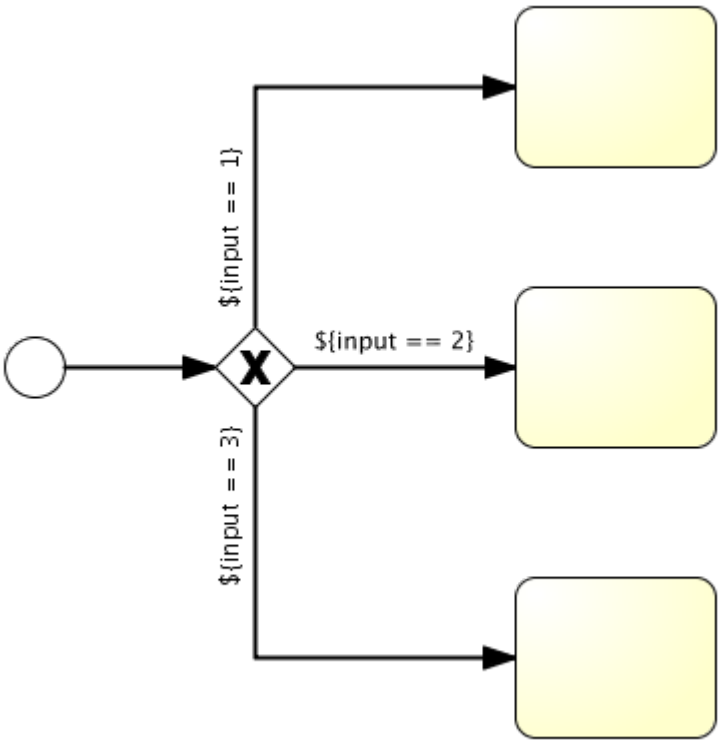
图形化符号

exclusive gateway 被形象化为里面有一个'X'的特殊 gateway（即一个菱形），XOR 的语义。注意，不带图标的 gateway 默认是 exclusive gateway。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

Exclusive gateway 的 XML 表示是很直观的：定义 gateway 的行以及定义在输出顺序流上的条件表达式。参看关于带条件的顺序流的一节来查看该表达式有那些选项。

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（parallel gateway）

描述

Gateway 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 gateway 就是 Parallel Gateway，它能拆分出多个执行路径，或多个输入执行路径进行合并。

Parallel gateway 的功能要根据输入和输出的顺序流：

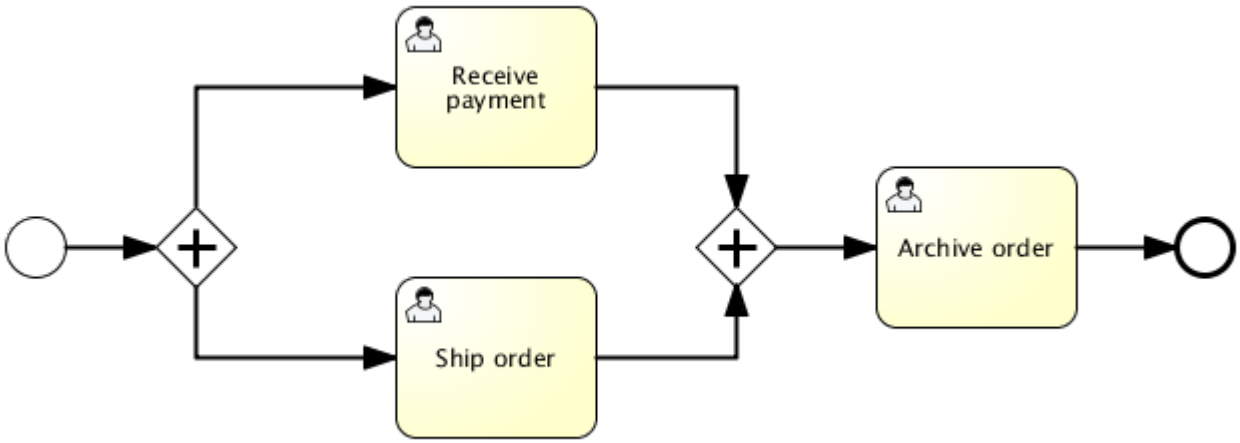
- 拆分（fork）：并行执行所有的输出顺序流，为每一个顺序流创建一个并行执行路径。
- 合并（join）：所有到达 parallel gataway 的并发性的执行路径都等待于此，直到每个输入流都执行到。然后，流程经由 joining gateway 继续向下执行。

注意：如果 parallel gateway 有多个输入流和输出流，那么它可以同时具有 fork 和 join 行为。那样，gateway 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 gateway 一个重要的不同点是 parallel gateway 不会计算条件。如果在连接 parallel gateway 上定义了条件，那么那些条件会被简单的忽略掉。

图形化符号

Parallel gateway 被形象化成里面有‘加号’的 gateway（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 parallel gateway：

```
<parallelGateway id="myParallelGateway" />
```

实际的行为（fork、join）由连接到 parallel gateway 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" />

<parallelGateway id="fork" />
<sequenceFlow sourceRef="fork" targetRef="receivePayment" />
<sequenceFlow sourceRef="fork" targetRef="shipOrder" />

<userTask id="receivePayment" name="Receive Payment" />
<sequenceFlow sourceRef="receivePayment" targetRef="join" />

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

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

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

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

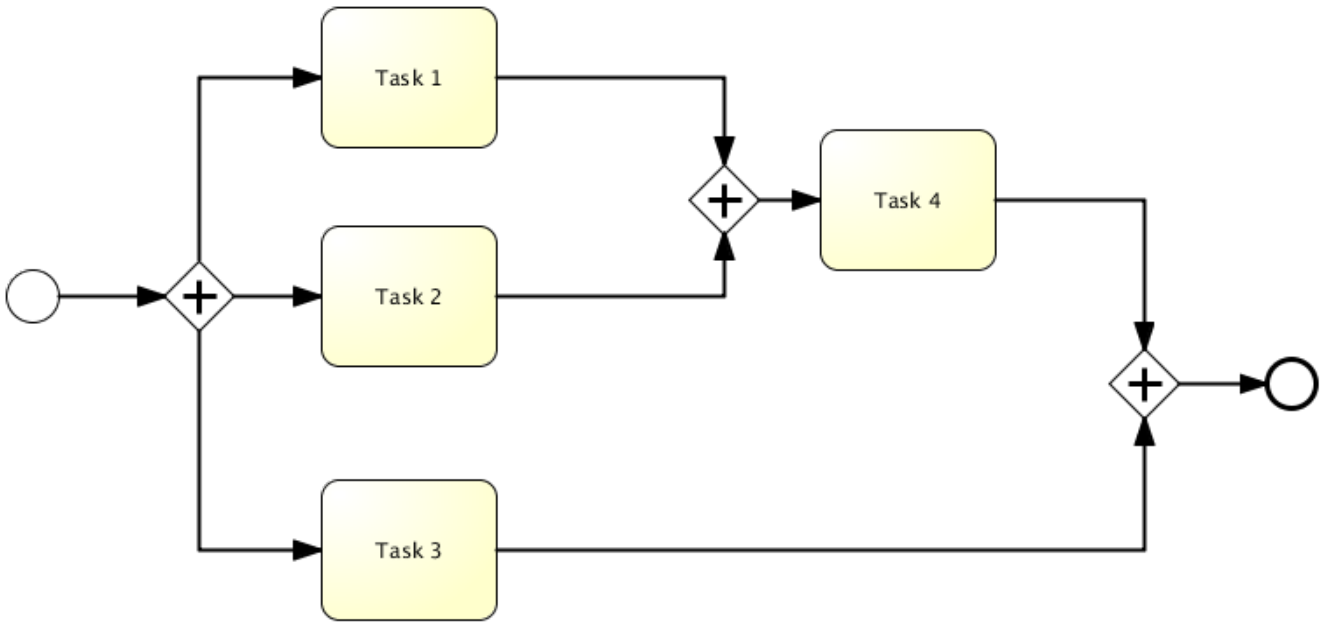
上面的示例中，在启动流程后，将创建两个任务：

```
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 parallel gateway 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’parallel gateway（即，为 parallel gateway 匹配输入/输出顺序流得数量）。Parallel gateway 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：



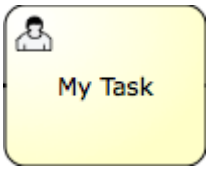
7.5.15 用户任务

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，会在分配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用户图标的特殊任务（椭圆）。



XML 表示

在 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" />
```

用户任务可以含有描述。事实上，任何 BPMN 2.0 中的元素都可以有描述。通过添加 **documentation** 元素来定义描述。

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

描述文本可以以标准 java 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 可以用来在某个时间点前或后查询任务是否过期。

有个 activity 的扩展，允许在创建任务时，在任务定义中指定一个表达式来设置任务初始的超期时间。该表达式结果必须为 **java.util.Date** 或 **null**。例如，你可以使用由流程中前面表单输入或前面 Service Task 计算的日期。

```
<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 来修改。

用户的分配

可以将用户任务直接分配给用户。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**。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**候选任务列表**中。这时，就要利用 **potentialOwner** 了。用法类似于 *humanPerformer*。一定要注意需要定义 formal 表达式中的每个元素以指名是用户还是用户组（流程引擎是猜测不到的）。

```
<process ... >
```

```
...

<userTask id='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（或类似于在分配了代理者任务中使用 *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)*。这也会获得所以分配在 kermit 所在组（例如，*group(management)*，如果 kermit 是那个用户组的成员，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。

  ```
  <userTask id="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。

  ```
  <userTask id="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。

  ```
  <userTask id="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，那么可以委托给使用了 create 事件上的任务监听器的自定义分配逻辑：

```
<userTask id="task1" name="My task" >
  <extensionElements>
    <activiti:taskListener event="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：

```
public class MyAssignmentHandler implements TaskListener {

  public void 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 Spring bean ldapService 中的方法 findManagerOfEmployee 来设置的。传递的 *emp* 参数，是个流程变量。

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

这对于候选用户和候选组的情况效果是类似的：

```
<userTask id="task" name="My Task" activiti:candidateUsers="${ldapService.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）才能生效。

```
public class FakeLdapService {

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

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

}
```

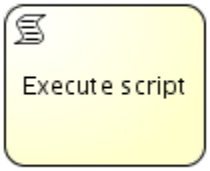
7.5.16 脚本任务（script task）

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 plateform）兼容的名称。默认 Groovy jar 是添加在 Activiti 中一起传递的。如果你想要使用其他（JSR-223 兼容的）脚本引擎，需将足够的 jar 添加到类路径下，然后使用恰当的名称。

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设置脚本中的流程变量。在上面的例子中，在脚本任务执行完成后*'sum'*变量会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：下面的名称被保留，**不能用来做**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表示流程变量名，可以将脚本任务的返回值分配到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" activiti: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）被设置到叫做*'myVar'*的流程变量。

7.5.17 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的椭圆。

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 **activiti:expression** 指定必然会被计算的 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** 来指定必然会被计算的 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的流程变量和（如果适用）Spring 上下文中被解析的。

实现

要实现在流程执行期间中调用类，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。

让我们创建一个 Java 类，该类用来将流程变量字符串该为大写。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 DelegateExcution 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）与实际运行的环境常常是不同的。比如，在使用 *ant* 或在 Activiti Probe 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，应该不要这样做。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。支持下面的注入类型：

- 固定字符串值
- 表达式

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，在实际字段注入声明前面，我们需要声明'extensionElements'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:field name="text" stringValue="Hello World" />
    </extensionElements>
</serviceTask>
```

类 ToUpperCaseFieldInjected 有一个类型为 org.activiti.engine.delegate.Expressiontext 的 text 字段。当调用 text.getValue(execution)时，返回配置了的“Hello world”字符串。

或者，对于长文本，可以使用子元素“activiti:string”（例如，一行 e-mial）：

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

要注入运行时自动解析的值，可以使用表达式。这些表达式可以使用流程变量，或 Spring 定义的 bean（如果使用了 Spring）。就像服务任务实现中注意的，所有流程实例共享服务任务中的 Java 类。要想向字段动态注入值，可以将值表达式和方法表达式注入给 org.activiti.engine.delegate.Expression，使用 execute 方法中传进来的 DelegateExecution 可以对 org.activiti.engine.delegate.Expression 进行计算/调用。

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

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

以下的示例类使用了注入的表达式，然后使用当前 DelegateExecution 来解析这些表达式。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:field name="text1" expression="${genderBean.getGenderString(gender)}" />
<activiti:field name="text1" expression="Hello ${gender == 'male' ? 'Mr.' : 'Mrs.'} ${name}" />
```

由于该 java 类的实例是可重用的，所以注入只在第一次调用 serviceTask 时发生。一旦在代码中修改过这些字段，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

通过给服务任务定义的属性*'activiti:resultVariable'*指定一个字符串表示的流程变量名，可以将服务执行的返回值分配到现有的或新的流程变量上。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，服务执行结果（调用流程变量或 Spring bean 中可用的名为'myService'的对象上方法*'doSomething()'*的返回值）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完成的。

```
<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);
    }

}
```

7.5.18 WebService 任务

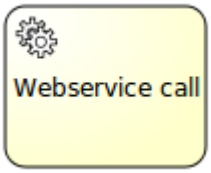
[试验性的]

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“prottyPrint”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：

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

<itemDefinition id="prettyPrintCountRequestItem" structureRef="counter:prettyPrintCount" />
<itemDefinition id="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’。每个消息我们都用到了前面定义消息。例如，下面的声明定义了’counter’接口和’prettyPrintCountOperation’操作：

```
<interface name="Counter Interface" implementationRef="counter:Counter">
```

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

然后，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。

```
<serviceTask id="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 规范来表述哪些是输入任务、哪些是输出任务。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>
    <dataInput itemSubjectRef="tns:prettyPrintCountRequestItem" id="dataInputOfServiceTask" />
    <dataOutput itemSubjectRef="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>
```

7.5.19 业务规则任务

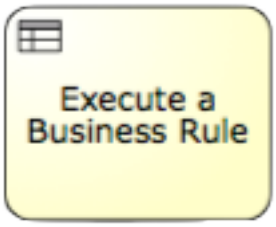
[试验性的]

描述

业务规则任务用于同步执行一个或更多规则集或规则。Activiti 使用 Drools Expert 和 Drool 规则引擎来执行业务规则。为此，包含着业务规则的.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。这意味着流程中使用的所有.drl 文件必须像任务表单一样被打包到流程的 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，用来将执行业务规则的输出对象存储到流程变量中。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.OUPUT。

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：

```
<process id="simpleBusinessRuleProcess">

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

  <businessRuleTask id="businessRuleTask" activiti:ruleVariablesInput="${order}"
      activiti:resultVariable="rulesOutput" />

  <sequenceFlow sourceRef="businessRuleTask" targetRef="theEnd" />

  <endEvent id="theEnd" />

</process>
```

也可以配置业务规则任务让它只执行部署的.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：

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

这个例子中，只有 rule1 和 role2 会执行。

你也可以定义一个不会被执行的规则列表。

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

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

7.5.20 Email 任务

Activiti 允许利用自动的 mail serice 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，邮件任务不是 BPMN 2.0 规范中的“办公”任务（因此，它并没有专门的图标）。因此，Activiti 中 mail 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 e-mails。实际发送 e-mails 时，引擎需要知道如何连接到该邮件服务器。下列属性可以在文件 *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。默认不设置。
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。默认不设置。

定义邮件任务

Email 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。

```
<serviceTask id="sendMail" activiti:type="mail">
```

Email 任务是通过字段注入来配置。所有那些属性的值都可以含有流程执行期间解析的 EL 表达式。可以设置下面的属性：

表 7.2 邮件服务器的配置

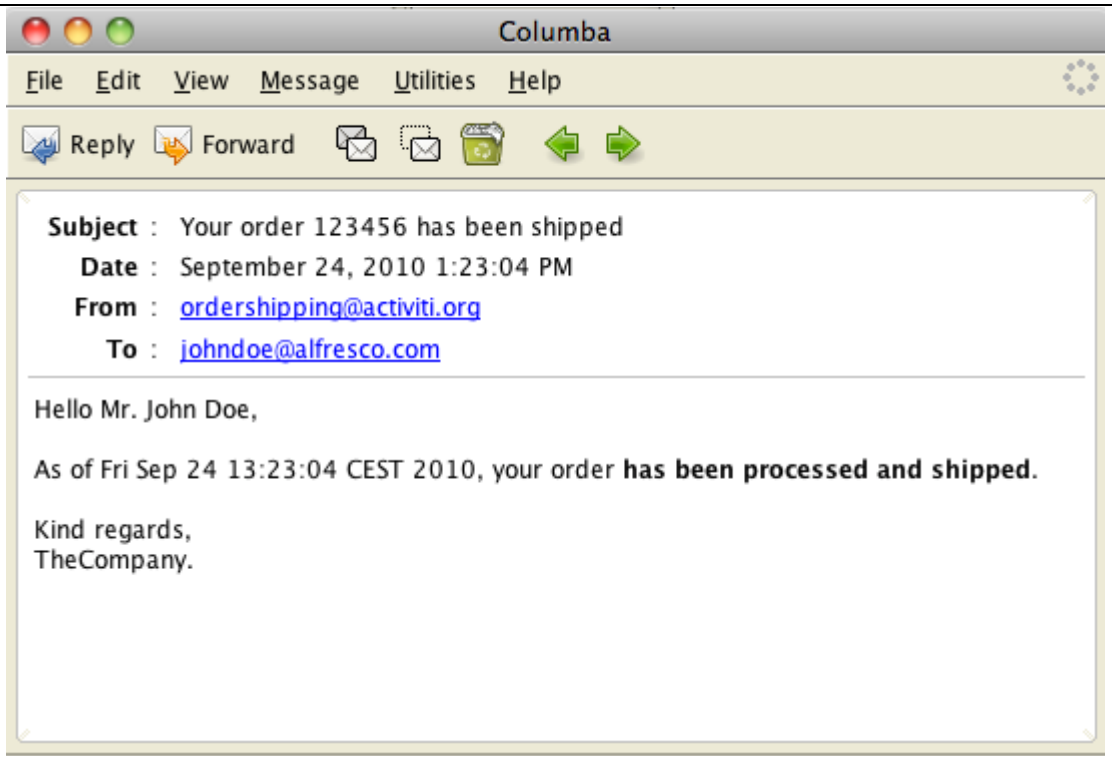
属性	是否必须?	描述
to	是	如果是 e-mail，表示收件人。可以在逗号分隔的列表中定义多个收件人。
form	否	发件人 e-mail 地址。如果没提供，使用地址的默认配置。
subject	否	e-mail 的主题。
cc	否	e-mail 的 cc。可以在逗号分隔的列表中定义多个收件人。
bcc	否	e-mail 的 bcc。可以在逗号分隔的列表中定义多个收件人。
html	否	e-mail 内容的 HTML 片段。
text	否	e-mail 的内容，如果需要非丰富的 e-mail。对于不支持丰富内容的 e-mail 客户端，可以结合 html 使用。客户端会选择使用纯文本。

用法举例

下面的 XML 片段展示了使用 Email 任务的例子。

```
<serviceTask id="sendMail" activiti:type="mail">
  <extensionElements>
    <activiti:field name="from" stringValue="order-shipping@thecompany.com" />
    <activiti:field name="to" expression="${recipient}" />
    <activiti:field name="subject" expression="Your order ${orderId} has been shipped" />
    <activiti:field name="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:field>
  </extensionElements>
</serviceTask>
```

结果如下：



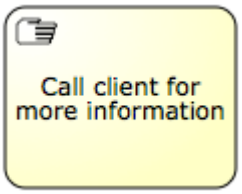
7.5.21 手动任务

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（pass-through）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“图标的椭圆。



XML 表示

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

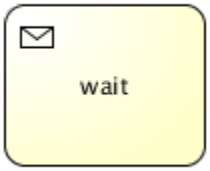
7.5.22 Java 接收任务

描述

接受任务是等着消息到来的简单任务。目前，我们仅实现了该任务的 Java 语义。当流程执行到接受任务时，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" />
```

要继续当前等待在那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());
```

7.5.23 执行（execution）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。新公开可用的 API 中，不能再访问 ExecutionListenerExecution.getEventSource()了。除了消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
- 活动的开始和结束

下面的流程定义包含 3 个执行监听器：

```
<process id="executionListenersProcess">

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

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

    <userTask id="firstTask" />
    <sequenceFlow sourceRef="firstTask" targetRef="secondTask">
    <extensionElements>
      <activiti:executionListener class="org.activiti.examples.bpmn.executionListener.ExampleExecutionListenerTwo" />
    </extensionElements>
    </sequenceFlow>

    <userTask id="secondTask" >
    <extensionElements>
      <activiti:executionListener expression="${myPojo.myMethod(execution.event)}" event="end" />
    </extensionElements>
    </userTask>
    <sequenceFlow sourceRef="secondTask" targetRef="thirdTask" />

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

    <endEvent id="theEnd" />

  </process>
```

第一个执行监听器在启动流程时被通知。监听器是个外部的 Java 类（如 ExampleExecutionListenerOne），并实现了 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)。

```
public class ExampleExecutionListenerOne implements ExecutionListener {

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

```
}
```

也可以使用实现了接口 org.activiti.engine.delegate.JavaDelegate 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取事件。**定义在迁移上的监听器的 event 属性值会被忽略掉。**

最后一个执行监听器在结束活动 secondTask 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，其会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，在此也可以使用变量 execution。因为 execution 实现类的对象有一个表示该事件名的属性，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);
}
```

7.5.24 任务监听器

任务监听器用来执行自定义的 Java 逻辑、或事件相关的表达式。

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>
```

任务监听器支持如下属性：

- **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**：在创建任务、并且任务的所有属性被设置后发生。
  - **assignment**：在任务分配给某人后发生。注意：当流程执行到 userTask，首先触发 *assignment* 事件，然后触发 *create* 事件。这像是一个反常的顺序，但原因很务实：接收到 create 事件，我们常常会要查看包括代

理人在内的所有任务的属性。

- **complete**：在任务完成，并且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) {
    // Custom logic goes here
  }

}
```

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（就想 Activiti 中其它类的代理一样），这意味着所有执行的流程实例共享此实例。

- **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**：允许指定一个结果是实现了 TaskListener 接口的对象的表达式，类似于服务任务。

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

7.5.25 多实例（for each）

描述

多实例活动是为流程中某个步骤定义重复的一种方式。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或并行地执行某个步骤、或者甚至执行完整的子流程。

*多实例*是个普通的活动，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。下面的活动可以成为多实例活动：

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

gateway、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如下的变量：

- **nrOfInstances**：实例总数。
- **nrOfActiveInstances**：当前活跃的实例，也就是还没完成的实例，的个数。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的变量（即，对其它执行路径是不可见的，并且不是存储在流程实例一级）：

- **loopConter**: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*垂直竖线*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 属性表示是否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。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>
```

也可以使用结果是正数的表示式。

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

另一种定义实例个数的方式是指定变量名，它是一个利用 loopDataInputRef 子元素的集合。会为集合中的每一项创建一个实例。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。如下面的 XML 例子展示的：

```
<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]。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，此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>
```

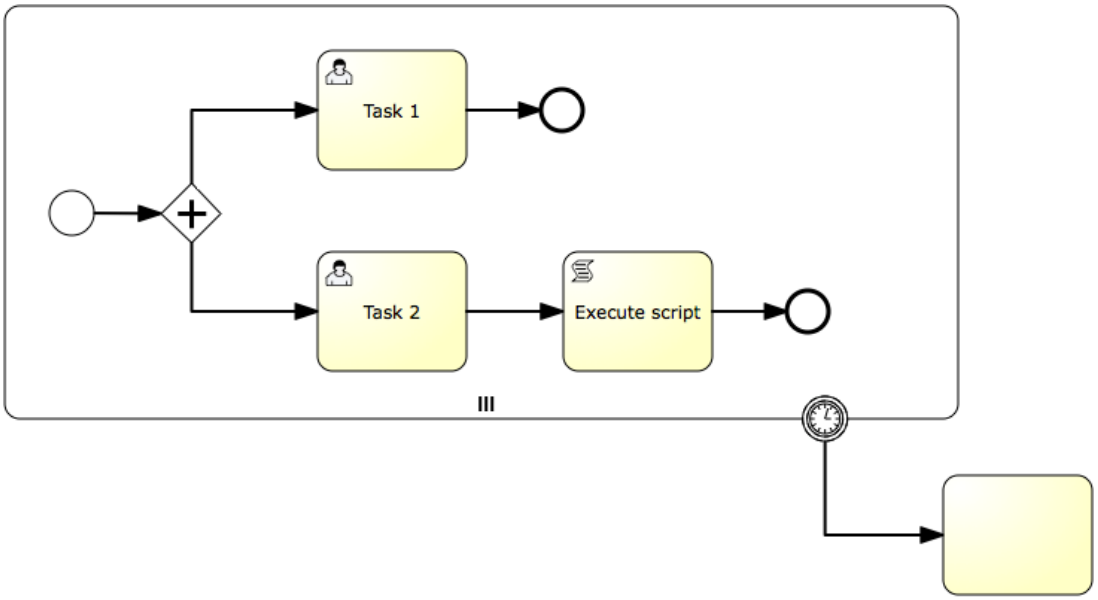
当所有实例完成后，多实例活动就结束了。然而，可以指定一个表达式每次实例结束时就对其进行计算。当该表达式值为 true 时，就会销毁所有剩余的实例，并结束该多实例活动，向下继续执行流程。这样的表达式必须定义在子元素 **completionCondition** 内。

```
<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，在任务完成了 60%的时候，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。举个采用多实例子流程的例子：



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，或是否有哪些内部活动还没完成。

7.5.26 边界事件

边界事件是关联在活动上的*捕获型*的事件（不能将边界事件抛出）。这意味着在活动运行时，该事件会一直监听着某一类型的触发器。当捕获到到事件时，活动被打断，沿事件输出的顺序流继续执行。

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）
- 使用 **attachedToRef** 属性定义的该事件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）。
- 定义了边界事件类型的一个 *XXXEventDefinition* 格式的 XML 子元素（如，*TimerEventDefinition*、*ErrorEventDefinition* 等等）。更多详细介绍，见指定边界事件的类型。

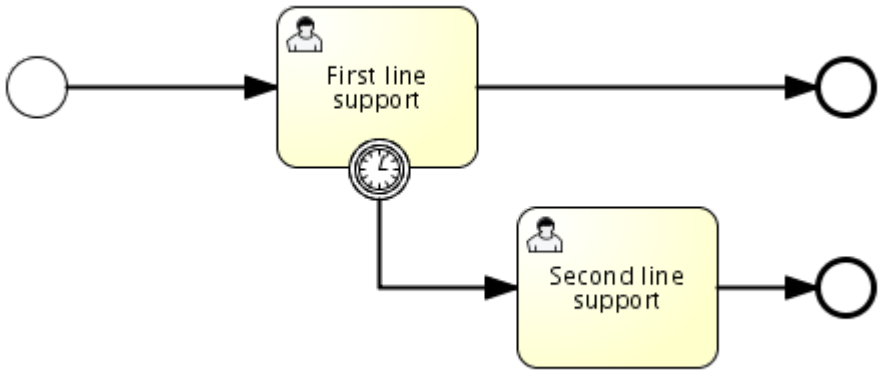
7.5.27 定时器边界事件

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，启动定时器。当触发定时器时（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）。



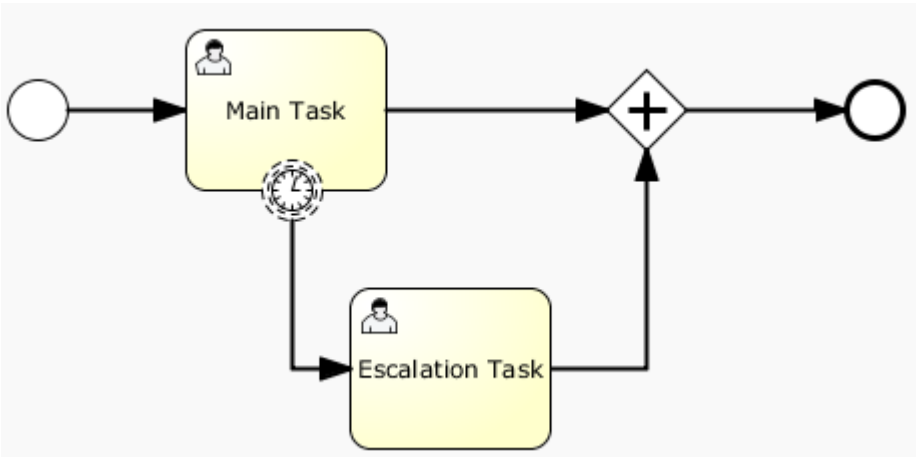
XML 表示

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是 **timerEventDefinition**。

```
<boundaryEvent id="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，如你在上面看到的，圆的边线是虚线：

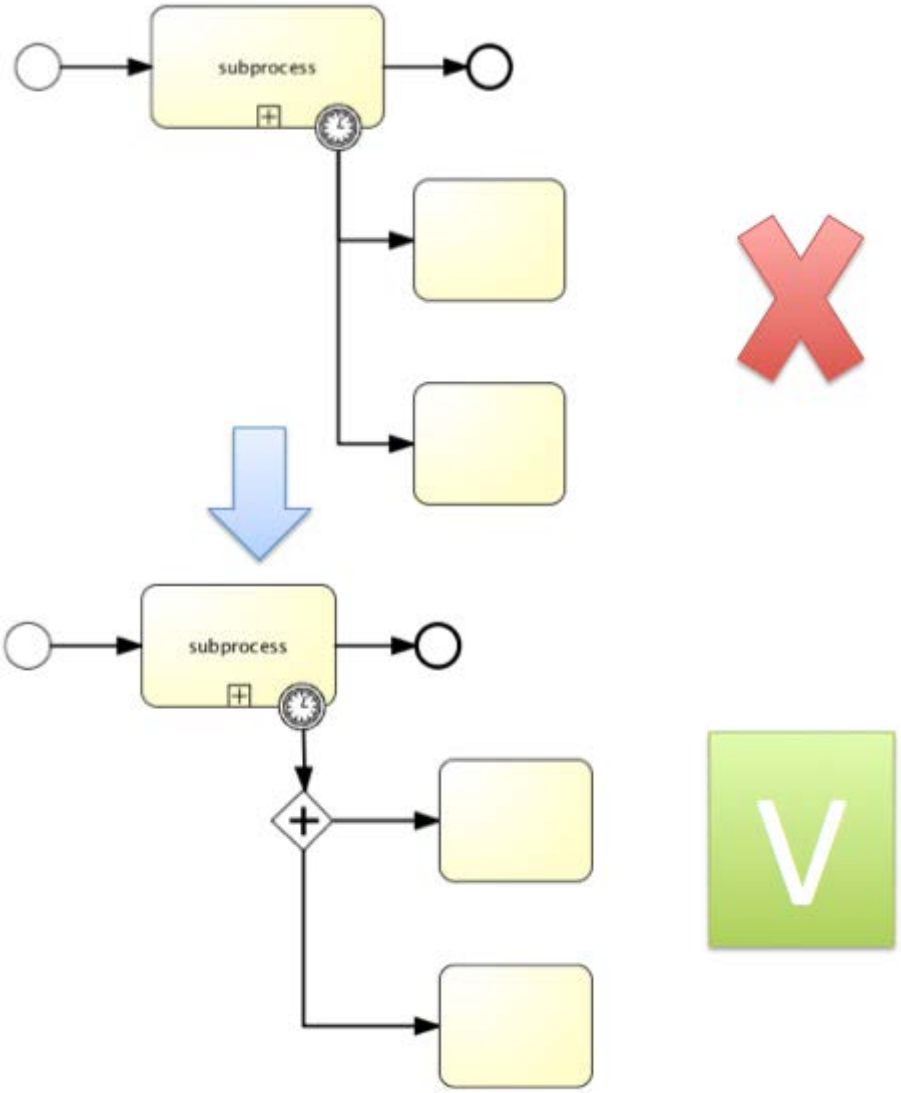


一个典型的用法是额外发送升级的 email，但不打断正常的流程执行流。

注意：边界定时器事件只有当开启作业执行器（job executor）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（job executor）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。



7.5.28 Error 边界事件

描述

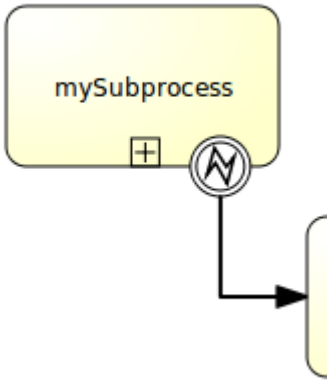
捕获活动边界上 error 的媒介，或简称**边界 error 事件**，捕获定义该活动的作用域内的抛出的 errors。

在嵌入的子流程、或调用活动上定义边界 error 是很容易理解的，因为子流程为所有子流程内的活动创建一个作用域。Errors 是由 error 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 error 事件定义匹配为止。

当捕获到 error 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）。流程沿着边界事件的输出流继续执行。

图形化符号

边界 error 事件被形象化为内部带有 error 图标的边界上的典型媒介事件（圆圈内套小圆圈）。error 图标是白色的，表示*捕获*的含义。



XML 表示

边界 error 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 error end 事件，*errorRef* 引用了流程元素以外定义的 error。

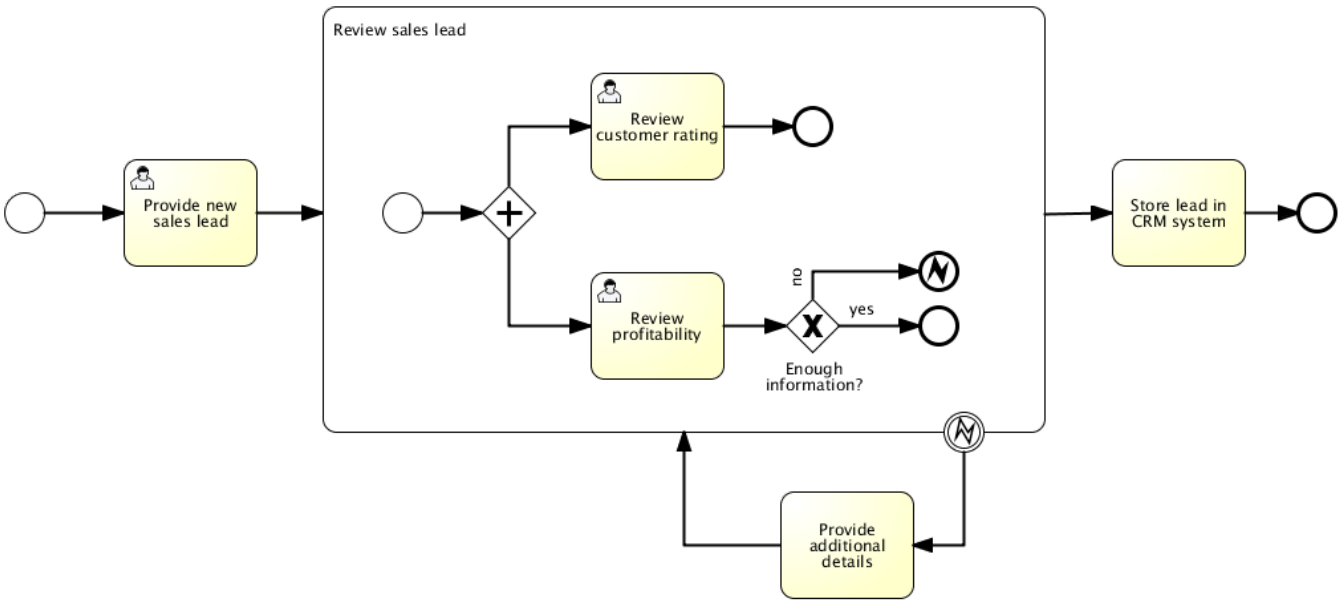
```
<error id="myError" errorCode="123" />
...
<process id="myProcess">
...
```

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 是什么。
- 如果提供了 *errorRef*，并指向了一个现有的 error，那么此边界事件将**只捕获同样出错码的 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利用 error end 事件。如果不能提供足够信息，会完成用户任务*'Review profitability'*，然后抛出 error。当在子流程的边界上捕获到该 error 后，子流程*'Review sales lead'*内所有活跃着的活动将被销毁（即使*'Review customer rating'*还没有完成），然后创建用户任务*'Provide additional details'*。（译注，这段话是对下面图形的业务解释）



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。

7.5.29 中间媒介捕获事件（intermediate catching events）

所有中间媒介捕获事件的定义都是按方式来定义的：

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

中间媒介捕获事件使用如下进行定义

- 一个唯一标识符（流程范围内的）
- 定义了中间媒介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。

7.5.30 定时器中间媒介捕获事件（Timer intermediate catching event）

描述

定时器中间媒介事件就像跑表一样。流程执行到捕获事件的活动时，启动定时器。定时器被触发后（也就是说，间隔一段时间后），会沿着定义器中间媒介事件的输出顺序流执行。

图形化符号

定时器中间媒介事件被形象化成内部带有定时器图标的中间媒介捕获事件。

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

7.5.31 子流程

描述

*子流程*是含有其它活动、gateways、事件等的活动。它是一个有它自己形式的流程，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：

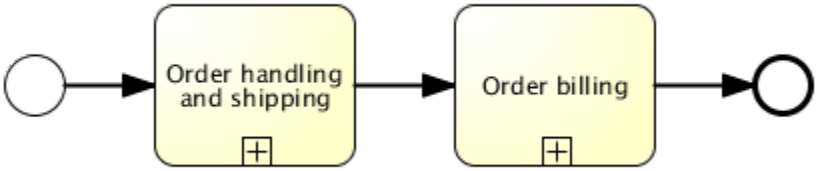
- 子流程允许进行**分层建模**。很多建模工具都允许将子流程进行*折叠*，来隐藏子流程的所有细节，以展示出了高层次互相衔接的业务流程概况图。
- 子流程**为事件创建了新的作用域**。子流程执行期间抛出事件可以被子流程边界上的边界事件捕获，因此针对事件创建的作用域只局限在子流程内。

使用子流程不会强加任何限制：

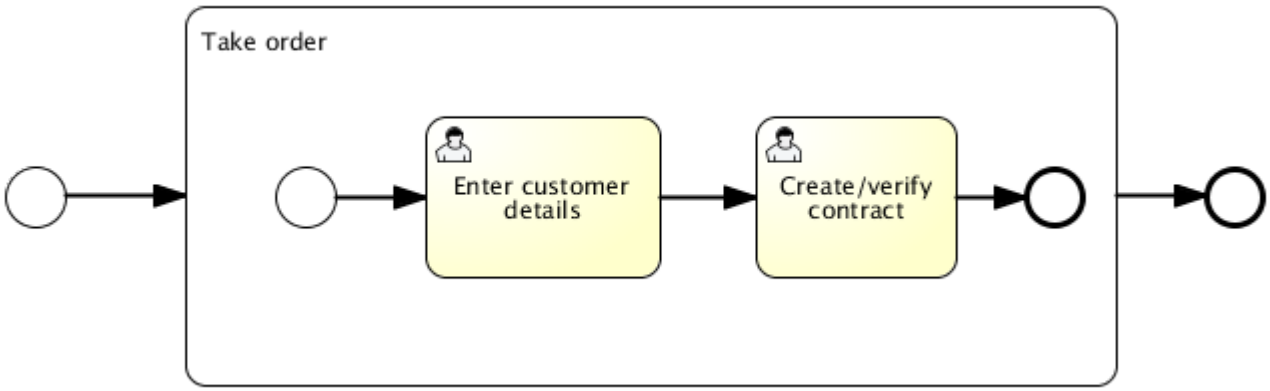
- 子流程仅能包含**一个 none start 事件**，不允许有其它类型的 start event。子流程必须**至少包含一个 end 事件**。注意，BPMN 2.0 规范允许在子流程中省略启动事件和结束事件，但是目前 Activiti 的实现不支持这样做。
- **顺序流不能跨子流程边界**。

图形化符号

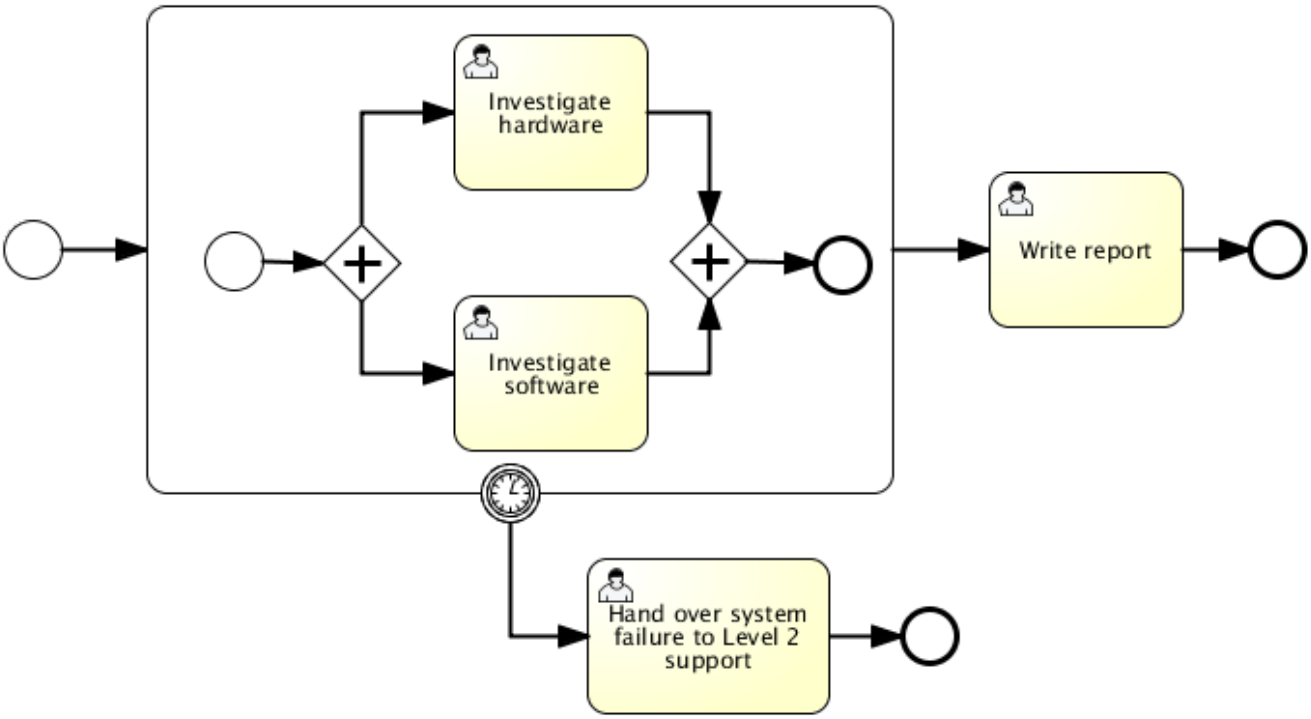
子流程被形象成一个典型的活动，即一个椭圆形。如果子流程被*折叠*起来，那么只显示名称和加号来提供流程的一个高层次的视图：



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

子流程是通过 *subprocess* 元素来定义的。所有属于子流程的活动、gateways、事件等都要封装在这个元素内。

```
<subProcess id="subProcess">

  <startEvent id="subProcessStart" />

  ... other subprocess elements ...

  <endEvent id="subProcessEnd" />

</subProcess>
```

7.5.32 调用活动（子流程）

描绘

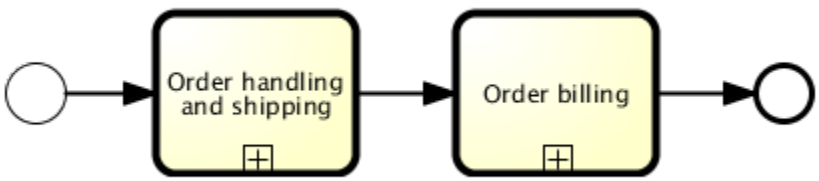
BPMN 2.0 对普通的子流程（常常称之为嵌入的子流程）和调用活动（看上去与自流程非常相似）进行了区分。从概念的角度来看，两者都会在流程执行到此活动时调用子过程。

不同的是调用活动引用的是流程定义之外的流程，而子流程是嵌入在流程定义内的。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流程定义。

当流程执行到*调用活动*时，为执行到该调用活动的执行路径创建新的子执行路径。该子执行路径接下来用来执行这个子流程，就像在普通流程中一样，子流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）。根据建模工具，调用活动也可以展开，但是默认用折叠的子流程表示。



XML 表示

调用活动是个普通活动，它需要使用 *calledElement* 利用流程定义的 **key** 来引用流程定义。实际上，这意味着 calledElement 使用的是流程的 id。

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

注意，子流程的流程定义是**在运行时解析的**。这意味着如果需要的话，子流程可以脱离调用流程进行独立部署。

传递变量

你可以将流程变量传递给子流程，反过来也一样。子流程启动时，这些数据被拷贝至子流程内，并且在子流程结束时，将这些数据拷贝回主流程。

```
<callActivity id="callSubProcess" calledElement="checkCreditProcess" >
  <extensionElements>
        <activiti:in source="someVariableInMainProcess" target="nameOfVariableInSubProcess" />
        <activiti:out source="someVariableInSubProcss" 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 的标准方式来声明流程变量时它们才能生效。

也可以在此使用表达式：

```
<callActivity id="callSubProcess" calledElement="checkCreditProcess" >
      <extensionElements>
        <activiti:in sourceExpression="${x+5}"" target="y" />
        <activiti:out source="${y+5}" target="z" />
      </extensionElements>
</callActivity>
```

结果是 z = y+5 = x+5+5

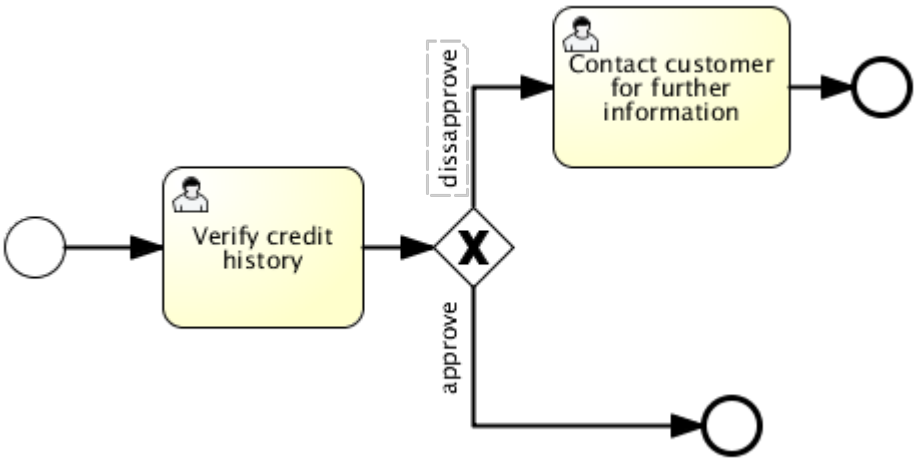
示例

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也是共同的，所以 *check credit 这步*在此被建模成一个调用活动。

流程如下：

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

<manualTask id="receiveOrder" name="Receive Order" />
<sequenceFlow id="flow2" sourceRef="receiveOrder" targetRef="callCheckCreditProcess" />

<callActivity id="callCheckCreditProcess" name="Check credit" calledElement="checkCreditProcess" />
<sequenceFlow id="flow3" sourceRef="callCheckCreditProcess" targetRef="prepareAndShipTask" />

<userTask id="prepareAndShipTask" name="Prepare and Ship" />
<sequenceFlow id="flow4" sourceRef="prepareAndShipTask" targetRef="end" />

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

子流程如下：



子流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。

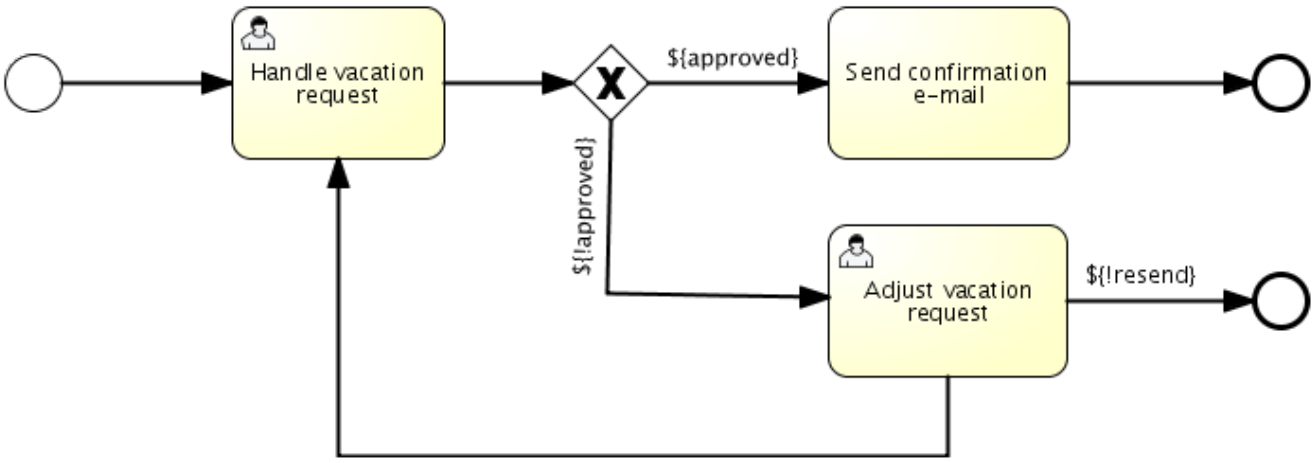
第八章、表单

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：内置的表单渲染和外部的表单渲染。

8.1 内置的表单渲染

内置的表单渲染是最容易入手的。我们将借助一个例子进行介绍。

演示设置的脚本安装了 *vacationRequest* 业务流程作为 Activiti Explorer 使用任务表单的一个例子。请查看例子整个源码。业务流程图如下：



要想使用内置的渲染，部署时必须包含表单文件。可以通过编程来完成，如下：

```
Deployment deployment = repositoryService.createDeployment()
  .addClasspathResource("org/activiti/examples/taskforms/VacationRequest.bpmn20.xml")
  .addClasspathResource("org/activiti/examples/taskforms/approve.form")
  .addClasspathResource("org/activiti/examples/taskforms/request.form")
  .addClasspathResource("org/activiti/examples/taskforms/adjustRequest.form")
  .deploy();
```

其它部署流程/表单的方法，可以在部署一章内找到。

BPMN 2.0 规范并没有规定任务或任务表单是如何被渲染的，因为这样的表单是利用 Activiti 特有的构造来定义的。可以在 start 事件和用户任务上指定属性 activiti:formKey。

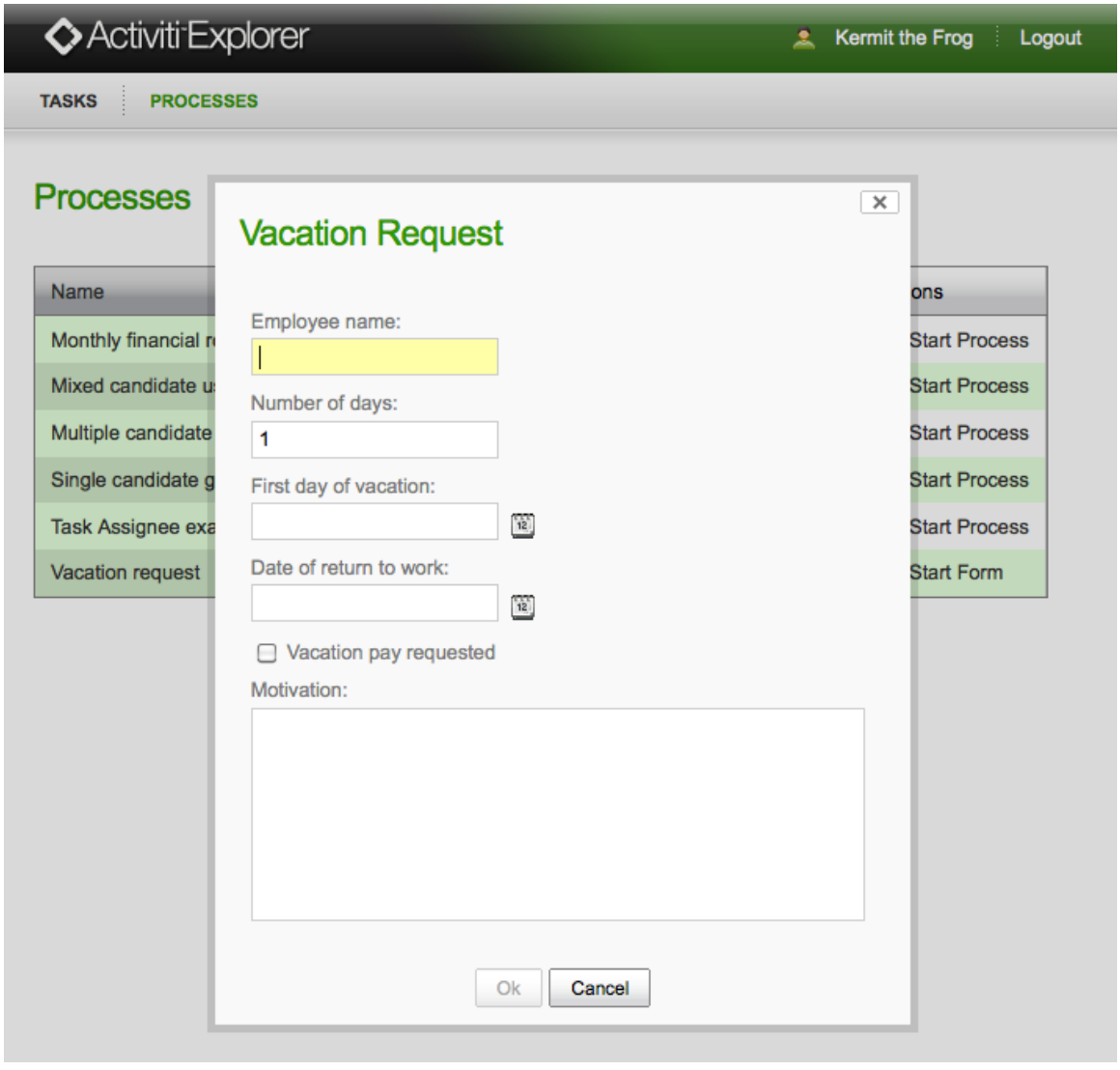
```
<startEvent id="request"
            activiti:formKey="org/activiti/examples/taskforms/request.form" />

<sequenceFlow id="flow1" sourceRef="request" targetRef="handleRequest" />
```

```
<userTask id="handleRequest" name="Handle vacation request"
          activiti:formKey="org/activiti/examples/taskforms/approve.form" >
  <documentation>
      Vacation request by ${employeeName}
  </documentation>
  ...
</userTask>
```

如果你要自己完成表单的渲染，activity:formKey 属性可以包含任何文本，用于标识你的表单。但对于内置的表单渲染 activity:formKey 的值最好是所在的同一个流程归档文件（=部署）内的资源的引用。资源是由部署 BAR 文件中完整资源文件路径来标识的。在我们的例子中，表单 request.form 被部署在了业务归档文件中的 org/activity/examples/taskforms/文件夹内。

Activiti Explorer 使用内置的表单渲染引擎。目前，只配置了一个表单渲染引擎，Juel。所以，Activiti Explorer 是按照 Juel 表达式来解析资源文件的，然后将结果 HTML 字符串发送给客户端。不久的将来，我们希望能添加 FreeMarker 表单引擎，但这样将需要更多库的依赖，所以我们选择了 Juel 作为默认的表单引擎。

这是定义在资源 org/activity/examples/taskforms/request.form 中被渲染过的表单。它是用来收集启动新流程实例所需数据的表单。



这是该表单文件的内容：

```
<h1>Vacation Request</h1>
<table>
  <tr>
    <td>
      <label>
        Employee name:<br/>
        <input type="text" name="employeeName" value="" />
        <input type="hidden" name="employeeName_required" value="true" />
        <input type="hidden" name="employeeName_type" value="User" />
      </label><br/>
    </td>
  </tr>
  <tr>
    <td>
      <label>
        Number of days:<br/>
        <input type="number" name="numberOfDays" value="1" min="1" />
        <input type="hidden" name="numberOfDays_type" value="Integer" />
      </label>
    </td>
  </tr>
  <tr>
    <td>
      <label>
          First day of vacation: (YYYY-MM-DD)<br/>
          <input type="date" name="startDate"/>
          <input type="hidden" name="startDate_type" value="Date"/>
      </label>
    </td>
  </tr>
  <tr>
    <td>
      <label>
          Date of return to work: (YYYY-MM-DD)<br/>
          <input type="date" name="returnDate"/>
          <input type="hidden" name="returnDate_type" value="Date"/>
      </label>
    </td>
  </tr>
  <tr>
    <td>
      <label>
        <input type="checkbox" name="vacationPay"/> Vacation pay requested
        <input type="hidden" name="vacationPay_boolean" value="true"/>
```

```
        </label>
      </td>
    </tr>
    <tr>
      <td>
        <label>
          Motivation:<br/>
         <textarea name="vacationMotivation" value=""></textarea>
        </label>
      </td>
    </tr>
</table>
```

[试验性的] 利用隐藏域提供诸如类型及必需的额外信息的这一机制将会在接下来发布的某个版本中做适当修改。以后会根据 FormService.getStartFormData 以及 FormService.getTaskFormData 来获得这类元数据。

隐藏域为 Activiti Explorer 客户端应用程序提供了额外的信息。所以浏览器内的 Javascript 能够利用这些隐藏域来增强相应的输入域。例如，可以将某个文本域指定为日期类型，这样 Activiti Exploer 就会显示出一个日期选择器。

- 变量名必须是驼峰大小写的格式
- 被存储的流程变量的默认类型是 string 类型。利用输入域变量名后面跟'*_type*'的隐藏域来定义此类型（这也定义了由 HTML 输入域到变量的转换）：

  ```
  <input type="hidden" name="numberOfDays_type" value="Integer" />
  ```

  目前，支持 String, Integer, Boolean, Date。
- 通过添加一个名称为输入域变量名后跟'_required'的隐藏域可以将输入域变量声明为必填项。

  ```
  <input type="hidden" name="employeeName_required" value="true" />
  ```
- 在 Activiti Explorer 内，*Date* 类型必须是 ISO 8601（*YYYY-MM-DD*）的格式。该域将使用浏览器中的本地日期选择器工具（利用 HTML 5 中的 input type="date"），或者退而使用 YUI 日历组建的弹出式日历选择器。当然，仍然可以手动输入日期，在你输入的同时会被验证。
- 演示中的 Integer 类型的表单域使用了 HTML 5 的 input *type=number* 进行了增强，它允许了在支持 HTML 5 的浏览器内进行本地校验以及自定义输入域，尽管 Activiti-Explorer 也提供了客户端的校验。

预计以后 Activiti Explorer 将使用 FormService.getStartFormData 来代替这些隐藏域的值来增强表单输入域。这就是为什么隐藏域的部分被标记了[试验性的]。

使用默认的表单引擎（JUEL），利用 FormService.getRenderStartForm 来取得渲染过的表单字符串：

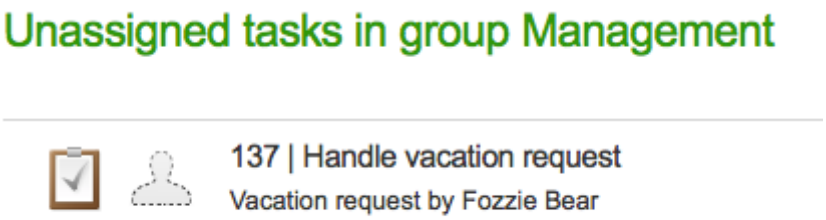
```
String FormService.getRenderedStartForm(String processDefinitionId)
```

使用 FormService.submitStartFormData 来启动用户在表单输入的属性的新流程实例：

```
ProcessDefinition FormService.submitStartFormData(String processDefinitionId, Map<String, String> properties)
```

要想了解使用 FormService 的方法来启动新的流程实例与使用 ProcessInstance RuntimeService.startProcessInstanceById(String processDefinitionId)之间的不同，请阅读“表单属性”一节。

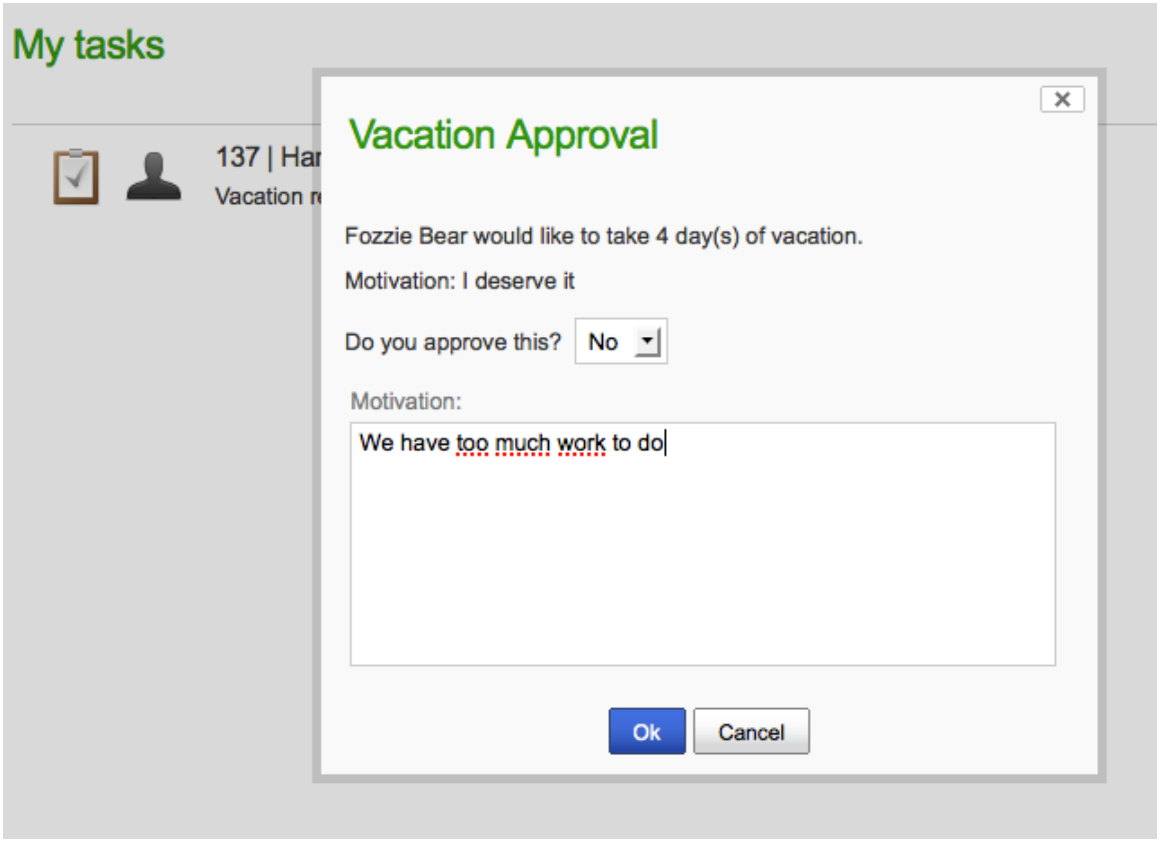
提交表单后，流程实例被启动，然后由管理层的一员来处理该请求。



相应的用户任务关联了一个任务表单，其使用了员工在启动表单中输入的变量。这些变量是作为表达式引用的，且在运行时被解析为文本。

```
<h1>Vacation Approval</h1>
<p>
  ${employeeName} would like to take ${numberOfDays} day(s) of vacation.
</p>
<p>
  Motivation: ${vacationMotivation}
</p>
<p>
  Do you approve this?
  <select name="vacationApproved">
    <option value="true">Yes</option>
    <option value="false">No</option>
   </select>
   <input type="hidden" name="vacationApproved_type" value="Boolean" />
</p>
<p>
  <label>
    Motivation:<br/>
    <textarea name="managerMotivation" value=""></textarea>
  </label>
</p>
```

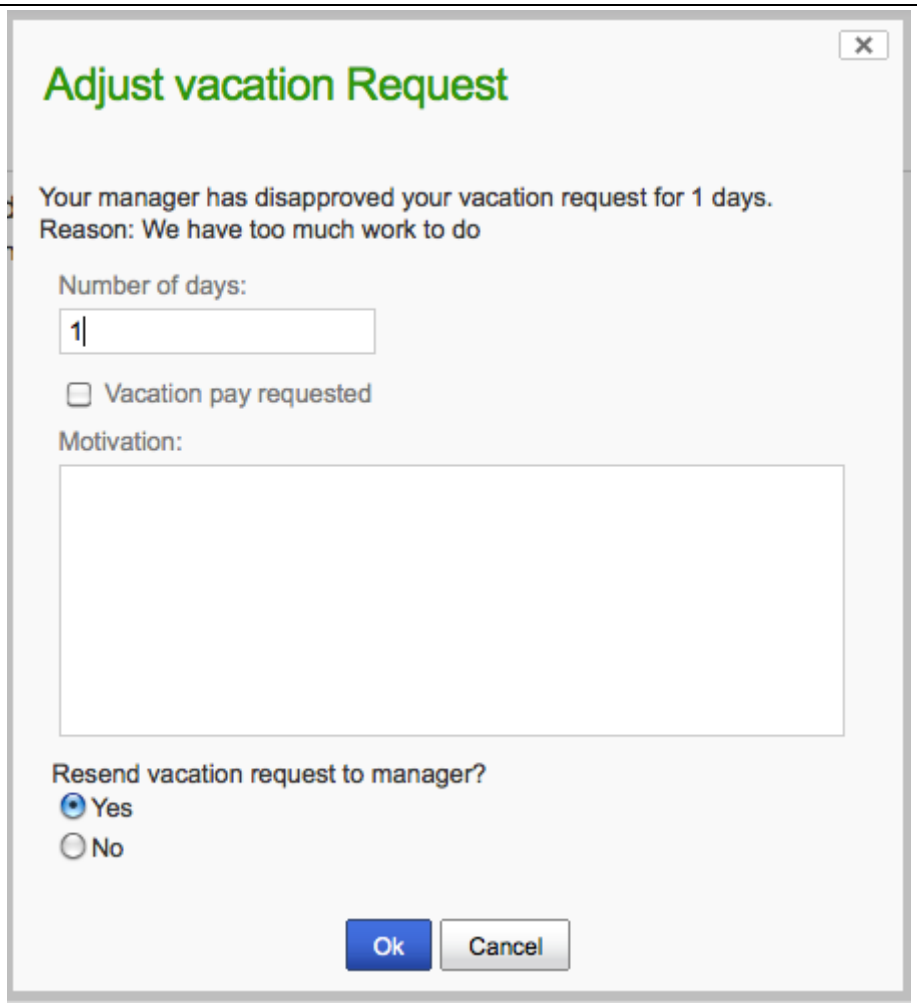
经理现在就可以在表单中选择合适的输入来表明是不是批准休假申请。



结果作为流程变量被存储起来，表单被提交后，其将使用在单一分支上。

```
<sequenceFlow id="flow5" sourceRef="requestApprovedDecision" targetRef="adjustVacationRequestTask">
  <conditionExpression xsi:type="tFormalExpression">${!vacationApproved}</conditionExpression>
</sequenceFlow>
```

根据经理在用户任务输入的结果，会在员工的个人任务列表中添加一个新的任务，写的是请假申请没有被批准，如果想请假必须重填申请。



员工现在可以选择重新发送请假申请，将流程重新带回经理的用户任务。或者，员工可以放弃申请，从而结束该流程。

```
<h1>Adjust vacation Request</h1>
<p>
   Your manager has disapproved your vacation request for ${numberOfDays} days. <br/>
   Reason: ${managerMotivation}
</p>
<p>
   Number of days:<br/>
   <input type="text" name="numberOfDays" value="${numberOfDays}" />
   <input type="hidden" name="numberOfDays_type" value="Integer" />
</p
...
```

8.2 外部的表单渲染

我们在上面展示了内置的任务表单渲染。但该 API 也允许执行 Activiti 引擎之外的你自己的任务的表单渲染。这些步骤解释了用于自己渲染任务表单的技术。

本质上，所有需要渲染表单的数据都是在这两个服务方法之一内进行装配的：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，请参看“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）。可将其作为部署的资源来使用，其可以这样来获得：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，任何其它的目的，你都可以使用此功能来访问部署的资源。

属性<userTask activiti:formKey=”…”是由流程 API 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 值存储在此表单属性中，然后使用算法或转换来得到实际需要使用的模板。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，甚至可以是适用 IM 的表单和 email 的表单模板。

8.3 表单属性

业务流程的相关所有信息或者包含在流程变量内，或者可以通过流程变量来引用。Activiti 支持存储复杂 Java 对象类型的流程变量，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 Map<String, String>属性的一个值的转换逻辑（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）。

然后，任何 UI 技术都可以在这些属性之上构建表单。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 StartFormData FormService.getStartFormData(String processDefinitionId)和 TaskFormdata FormService.getTaskFormData(String taskId)的返回值 FormData 中取得。这些属性是从流程变量获取的。

默认，内置的表单引擎‘对待’这些属性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。比如，以下声明：

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

表单能看到所有流程变量，但 formService.getStartFormData(String processDefinitionId).getFormProperties()将为空。

上面的例子中，所有提交的属性会作为流程变量被存储。这意味着仅通过在表单内添加新的输入域就能存储新的变量。

属性是由流程变量得来的，但没有必要非得将其作为流程变量存储起来。比如，流程变量有可能是个 Address 类的 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起来或者使用 UEL 值表达式将其作为流程变量的嵌套属性存储起来，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起来。

同时类型转换也可以作为处理表单属性和流程变量的之间的一部分。

例如：

```
<userTask id="task">
  <extensionElements>
    <activiti:formProperty id="room" />
    <activiti:formProperty id="duration" type="long"/>
    <activiti:formProperty id="speaker" variable="SpeakerName" writable="false" />
    <activiti:formProperty id="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。其只能在 TaskFormData 中使用。如果属性 speaker 被提交，就会抛出 ActivitiException。类似的，设置为 readable="false"的属性将从 FormData 内被排除掉，但在提交的时候然后会对其进行处理。
- 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。如果设置为 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。

也可以提供 FormData 的部分类型元数据，其可由方法 StartFormData FormService.getStartFormData(String processDefinitionId)和 TaskFormdata FormService.getTaskFormData(String taskId)返回。

我们支持以下表单属性的类型：

- 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）

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

```
public interface FormProperty {
  /** the key used to submit the property in {@link FormService#submitStartFormData(String, java.util.Map)}
   * or {@link FormService#submitTaskFormData(String, java.util.Map)} */
  String getId();
  /** the display label */
  String getName();
  /** one of the types defined in this interface like e.g. {@link #TYPE_STRING} */
```

```
  FormType getType();
  /** optional value that should be used to display in this property */
  String getValue();
  /** is this property read to be displayed in the form and made accessible with the methods
   * {@link FormService#getStartFormData(String)} and {@link FormService#getTaskFormData(String)}. */
  boolean isReadable();
  /** is this property expected when a user submits the form? */
  boolean isWritable();
  /** is this property a required input field */
  boolean isRequired();
}
```

例如：

```
<startEvent id="start">
  <extensionElements>
    <activiti:formProperty id="speaker"
      name="Speaker"
      variable="SpeakerName"
      type="string" />

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

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

  </extensionElements>
</startEvent>
```

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

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时填充，也可以由服务任务产生。
- 重用现有的领域模型，而无需编写显示获取实体及更新其值的服务。
- 根据现有实体的属性做出判断（gateways）。
- …

9.1 要求

只支持符合如下要求的实体：

- 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。
- 实体中要有利用@Id 注解的主键，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（不包括 boolean）、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我们自己定义的 EntityManagerFactory（此例中，为 open-jpa 实体管理器）。注意下面的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 的完整可行的例子可以在 actviti-spring-examples 下找到（/activiti-spring/src/test/java/org/activiti/spring/test/jpa/JPASpringTest.java）。

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

<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
  <property name="dataSource" ref="dataSource" />
  <property name="transactionManager" ref="transactionManager" />
  <property name="databaseSchemaUpdate" value="true" />
  <property name="jpaEntityManagerFactory" ref="entityManagerFactory" />
  <property name="jpaHandleTransaction" value="true" />
  <property name="jpaCloseEntityManager" value="true" />
  <property name="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、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，根据 META-INF/persistence.xml 为持久化单元创建 EntityManagerFactory。包括持久化单元所包含的类，及一些提供商特有的配置。

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。在运行测试前，我们会创建一个实体并将其保存起来。

```
@Entity(name = "JPA_ENTITY_FIELD")
public class 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;
   }

   public void setId(Long id) {
      this.id = id;
   }

   public String getValue() {
      return value;
   }

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

启动一个新的流程实例，作为变量添加了该实体。至于其它变量，将被存储到引擎的持久化数据库中。下次请求该变量时，将根据该类和存储的 Id 从 EntityManaer 加载它。

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

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

流程定义的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，其是从与当前引擎上下文关联的 EntityManager 中加载的。

```
<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，且对实体的修改也被推进了数据库。在获取变量 entityToUpdate 的 value 时，会再次加载该变量且我们获得是 value 属性被设置为 updateValue 的实体。

```
//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

可以查询拥有某一 JPA 实体变量的 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。

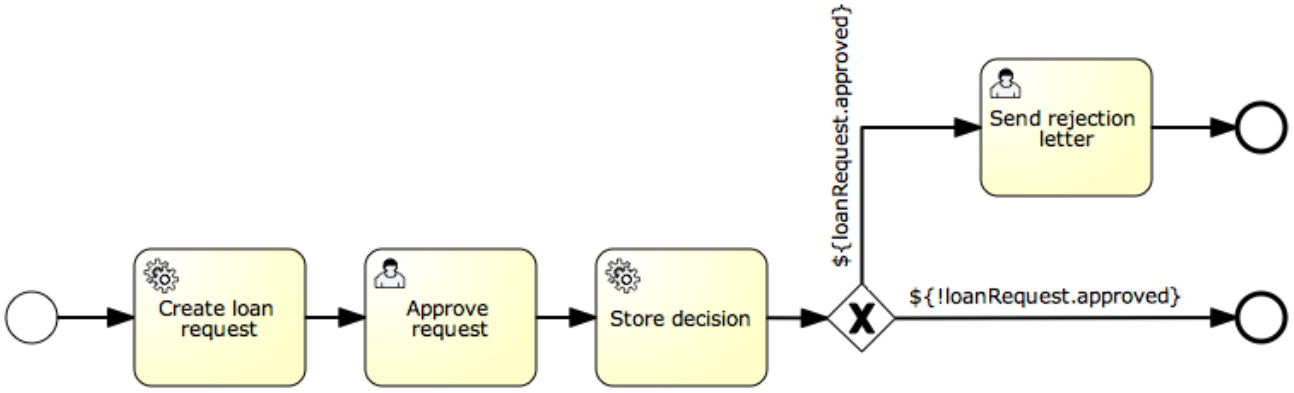
```
ProcessInstance result = runtimeService.createProcessInstanceQuery().variableValueEquals("entityToQuery",
entityToQuery).singleResult();
```

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。
- 利用 Activiti，我们可以使用通过现有 bean 获得的现有实体，并可以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，使用现有的 LoanRequestBean 和启动流程时接收的变量（比如，可以来自于开始的表单）来创建新的 LoanRequest。使用作为变量存储表达式结果的 activiti:resultVariable 将被创建的实体作为变量存储起来。
  - 用户任务，允许经理查看请求，然后批准/不批准，将以一个 boolean 类型的变量 approvedByManager 存储。
  - 服务任务，用来更新贷款请求实体来使实体与流程同步。
  - 根据实体属性 approved 的值，利用单一分支来决定下一步选择哪条路径：当批准了请求时，流程将结束；否则，会另有一个任务（发送拒绝信），这样就可以使用拒绝信手工通知用户了。（译注，下图中单一分支上的条件正好颠倒了）

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"?>
<definitions id="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">

   <process id="LoanRequestProcess" name="Process creating and handling loan request">
      <startEvent id='theStart' />
      <sequenceFlow id='flow1' sourceRef='theStart' targetRef='createLoanRequest' />

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

      <userTask id="approveTask" name="Approve request" />
      <sequenceFlow id='flow3' sourceRef='approveTask' targetRef='approveOrDissaprove' />

      <serviceTask id='approveOrDissaprove' name='Store decision'
         activiti:expression="${loanRequest.setApproved(approvedByManager)}" />
      <sequenceFlow id='flow4' sourceRef='approveOrDissaprove' targetRef='exclusiveGw' />

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

      <userTask id="sendRejectionLetter" name="Send rejection letter" />
      <sequenceFlow id='flow5' sourceRef='sendRejectionLetter' targetRef='theOtherEnd' />
```

```
    <endEvent id='theEnd' />
    <endEvent id='theOtherEnd' />
  </process>

</definitions>
```

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

第十章、历史（History）

历史是这样的组件：捕获流程执行期间发生了什么并将其永久性的存储起来。与运行时的数据相反，历史的数据将一直保留在数据库中，即便是流程实例已经完成。

存在 4 类历史的实例：

- HistoricProcessInstances，包含着有关当前的以及结束了的流程实例的信息。
- HistoricActivityInstances，包含着有关一个活动执行的信息。
- 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。

因为数据库中包含着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的流程实例的数据的访问，这样也能保持高性能的执行。

后面，将在 Activiti Explorer 和 Activiti Probe 中使用此信息。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。完成可能的描述可以在 javadocs 中 org.activiti.engine.history 包找到。

10.1.1 HistoricProcessInstanceQuery

获取完成的且流程定义’XXX’中所有完成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

获取类型为’serviceTask’，使用的流程定义的 id 为’XXX’，且已经完成了的最后一个 HistoricActivitiInstance。

```
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取 id 为 123 的流程所完成的所有变量更新。该查询只返回 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()
```

最后这个例子获取了所有在 id 为”123”的任务上执行的变量更新。返回设置在任务上的变量的所有 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 TaskLinstener 内的 DelegateTask 上对任务的本地变量进行设置。

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

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

10.1.4 HistoricTaskInstanceQuery

获取完成了的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);
```

获取删除原因中包含”invalid”的被删除的，且最终分配给用户’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定义在 ProcessEngineConfiguration 上的 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>
```

可以配置以下历史的级别：

- none：忽略所有历史归档。这时运行时流程的执行效率最高，但没有任何历史性的信息可供使用。
- acitivity：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并查证。
- 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]目前像在 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

看用户任务所设置的值。查看 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 的工具堆栈提供了 Actiti Modeler 和 Activiti Designer 这两种建模/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应该有任何技术性知识。然后可以利用 Activiti Designer 添加一些必要的技术参数，像 Java 服务任务、执行监听器等。因为 Activiti Designer 功能的引入，这一工作流得到了很好的支持。

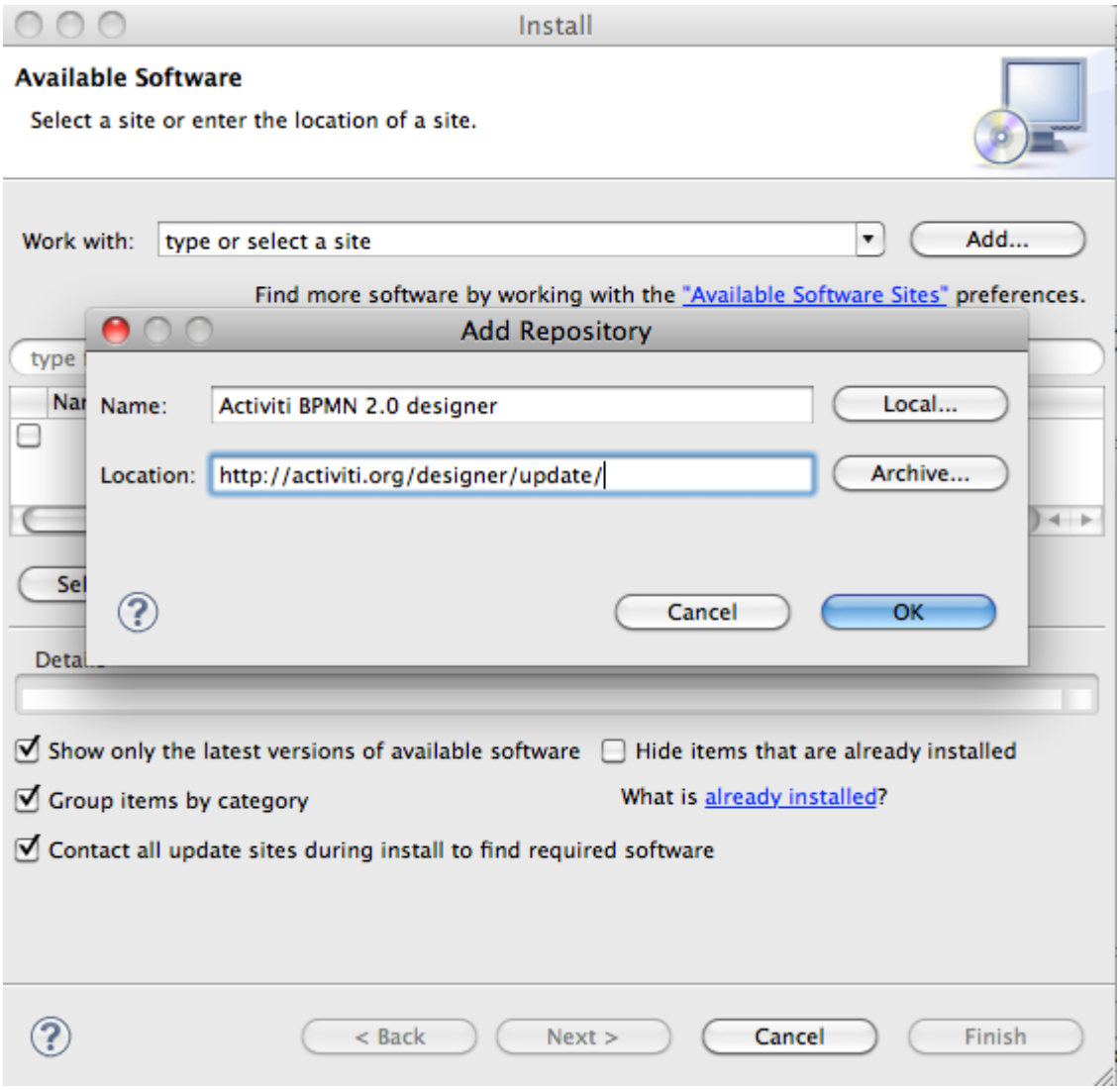
请注意：**Activiti Designer 仍然还是"beta"状态**。这意味着我们还不支持版本的兼容性。尽管，利用 Activiti Designer 支持的 BPMN 2.0 XML 你能够非常容易地将你的流程定义更新到新版的 Activiti Designer。同时要注意并不是所有 Activiti 引擎支持的 BPMN 2.0 的构造都被 Activiti Designer 所支持。

11.1 安装

以下安装指南是在 Eclipse 经典 Helios 上验证的。（译注，请下载 Eclipse IDE for Java EE Developers）

打开 Help→Install New Software。在如下面板中，点击 *Add* 按钮，然后填写下列字段：

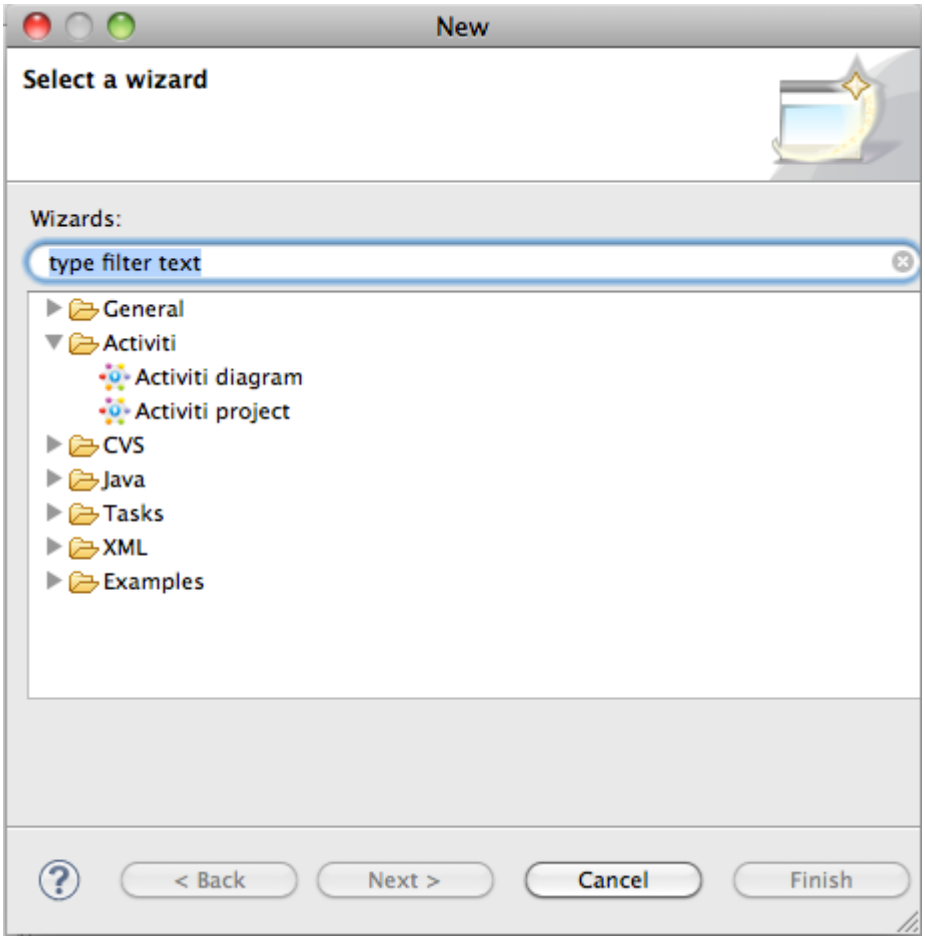
- **Name**：Activiti BPMN 2.0 designer
- **Location**：http://activiti.org/designer/update/



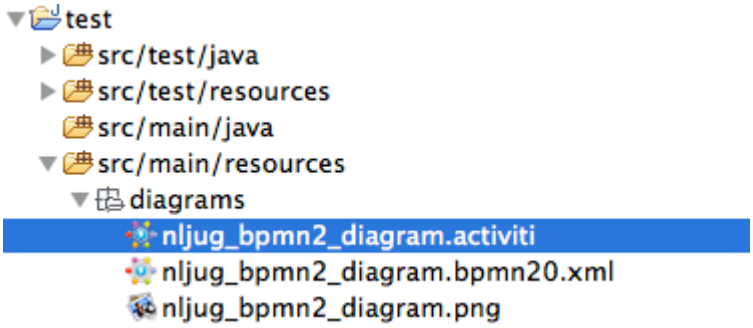
务必不要选中**”Contact all updates sites..”**，因为所有必需的插件都能从 Activiti 更新站点下载。

11.2 Activiti Designer 编辑器的特性

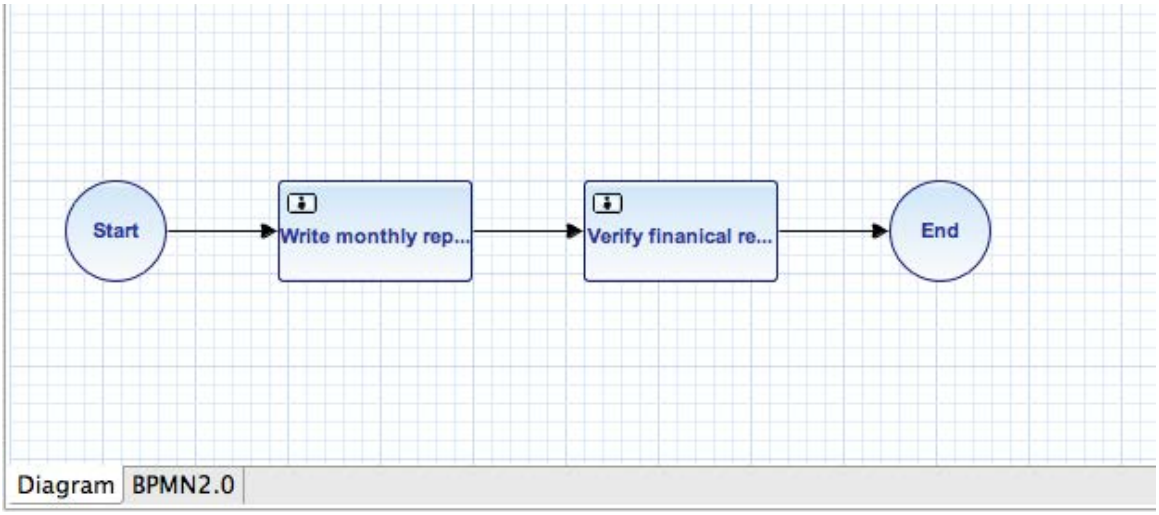
- 创建 Activiti 项目和图形



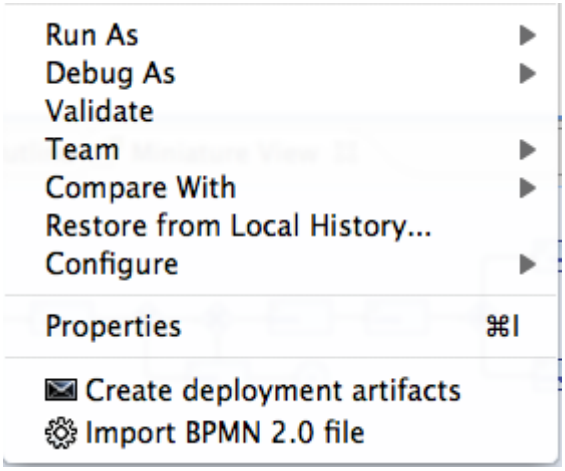
- 每次保存 Activiti 图形时都会自动生成一个 BPMN 2.0 XML 文件和一张流程图片（可以在 Eclipse preference 中的 Activiti 标签内将关闭自动生成）。



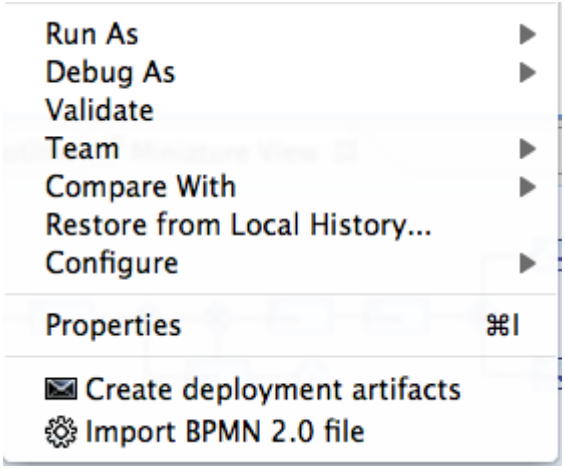
- Activiti 设计器被实现为多页的设计器。指的是第一页展示的是流程图，第二夜展示的 BPMN 2.0 XML。每次保存图形时都会将图形上的修改输出到 BPMN 2.0 XML 文件上。这同样适用于 BPMN 2.0 XML 文件。当你编辑了 BPMN 2.0 XML 文件，在保存文件时修改也会被保存进图形上。注意这只支持在 diagram 试图下编辑、创建的构造和属性。



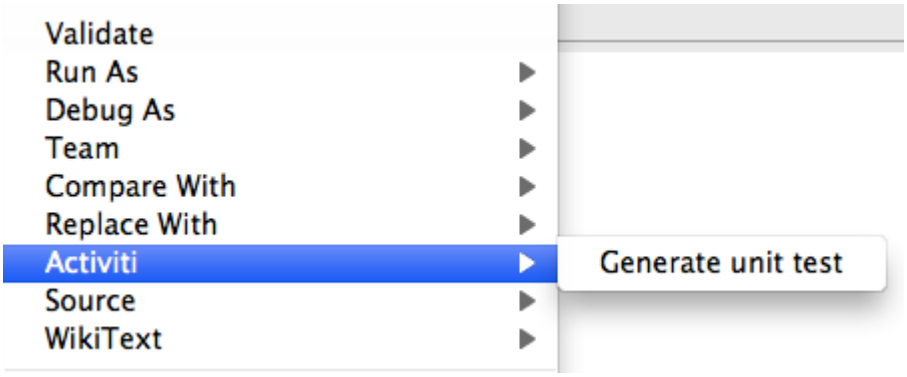
- BPMN 2.0 的 XML 文件可以导进 Activiti Designer，还会有图片被创建。支持两种导入 BPMN 2.0 的 XML 文件。一是右击 package exporler 中的 Activiti 项目，然后选择弹出菜单底部的 Import BPMN 2.0 file 选项。接下来就可以通过文件选择框选择 BPMN 2.0 的 XML 文件并创建图形了。第二个选择是将 BPMN 2.0 的 XML 文件导入到项目下的 src/main/resources/diagrams 文件夹内（文件名必须以.bpmn20.xml 结尾）。接下来在打开此 BPMN 2.0 的 XML 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，但在这个版本中仅仅是对 BPMN 2.0 的 XML 进行解析并分析后创建的图形，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。



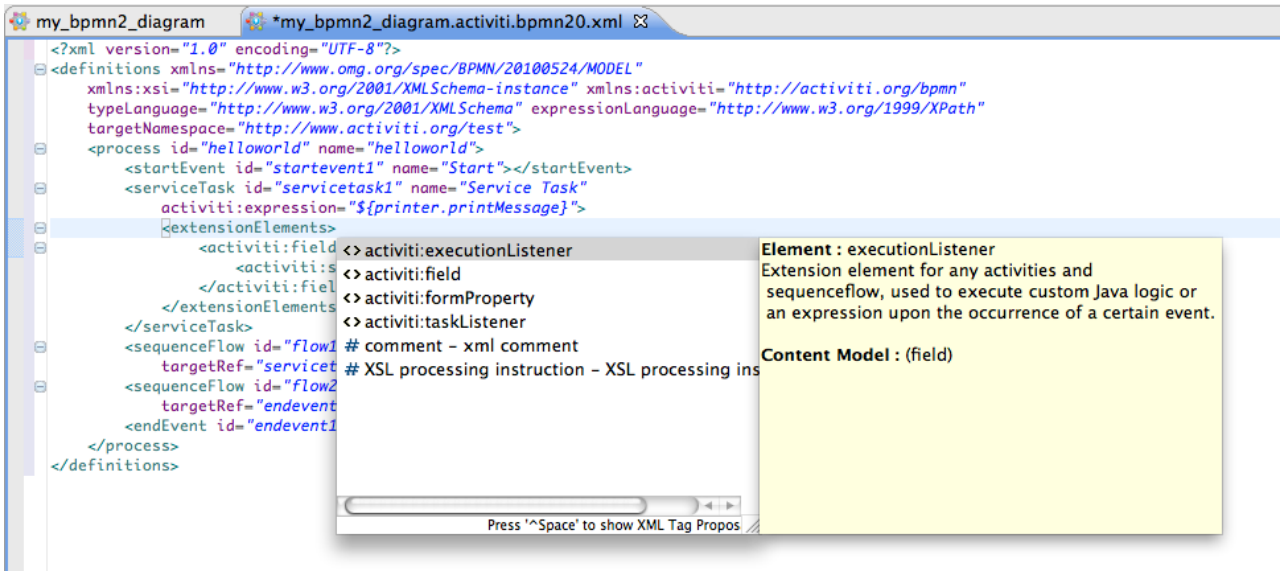
- 对于部署的 BAR 文件以及可选的 JAR 文件，可以通过右击 package exporler 视图中的 Activiti 项目，然后选择弹出菜单底部的 *Create deployment artifacts* 选项由 Activiti Designer 来创建。更多关于 Designer 的部署功能，见部署一节。



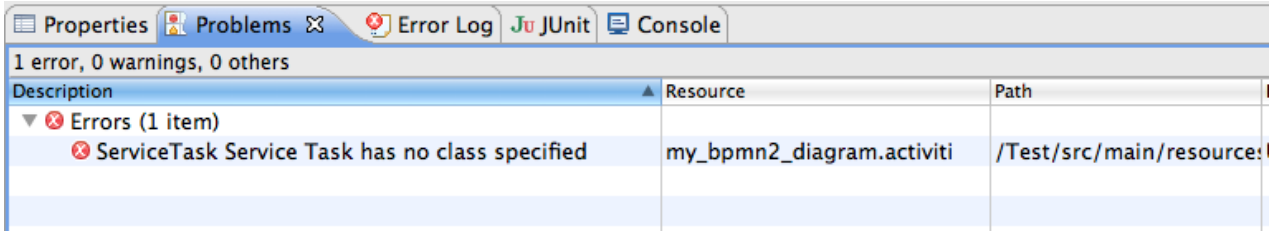
- 生成单元测试（在 package explorer 视图下右击 BPMN 2.0 的 XML 文件，然后选择 *generate unit test*）。与 Activiti 配置一块生成的单元测试是运行在 H2 数据库上的。你现在就可以运行单元测试来测试你的流程定义了。



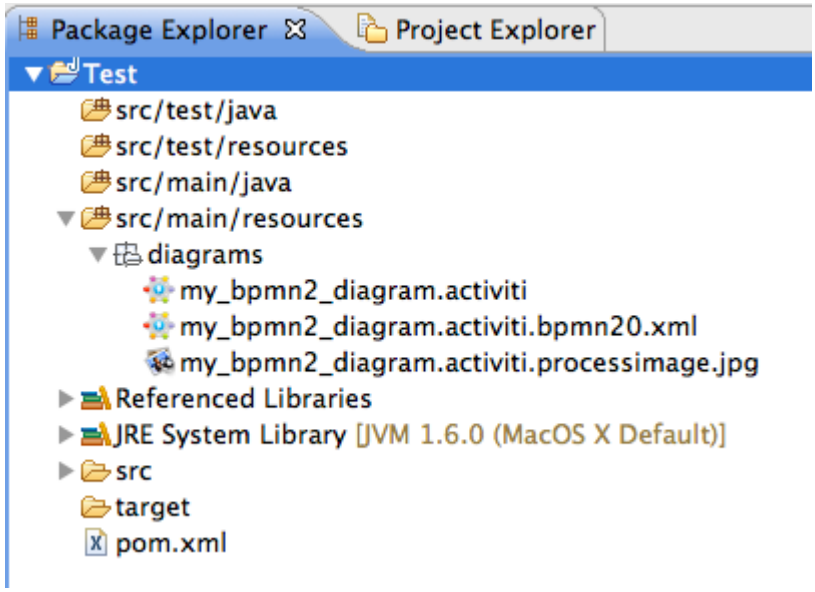
- BPMN 2.0 的 XML 文件是在 Activiti XML 的编辑器中打开的，它提供了内容提示。注意配置了 2 个主要的 XSDs，BPMN 2.0 规范的 XSD 和 Activiti 扩展 XSD。这两个 XSDs 还没很好地统一。



- 每次 Activiti 图形保存时都会执行基本的校验，错误会在 Eclipse 的 problem 视图内显示。

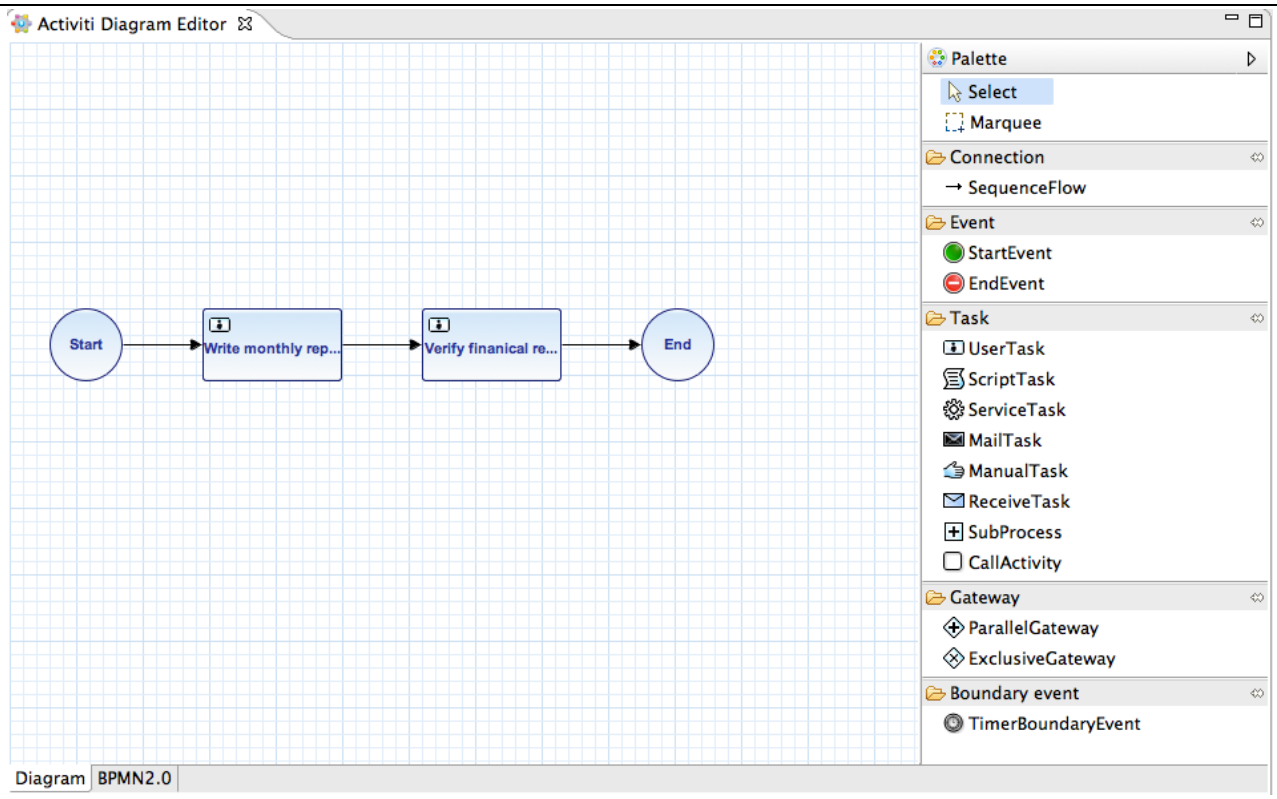


- Activiti 项目可生成为 Maven 项目（译注，生成单元测试时同时生成了一个 pom.xml 文件）。要配置依赖，需要运行 *mvn eclipse:eclipse*，这样 Maven 的依赖才像预期的那样被配置。注意流程设计是不需要那些 Maven 依赖的。只是在运行单元测试时需要。

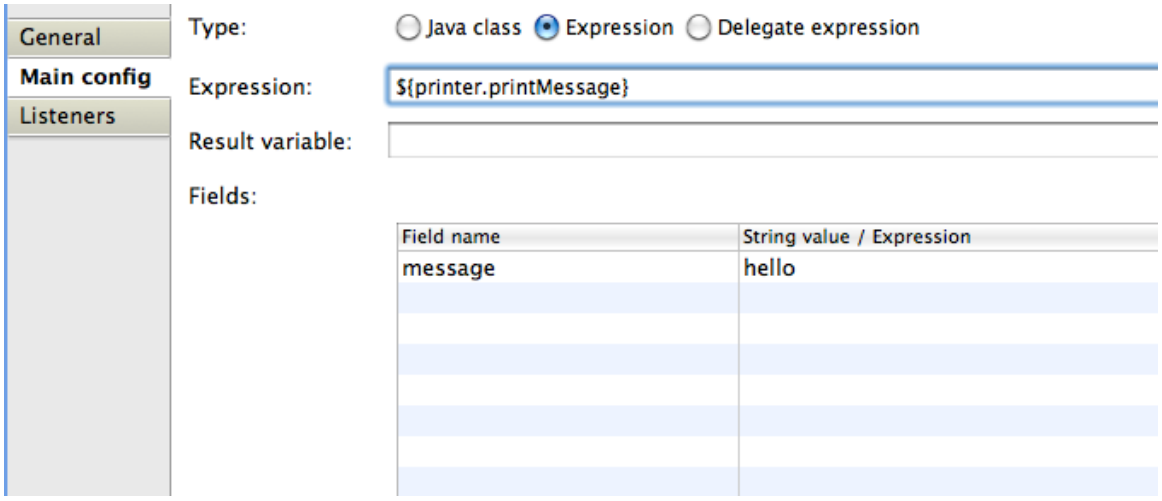


11.3 Activiti Designer 的 BPMN 特性

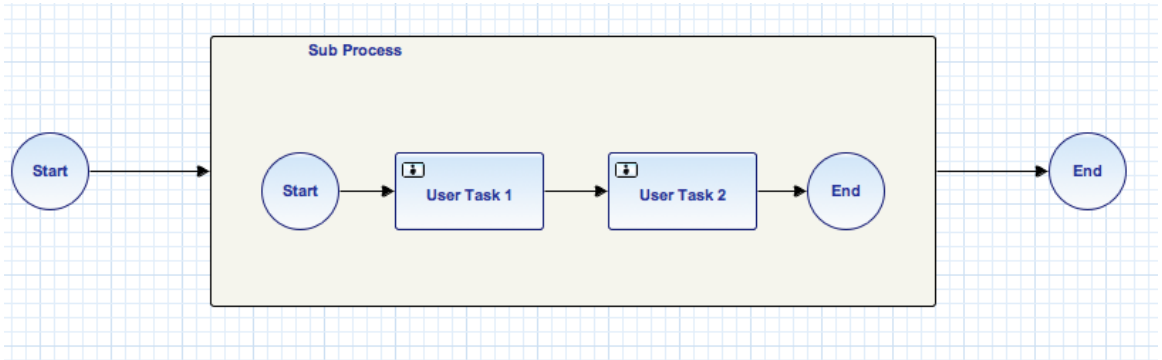
- 支持 start 事件、end 事件、顺序流、并行分支、单一分支、嵌入子流程、调用活动、脚本任务、用户任务、服务任务、邮件任务、手工任务以及定时器边界事件。



- 支持对 Java 服务任务的 Java 类、表达式以及代理表达式的配置。可以在添加的字段中配置表达式。

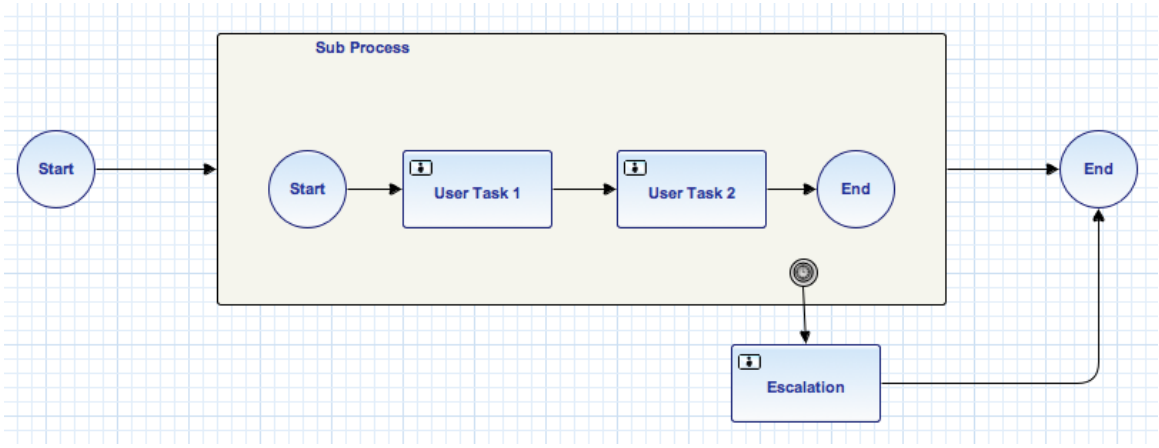


- 支持扩展的嵌入子流程。这意味着不能在将嵌入子流程添加进另一个嵌入子流程。

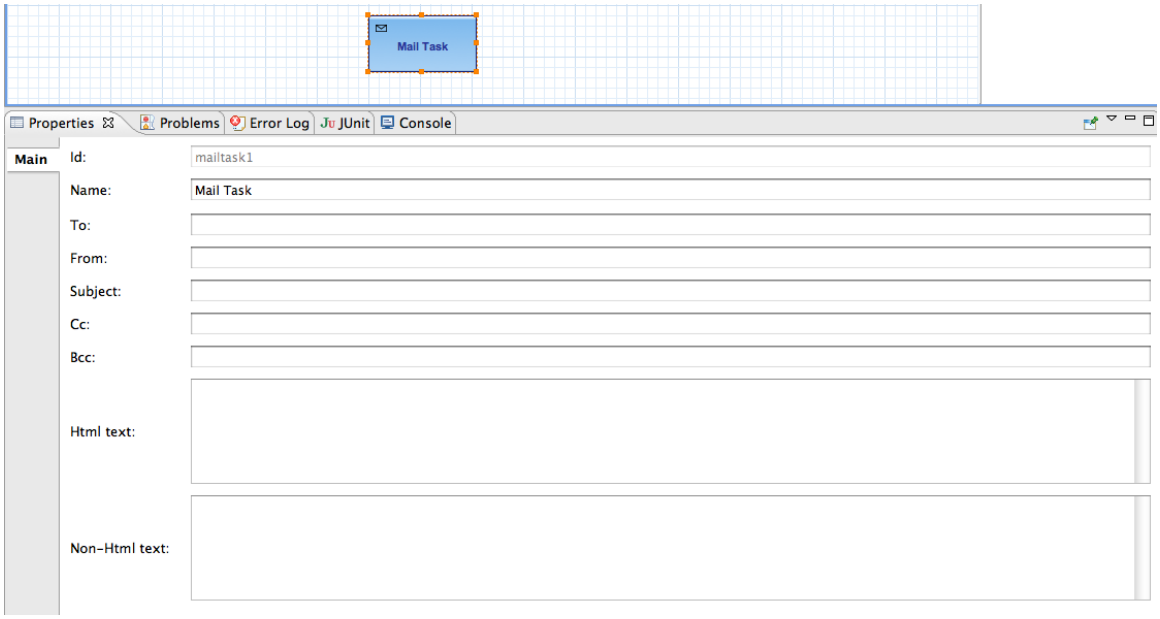


- 支持任务和嵌入子流程上的定时器边界事件。定时器边界事件最大的意义在于可以在 Activiti Designer 中的用户

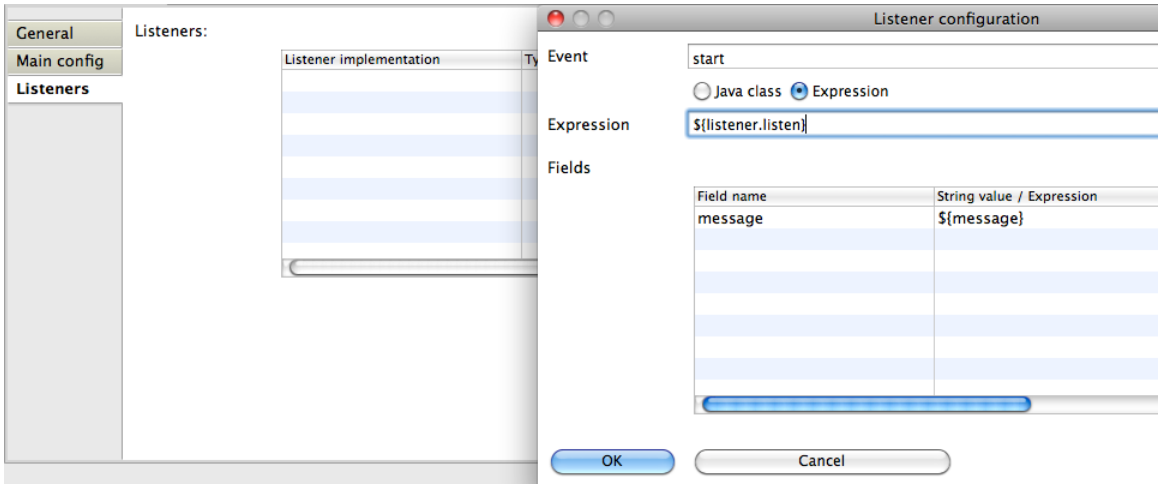
任务或嵌入子流程上使用。



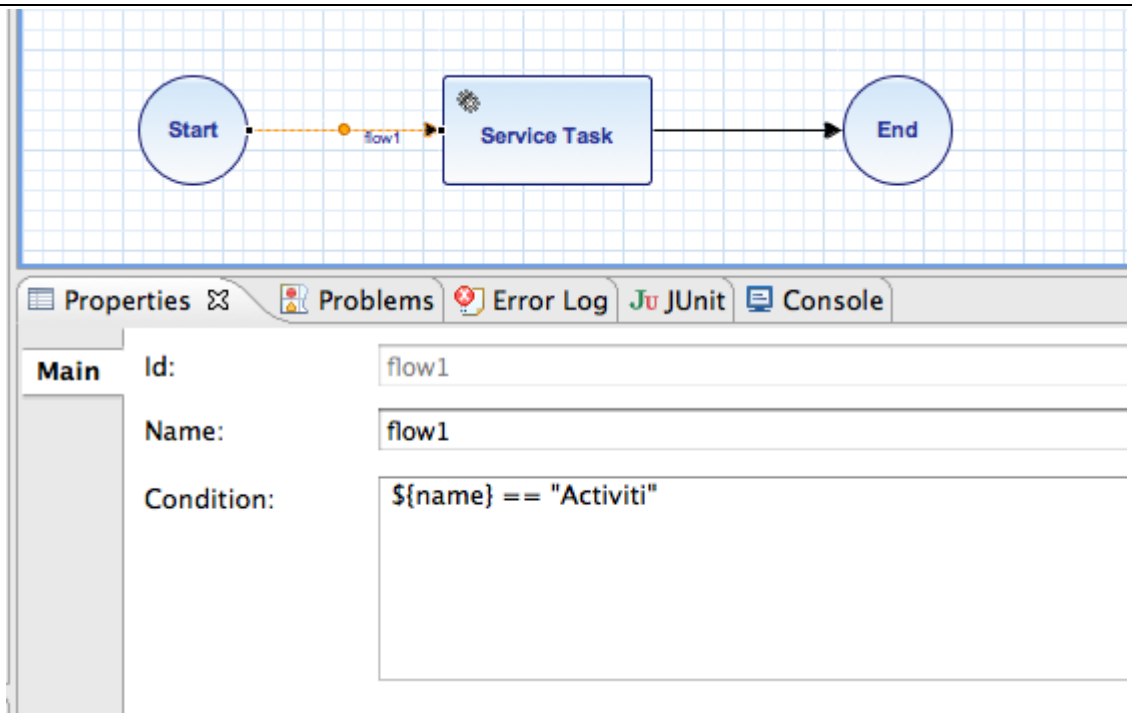
- 支持另外的 Activiti 的扩展，如服务任务、用户任务候选者配置以及脚本任务的配置。



- 支持 Activiti 执行监听和任务监听。可以为执行监听添加字段扩展。

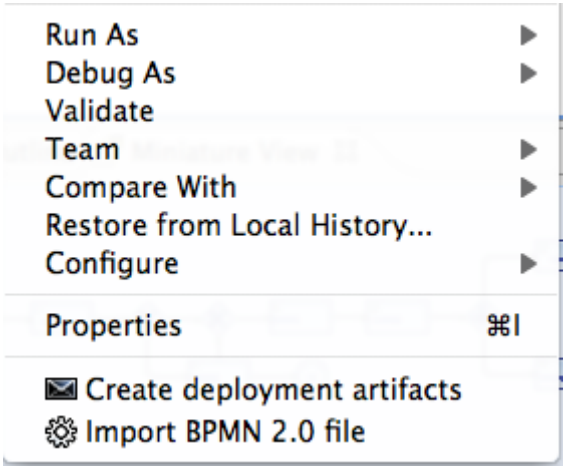


- 支持顺序流上的条件。

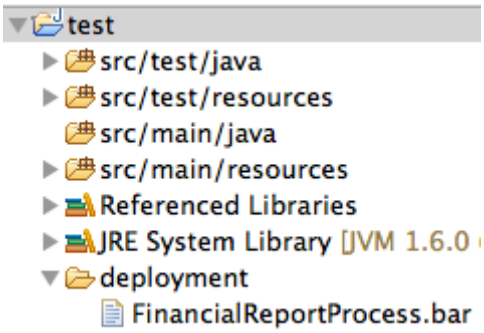


11.4 Activiti Designer 的部署特性

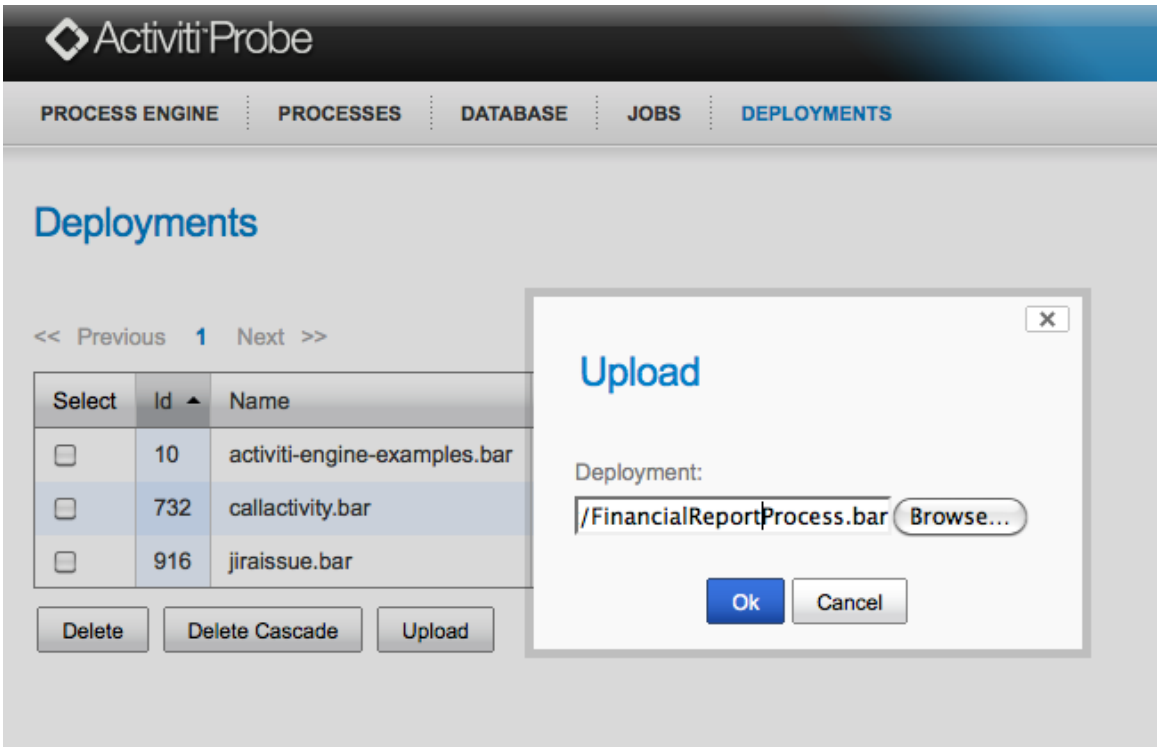
在 Activiti 引擎上不俗流程定义和任务表单并不难。要准备一个 BAR 文件，它包含了流程定义的 BPMN 2.0 的 XML 文件，还可以有任务表单及可以在 Activiti Explorer 中浏览的流程图片。在 Activiti Designer 中创建 BAR 文件是非常简单的。当完成了你的流程实现后，在 package explorer 视图下右击你的 Activiti 项目，然后选择弹出菜单底部的 **Create deployment artifacts** 选项。



接下来，就会创建包含 BAR 文件以及可选的你的 Activiti 项目中的 Java 类的 JAR 文件的 deployment 目录。（译注，bar 文件及 jar 文件都会生成在这一目录下）



现在就可以使用 Activiti Probe 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。



当你的项目中包含了 Java 类时，部署要稍微麻烦一点。这种情况下，Activiti Designer 中的 Create deployment artifacts 同时会创建包含编译过的类的 JAR 文件。必须将这个 JAR 文件部署到你的 Activiti Tomcat 安装路径下的 webapps/activiti-rest/WEB-INF/lib。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。Acitiviti 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 提供的任何或全部 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）。那样的 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依赖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。由你决定是利用 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 文件内添加集成依赖的函数库，将集成的函数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，但这样你可以利用清单文件的一个特定的属性为你的扩展提供名称（这个名称可以显示在 Designer 的某处，它主要是为以后使用在如果 Designer 中有几个扩展的时候）。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。

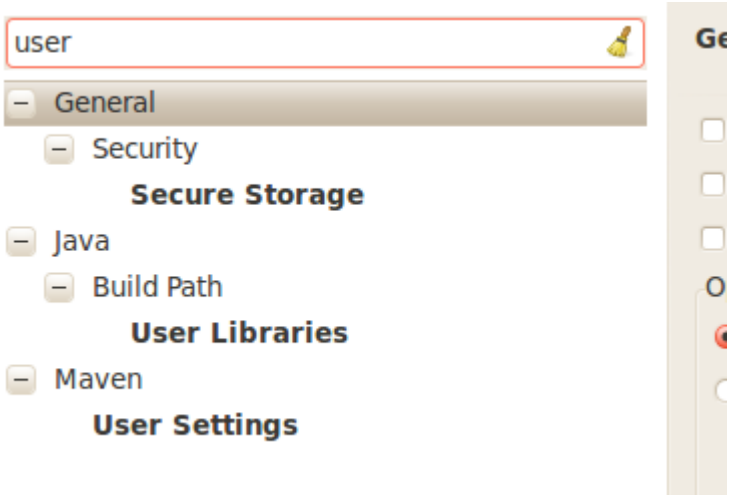
注意
当然你也可以使用 m2eclipse 插件，从项目的上下文菜单（右击）。然后在项目上下文菜单中选择 Maven→Update project configuration。

设置就这样。现在就可以开始创建 Activiti Designer 的定制了。

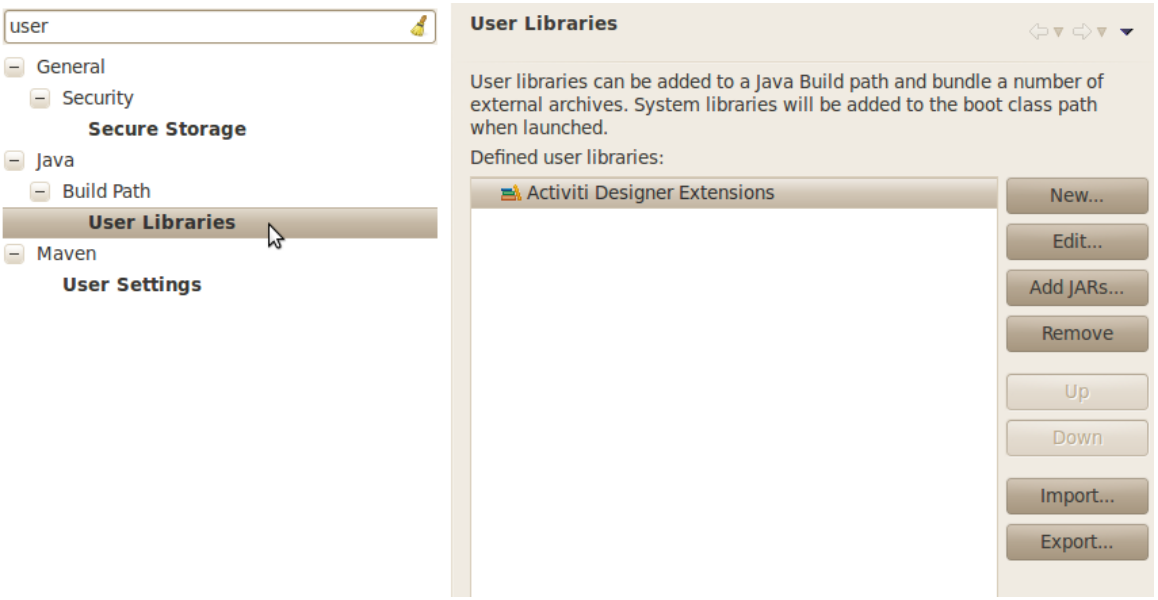
11.5.1.2 将扩展应用到 Activiti Designer

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：

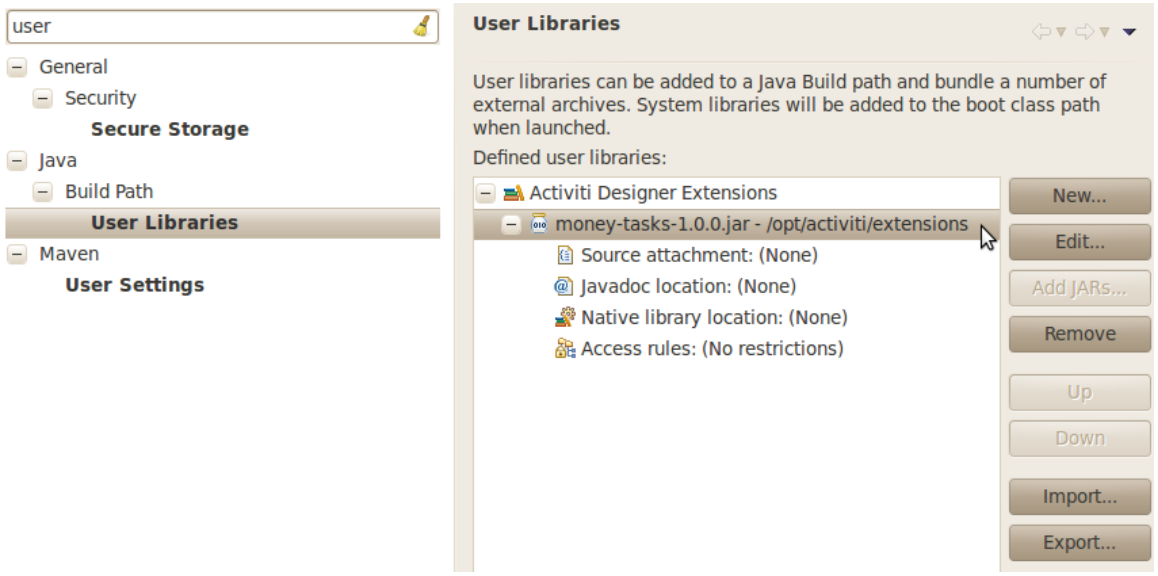
- 一旦创建完扩展的 JAR（例如，通过执行项目内使用了 Maven 的 mvn 的安装程序来构建的 JAR），需要将其放在计算机中安装 Activiti Designer 的地方；
- 将该扩展存储到硬盘中一个能保留并记住的地方；
- 启动 Activiti Designer，选择菜单栏中的 Window → Preferences；
- 在 Preferences 屏幕中，输入 user 关键字。在 Java 部分中你应该能看到访问 Eclipse 中的 User Libraries 一项。



- 选择 User Libraries，在右侧显示的树形视图中你可以添加函数库。你会看到用来向 Activiti Designer 添加扩展的默认分组（根据安装的 Eclipse，可能也会看到另外几个分组）。



- 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，preference 屏上会显示作为 Activiti Designer Extensions 分组一部分的这一扩展，如下显示的。



- 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 视图中你可以看到用户函数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

随着项目的设置，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 文件中的类来代表的。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。形状所参照的运行时的类要被引擎使用。就像 Activiti 中所有服务任务一样，这个类必须实现 JavaDelegate。

一个形状的类就是添加了一些注解的普通 Java 类。此类必须实现 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了的方法的说明。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拥有的名称。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 可以使字段成为必需项。如

果用户不填写该字段，就会有消息和红色背景出现。

如果要确保类中变量的属性是它们在 property 窗口出现的顺序，应该指定@Property 注解的 order 属性。

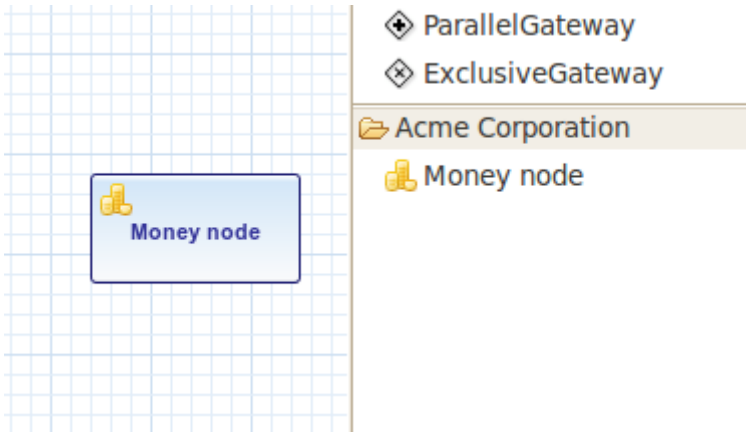
正如你所看到了，有个@Help 注解用来在填写字段时给用户提供一些引导。也可以将@Help 注解用于类本身-信息就会显示在呈现给用户的属性表上方。

下面列出了对 MoneyTask 的详细阐述。添加一个评论字段，且可以看到该节点包含了一个图标。

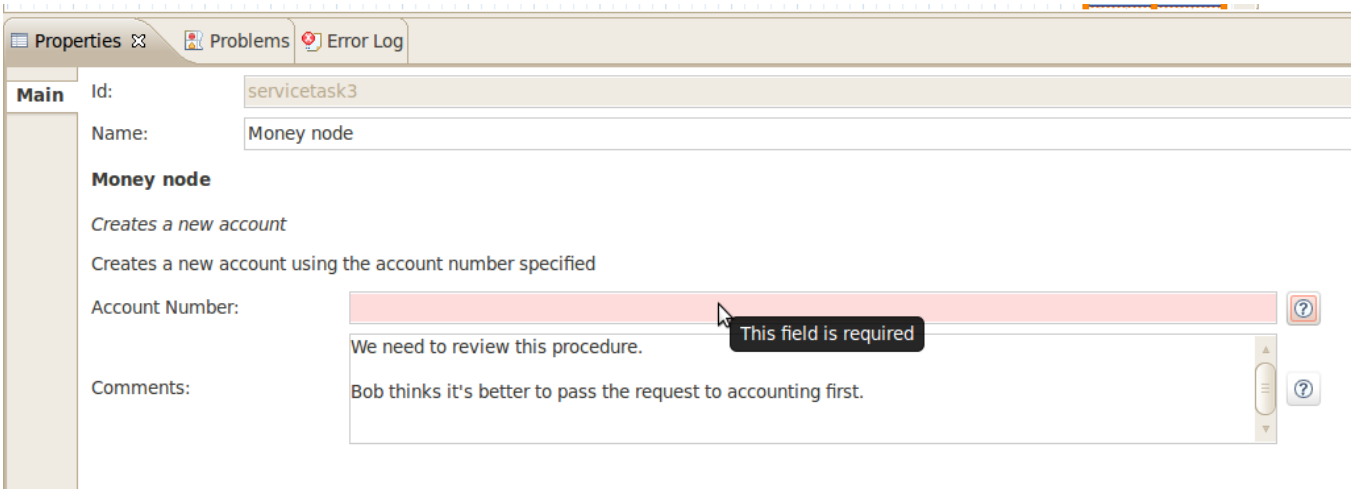
```
/**
 * @author John Doe
 * @version 1
 * @since 1.0.0
 */
@Runtime(delegationClass = "org.acme.runtime.AcmeMoneyJavaDelegation")
@Help(displayHelpShort = "Creates a new account", displayHelpLong = "Creates a new account using the account number specified")
public class AcmeMoneyTask extends AbstractCustomServiceTask {

   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";
   }
}
```

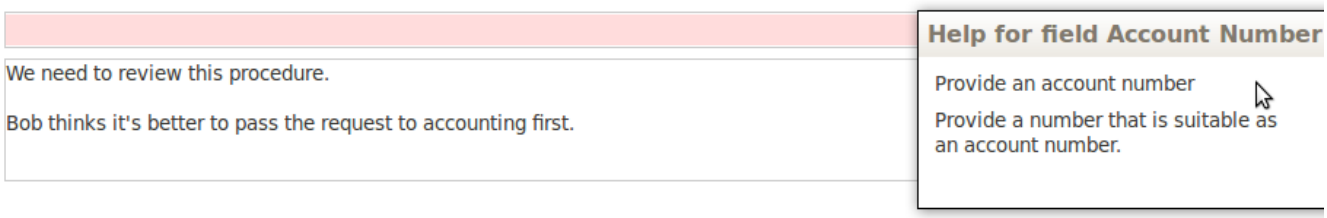
如果拿这个形状来扩展 Activiti Designer，画板和对应的节点看起来如此：



money 任务的属性窗格显示如下。注意对于 accountNumber 字段所要求的信息。



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，如下展示的。



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。指定给 delegationClass 属性的值必须是运行时类的标准名称。注意该运行时的类不必在扩展 JAR 文件内，因为它是依赖于 Activiti 函数库的。

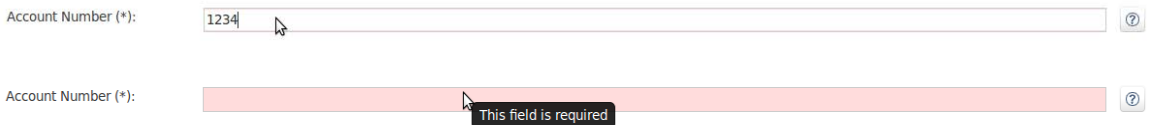
```
@Runtime(delegationClass = "org.acme.runtime.AcmeMoneyJavaDelegation")
```

11.5.1.4 属性的类型

这一节描述属性的类型，你可以将其用于自定义服务任务，通过将自定义服务任务的 type 属性设置为一个 PropertyType 值。

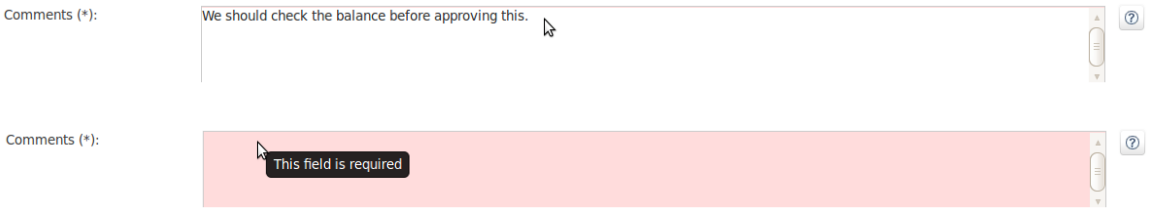
PropertyType.TEXT

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。



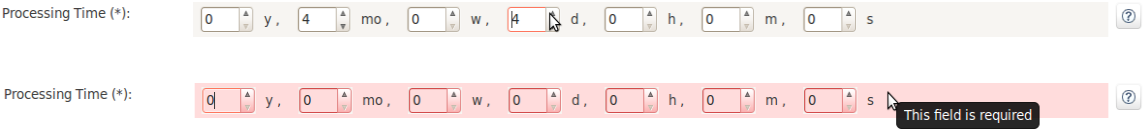
PropertyType.MULTILINE_TEXT

如下所示，创建多行文本域（高度固定为 80 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。



PropertyType.PERIOD

创建结构化的编辑器，使用微调控件修改一些单位来确定一段时间。结果如下所示。可以是必填项（解释为不是所有值都为 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。该字段的值存储为格式为 1y 2mo 3w 4d 5h 6m 7s 的字符串，表示 1 年，2 个月，3 个星期，4 天，5 小时，6 分钟，7 秒。总是对整个字符串进行存储，尽是每个都是 0。



PropertyType.BOOLEAN_CHOICE

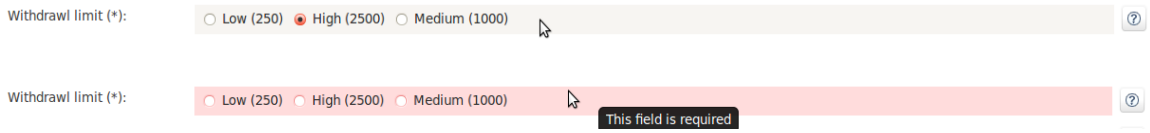
创建单个复选框以控制逻辑或切换的选择。注意你可以指定 Property 注解的 required 属性，但它不会被计算，因为那会让用户不能选择，不管是不是选中了选择框。图形存储的值是 java.lang.Boolean.toString(boolean)，其结果是”true”或”false”。

PropertyType.RADIO_CHOICE

如下所示，创建一组单选按钮。选择任何单选按钮与选择任何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。

这个属性类型希望注解的类的成员同时有@PropertyItems 注解（例子，如下）。利用这个附加的注解，你可以指定必须以字符串数组提供的项目列表。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;
```



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。

这个属性类型希望注解的类的成员同时有@PropertyItems 注解（例子，如下）。利用这个附加的注解，你可以指定必须以字符串数组提供的项目列表。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 })
```

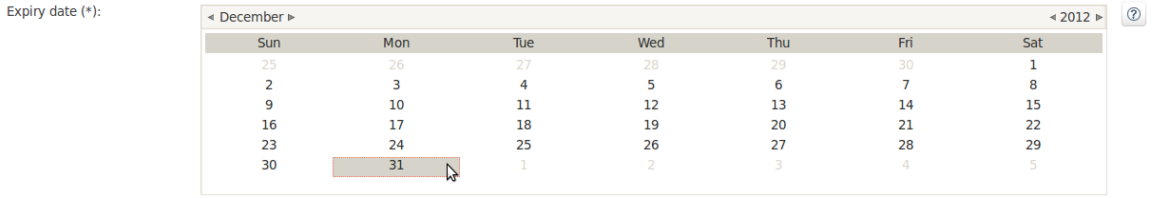
```
private String accountType;
```

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，使用的空间会自动设置对系统日期的选择，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。

这个属性类型希望注解的类成员同时有@ DatePickerProperty 注解（示例，见下文）。利用这个附加的注解，你可以指定图形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，如果不指定它们都有使用的默认值（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，要指定 SWT 的 DateTime 控件所支持的一个整数，因为就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;
```



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。

这个属性类型希望注解的类成员同时有@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的所特有的属性。需要使用 itemClass 属性引用另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，你要在 GroceryListItem 类上定义表格的列。在你的 CustomServiceTask 内，会想要它是像这样的：

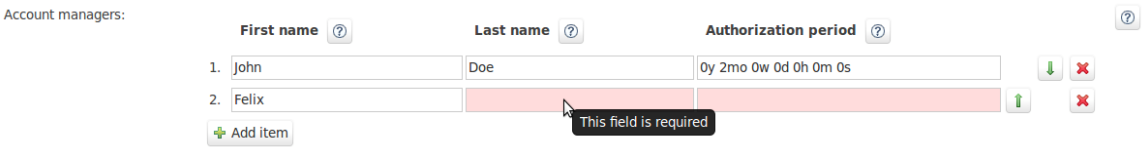
```
@Property(type = PropertyType.DATA_GRID, displayName = "Grocery List")
@DataGridProperty(itemClass = GroceryListItem.class)
private List<GroceryListItem> groceryList;
```

除了使用了数据表格，“itemClass”类使用的注解与你用来指定 CustomServiceTask 的字段的注解是一样的。具体的，目前支持 TEXT、MULTILINE_TEXT 以及 PERIOD。你会注意到表格会为每个字段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。如果，打个比方，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，添加在每个字段上的双击机制都会弹出一个大的 PropertyType 编辑器。用户点击 OK 后，值被存储到字段内，因此它能在表格中被看到。

必填属性的处理类似于处理 TEXT 类型的普通字段，整个表格在任意字段失去焦点时被校验。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，此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。



11.5.1.5 禁用画板中默认形状

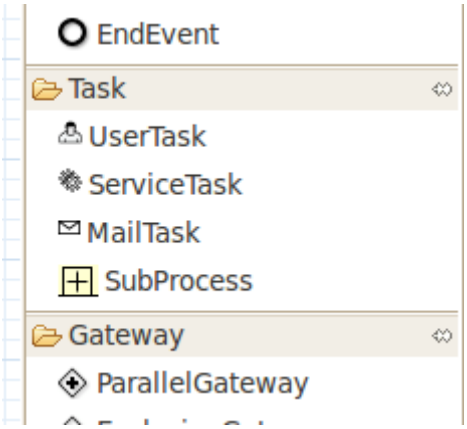
这个定制需要在扩展中包含一个实现了 DefaultPaletteCustomizer 接口的类。不要直接实现这个接口，要创建 AbstractDefaultPaletteCustomizer 基础类的子类。目前，这个类没提供任何功能，但 DefaultPaletteCustomizer 接口今后的版本会提供更多的能力让这个基础类拥有一些合适的默认行为，因为最好继承，这样你的扩展会兼容将来的发布。

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，通过添加将它对应的 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，某个别的抽屉内没有剩余的形状，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某人形状，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。

```
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;
```

```
  }

}
```

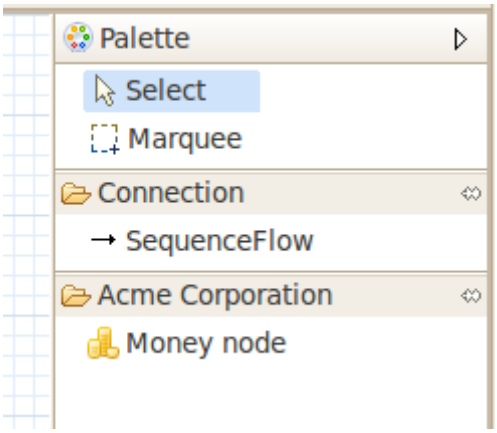
应用这个扩展的结果显示为如下图片。正如你看到的，手工任务和脚本任何不再在任务抽屉中了。



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。默认该工具内已经存在 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。

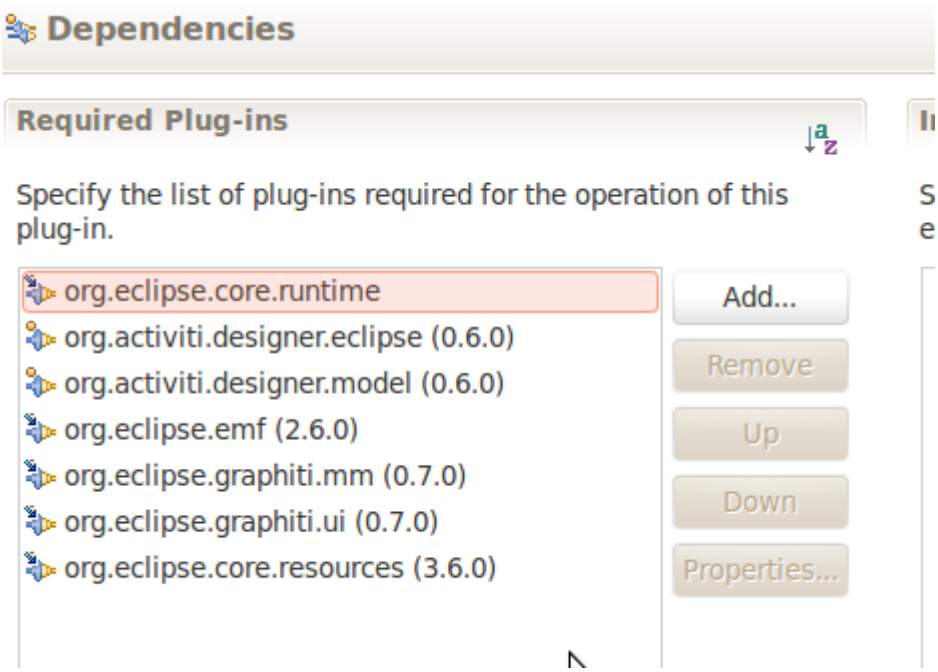
你也可以让 Activiti Designer 在保存图形时公布一些额外的格式。这样的扩展被称为输出装配器，每次用户保存时都会由 Activiti Designer 自动调用。通过在 Eclipse 设置对话框中对保存格式进行设置来启用或禁用这一个行为。

你可以将这两种扩展与 BPMN 2.0 的校验和 BPMN 2.0 的输出进行对比，Activiti Designer 默认在保存时执行流程图片的保存。事实上，这些这些功能使用了相同的扩展特性，你可以将其用到保存你自己的格式中。

往往，会想将 ProcessValidator 和 ExportMarshaller 进行联合。假如你有一些其属性要在生成的流程中使用的 CustomSerivceTask。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 ExportMarshller 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函数库提供的 API。特别是，你将使用存在于 Eclipse 本身中的类。所以要开始，必须先创建一个 Eclipse 插件（通过使用 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 插件的所设计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。

必须依赖于以下函数库：



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，那里继承了一些有用的方法。利用这些方法可以创建显示在 Eclipse 中 problesm 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工作人员的标记（标记自动链接到工作人员，清除一个工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的文章。

11.5.2.1 创建 ProcessValidator 扩展

在 plugin.xml 文件内为 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>
```

```
public class AcmeProcessValidator extends AbstractProcessValidator {
}
```

你必须实现一些方法。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，根据具体功能在这里放什么样的代码。但通常一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、比较并校验数据。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
}
```

在你完成校验时，不要忘记调用 addProblemToDiagram()和/或 addWarningToDiagram()，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。这个结果可以在调用 ExportMarshaller 来决定下一步操作时使用。

11.5.2.2 创建 ExportMarshaller 扩展

在 plugin.xml 文件内为 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。这个抽象基础类提供了在编组到你自己格式时所使用的一些有帮助的方法，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>
   <extension point="org.activiti.designer.eclipse.extension.export.ExportMarshaller">
      <ExportMarshaller class="org.acme.export.AcmeExportMarshaller">
      </ExportMarshaller>
   </extension>
</plugin>
```

```
public class AcmeExportMarshaller extends AbstractExportMarshaller {
}
```

需要你来实现一些方法，如 getMarshllerName()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。提供给你了 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了。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，甚至是创建一个不同的资源。比如：

```
final boolean 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 {
   //proceed with marshalling
}
```

如你所见，如果校验器返回的结果为 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，并且这展示出了在 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()方法来创建包含了你的数据的文件了。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为输出的资源构造文件名。有几个你已经解析过了的有帮助的变量，允许你来创建像<original-filename>_<my-format-name>.xml 的文件名。在 Javadocs 内对这些变量进行了说明，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.axml";
...
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），该模式利用 ExportMarshaller.PLACEHOLDER_ORIGINAL_FILENAME 常量将一个变量插入到了原始文件名内。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()提供了一个到你数据的 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。

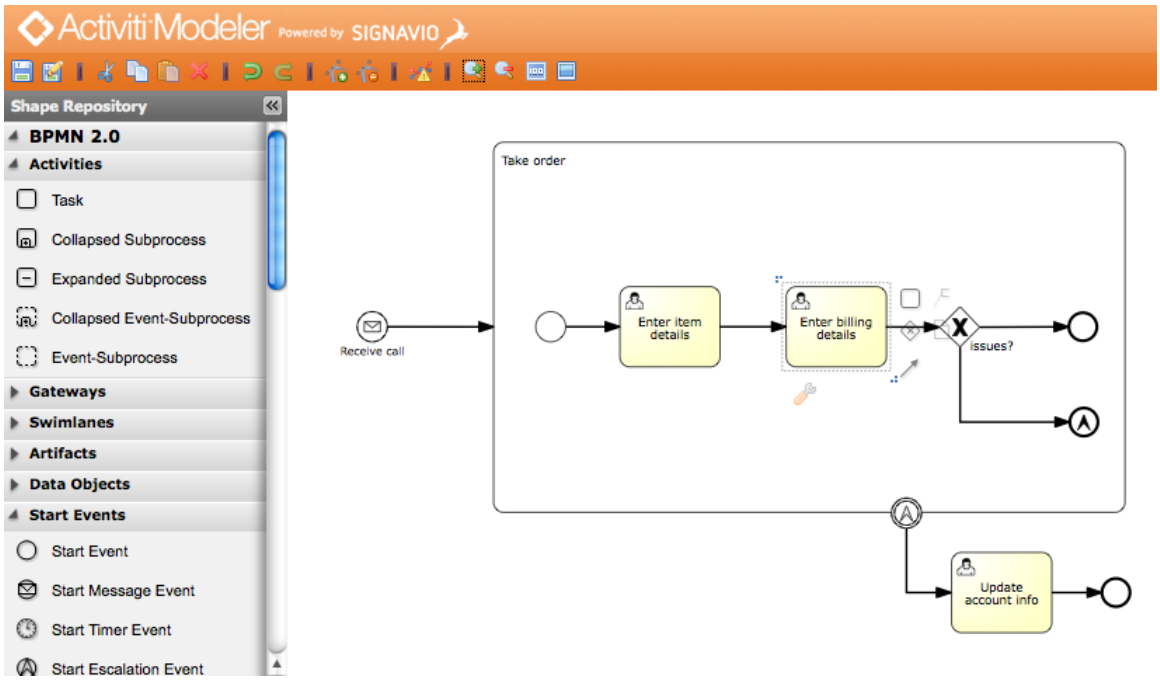
第十二章、Activiti Modeler

Activiti Modeler 是一个基于 web 的流程设计器，可用来编写图形化的 BPMN 2.0 的流程，其使用的只不过是普通的浏览器。服务器将流程文件存储到文件系统中，这样是很方便的，省去了引入任何 Java IDE 带来的麻烦。

Activiti Modeler 是由 Signavio 开发的独立组件，寄存在 goole 代码项目内。Activiti Modeler 的许可协议使用的是 MIT。

可以将错误和问题报告在 Signavio 的核心组件问题追踪器上。

Activiti Modeler 在演示设置中自动被安装。



12.1 修改流程模型库的位置

运行了演示设置后，流程的 XML 文件就会被存储到 workspace/activity-modeler-examples/文件夹下。通过修改（**分解开的**）**activiti-modeler.war/WEB-INF/classes/configuration.properties** 文件中的 **fileSystemRootDirectory** 属性可以切换该基于文件的库的位置。

12.2 修改 Modeler 的主机

运行了演示设置后，Activiti Modeler 只能在本地主机地址下访问。当你想要修改 Modeler 的主机，例如将其集中运行在服务器上，这时就要修改（**分解开的**）**activiti-modeler.war/WEB-INF/classes/configuration.properties** 文件中的 **host** 属性

```
> cat configuration.properties
host = http://192.168.1.101:8080
fileSystemRootDirectory = /Users/jbarrez/Development/model-repo
```

12.3 为 Activiti Modeler 配置 Apache Tomcat

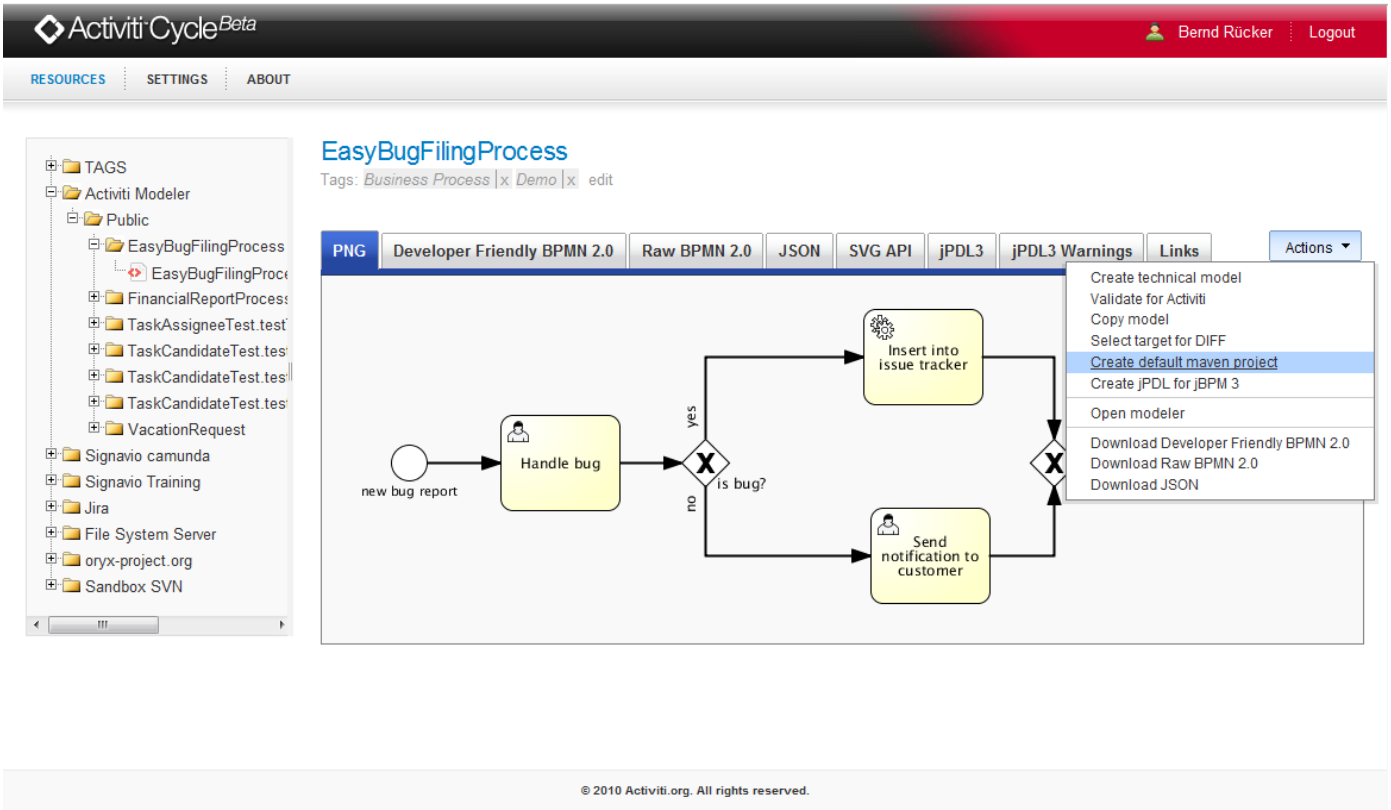
在 Activiti 的演示设置上下文中，已经正确配置了 Apache Tomcat。如果要手动配置 Apache Tomcat，要确保访问文件系统和 URI 编码使用的是 UTF-8 编码。

- 确保将-Dfile.encoding=UTF-8 添加到${catalina.home}/bin/catalina.bat 或${catalina.home}/bin/catalina.sh 内的 java 调用上。
- 确保${catalina.home}/conf/server.xml 内 HTTP 连接器上设置了 **URIEncoding="UTF-8"**，比如像这样：

```
<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443"
URIEncoding="UTF-8" />
```

第十三章、Activiti Cycle

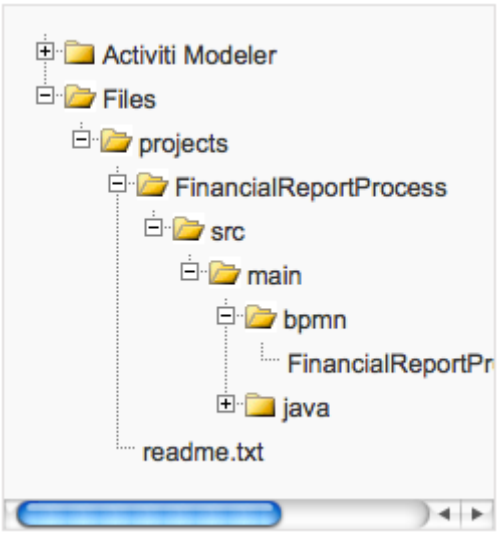
Activiti Cycle 是个 web 应用程序，它为 BPM 项目的相关各方（企业、开发者、IT 运营....）提供了一个协作平台。它将 Activiti Modeler 仓库、Subversion 以及你本地的文件系统等这些不同的仓库集成到一个视图，以方便浏览所有在创建可执行业务流程过程中所涉及到的物件。除此之外，Activiti Cycle 也提供了像仓库之间移动物件，下载不同格式的流程模型这样的一些内置行为，以及针对自定义行为的可插入式的基础结构。下面是默认演示设置中预配置的 Cycle 的抓屏。请注意：**Activiti Cycle 还是"beta"状态**。有了 Activiti Cycle 这样的协作工具就不再像使用核心引擎那样需要有多年的经验了。目前，大部分特性已经稳定了，但随着更多用例及特性的添加，我们会对该 API 或数据库的表做些调整。举例来说，这意思着插件 API 可能会被修改，这样一来你必须得知道什么时候可以编写你自己的插件；或者数据库可能会在创新的数据库迁移脚本还不存在的情况下被修改。但肯定的是：我们一直在努力消除这一 beta 状态，这意味着很快就会有稳定的 API 和数据库模式了。



13.1 概述

因为 Activiti Cycle 旨在给具有完全不同的角色、背景以及需要的人来使用，所以在构建它是时刻考虑到了它的可定制性和扩展性。然而，已经存在于当前发布中的插件的基础架构只是使 Cycle 满足你特定需求的第一步。将来的版本将提供改进的插件机制，配置及插入仓库、标签以动态地向构件中添加元数据的图形用户接口，以及直接协作于构件层级之上的注释功能。在文章 Activiti Cycle explained 中你可以找到有关对 Activiti Cycle 最初版本和其后愿景的简单介绍。

13.2 仓库



上面的图片显示的是 Activiti Cycle 仓库树。默认，它提供了对 Activiti Modeler 仓库和位于你本地文件系统 ${activiti.home}/workspace/activiti-cycle-examples 内的演示 Eclipse 工作空间的访问。仓库是作为仓库连接器被插入进来的。在 Activiti 数据库中表 CY_CONN_CONFIG 中可以找到相关的配置信息，可以按“查看数据库”一节介绍的方式对其进行访问。

其数据库模式有 6 列：

1. PLUGIN_ID_：连接器的插件 id（比如，文件系统连接器"org.activiti.cycle.impl.connector.fs.FileSystemConnector" 以及对于 Activiti Modeler 的连接器”org.activiti.cycle.impl.connector.signavio.SignavioConnector"）。
2. INSTANCE_NAME_：连接器实例的名称。正是这个字符串作为连接器的根文件夹显示在 Cycle 仓库树内。注意连接相同类型的不同仓库是可以多次使用同一连接器的（比如，使用文件系统连接器连接多个 Eclipse 工作空间）。这样的话 CY_CONN_CONFIG 会包含多条 PLUGIN_ID_相同，INSTANCE_NAME_s 和 INSTANCE_ID_s 不同的记录（见下文）。
3. INSTANCE_ID_：用于内部寻址连接器实例。
4. USER_：当前连接器实例被设置给的用户（注意：这并不是连接器的登录名，而是 Activiti Cycle 的用户名，例如，演示配置中的”kermit”）。
5. GROUP_：连接器实例可以配置给单个用户，也可以配置给一个组。如果给定的连接器实例设置了” GROUP_”字段，那么 Cycle 会为每个属于这个组的用户都加在一次这个实例。
6. VALUES_：包含着当前配置值的 XML 字符串。它是个被序列化过 map，格式为 <map>(<entry><string>KEY</string><string>VALUE</string></entry>)*</map>。

执行如下 SQL 语句设置”kermit”的默认配置，包括一个文件系统连接器实例和一个用于访问 Activiti Modeler 仓库的 Signavio 连接器实例：

```
insert into ACT_CY_CONN_CONFIG values ('1',
    'org.activiti.cycle.impl.connector.fs.FileSystemConnector',
    'Eclipse Workspace (File System)',
    'Workspace',
    'kermit', '',
    '<map><entry><string>basePath</string><string>/path-to-activiti-home/activiti-5.0.beta2/workspace/activiti-cycle-exampl
```

```
es</string></entry></map>');

insert into ACT_CY_CONN_CONFIG values ('2',
    'org.activiti.cycle.impl.connector.signavio.SignavioConnector',
    'Activiti Modeler',
    'Activiti',
    'kermit', '',
    '<map>
        <entry><string>signavioBaseUrl</string><string>http://localhost:8080/activiti-modeler/</string></entry>
        <entry><string>loginRequired</string><boolean>false</boolean></entry>
    </map>');
```

这时，比如，如果想要添加 Signavio Process Modeler 仓库，你就可以增加对 PLUGIN_ID_'org.activiti.cycle.impl.connector.signavio.SignavioConnector'的配置。如果需要 Signavio 的清单，可以在 Signavio 的 web 站点上免费注册试用。这是个配置的例子：

```
<map>
    <entry><string>signavioBaseUrl</string><string>https://editor.signavio.com/</string></entry>
    <entry><string>loginRequired</string><boolean>true</boolean></entry>
    <entry><string>username</string><string>你的用户名</string></entry>
    <entry><string>password</string><string>密码</string></entry>
</map>
```

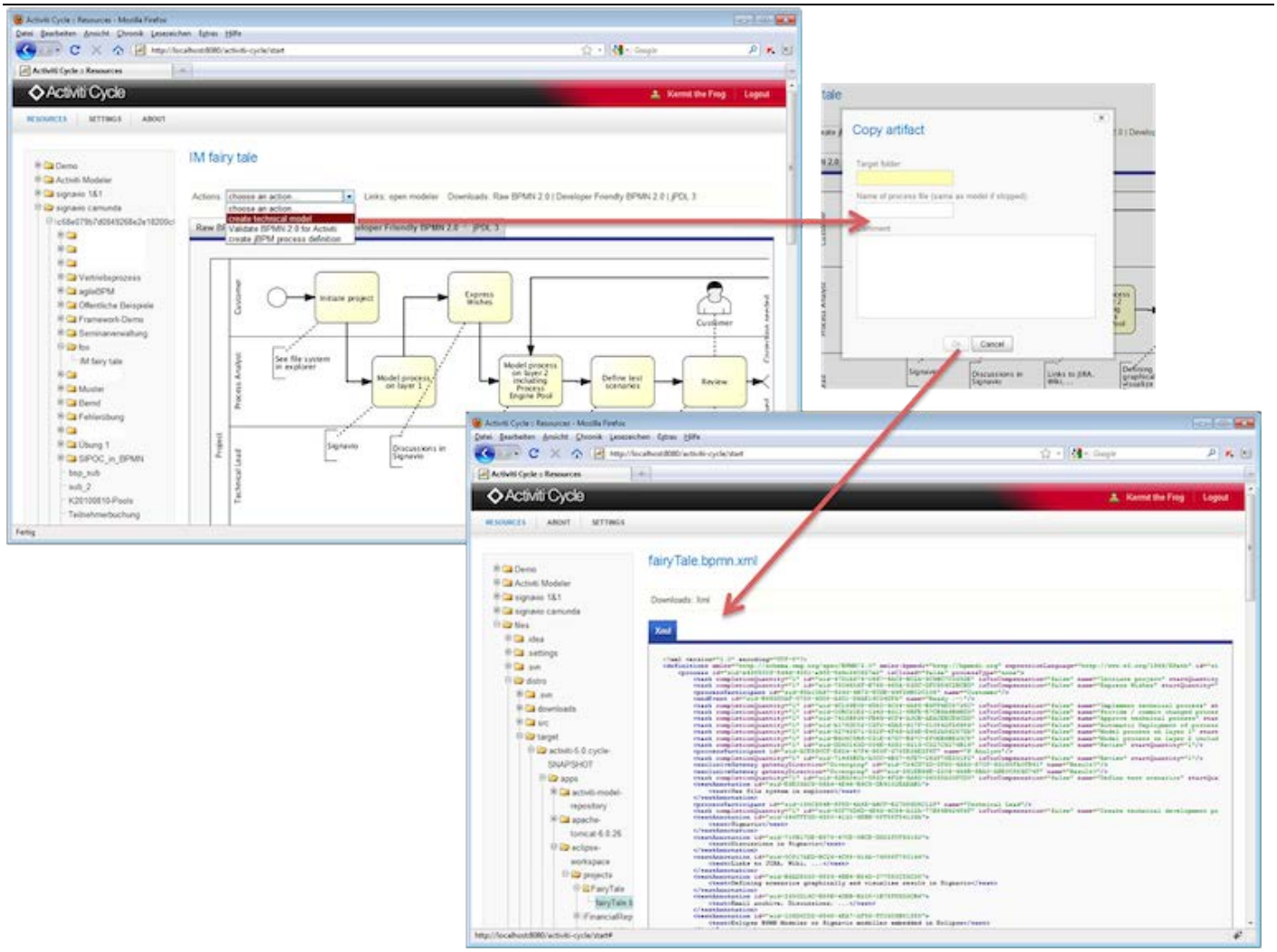
也可以选择不在数据库中保存用户名和密码，空着”username”和”password”。需要时 Cycle 会向你要。

另一个可能会考虑的是从 Oryx 项目中添加一批 BPMN 的例子。为了做到这一点，要对 PLUGIN_ID_ 'org.activiti.cycle.impl.connector.signavio.SignavioConnector'的连接器进行配置：

```
<map>
    <entry><string>type</string><string>oryx</string></entry>
    <entry><string>signavioBaseUrl</string><string>http://oryx-project.org/</string></entry>
    <entry><string>loginRequired</string><boolean>false</boolean></entry>
</map>
```

13.3 构建和行为

点击仓库树中的一个项目时（Activiti Cycle 我们称之为构件），详细信息会显示在右侧。目前，并不是对所有文件都支持，但我们的目标是为那些最常见的文件类型提供一个用帮助的默认视图。除了这些视图，构件上还存在一系列可以被执行的操作。



这些操作可以视为 Cycle 所能为你做的例子，将来还会被扩展，还可以作为额外行为的起点编写在你自己的插件内。

13.4 更多插件

还有更多 Cycle 插件，比如 SVN 和 SFTP 连接器，以及 Maven Project Wizard。这些插件都可以单独下载以保持 Cycle 的核心小巧，易于管理并允许创建需要那些还未通过 Apache 兼容许可的依赖的插件。可以在 Cycle 的插件页面上找到一些可供下载的插件。

第十四章、REST API

[试验性的]…整个 REST 接口都还是试验性的。

Activiti 中包含了操作引擎的 REST API，在运行演示设置时，会将它部署到你的服务器内。REST API 采用了 JSON 格式（http://www.json.org），建立在 Spring Webscript（http://www.springsurf.org）之上。

如果将浏览器指向 http://localhost:8080/activiti-rest/service/index，并以管理员用户（kermit）进行登录，就可以浏览 REST API 了。如果点击”Browse by Web Script Package”链接，会得到部署在服务器上的包的概况，可以轻松地导航进一个包查看为那个包所定义的 REST API 调用。可以忽略"Alfresco JavaScript Debugger"，因为使用的是 Java 而不是 JavaScript 来实现的该逻辑。

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 的调用（除了登录的服务）。认证使用的是基本的 HTTP 认证，所以如果是以管理员（比如 kermit）登录浏览该 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无限数量的对象可以被使用，POST 会被用在所有这些操作上。使用 POST 的原因是 HTTP 的 DELETE 方法隐含是不允许请求体的，因此，理论上，使用了 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 POST，即使是这时可能已经使用了 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。

请查看下面看看目前有哪些 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，并不使用 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(fi

leName).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)
- **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:\//<path-to-deployment>\/activiti-cfg.jar!\/activiti.properties",
  "exception": null,
  "version": "5.4"
}
```

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

创建基于流程定义的流程实例，返回关于最新创建流程实例的详细信息。附加变量（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。利用一个名称为”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。如果没有用到类型描述符，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 将作为整数并非”123”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，所以不可能再 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原数据模型，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

为特定用户返回任务概述：分配给该用户任务个数，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。该列表是基于特定角色的：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（表单中的）附加的变量。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

返回由流程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[, groupType])
- **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，或者如果作业重试过，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_"],
  "columnNames": ["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
}
```

第十五章、Activiti Explorer

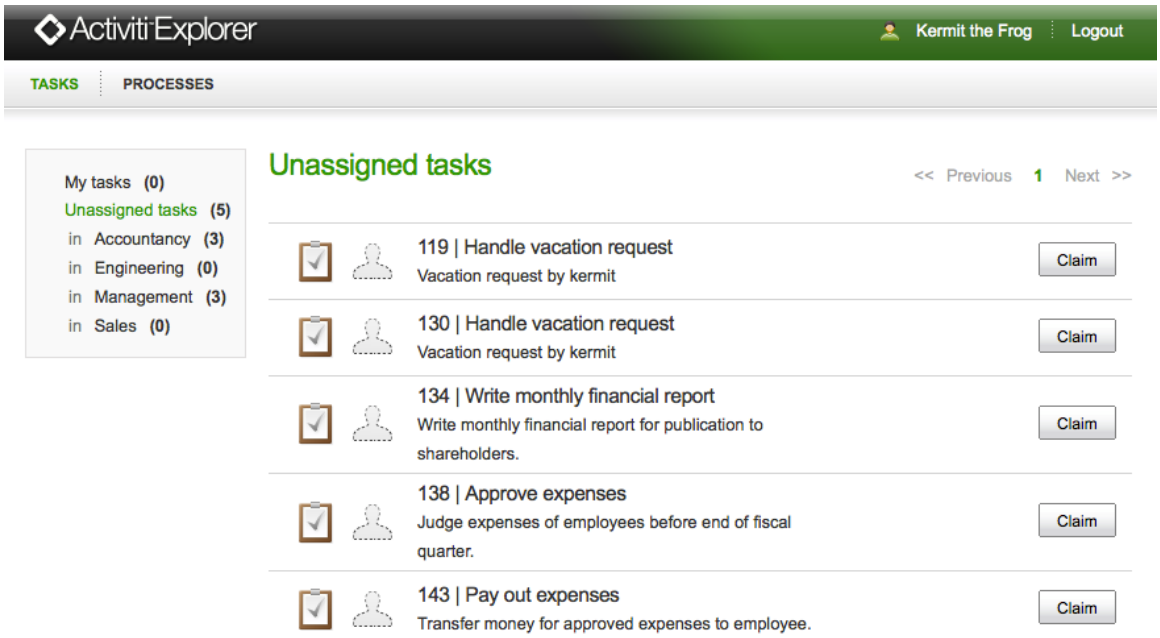
15.1 概述

Activiti Explorer 是用于业务流程中任务管理的 web 应用程序。

15.2 任务的管理

登陆 Activiti Explorer 后，首先会任务页面。此页面列出了所有当前可用的任务，且能够使用左侧面板中的条件过滤该视图。从这里，所有用户都能够认领任务和完成任务。

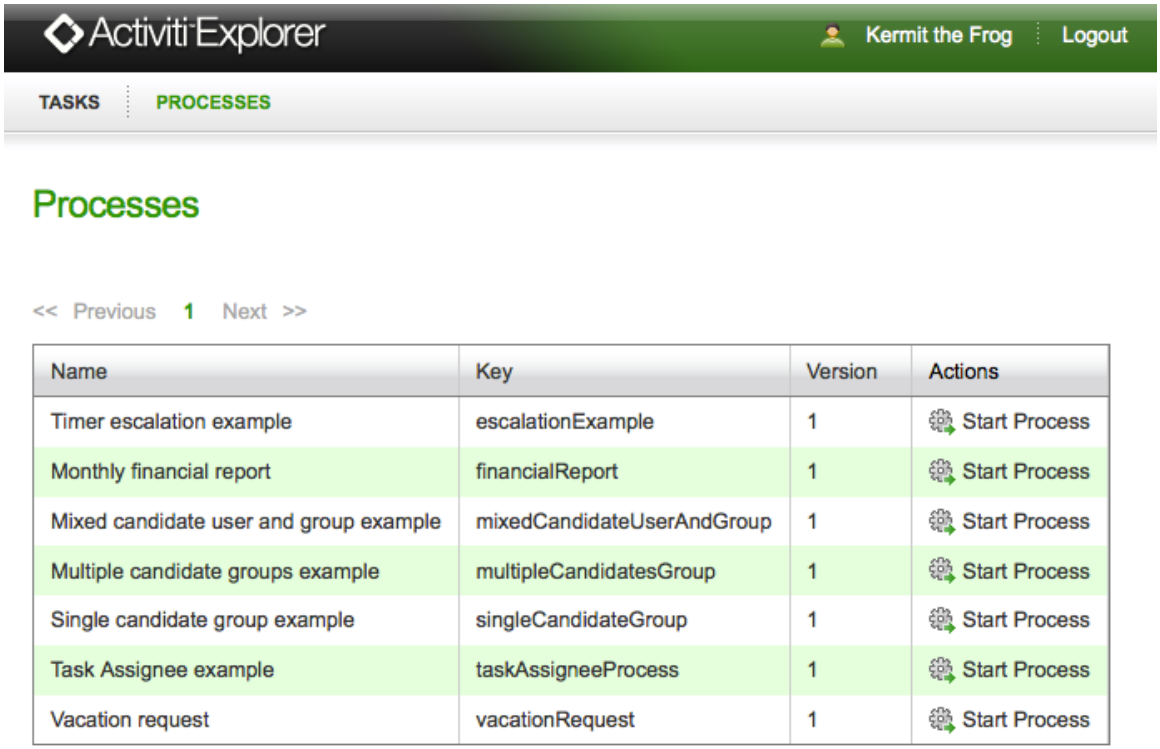
任务表单可以直接从 URL 启动，使到表单的链接能嵌入到某些地方，如邮件。URL 为 http://localhost:8080/activiti-explorer/start?taskId=119，其中 119 是有关联表单的任务的 ID。



15.3 启动流程

Process 标签让用户能够浏览到所有部署的流程定义，并提供了启动新流程实例的能力。一旦启动了新的流程，所有相关表单会被自动显示出来。至于任务，可直接从 URL 启动流程。其中 URL 采用的格式为：http://localhost:8080/activiti-explorer/processes#start?id=financialReport:1，其中 financialReport:1 是你要启动流程的 ID。流程会被立即启动，并在屏幕上显示出确认信息。

流程列表是分页的，可排序。通过扩展流程的 WebScript 可以对 Action 列进行扩展以允许流程实例上的其它功能（如，将实例链接到 Activiti Modeler 的安装程序让用户在启动实例前就能浏览到流程）。


Activiti Explorer
Kermit the Frog
Logout
TASKS
PROCESSES
Processes
<< Previous 1 Next >>
Name
Key
Version
Actions
Timer escalation example
escalationExample
1
Start Process
Monthly financial report
financialReport
1
Start Process
Mixed candidate user and group example
mixedCandidateUserAndGroup
1
Start Process
Multiple candidate groups example
multipleCandidatesGroup
1
Start Process
Single candidate group example
singleCandidateGroup
1
Start Process
Task Assignee example
taskAssigneeProcess
1
Start Process
Vacation request
vacationRequest
1
Start Process

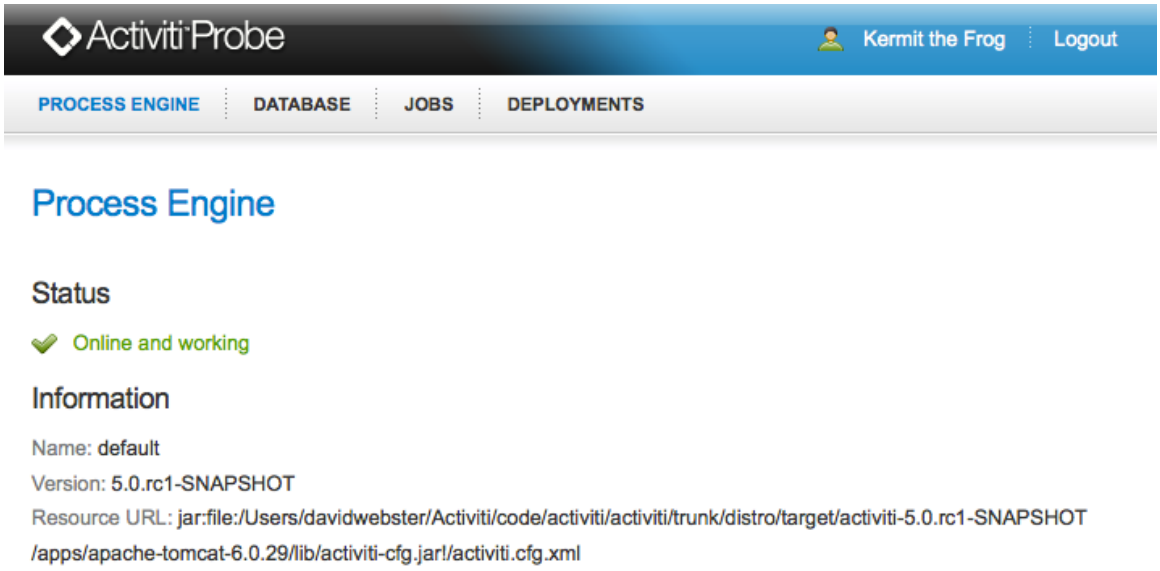
第十六章、Activiti Probe

16.1 概述

Activiti Probe 是个 web 应用程序，它提供了管理和监控的能力，以保持 Activiti 引擎实例启动和运行。此应用程序是针对负责维护系统和基本结构启动并运行的系统管理员和操作员的。

16.2 流程引擎的状态

起始页给出了包括正在使用的 Activiti 版本在内的，关于 Activit 引擎实例状态的概述。

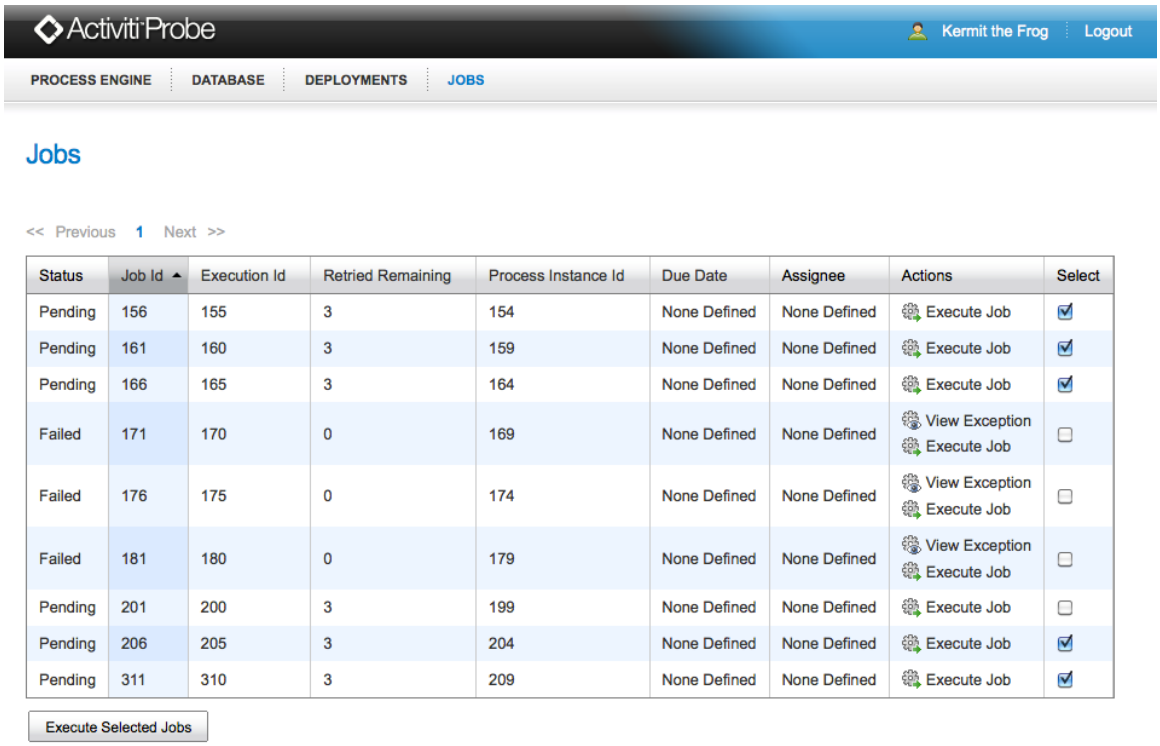


16.3 作业管理

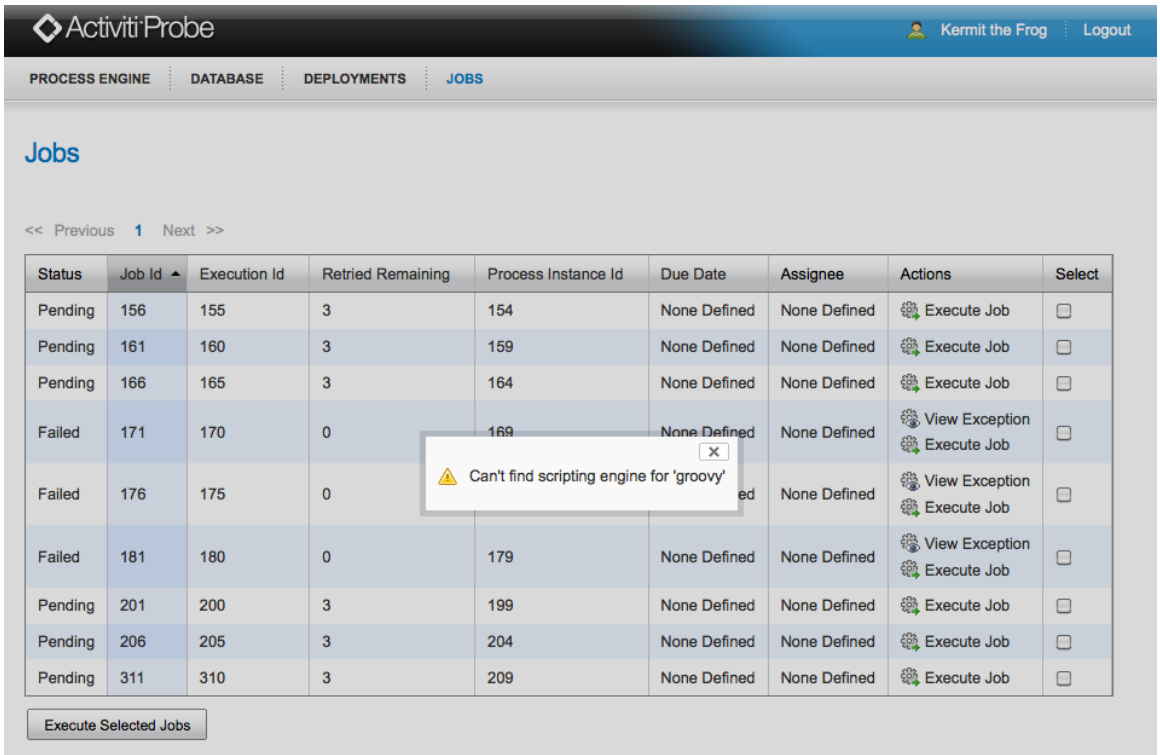
作业管理页面允许你执行若干任务相关的作业，包括：

- 查看作业列表和作业的详细信息
- 确定作业状态
- 重试失败的作业
- 立即发送要执行的挂起作业
- 查看失败了的作业的异常信息

在执行作业时，它会被发送到执行队列，然后异步执行，这意味着执行作业的结果不会立即在数据表中看到。利用每行右侧末尾的复选框，可以在同一个请求中发送多个要执行的作业。



如果作业失败了，会有查看作业中异常或错误信息的选项-显示在同一页面上的弹出框。

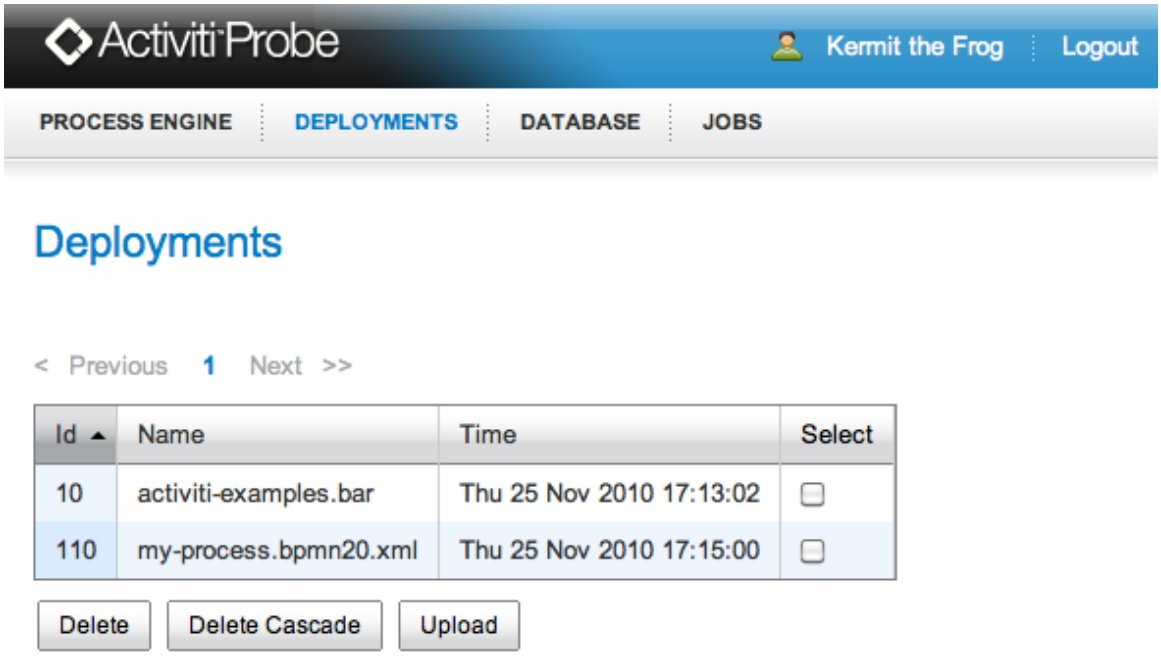


16.4 部署

在部署管理页面你可以看到 Activiti 引擎中的每个部署以及部署时间。

对于每个部署的管理选项就是将其删除（”Delete”），或者删除部署及所有相关文件，如流程和作业（”Delete Cascade”）。通过选择数据表中的多行可以同时删除多个部署。

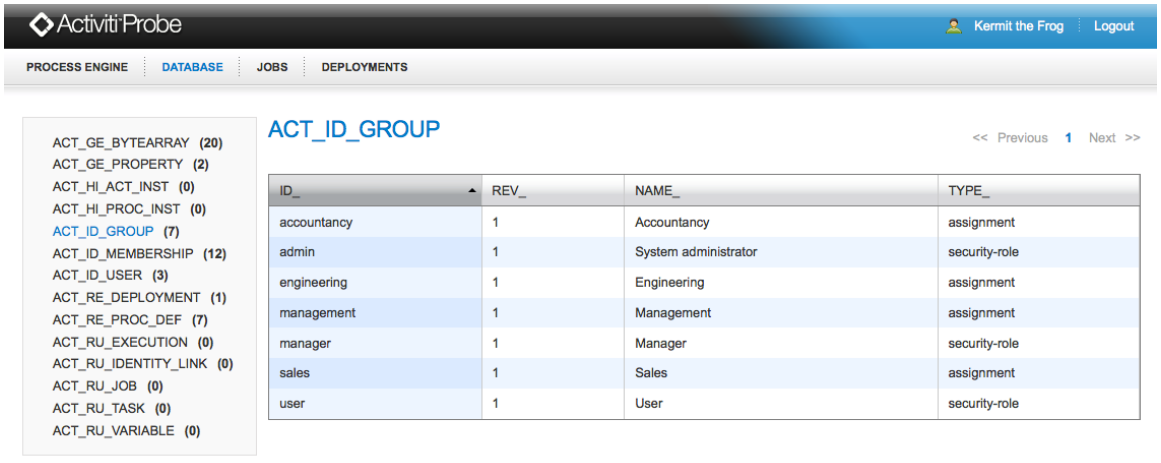
数据表可以按 Id、名称或部署时间进行排序。



利用 Probe，你也可以从浏览器上传新的部署。点击上传按钮后，选择要上传的文件，其必须是.bpmn20.xml、.zip 或.bar 的文件。一旦部署执行成功，部署就会显示在该列表中，流程可以随时使用。

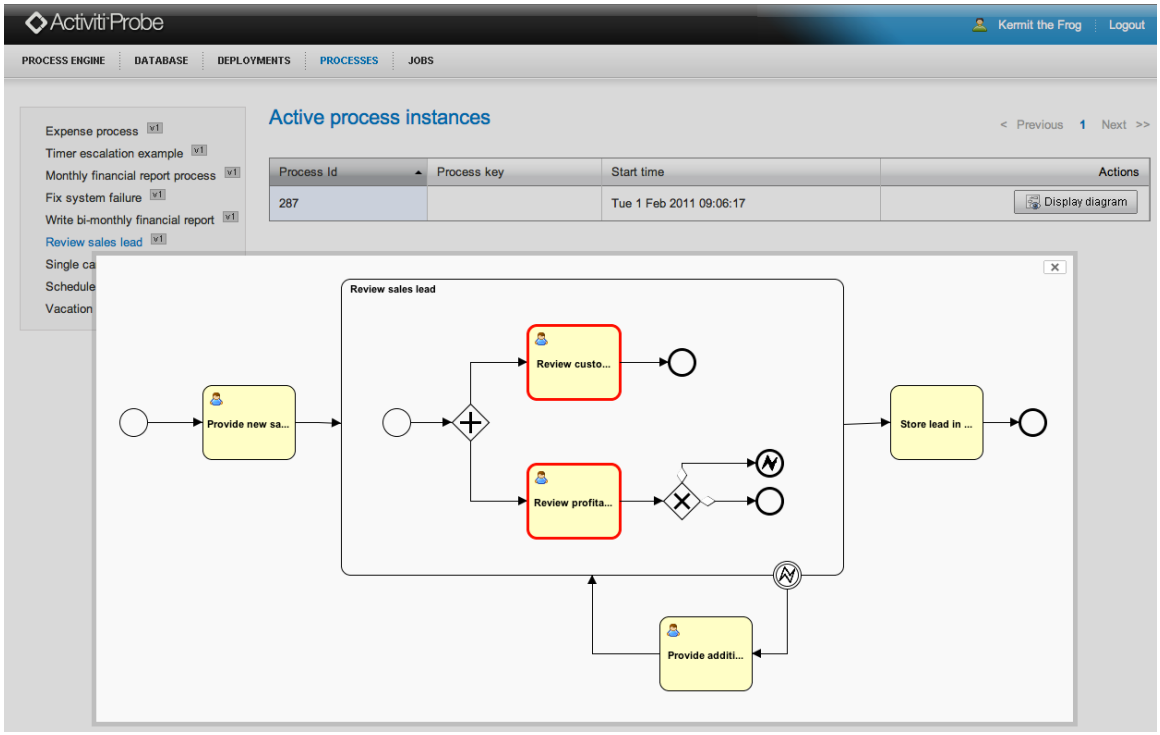
16.5 数据库

数据库视图为管理员用户提供了一个关于数据库中原始数据的视图。为了保证数据的一致性，从此视图只能读取数据，不能对数据进行修改 – 必须使用提供的公开的 API 方法才能修改这些记录。数据库视图显示为下列表格：



16.6 流程

流程视图显示了所有部署的流程中的活跃的流程实例。如果流程定义中包含图形交互的信息，点击’Display diagram’按钮后，就会看到流程实例中当前的活动。



第十七章、Activiti KickStart

17.1 概述

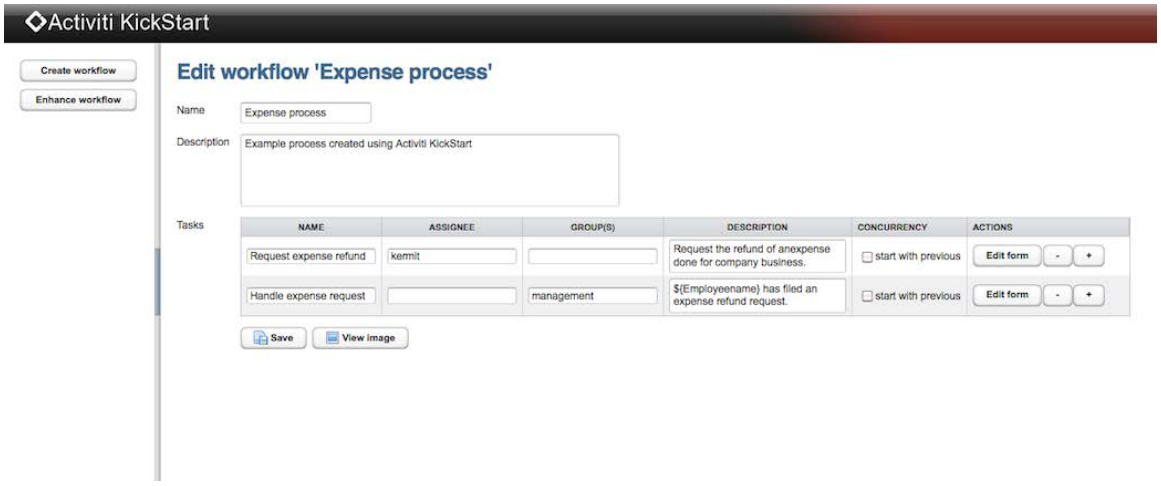
Activiti KickStart 是利用 Activiti 引擎的可用构造的子集来快速创建'临时安排的(adhoc)'业务流程的一个基于 web 的工具。KickStart 提供了一个简单的图形界面，且它是不需要了解 BPMN 或任何建模环境的，因为它使用的是每个业务人员所熟悉的概念。然而，使用 KickStart 创建的流程却是完全符合 BPMN 2.0 的，可作为更为复杂的 BPM 努力的起点。

KickStart 完美地集成了 Activiti 引擎。这样一来，利用 KickStart 创建的流程能立即使用在 Activiti Explorer 内，且能立即显示在 Probe 中。

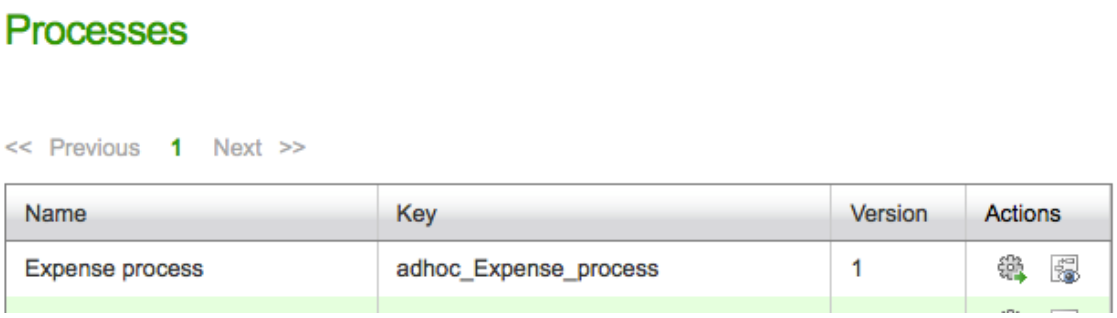
KickStart 服务很多业务用例，但一下三种情况恐怕是最常见的：

- **简单业务流程**：有些流程本身就很简单，且每个公司都有。想象一下报销流程、假日休假流程、雇佣流程，等等…这些类型的流程可能都是使用纸张或 e-mail 来完成的。KickStart 能对这些流程快速建模，并在必要时对它们进行修改。这样一来，KickStart 真正降低了自动化这些流程的门槛。
- **原型**：在深入钻研复杂的 BPMN 2.0 建模及考虑流程极端情况前，将所有涉及到的人员集中起来，制定出能展示所要完成操作的蓝图的原型常常是明智的。KickStart 允许照此操作：即时创建业务流程原型，将你的想法为大家所见。
- **临时安排的（ad hoc）工作**：在某些情况下，是需要公司内部不同人员或团队之间配合的。您是知道通常所发生的：这边发邮件，那边打电话…其结果是没有人知道什么需要做或何时做。然而，像 Activiti 这样的业务流程管理平台是分配及后续一切的极佳方式，因为它就是为追踪那些事情的。KickStart 允许在几分钟内为临时安排的工作创建流程，然后很容易地将任务在人员之间分配并调整。

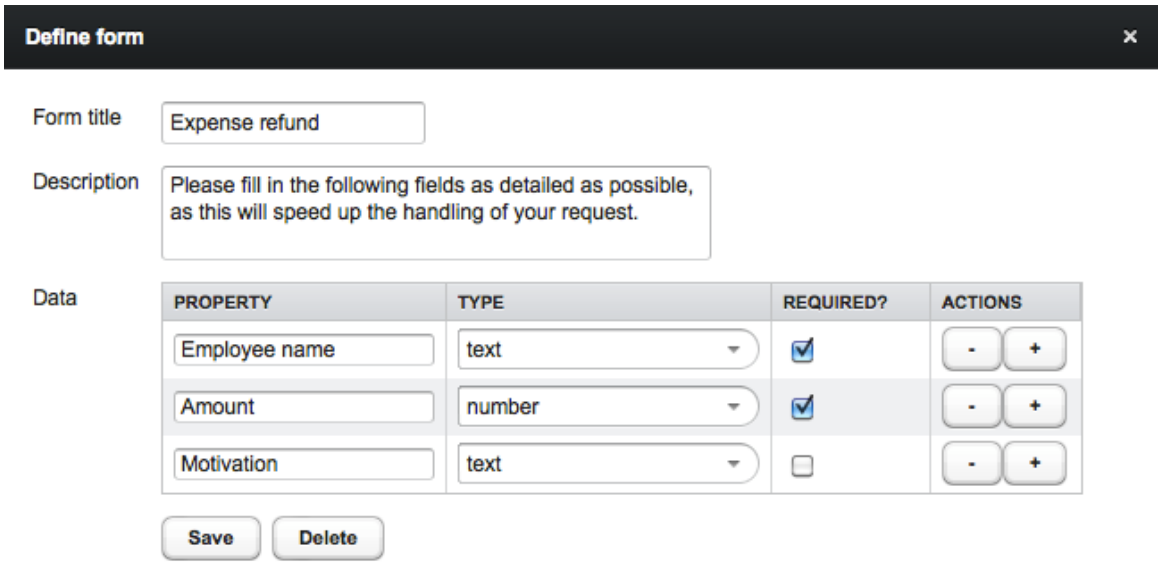
下面的截屏显示了 Activiti KickStart 的功能。拿接下来的图片来举例。它显示了报销流程是如何在几分钟内被创建的。



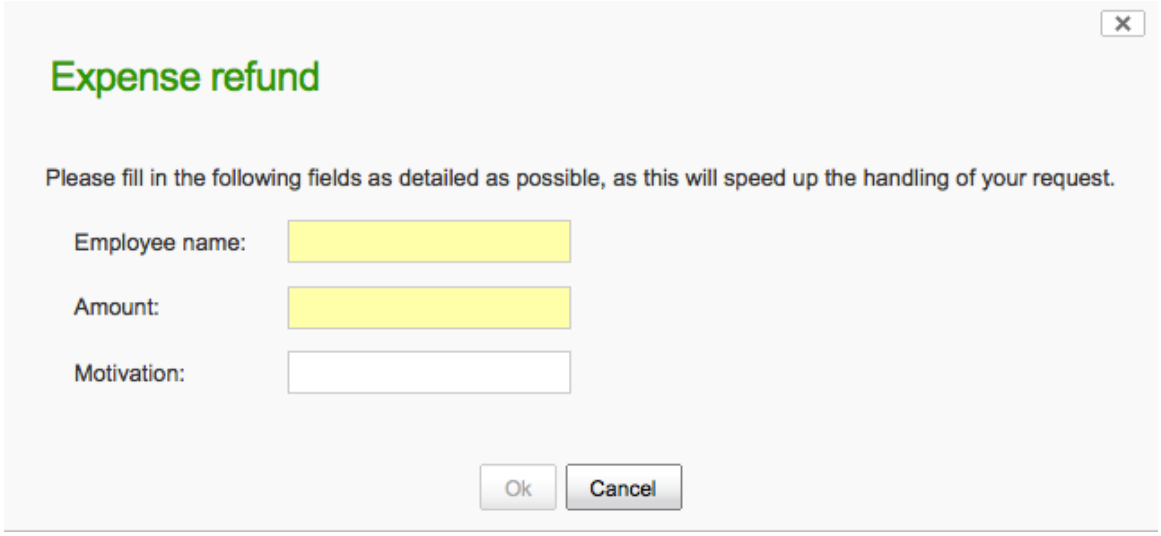
点击保存按钮后，该流程可立即使用在 Activiti Explorer 内。



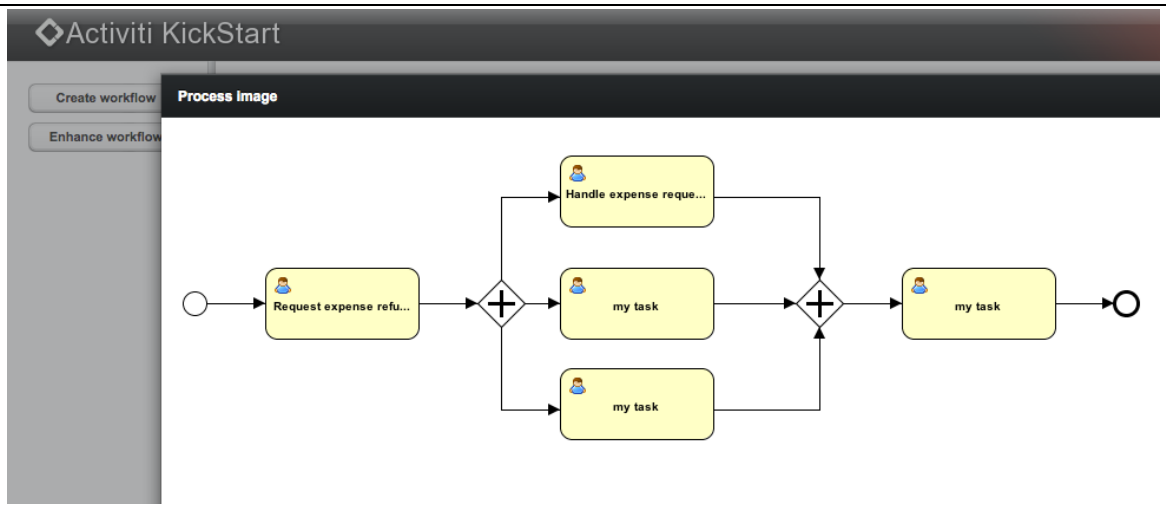
KickStart 也允许为每个任务定义表单。



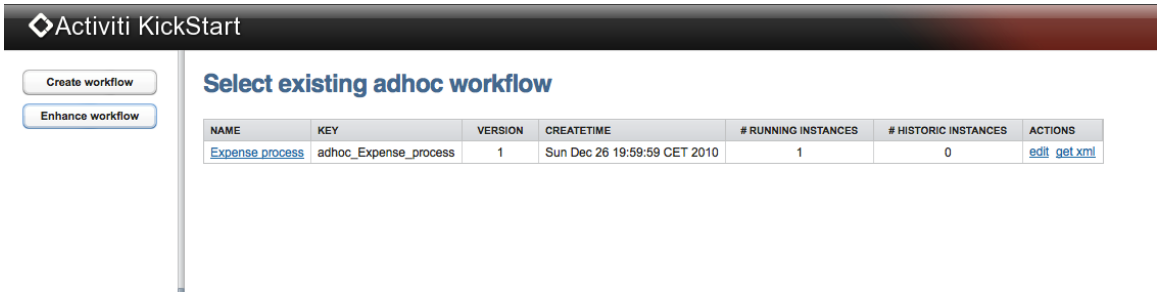
其显然可直接用于 Activiti Explorer。



在流程创建期间的任何时候，都可以查看业务流程相应的 BPMN 2.0 的图形。



只要需要，可以随时打开并修改利用 KickStart 定义的流程。



使用 KickStart 创建的流程是完全符合 BPMN 2.0 XML 的，这意味着可将该流程导入进任何 BPMN 2.0 的编辑器。

17.2 修改数据库

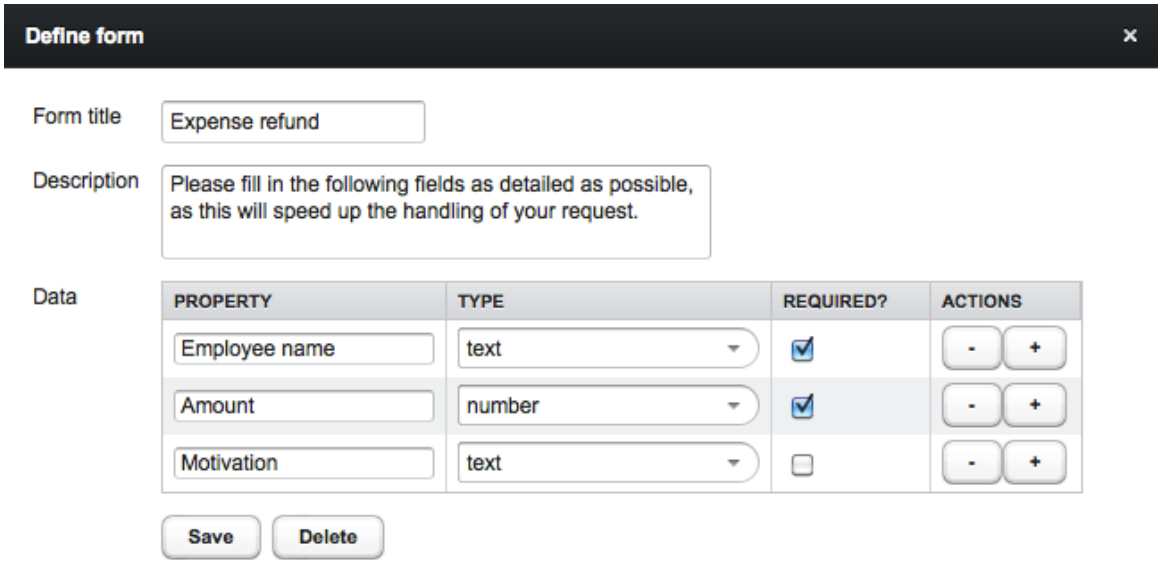
Activiti KickStart 目前还没有使用 REST API，其通过作为函数库包含进 Activiti 引擎的方式使用了服务的 API。虽然 Explorer、Probe 以及 Cycle 对于修改数据库使用的是同一机制（见这里），但这一机制不适用于 KickStart。

要修改演示设置中 KickStart 使用的数据库，需要创建新的 activiti.cfg.jar 文件，并将其置于

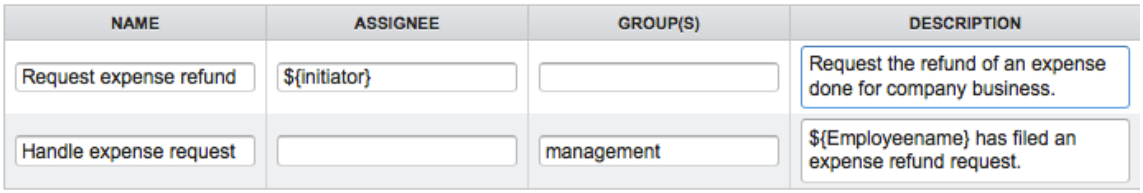
apps/apache-tomcat-6.x/webapps/activiti-kickstart/WEB-INF/lib 文件夹下。同样，必须将数据库驱动程序置于同一文件夹下。可以在 setup/files/dependencies/libs/下找到你的数据库驱动程序（除了 Oracle）。

17.3 引用表单属性

通过引用表单属性的名称可以很容易地使用到用户为表单属性填入的值。拿下面的表单举例：



该表单有 3 个表单属性：Employee name、Amount 和 Motivation，其值接下来可用于填充分配、描述以及表单标题/描述字段。例如，Employee name 属性用在了下面任务的描述中。

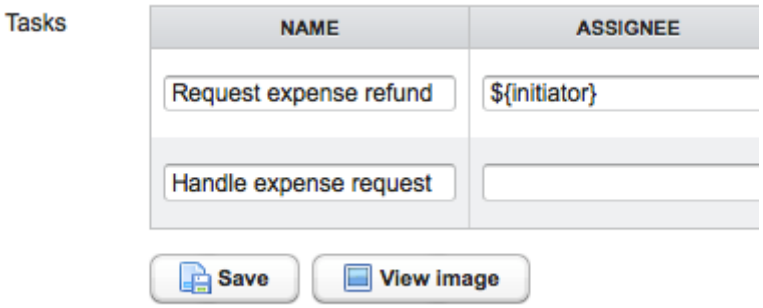


注意，如果属性有空格，保留或省略空格都可以引用到它。

17.4 捕捉流程的启动程序

当你可以捕捉流程实例的启动时，业务流程就变得真正有用了。拿随发布发送的报销流程举例。当某人启动了一个流程实例打算报销他的费用时，我们想知道他是谁，所以，举例来说，我们可以将第一个任务分配给这个人。

所有利用 KickStart 生成的流程都可以在 initiator 属性中捕捉到流程的启动程序。只需按下面图片展示的那样引用'initiator'属性。这个例子中，'Request expense refund'任务会分配给任何启动该流程实例的人。


Tasks
NAME
ASSIGNEE
Request expense refund
${initiator}
Handle expense request
Save
View image

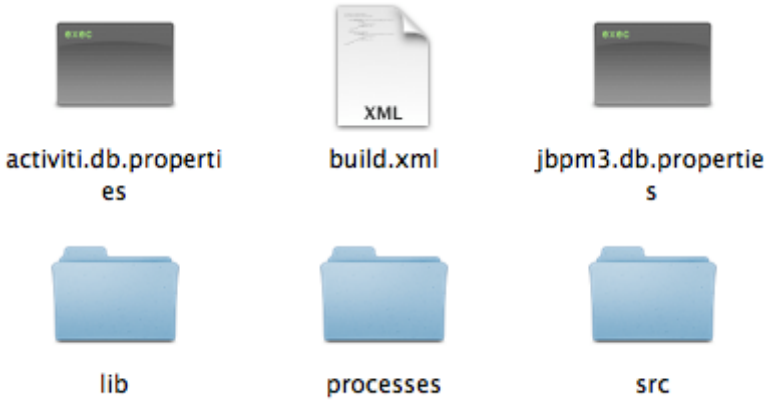
第十八章 JBPM 迁移

jBPM 迁移被看作是[试验性的]。

可以将 jBPMN 版本 3 的现有安装迁移到 Activiti。迁移包括流程定义转换（如从 JPDL 3 到 BPMN 2.0）以及数据迁移。

此迁移工具目前只是作为‘试行’被提供的！此时此刻，该迁移所覆盖的还不足以包括生产环境下的流程定义和数据库的所有情况。同时也要注意到该迁移是**‘尽力而为’**的，这意味着你可能需要扩展迁移逻辑以覆盖所有你的用例。

该迁移工具可单独以压缩文件的形式从 activiti.org 下载站点获得。解压文件后，将会看到以下文件和文件夹：



- **build.xml**：该 ant 构建文件内含有执行流程定义转换和数据库迁移的任务。
- **activity.db.properties** 和 **jbpm3.db.properties**：这些 properties 文件内含有 jBPMN 3 和 Activiti 的数据库的连接参数。在执行数据库迁移时，必须对这些 properties 文件进行修改。
- **processes**：在执行流程定义转换时，必须将 jBPM 3 的流程定义文件放到这个文件夹内。
- **src**：这个文件夹中包含了该迁移工具的源码。如果想要定制或扩展其中的迁移代码以满足你的需要，请阅读扩展一节。
- **lib**：这个文件夹中包含了执行该迁移逻辑所需的所有 jar 文件。

18.1 数据库迁移

可以将 jBPMN 3 的数据库表中的数据迁移到 Activiti 数据库模式。为此，必须修改以下文件以将其指向正确的数据库：

- **jbpm3.db.properties**：含有连接到 jBPM 3 的数据库的参数。同时必须提供 Hibernate 方言。
- **activity.db.properties**：含有连接到迁移数据将被写入的数据库的参数。其可以是与 jBPM 数据库模式相同的数据库模式（即，jBPM 和 Activiti 在表、索引、外键等是不存在命名冲突的）。

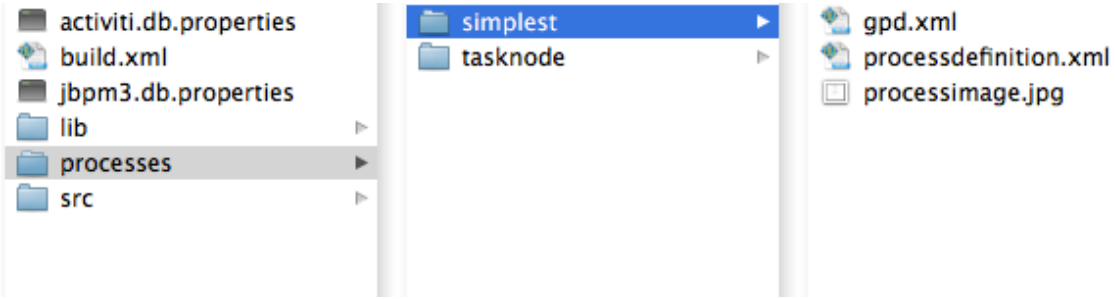
数据库迁移将利用 jBPM 3 的数据表中的数据进行：

- 将流程定义转换为 BPMN 2.0 对应的流程定义。这意味着被转换的流程定义作为迁移的一部分将被部署到 Activiti 数据表中。其负面作用是，这也会产生 BPMN 2.0 的 XML 文件，类似于在执行流程转换时所发生的。
- 将流程中的运行时和历史数据迁移到 Activiti 的数据表中。在此版本中，**这还没有实现！**

从 jBPM 3 的数据表获取数据使用的是 jBPM 本身的 Hibernate 查询和 Hibernate 映射。

18.2 流程转换

可以只进行从 JPDL 到可在 Activiti 引擎上执行的 BPMN 2.0 的流程定义 xml 文件的转换。为此，必须将 jBPM 3 的流程放到 processes 文件夹内。其内可包含任意数目的（嵌套）子文件，此工具会扫描 processes 文件夹内的每个（子）文件夹以寻找 processdefinition.xml 文件（文件名必须为 processdefinition.xml！）。



被找到的流程会被解析，然后被部署到内存数据库中，这样由包含在数据库迁移中的 jBPM 的部署数据表进行的流程定义的反向工程的逻辑就可以在这两种情况下使用了。

一旦将 jBPM 3 的流程放到了 processes 文件夹内，就可以执行解压开的迁移工具文件夹根目录下的 convert.processes 任务了：

```
ant convert.processes
```

```
jenova:activiti-jbpm-migration-5.2 jbarrez$ ls
activiti.db.properties  jbpm3.db.properties     processes
build.xml               lib                      src
jenova:activiti-jbpm-migration-5.2 jbarrez$ ant convert.processes
```

转换过程中，你会看到有大量描述流程转换执行到何处以及是如何执行的日志通过。转换结束时，你将看到有以下日志显示：

```
     [java] Jan 27, 2011 8:38:21 AM org.activiti.migration.ProcessMigration writeProcessToFile
     [java] INFO: Writing converted process to /Users/jbarrez/Desktop/activiti-jbpm-migration-5.2/
converted-processes-2011-01-27-08:38:21/pooledActorsProcess.bpmn20.xml
     [java] Jan 27, 2011 8:38:21 AM org.activiti.migration.ProcessMigration writeProcessToFile
     [java] INFO: Writing converted process to /Users/jbarrez/Desktop/activiti-jbpm-migration-5.2/
converted-processes-2011-01-27-08:38:21/actorIdProcess.bpmn20.xml
     [java] Jan 27, 2011 8:38:21 AM org.activiti.migration.ProcessMigration writeProcessToFile
     [java] INFO: Writing converted process to /Users/jbarrez/Desktop/activiti-jbpm-migration-5.2/
converted-processes-2011-01-27-08:38:21/simplest.bpmn20.xml
```

如日志所示，生成的 BMPN 2.0 的流程可以在 converted-process-xxxx 文件夹内找到，其中 xxxx 是转换时的时间戳。

此版本中，只限于支持的开始、结束、等待以及任务节点被实现了。将来，覆盖的情况会扩大。

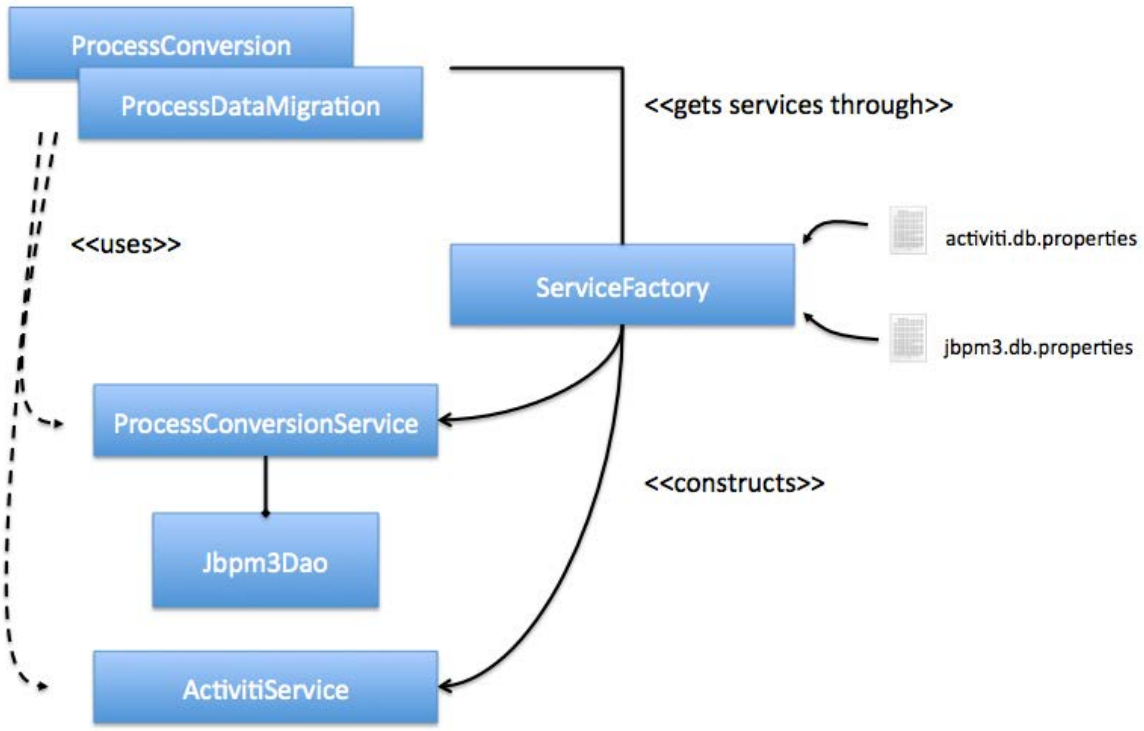
18.3 扩展迁移逻辑

该迁移逻辑是采用这样一种方式来编写的，即易于扩展以满足你的需要。源码是以下载的压缩文件内的 src 子文件夹内的 Maven 2 项目出现的。要构建新的压缩文件，在修改或扩展此逻辑后，只需执行

```
mvn clean install
```

来在 target 文件夹中创建新的压缩文件。

以下图片给出了对迁移逻辑中类的一个较高层次的概括。



- **ProcessConversion** 和 **ProcessDataMigration** 两个类中都有从迁移压缩文件根目录下的 ant 构建脚本直接调用的主调方法。
- 这两个类都是使用静态工厂方法基于两个属性文件来构造 **ServiceFactory** 的

```
ServiceFactory.configureFromProperties(jbpmDbProperties, activitiDbProperties);
```

- 那些服务是由 ServiceFactory 构造的（例如，getProcessConversionService()），被用来执行迁移的逻辑。

```
public void execute() throws IOException {

  // 转换流程
  ServiceFactory serviceFactory = createServiceFactory();
  ProcessConversionService processConversionService = serviceFactory.getProcessConversionService();
  Map<String, Document> migratedProcesses = processConversionService.convertAllProcessDefinitions();

  // 将结果写入bpmn20.xml文件
  writeConvertedProcesses(migratedProcesses, workingDir);

  // 将流程部署到Activiti
  ActivitiService activitiService = serviceFactory.getActivitiService();
  activitiService.deployConvertedProcesses(migratedProcesses);
```

```
    // 数据迁移
    ...
}
```

- **ProcessConversionService** 是个接口，其中包含有流程转换及流程定义数据获取的操作。它使用了 **Jbpm3Dao** 的实现。该类的默认实现使用了 Hibernate 中的 **SessionFactory** 来从 jBPM 3 的数据表中获取所有数据。
  ActivitiService 提供的操作是需要获得 Activiti 数据库表中的被迁移的数据的。例如，部署转换的流程定义就是这样的一个方法。
- 所有的这些依赖，ProcessConversionService, Jbpm3Dao, Sessionfactory, ActivitiService 和 ProcessEngine，都是接口，都可以由你自己的实现来实现。使用正规的 JavaBean 的 setter 方法，可以将它们注入给 ServiceFactory。当没有设置这样的自定义的实现时，ServiceFactory 会回过头来创建默认的实现：

```
public ProcessConversionService getProcessConversionService() {
    if (processConversionService == null) {
        this.processConversionService = createDefaultProcessConversionService();
    }
    return processConversionService;
}

protected ProcessConversionService createDefaultProcessConversionService() {
    ProcessConversionServiceImpl service = new ProcessConversionServiceImpl(getJbpm3Dao());
    return service;
}
```

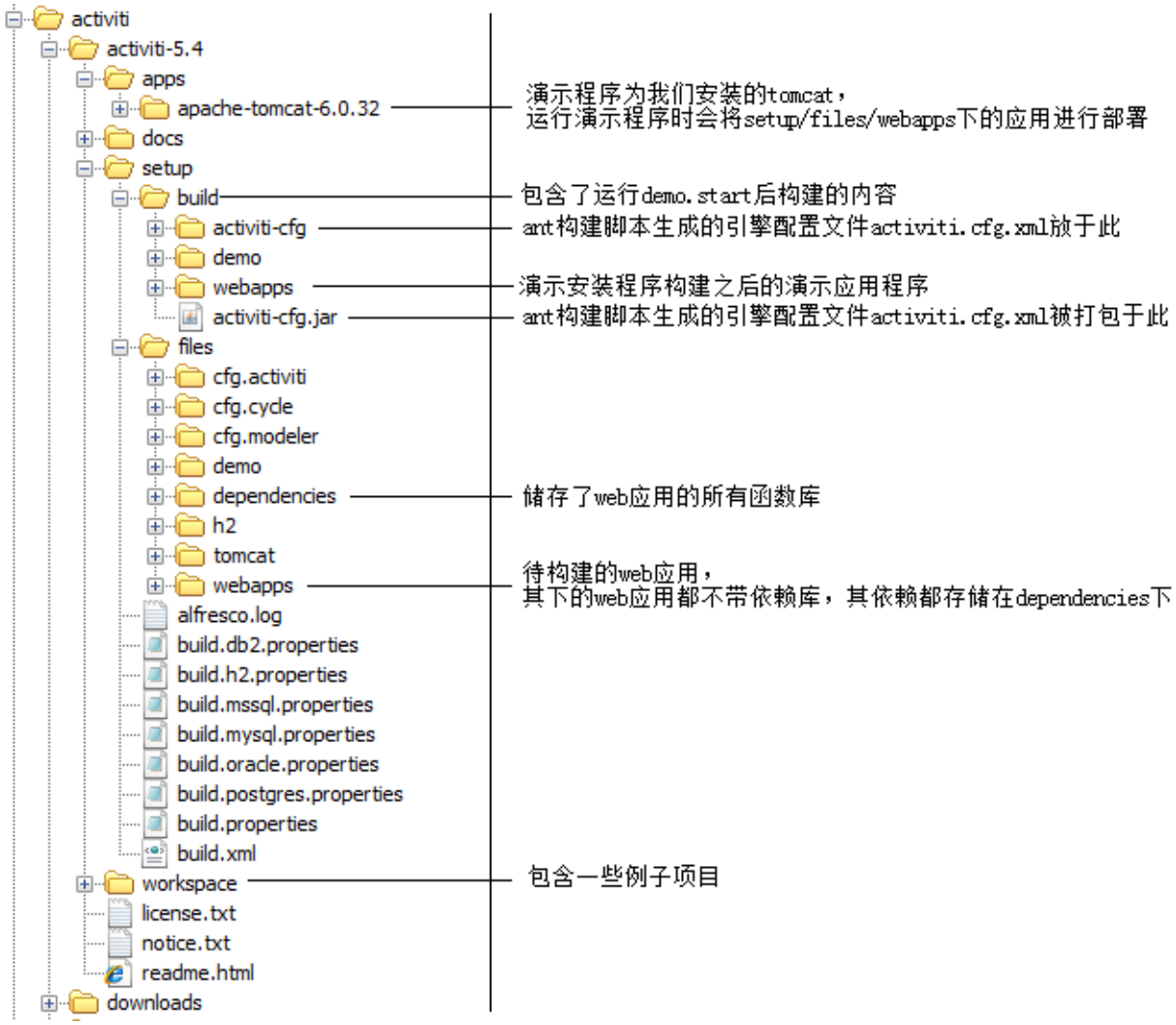
附录

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.creata：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. 2011/4/6 ： 启动。准确的说应该是之后的某天，因为 2011/4/19 我看了下解压开的 activiti-5.4 的日期是 2011/4/6 9:26，因为忙于工作，推迟了几天才开始着手翻译，其间认真读完《钢铁是怎样炼成的》。

2. 2011/4/19 10:19 ： 经过几天的努力，终于将 1-7 章的主要内容翻译完。今天是个好天气。

3. 2011/4/19 晚 至 2011/4/25 晚 ： 今天终于把第 7 章校验完。感觉时间好长啊。看到还有一半多的工作要做，不禁感到了很大的压力。

4. 截至 2011/5/2 中午 12:41 终于在许巍的《难忘的一天》的旋律下完成了 1-6 章的校验，以及第 8 章的部分翻译。

5. 2011/5/4 晚，完成第 8 章的校验。

6. 2011/5/6 晚，完成第 9 章的翻译。

7. 2011/5/7 上午，完成第 10 章的翻译。

8. 2011/5/13 凌晨 0:57，终于完成第 11 章的翻译与校验，其间不只一次的发现该英文原版手册中存在有病句的情况，而且很多句子的断句非常不理想。

9. 2011/5/13 23:12 完成第 12 章的翻译。

10. 2011/5/15 下午完成第 13 章的翻译。

11. 2011/5/19 1:07 完成第 14 章的翻译，感觉好困啊，眼睛真的要睁不开了。

12. 2011/5/19 晚 23:20 完成第 15、16 章的翻译。今天决定早点休息。

13. 2011/5/21 0:27 完成第 17 章的翻译及校验。估计明天，奥，应该是今天，现在已经是凌晨了，可以将第一稿截稿了。好困啊，两眼已经开始模糊了。听着《难忘的一天》那熟悉而又优美的旋律，心里又开始此起彼伏！

14. 2011/5/21 23:42 终于完成了该文档的全部翻译的工作，打算明天就文档内的一些相关的格式在做些整理。

15. 2011/5/22 晚 终于在博客中发表了关于此文档的日志。

16. 2011/5/26 晚 完成了本文档的最终排版。打算明天再写些无关紧要的说明性的东西。

17. 2011/5/27 晚 结稿。

关于译者及此文档

译者

姓　　名：栗建涛　　民　　族：汉

专　　业：信息管理与信息系统　　学　　历：本科

毕业学校：石家庄经济学院（本）　　邮　　箱：java_123@yeah.net

关于文档

经过近两个月的奋战，终于将《Activiti 5.4 用户指南（中文版）》翻译完成。能力有限，文档中难免会出现翻译不当之处，愿接受广大朋友的批评与指正，笔者将如期更正文档中的错误，并将您的大名一并写入文档修正日志中，以对您表示衷心的感谢。

Activiti 5.5 新近已经推出，由于渴望此 5.4 版本的文档能早日与大家见面，故没有再融进 Activiti 5.5 的翻译工作，笔者会依时间安排，尽快着手 5.5 版本的翻译工作，以方便国内朋友的学习，相关信息将发布到我的博客 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，2011 年 5 月 27 日，于石家庄

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，否则笔者将追究其法律责任。