Activi7工作流经典实战

=== 图灵： 楼兰 ===

一、Activiti7介绍

Activiti是目前使用最为广泛的开源工作流引擎，2010年5月就正是启动了。在了解Activiti之前，我们首先要了解下什么是工作流。

1.1 工作流WorkFlow

关于什么是工作流，有一个官方的定义： 工作流是指一类能够完全自动执行的经营过程，根据一系列规程规则，将文档、信息或任务在不同的执行者之间进行传递和执行。其实说直白一点，就是业务上一个完整的审批流程。例如员工的入职、请假、出差、采购等等、还有一些关键业务如订单申请、合同审核等等，这些过程，都是一个工作流。

对于工作流，传统的处理方式往往需要有人拿着各类的文件，在多个执行部门之间不断的审批。而当我们开始用软件来协助处理这一类审批流程时，就开始出现了工作流系统。工作流系统可以减少大量的线下沟通成本，提高工作效率。

有了工作流系统之后，才开始出现工作流引擎。在没有专门的工作流引擎之前，我们为了实现这样的流程控制，通常的做法都是采用状态字段的方式来跟踪流程的变化情况。例如对一个员工请假请求，我们会定义已申请、组长已审核、部门经理已审核等等这样一些状态，然后通过这些状态来控制不同的业务行为，比如部门经理角色只能看到组长已审核通过的，并且请假天数超过3天的订单等等。

这种实现方式实现起来比较简单，也是软件系统中非常常用的一种方式。但是这种通过状态字段来进行流程控制的方式还是有他的弊端。

一方面：整个流程定义不够清晰。业务流程是分散在各个业务阶段中的，从代码的角度非常难以看到整个流程是如何定义的。

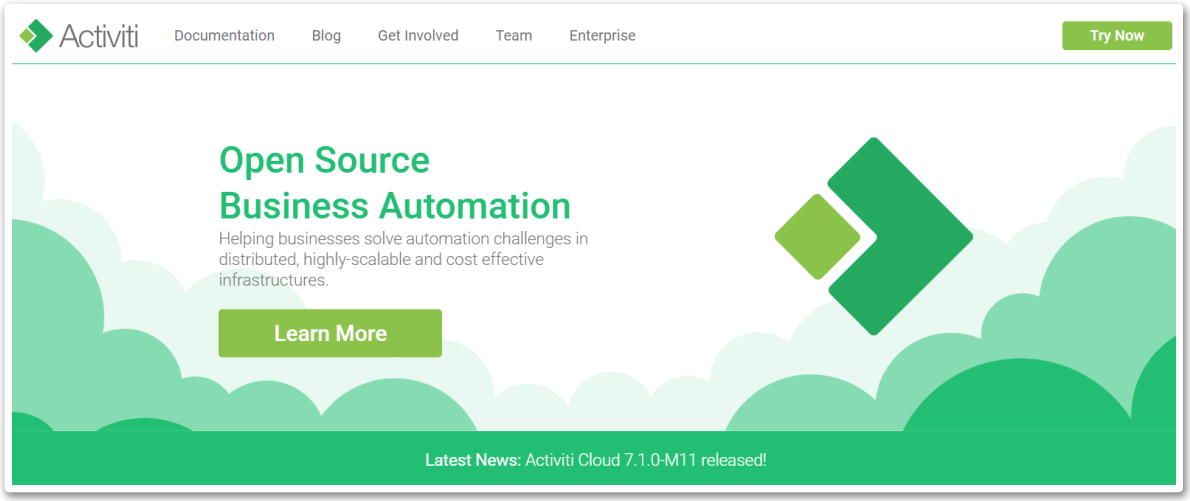
另一方面：当流程发生变更时，这种方式编写的代码就需要做非常大的变更。例如从三级审批要增加为四级审批甚至是协同审批，那各个业务阶段的审批流程都需要随之做大量的变更。

正是出于这些痛点，后面才有了工作流引擎。使用工作流引擎后，整个审批流程可以在同一个地方进行整体设计，并且当审批流程发生变更时，业务程序也可以不用改变。这样业务系统的适应能力就得到了极大提升。

> 其实引擎的思想无处不在。我们有Drools规则引擎，可以在程序不发生变动的情况下，集中定义业务规则并进行修改。Aviator表达式引擎，可以快速计算某一个表达式的结果。搜索引擎，可以快速进行统一搜索等等。其核心思想都是将业务之间的共性抽取出来，减少业务变动对程序的影响。

1.2 Activiti工作流引擎

Activiti正是目前使用最为广泛的开源工作流引擎。Activiti的官网地址是 https://www.activiti.org 历经6.x和5.x两个大的版本，目前最新的版本是 Activiti Cloud 7.1.0-M11。



他可以将业务系统中复杂的业务流程抽取出来，使用专门的建模语言BPMN2.0进行定义。业务流程按照预先定义的流程执行，整个实现流程完全由activiti进行管理，从而减少业务系统由于流程变更进行系统改造的工作量，从而减少系统开发维护成本，提高系统的健壮性。所以使用Activiti，重点就是两个步骤，首先使用BPMN定义流程，然后使用Activiti框架实现流程。

1.3 建模语言BPMN

谈到BPMN，首先就要谈BPM。 BPM即Business Process Managemenet，业务流程管理。是一种规范化的构造端到端的业务流程，以持续的提高组织业务效率。在常见的商业管理教育如EMBA、MBA中都包含了BPM的课程。

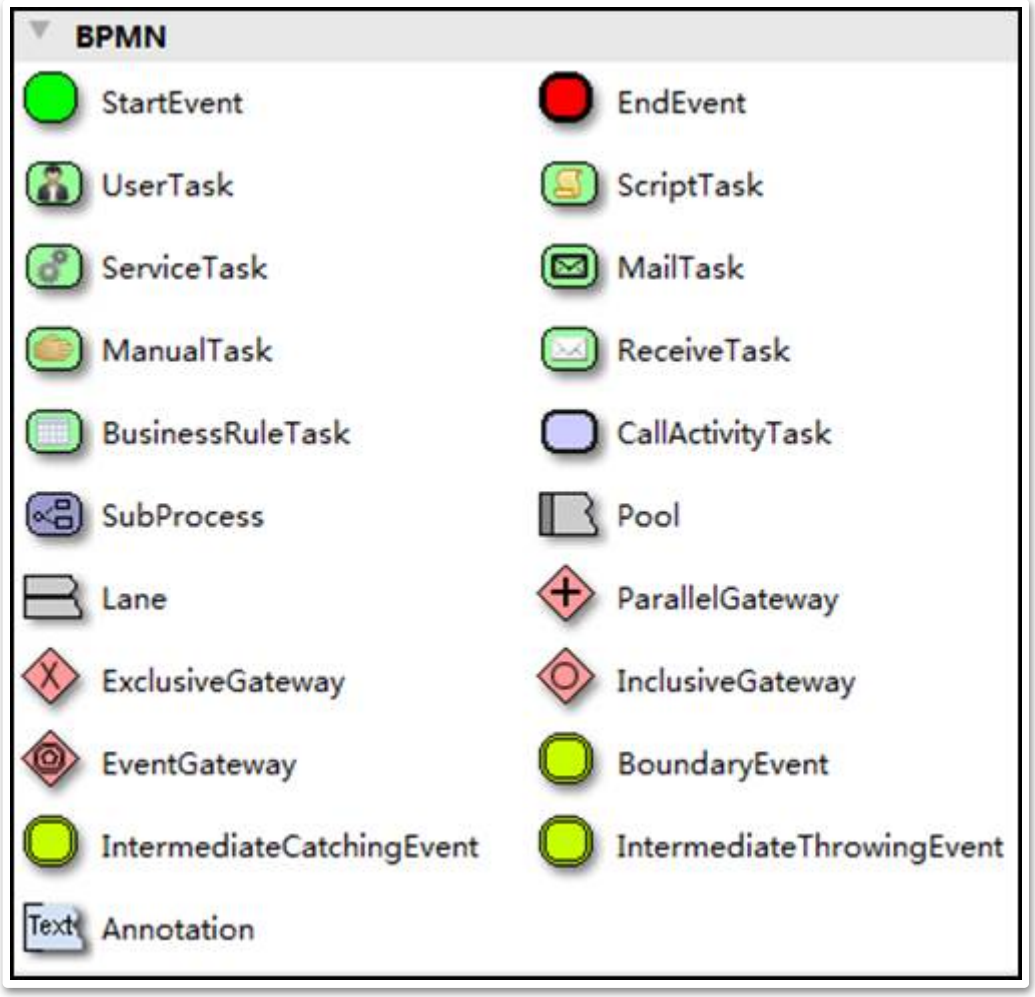
有了BPM的需求，就出现了BPM软件。他是根据企业中业务环境的变化，推进人与人之间，人与系统之间以及系统与系统之间的整合及调整的经营方法域解决方案的IT工具。通过对企业业务流程的整个生命周期进行建模、自动化、管理监控和优化，使企业成本降低，利润得到提升。BPM软件在企业中应用非常广泛，凡是有业

务流程的地方都可以使用BPM进行管理。比如企业人事办公管理、采购流程管理、公文审批流程管理、财务管理等。

而BPMN是Business Process Model And Notation 业务流程模型和符号，就是用来描述业务流程的一种建模标准。BPMN最早由BPMI(BusinessProcess Management Initiative)方案提出。由一整套标准的业务流程建模符号组成。使用BPMN可以快速定义业务流程。

BPMN最早在2004年5月发布。2005年9月开始并入OMG(The Object Managemenet Group)组织。OMG于2011年1月发布BPMN2.0的最终版本。BPMN是目前被各大BPM厂商广泛接受的BPM标准。Activiti就是使用BPMN2.0进行流程建模、流程执行管理。

整个BPMN是用一组符号来描述业务流程中发生的各种事件的。BPMN通过在这些符号事件之间连线来描述一个完整的业务流程。



而对于一个完整的BPMN图形流程，其实最终是通过XML进行描述的。通常，会将BPMN流程最终保存为一个.bpmn的文件，然后可以使用文本编辑器打开进行查看。而图形与xml文件之间，会有专门的软件来进行转换。

关于如何配置一个工作流，在后面的实战过程中我们会接触到。

1.4 Activiti使用步骤

通常使用Activiti时包含以下几个步骤：

- 部署activiti： Activiti包含一堆Jar包，因此需要把业务系统和Activiti的环境集成在一起进行部署。
- 定义流程： 使用Activiti的建模工具定义业务流程.bpmn文件。
- 部署流程定义： 使用Activiti提供的API把流程定义内容存储起来，在Acitivti执行过程汇总可以查询定义的内容。Activiti是通过数据库来存储业务流程的。
- 启动流程实例： 流程实例也叫ProcessInstance。启动一个流程实例表示开始一次业务流程的运作。例如员工提交请假申请后，就可以开启一个流程实例，从而推动后续的审批等操作。
- 用户查询待办任务(task)：因为现在系统的业务流程都交给了activiti管理，通过activiti就可以查询当前流程执行到哪个步骤了。当前用户需要办理哪些任务也就同样可以由activiti帮我们管理，开发人员不需要自己编写sql语句进行查询了。
- 用户办理任务：用户查询到自己的待办任务后，就可以办理某个业务，如果这个业务办理完成还需要其他用户办理，就可以由activiti帮我们把工作流程往后面的步骤推动。
- 流程结束：当任务办理完成没有下一个任务节点后，这个流程实例就执行完成了。

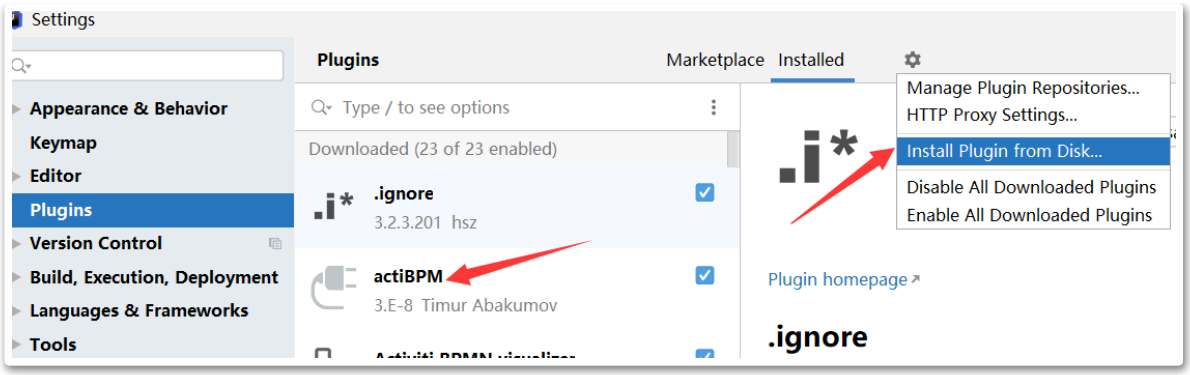
了解这些后，我们来开始进入实战内容。

二、Activiti环境搭建

使用Activiti需要的基本环境包括： JDK 8或以上版本；然后需要一个数据库用来保存流程定义数据，建议mysql 5或以上版本。

2.1 安装插件

开发工具IDEA，在IDEA中需要安装Activiti的流程定义工具插件actiBPM。目前该插件从2014年11月后就没有再更新，对于IDEA版本只支持到2019.1。新版本的IDEA已经无法从插件市场搜索到该插件。安装时，可以到jetBrain的插件市场 https://plugins.jetbrains.com/ 搜索actiBPM插件，下载到本地后，从本地安装该插件。



安装完成后，就可以使用这个插件在项目中编辑.bpmn的文件来定义业务流程了。但是这个文件之前介绍过，他的本质是一个xml文本文件，所以还是需要更多的了解xml的配置方式。

2.2 初始化数据库表

activiti支持多种数据库，详细的版本情况如下：

数据库类型	版本	JDBC连接示例	说明
h2	1.3.168	jdbc:h2:tcp://localhost/activiti	默认配置的数据库
mysql	5.1.21	jdbc:mysql://localhost:3306/activiti?autoReconnect=true	使用 mysql-connector-java 驱动测试
oracle	11.2.0.1.0	jdbc:oracle:thin:@localhost:1521:xe	
postgres	8.1	jdbc:postgresql://localhost:5432/activiti	
db2	DB2 10.1 using db2jcc4	jdbc:db2://localhost:50000/activiti	
mssql	2008 using sqljdbc4	jdbc:sqlserver://localhost:1433/activiti	

我们这里选择mysql数据库。接下来按照以下步骤来初始化activiti所需要的数据表。

1- 在mysql中创建一个数据库activiti，将会用来创建activiti相关的表。

```
CREATE DATABASE activiti DEFAULT CHARACTER SET utf8;
```

然后，activiti需要依赖的业务表有25张，而activiti中提供了工具帮我们生成所需要的表。

2- 创建一个maven工程BasicDemo，在pom.xml中引入以下依赖：

```
<properties>
    <slf4j.version>1.6.6</slf4j.version>
    <log4j.version>1.2.12</log4j.version>
    <activiti.version>7.1.0.M6</activiti.version>
    <activiti.cloud.version>7.0.0.Beta1</activiti.cloud.version>
    <mysql.version>8.0.20</mysql.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.activiti</groupId>
        <artifactId>activiti-engine</artifactId>
        <version>${activiti.version}</version>
    </dependency>
    <dependency>
        <groupId>org.activiti</groupId>
        <artifactId>activiti-spring</artifactId>
        <version>${activiti.version}</version>
    </dependency>
    <!-- bpmn 模型处理 -->
    <dependency>
        <groupId>org.activiti</groupId>
        <artifactId>activiti-bpmn-model</artifactId>
        <version>${activiti.version}</version>
    </dependency>
    <!-- bpmn 转换 -->
    <dependency>
        <groupId>org.activiti</groupId>
        <artifactId>activiti-bpmn-converter</artifactId>
        <version>${activiti.version}</version>
    </dependency>
    <!-- bpmn json数据转换 -->
    <dependency>
        <groupId>org.activiti</groupId>
        <artifactId>activiti-json-converter</artifactId>
        <version>${activiti.version}</version>
    </dependency>
    <!-- bpmn 布局 -->
    <dependency>
        <groupId>org.activiti</groupId>
        <artifactId>activiti-bpmn-layout</artifactId>
```

```
        <version>${activiti.version}</version>
    </dependency>
    <!-- activiti 云支持 -->
    <dependency>
        <groupId>org.activiti.cloud</groupId>
        <artifactId>activiti-cloud-services-api</artifactId>
        <version>${activiti.cloud.version}</version>
    </dependency>
    <!-- mysql驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>${mysql.version}</version>
    </dependency>
    <!-- mybatis -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.4.5</version>
    </dependency>
    <!-- 链接池 -->
    <dependency>
        <groupId>commons-dbcp</groupId>
        <artifactId>commons-dbcp</artifactId>
        <version>1.4</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <!-- log start -->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>${log4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>${slf4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>${slf4j.version}</version>
    </dependency>
</dependencies>
```

3- 添加log4j日志配置

这里采用的是log4j来记录日志，所以需要在resources目录下创建log4j.properties文件来对日志进行配置

```
# Set root category priority to INFO and its only appender to CONSOLE.
#log4j.rootCategory=INFO, CONSOLE debug info warn error fatal
log4j.rootCategory=debug, CONSOLE, LOGFILE
# Set the enterprise logger category to FATAL and its only appender to CONSOLE.
log4j.logger.org.apache.axis.enterprise=FATAL, CONSOLE
# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} %-6r[%15.15t] %-5p %30.30c %x - %m\n
# LOGFILE is set to be a File appender using a PatternLayout.
log4j.appender.LOGFILE=org.apache.log4j.FileAppender
log4j.appender.LOGFILE.File=f:\act\activiti.log
log4j.appender.LOGFILE.Append=true
log4j.appender.LOGFILE.layout=org.apache.log4j.PatternLayout
log4j.appender.LOGFILE.layout.ConversionPattern=%d{ISO8601} %-6r[%15.15t] %-5p %30.30c %x - %m\n
```

4- 添加activiti的配置文件

activiti默认就会使用mysql来创建表。创建时需要先创建一个配置文件activiti.cfg.xml，来对数据源信息进行定义。

在resources目录下创建activiti.cfg.xml文件。

> 注意：这个目录其实就是classpath下的默认位置。这是activiti默认读取的目录和文件。
>
> 创建在其他目录下也是可以的，但是就需要在生成时指定文件的目录和名字。

配置文件的基础内容如下： -这里主要是定义几个namespace。

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

5- 在activiti.cfg.xml中进行配置

我们可以在activiti.cfg.xml中添加关于数据库的基础配置

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

    <!-- 这里可以使用 链接池 dbcp-->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/activiti?serverTimezone=GMT%2B8" />
        <property name="username" value="root" />
        <property name="password" value="root" />
        <property name="maxActive" value="3" />
        <property name="maxIdle" value="1" />
    </bean>

    <bean id="processEngineConfiguration"

 class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
        <!-- 引用数据源 上面已经设置好了-->
```

```
        <property name="dataSource" ref="dataSource" />
        <!-- activiti数据库表处理策略 -->
        <property name="databaseSchemaUpdate" value="true"/>
    </bean>
</beans>
```

> 注意：1、processEngineConfiguration这个名字最好不要修改。这是activiti读取的默认Bean名字。
>
> 2、在processEngineConfiguration中也可以直接配置jdbcDriver、jdbcUrl、jdbcUsername、jdbcPassword几个属性。
>
> 3、关于databaseSchemaUpdate这个属性，稍微跟踪一下源码就能看到他的配置方式：
>
> 默认是false；表示不创建数据库，只是检查数据库中的表结构，不满足就会抛出异常
>
> create-drop：表示在引擎启动时创建表结构，引擎处理结束时删除表结构。
>
> true：表示创建完整表机构，并在必要时更新表结构。

6- 编写java程序生成表。

创建一个测试类，调用activiti的工具类，直接生成activiti需要的数据库表。代码如下：

```
package com.roy;

import org.activiti.engine.ProcessEngine;
import org.activiti.engine.ProcessEngines;
import org.junit.Test;

/**
 * @author : 楼兰
 * @date : Created in 2021/4/7
 * @description:
 **/

public class TestCreateTable {
    /**
     * 生成 activiti的数据库表
     */
```

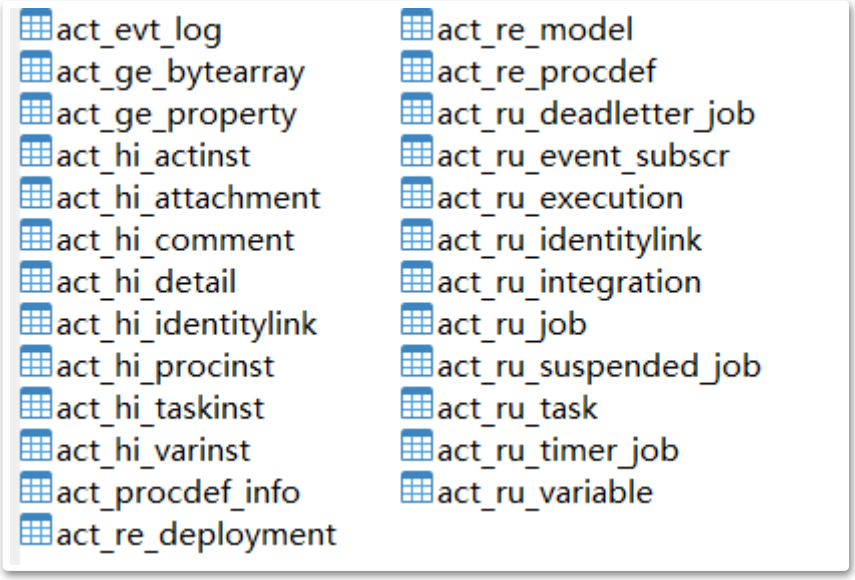
```
17          @Test
18      public void testCreateDbTable() {
19          //默认创建方式
20          ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
21          //通用的创建方式，指定配置文件名和Bean名称
22  //          ProcessEngineConfiguration processEngineConfiguration =
ProcessEngineConfiguration.createProcessEngineConfigurationFromResource("ac
tiviti.cfg.xml", "processEngineConfiguration");
23  //          ProcessEngine processEngine1 =
processEngineConfiguration.buildProcessEngine();
24          System.out.println(processEngine);
25      }
26  }
```

> 注意：从这个代码就能看出我们之前那些默认配置的作用。ProcessEngines.getDefaultProcessEngine()这行代码默认就会去读取classpath:下的activiti.cfg.xml和activiti-context.xml两个配置文件。并且从spring容器中加载名为processEngineConfiguration的Bean。

执行这个脚本就会完成mysql的表结构创建。如果执行正常，可以看到执行了一大堆的sql语句，最终打印出一行日志

```
1  org.activiti.engine.impl.ProcessEngineImpl@77307458
```

这就表示引擎创建成功了。同时在mysql中可以看到activiti用到的25张表。



> 这些表机构通常也可以导出成sql文件，然后直接进行移植。但是考虑到不同版本可能会有微调，所以通常不建议以sql文件的方式移植。

2.3 表结构解读

从这些刚才创建的表中可以看到，activiti的表都以act_开头。第二个部分表示表的用途。用途也和服务的API对应。

ACT_RE：'RE'表示 repository。 这个前缀的表包含了流程定义和流程静态资源（图片，规则，等等）。
ACT_RU：'RU'表示 runtime。 这些运行时的表，包含流程实例，任务，变量，异步任务，等运行中的数据。 Activiti 只在流程实例执行过程中保存这些数据， 在流程结束时就会删除这些记录。 这样运行时表可以一直很小速度很快。
ACT_HI：'HI'表示 history。 这些表包含历史数据，比如历史流程实例， 变量，任务等等。
ACT_GE： GE 表示 general。 通用数据， 用于不同场景下

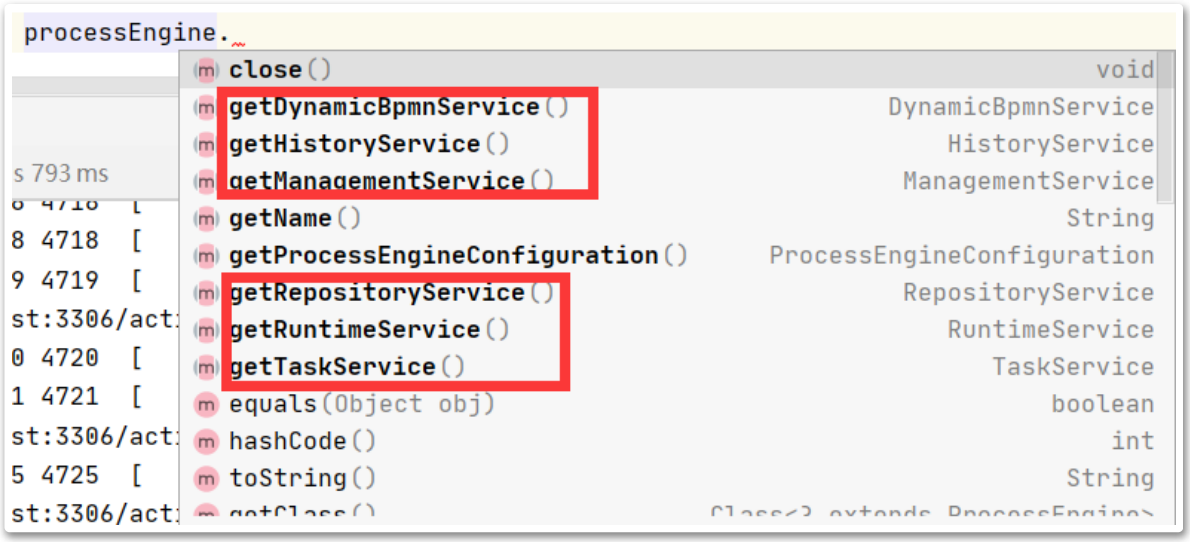
完整的数据库表作用如下：

表分类	表名	解释
一般数据		
	[ACT_GE_BYTEARRAY]	通用的流程定义和流程资源
	[ACT_GE_PROPERTY]	系统相关属性
流程历史记录		
	[ACT_HI_ACTINST]	历史的流程实例
	[ACT_HI_ATTACHMENT]	历史的流程附件
	[ACT_HI_COMMENT]	历史的说明性信息
	[ACT_HI_DETAIL]	历史的流程运行中的细节信息
	[ACT_HI_IDENTITYLINK]	历史的流程运行过程中用户关系
	[ACT_HI_PROCINST]	历史的流程实例
	[ACT_HI_TASKINST]	历史的任务实例
	[ACT_HI_VARINST]	历史的流程运行中的变量信息
流程定义表		
	[ACT_RE_DEPLOYMENT]	部署单元信息
	[ACT_RE_MODEL]	模型信息
	[ACT_RE_PROCDEF]	已部署的流程定义
运行实例表		
	[ACT_RU_EVENT_SUBSCR]	运行时事件
	[ACT_RU_EXECUTION]	运行时流程执行实例

表分类	表名	解释
	[ACT_RU_IDENTITYLINK]	运行时用户关系信息，存储任务节点与参与者的相关信息
	[ACT_RU_JOB]	运行时作业
	[ACT_RU_TASK]	运行时任务
	[ACT_RU_VARIABLE]	运行时变量表

2.4 Activiti核心类

当拿到ProcessEngine之后，我们可以简单的看一下他的方法



这几个service就是activiti最为核心的几个服务实现类。围绕activiti的核心业务功能大都通过这几个service来组成。

service名称	service作用
RepositoryService	activiti的资源管理类
RuntimeService	activiti的流程运行管理类
TaskService	activiti的任务管理类
HistoryService	activiti的历史管理类
ManagerService	activiti的引擎管理类

简单介绍：

RepositoryService

是activiti的资源管理类，提供了管理和控制流程发布包和流程定义的操作。使用工作流建模工具设计的业务流程图需要使用此service将流程定义文件的内容部署到计算机。

除了部署流程定义以外还可以：查询引擎中的发布包和流程定义。

暂停或激活发布包，对应全部和特定流程定义。 暂停意味着它们不能再执行任何操作了，激活是对应的反向操作。获得多种资源，像是包含在发布包里的文件， 或引擎自动生成的流程图。

获得流程定义的pojo版本， 可以用来通过java解析流程，而不必通过xml。

RuntimeService

Activiti的流程运行管理类。可以从这个服务类中获取很多关于流程执行相关的信息

TaskService

Activiti的任务管理类。可以从这个类中获取任务的信息。

HistoryService

Activiti的历史管理类，可以查询历史信息，执行流程时，引擎会保存很多数据（根据配置），比如流程实例启动时间，任务的参与者， 完成任务的时间，每个流程实例的执行路径，等等。 这个服务主要通过查询功能来获得这些数据。

ManagementService

Activiti的引擎管理类，提供了对 Activiti 流程引擎的管理和维护功能，这些功能不在工作流驱动的应用程序中使用，主要用于 Activiti 系统的日常维护。

三、Activiti入门

在这一章，我们就来创建一个Activiti工作流，并启动这个工作流。了解Activiti的基础开发流程。

创建Activiti工作流的主要步骤包含以下几步：

1. 定义流程。按照BPMN的规范，使用流程定义工具，将整个流程描述出来
2. 部署流程。把画好的BPMN流程定义文件加载到数据库中，生成相关的表数据
3. 启动流程。使用java代码来操作数据库表中的内容。

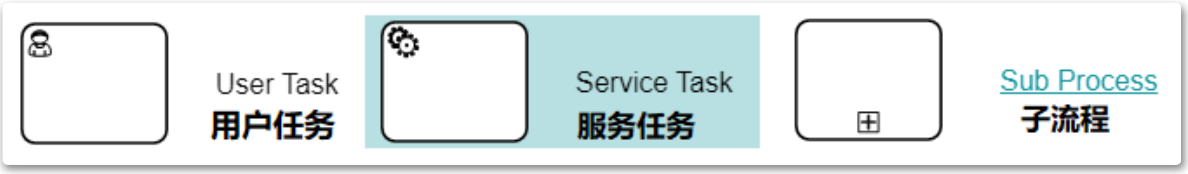
3.1 流程符号详解

接下来我们来了解下在流程设计中常见的符号。BPMN2.0的基本符号主要包含以下几类：

- 事件 Event

事件是驱动工作流发展的核心对象，在工作流的流程定制过程中会经常看到。

- 活动 Activity



活动是工作或任务的一个通用术语。一个活动可以是一个任务，也可以是当前流程的子处理流程。并且，活动会有不同的类型。例如Activiti中定义了UserTask,ScriptTask,ServiceTask,MailTask等等多种类型。这些活动是构成整个业务流程的主体。

- 网关 GateWay

网关是用来处理角色的，他决定了工作流的业务走向。有几种常用的网关需要了解一下：

- 排他网关

只有一条路径会被选择。流程执行到该网关时，会按照输出流的顺序逐个计算，当条件的结果为true时，继续执行当前网关的输出流。

如果有多条线路计算结构都是true，则会执行第一个值为true的路线。如果所有网关计算结果都没有true，引擎会抛出异常。

排他网关需要和条件顺序流结合使用，default属性指定默认顺序流，当所有的条件不满足时会执行默认顺序流。

- 并行网关

所有路径会被同时选择。

并行执行所有输出顺序流，为每一条顺序流创建一个并行执行路线。最终，在所有的执行路线都执行完后才继续向下执行。

- 包容网关

可以同时执行多条路线。相当于是排他网关和并行网关的结合。

可以在网关上设置条件，计算每条路线上的表达式。当表达式计算结果为true时，创建一个并行线路并继续执行。最终，当所有需要执行的线路都执行完成后才继续向下执行。

- 事件网关

专门为中间捕获事件而设置。允许设置多个输出流指向多个不同的中间捕获事件。当流程执行到事件网关后，流程出于等待状态，需要等待抛出对应的事件才能将等待状态转换为活动状态。

- 流向 Flow

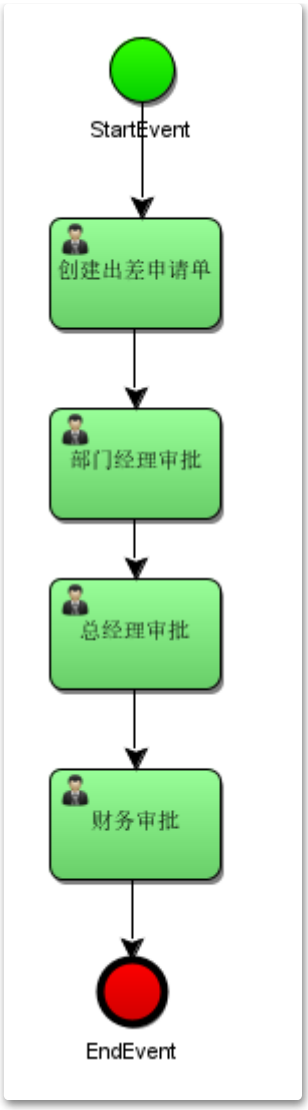
流就是连接两个流程节点的连线，代表了流程之间的关联关系。

> 注意：Activiti的BPMN符号是在标准符号基础上做了一定的扩展。你可以新建一个BPMN文件，然后对照actBPM工具提供的工作流图形来理解。

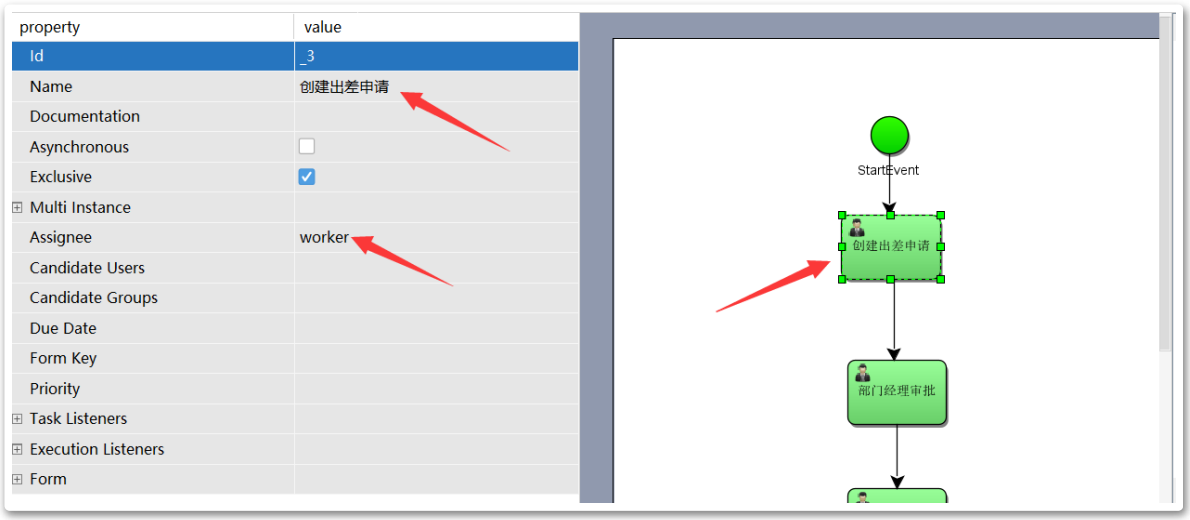
3.2 定制一个简单的请假流程

首先在resources目录下创建bpmn目录，然后在目录下创建一个Bomn文件，起名Leave，表示是一个请假流程。

然后我们在流程图中拖拽出下图的流程图

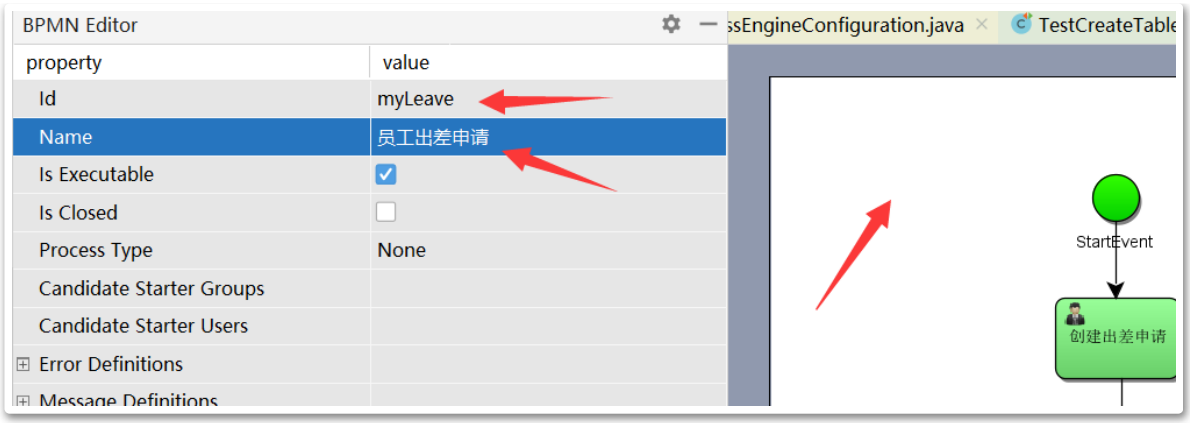


这里注意每个模块都可以选中后在左侧设置他的属性。



这里，Assignee属性表示是这个任务的负责人。这里我们给 创建出差申请 设置负责人 worker；部门经理审批 设置负责人 manager；财务审批 设置负责人 financer。

设置完成后，记得点击一下流程的空白页，在左侧设置整个流程的属性。



这样，我们整个员工出差申请的流程就定义完成了。之前介绍过，这个文件实际上是一个xml文件，所以，这个文件是可以用文本编辑器直接打开的。整个文件大致是这样

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:activiti="http://activiti.org/bpmn"
xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI" xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI" xmlns:tns="http://www.activiti.org/test"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
expressionLanguage="http://www.w3.org/1999/XPath" id="m1617786776055" name=""
targetNamespace="http://www.activiti.org/test" typeLanguage="http://www.w3.org/2001/XMLSchema">
  <process id="myLeave" isClosed="false" isExecutable="true" name="员工出差申请" processType="None">
    <startEvent id="_2" name="StartEvent"/>
    <userTask activiti:assignee="worker" activiti:exclusive="true" id="_3" name="创建出差申请"/>
    <userTask activiti:assignee="manager" activiti:exclusive="true" id="_4" name="部门经理审批"/>
    <userTask activiti:assignee="financer" activiti:exclusive="true" id="_5" name="财务审批"/>
    <endEvent id="_6" name="EndEvent"/>
    <sequenceFlow id="_7" sourceRef="_2" targetRef="_3"/>
    <sequenceFlow id="_8" sourceRef="_3" targetRef="_4"/>
    <sequenceFlow id="_9" sourceRef="_4" targetRef="_5"/>
    <sequenceFlow id="_10" sourceRef="_5" targetRef="_6"/>
  </process>
  <bpmndi:BPMNDiagram
  documentation="background=#FFFFFF;count=1;horizontalcount=1;orientation=0;width=842.4;height=1195.2;
  imageableWidth=832.4;imageableHeight=1185.2;imageableX=5.0;imageableY=5.0" id="Diagram-_1" name="New Diagram">
    <bpmndi:BPMNPlane bpmnElement="myLeave">
      <bpmndi:BPMNShape bpmnElement="_2" id="Shape-_2">
        <omgdc:Bounds height="32.0" width="32.0" x="200.0" y="45.0"/>
        <bpmndi:BPMNLabel>
```

其中根节点是definitions节点。在这个节点中，可以定义多个工作流程process节点。这也就意味着可以在一个图中定义多个工作流，但是通常在使用过程中建议一个文件只包含一个流程定义，这样可以简化维护难度。definitions节点中，xmlns和tagetNamespace两个属性是必须要包含的。

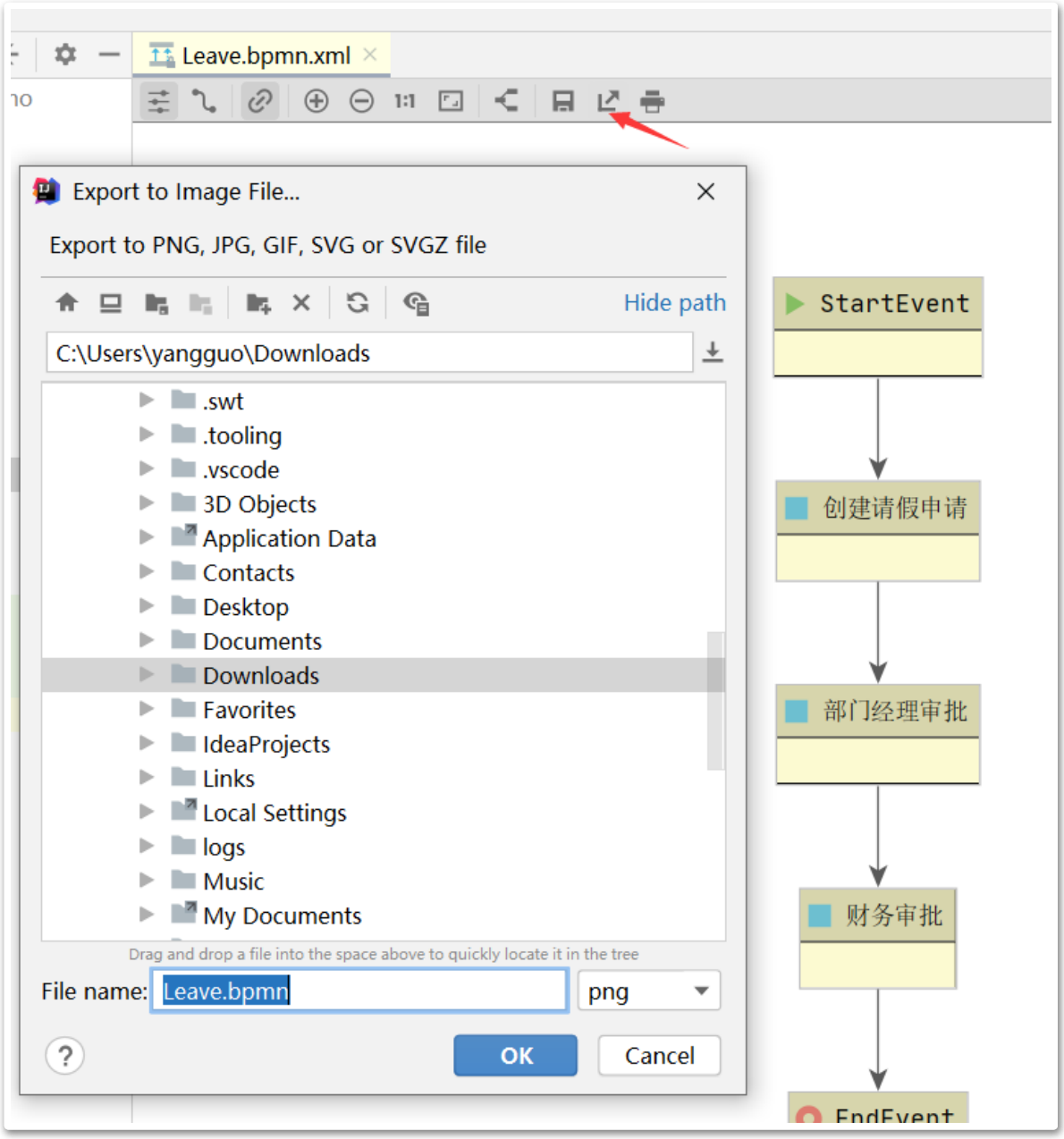
然后在文件中，包含了以标签描述的流程定义部分以及以<bpmndi:BPMNDiagram >标签描述的流程布局定义部分。分别用来定义工作流程以及流程图的布局信息。

> 文件关闭后，重新打开，可能会出现中文乱码的问题。这是因为IDEA的字符集问题。此时，需要修改IDEA的配置文件。 在IDEA中选Help -> Edit Custom VM Options 菜单，在打开的文件最后加上一行-Dfile.encoding=UTF-8配置，然后重启IDEA即可。

3.3 部署请假流程

接下来，需要将设计器中定义的流程部署到activiti的数据库中。activiti提供了api将流程定义的bpmn和png文件部署到activiti中。

要获取png图片文件，可以直接截图，也可以使用IDEA导出。导出时，先将文件修改为Leave.bpmn.xml，然后右键该文件，选择Diagrams -> Shwo BPMN 2.0 Diagrams，可以打开图片工具，然后选择上方的导出按钮，就可以导出一个png文件。



部署流程时，可以分别上传bpmn文件和png文件，也可以将两个文件打成zip压缩包一起上传。

```
1  public class ActivitiDemo {
2
3      /**
4       * 部署流程定义  文件上传方式
```

```
     */
    @Test
    public void testDeployment(){
//        1、创建ProcessEngine
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
//        2、得到RepositoryService实例
        RepositoryService repositoryService =
processEngine.getRepositoryService();
//        3、使用RepositoryService进行部署
        Deployment deployment = repositoryService.createDeployment()
                .addClasspathResource("bpmn/Leave.bpmn") // 添加bpmn资源
  //png资源命名是有规范的。Leave.[key].[png|jpg|gif|svg]  或者Leave.
[png|jpg|gif|svg]
            .addClasspathResource("bpmn/Leave.myLeave.png")  // 添加png资源
                .name("请假申请流程")
                .deploy();
//        4、输出部署信息
        System.out.println("流程部署id: " + deployment.getId());
        System.out.println("流程部署名称: " + deployment.getName());
    }

    /**
     * zip压缩文件上传方式
     */
    @Test
    public void deployProcessByZip() {
        // 定义zip输入流
        InputStream inputStream = this
                .getClass()
                .getClassLoader()
                .getResourceAsStream(
                        "bpmn/Leave.zip");
        ZipInputStream zipInputStream = new ZipInputStream(inputStream);
        // 获取repositoryService
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
        RepositoryService repositoryService = processEngine
                .getRepositoryService();
        // 流程部署
        Deployment deployment = repositoryService.createDeployment()
                .addZipInputStream(zipInputStream)
                .deploy();
        System.out.println("流程部署id: " + deployment.getId());
        System.out.println("流程部署名称: " + deployment.getName());
    }
}
```

这个过程中，最重要的就是要去找repositoryService。执行完成后可以在日志中看到部署的情况：

```
流程部署id：1
流程部署名称：请假申请流程
```

并且，从日志中可以分析出整个部署过程操作了三张数据表:

- act_re_deployment　　流程定义部署表，每部署一次增加一条记录
- act_re_procdef　　流程定义表，部署每个新的流程定义都会在这张表中增加一条记录。记录中的key就是流程定义中最为重要的字段。
- act_ge_bytearray　　流程资源表，每个流程定义对应两个资源记录，bpmn和png。

> 一次部署可以部署多个流程定义 即act_re_deployment和act_re_procdef中的数据其实是一对多的。但是在实际开发中，建议一次部署只不是一个流程。

3.4 启动流程实例

一个业务流程部署到activiti后，就可以使用了。例如这个出差申请的流程部署完成后，就可以启动一个流程进行一次出差申请了。流程的执行过程主要通过RuntimeService服务来管理。

```
 /**
     * 启动流程实例
     */
    @Test
    public void testStartProcess(){
//        1、创建ProcessEngine
        ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
//        2、获取RunTimeService
        RuntimeService runtimeService = processEngine.getRuntimeService();
//        3、根据流程定义Id启动流程
        ProcessInstance processInstance = runtimeService
                .startProcessInstanceByKey("myLeave");
//        输出内容
        System.out.println("流程定义id: " + processInstance.getProcessDefinitionId());
        System.out.println("流程实例id: " + processInstance.getId());
        System.out.println("当前活动Id: " + processInstance.getActivityId());
    }
```

执行结果可以看到流程实例的情况：

```
流程定义id：myLeave:1:4
流程实例id：2501
当前活动Id：null
```

继续查看日志，可以看到这个过程中涉及到的数据表：

- act_hi_actinst　流程实例执行历史
- act_hi_identitylink　流程的参与用户历史信息
- act_hi_procinst　流程实例历史信息
- act_hi_taskinst　流程任务历史信息
- act_ru_execution　流程执行信息
- act_ru_identitylink　流程的参与用户信息
- act_ru_task　任务信息

3.5 任务查询

流程启动后，任务的负责人就可以查询自己当前需要处理的待办任务了。任务相关的服务都是由TaskService管理。

```
/**
     * 查询当前个人待执行的任务
      */
    @Test
    public void testFindPersonalTaskList() {
//        任务负责人
        String assignee = "worker";
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
//        创建TaskService
        TaskService taskService = processEngine.getTaskService();
//        根据流程key 和 任务负责人 查询任务
        List<Task> list = taskService.createTaskQuery()
                .processDefinitionKey("myLeave") //流程Key
                .taskAssignee(assignee)//只查询该任务负责人的任务
                .list();

        for (Task task : list) {

            System.out.println("流程实例id：" + task.getProcessInstanceId());
            System.out.println("任务id：" + task.getId());
            System.out.println("任务负责人：" + task.getAssignee());
            System.out.println("任务名称：" + task.getName());
```

```
23
24          }
25      }
```

执行完成后可以看到当前流程的任务列表

```
流程实例id：2501
任务id：2505
任务负责人：worker
任务名称：创建请假申请
```

> 当前请假流程启动后，就该等待worker用户提交请假申请了。实际中的请假申请流程应该是从worker提交请假申请开始，但是在activiti工作流中，都是从starter事件开始，这个关系要理清楚。

3.6 流程任务处理

任务负责人查询到代办任务后，可以选择任务进行处理，完成任务。

```
// 完成任务
    @Test
    public void completTask(){
//        获取引擎
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
//        获取taskService
        TaskService taskService = processEngine.getTaskService();

//        根据流程key 和 任务的负责人 查询任务
//        返回一个任务对象
        Task task = taskService.createTaskQuery()
                .processDefinitionKey("myLeave") //流程Key
                .taskAssignee("worker")  //要查询的负责人
                .singleResult();

//        完成任务,参数：任务id
        taskService.complete(task.getId());
    }

```

这个任务完成后，这一个请假流程就推动到了下一个步骤，部门经理审批了。后续可以用不同的用户来推动流程结束。

其实在完成审批任务的过程中，可以针对这个taskId，进行其他一些补充操作。例如添加Comment，添加附件，添加子任务，添加候选负责人等等。具体可以看下taskService的API。

3.7 流程信息查询

这一步可以查询流程相关信息，包含流程定义，流程部署，流程版本。

```
    /**
     * 查询流程定义
     */
    @Test
    public void queryProcessDefinition(){
        //        获取引擎
        ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
//        repositoryService
        RepositoryService repositoryService = processEngine.getRepositoryService();
//        得到ProcessDefinitionQuery 对象
        ProcessDefinitionQuery processDefinitionQuery = repositoryService.createProcessDefinitionQuery();
//          查询出当前所有的流程定义
//          条件：processDefinitionKey =evection
//          orderByProcessDefinitionVersion 按照版本排序
//        desc倒叙
//        list 返回集合
        List<ProcessDefinition> definitionList = processDefinitionQuery.processDefinitionKey("myLeave")
                .orderByProcessDefinitionVersion()
                .desc()
                .list();
//      输出流程定义信息
        for (ProcessDefinition processDefinition : definitionList) {
            System.out.println("流程定义 id="+processDefinition.getId());
            System.out.println("流程定义 name="+processDefinition.getName());
            System.out.println("流程定义 key="+processDefinition.getKey());
            System.out.println("流程定义 Version="+processDefinition.getVersion());
            System.out.println("流程部署ID ="+processDefinition.getDeploymentId());
        }

    }
```

执行后可以看到查询结果

```
流程定义 id=myLeave:1:4
流程定义 name=员工请假审批流程
流程定义 key=myLeave
流程定义 Version=1
流程部署ID =1
```

3.8 删除流程

```
public void deleteDeployment() {
        // 流程部署id
        String deploymentId = "1";

    ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
    // 通过流程引擎获取repositoryService
        RepositoryService repositoryService = processEngine
                .getRepositoryService();
        //删除流程定义，如果该流程定义已有流程实例启动则删除时出错
        repositoryService.deleteDeployment(deploymentId);
        //设置true 级联删除流程定义，即使该流程有流程实例启动也可以删除，设置为false非级连删除方式
        //repositoryService.deleteDeployment(deploymentId, true);
    }
```

> 注：
>
> 1、这里只删除了流程定义，不会删除历史表信息
>
> 2、删除任务时，可以选择传入一个boolean型的变量cascade ，表示是否级联删除。默认是false，表示普通删除。
>
> 如果该流程下存在已经运行的流程，使用普通删除会报错，而级联删除可以将流程及相关记录全部删除。删除没有完成的流程节点后，就可以完全删除流程定义信息了。
>
> 项目开发中，级联删除操作一般只开放给管理员使用。

3.9 流程资源下载

在流程执行过程中，可以上传流程资源文件。我们之前在部署流程时，已经将bpmn和描述bpmn的png图片都上传了，并且在流程执行过程中，也可以上传资源文件。如果其他用户想要查看这些资源文件，可以从数据库中把资源文件下载下来。

但是文件是以Blob的方式存在数据库中的，要获取Blob文件，可以使用JDBC来处理。也可以使用activiti提供的api来辅助实现。我们这里采用activiti的方式来实现。

首先引入commons-io依赖

```
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.6</version>
</dependency>
```

然后，就可以通过流程定义对象来获取流程资源。这里获取我们之前上传的bpmn和png文件

```
import org.apache.commons.io.IOUtils;

@Test
    public void deleteDeployment(){
//        获取引擎
        ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
//        获取repositoryService
        RepositoryService repositoryService = processEngine.getRepositoryService();
//        根据部署id 删除部署信息,如果想要级联删除，可以添加第二个参数，true
        repositoryService.deleteDeployment("1");
    }

    public void  queryBpmnFile() throws IOException {
//        1、得到引擎
        ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
//        2、获取repositoryService
        RepositoryService repositoryService = processEngine.getRepositoryService();
//        3、得到查询器：ProcessDefinitionQuery，设置查询条件,得到想要的流程定义
        ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()
                .processDefinitionKey("myLeave")
                .singleResult();
//        4、通过流程定义信息，得到部署ID
        String deploymentId = processDefinition.getDeploymentId();
//        5、通过repositoryService的方法，实现读取图片信息和bpmn信息
//        png图片的流
```

```
        InputStream pngInput = repositoryService.getResourceAsStream(deploymentId, processDefinition.getDiagramResourceName());
//        bpmn文件的流
        InputStream bpmnInput = repositoryService.getResourceAsStream(deploymentId, processDefinition.getResourceName());
//        6、构造OutputStream流
        File file_png = new File("d:/myLeave.png");
        File file_bpmn = new File("d:/myLeave.bpmn");
        FileOutputStream bpmnOut = new FileOutputStream(file_bpmn);
        FileOutputStream pngOut = new FileOutputStream(file_png);
//        7、输入流，输出流的转换
        IOUtils.copy(pngInput,pngOut);
        IOUtils.copy(bpmnInput,bpmnOut);
//        8、关闭流
        pngOut.close();
        bpmnOut.close();
        pngInput.close();
        bpmnInput.close();
    }

```

> 注： 在获取资源文件名时，png图片资源的文件名是processDefinition.getDiagramResourceName()，他来自于ACT_RE_PROCDEF表中的DGRM_RESOURCE_NAME字段。这个字段的值是在部署流程时根据文件名后缀判断出来的。 支持的格式为[ResourceName].[key].[png|jpg|gif|svg]或者[ResourceName].[png|jpg|gif|svg]
>
> 而bpmn文件的文件名是processDefinition.getResourceName()，他来自于ACT_RE_PROCDEF表中的RESOURCE_NAME字段。

3.10 流程历史信息查看

流程的历史信息都保存在activiti的act_hi_*相关的表中，我们可以查询流程执行的历史信息。这里需要通过HistoryService来查看相关的历史记录。

```
/**
     * 查看历史信息
     */
    @Test
    public void findHistoryInfo(){
//        获取引擎
```

```
        ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
//         获取HistoryService
        HistoryService historyService = processEngine.getHistoryService();
//         获取 actinst表的查询对象
        HistoricActivityInstanceQuery instanceQuery = historyService.createHistoricActivityInstanceQuery();
//         查询 actinst表，条件：根据 InstanceId 查询，查询一个流程的所有历史信息
        instanceQuery.processInstanceId("25001");
//         查询 actinst表，条件：根据 DefinitionId 查询，查询一种流程的所有历史信息
//         instanceQuery.processDefinitionId("myLeave:1:22504");
//         增加排序操作,orderByHistoricActivityInstanceStartTime 根据开始时间排序 asc 升序
        instanceQuery.orderByHistoricActivityInstanceStartTime().asc();
//         查询所有内容
        List<HistoricActivityInstance> activityInstanceList = instanceQuery.list();
//         输出
        for (HistoricActivityInstance hi : activityInstanceList) {
            System.out.println(hi.getActivityId());
            System.out.println(hi.getActivityName());
            System.out.println(hi.getProcessDefinitionId());
            System.out.println(hi.getProcessInstanceId());
            System.out.println("<==========================>");
        }
    }
```

这样可以查询到之前的步骤处理结果

```
_2
StartEvent
myLeave:1:22504
25001
<==========================>
_3
创建请假申请
myLeave:1:22504
25001
<==========================>
_4
部门经理审批
myLeave:1:22504
25001
<==========================>
```

> 注：1、关于流程历史信息，要注意，在删除流程时，如果是采取级联删除的方式，那这个历史信息也会随着一起删除。而普通删除方式不会删除历史信息。
>
> 2、历史信息有不同的种类，具体可以通过historyService构建不同类型的Query对象来获取结果。

3.11 篇章总结

通过这一章的内容，我们已经完成了一个工作流的基础流程，也能对activiti的工作机制有个大致的了解。

activiti的强大之处在于，他围绕BPMN2.0构建的工作流程定义，提供了一系列完整的后台工作流功能。这些后台功能可以构成一个稳定的后台程序，针对不同的业务流程，只需要提供不同的BPMN定义文件，而不需要修改后台程度代码。并且，当业务流程发生变动时，也只需要修改BPMN文件中的流程定义，相应的后台代码基本不需要动。

但是也要看到，activiti只提供了后台功能，并没有配套的前端整合。并且，当需要对某一个任务或者某一个工作流做一些细致性的操作时，还是需要传入一些特定的业务参数，而这些业务参数，还是需要有个系统来进行整体的处理的。所以，activiti是一个强大的工作流引擎，但是距离一个完整的工作流系统还是有点差距的。

另外，在学习activiti时，数据库中的25张表也是非常关键的地方，这些最终保存的数据中包含了工作流运行过程中的所有数据。在实际使用中，我们基本不可能完全搞明白这25张表的数据内容，但是对于一些关键的操作数据还是需要了解下的，这是我们以后掌握整个工作流引擎运行状态的重要依据。

在我们对Activiti有了大致的理解后，接下来将深入一些Activiti的进阶功能。

四、Activiti进阶

4.1 流程定义与流程实例

流程定义 ProcessDefinition 和流程实例 ProcessInstance是Activiti中非常重要的两个概念。他们的关系其实类似于JAVA中类和对象的概念。

流程定义ProcessDefinition是以BPMN文件定义的一个工作流程，是一组工作规范。例如我们之前定义的请假流程。流程实例ProcessInstance则是指一个具体的业务流程。例如某个员工发起一次请假，就会实例化一个请假的流程实例，并且每个不同的流程实例之间是互不影响的。

在后台的表结构中，有很多张表都包含了流程定义ProcessDefinetion和流程实例ProcessInstance的字段。流程定义的字段通常是PROC_DEF_ID，而流程实例的字段通常是PROC_INST_ID。

4.1.1 启动流程实例时，添加Businesskey

在之前的简单案例中，我们启动一个流程实例的关键代码其实就是这一行。

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

当我们去查看下startProcessInstanceByKey这个方法时，会看到这个方法有好几个重载的实现方法，可以传一些不同的参数。其中几个重要的参数包括

- String processDefinitionKey：流程定义的唯一键 不能为空
- String businessKey：每个线程实例上下文中关联的唯一键。这个也是我们这一章节要介绍的重点。
- Map<String,Object> variables：在线程实例中传递的流程变量。这个流程变量可以在整个流程实例中使用，后面会介绍到。
- String tenantId：租户ID，这是Activiti的多租户设计。相当于每个租户可以上来获取一个相对独立的运行环境。

这一章节我们来介绍这个businessKey，业务关键字。这是Activiti提供的一个非常重要的便利，用来将activiti的工作流程与实际业务进行关联。

例如，当我们需要对一个业务订单进行审批时，订单的详细信息并不在activiti的数据当中，但是在审批时确实需要查看这些订单的详细信息。这个时候，就可以用这个businessKey来关联订单ID，这样在业务系统中，就可以通过这个订单ID去关联订单详细信息，审批人员就可以快速拿来进行参考。

进行实际业务整合时，这个businessKey可以根据业务场景，设计成不同的数据格式，比如关键信息逗号拼接，甚至是json都可以，唯一需要注意的是这个字段的数据库长度设计是255，不要超出了数据库的长度限制。

接下来，我们看看如何在流程实例执行过程中获取这个业务关键字：

```java
@Test
    public void queryProcessInstance() {
        // 流程定义key
        String processDefinitionKey = "myLeave";
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
        // 获取RunTimeService
        RuntimeService runtimeService = processEngine.getRuntimeService();
        List<ProcessInstance> list = runtimeService
                .createProcessInstanceQuery()
                .processDefinitionKey(processDefinitionKey)//
                .list();

        for (ProcessInstance processInstance : list) {
            System.out.println("----------------------------");
            System.out.println("流程实例id："
                    + processInstance.getProcessInstanceId());
            System.out.println("所属流程定义id："
                    + processInstance.getProcessDefinitionId());
            System.out.println("是否执行完成：" + processInstance.isEnded());
            System.out.println("是否暂停：" + processInstance.isSuspended());
            System.out.println("当前活动标识：" +
processInstance.getActivityId());
            System.out.println("业务关键
字："+processInstance.getBusinessKey());
        }
    }
```

通过最后面的一行processInstance.getBusinessKey()就能获取到当前流程实例中的业务关键字。在数据库中，act_ru_execution表中的BUSINESS_KEY字段就是用来保存这个业务关键字的。

4.1.2 挂起、激活流程实例

之前我们已经测试了如何删除一个流程，有很多时候，我们只是需要暂时停止一个流程，过一段时间就要恢复。例如月底不接受报销审批流程，年底不接受借贷审批流程，或者非工作日不接受售后报销流程等，这个时候，就可以将流程进行挂起操作。挂起后的流程就不会再继续执行。

在挂起流程时，有两种操作方式。

一种是将整个流程定义Process Definition挂起，这样，这个流程定义下的所有流程实例都将挂起，无法继续执行

```java
/**
```

```
     * 全部流程实例挂起与激活
     */
    @Test
    public void SuspendAllProcessInstance(){
//        获取processEngine
        ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
//        获取repositoryService
        RepositoryService repositoryService = processEngine.getRepositoryService();
//        查询流程定义的对象
        ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery().
                processDefinitionKey("myEvection").
                singleResult();
//        得到当前流程定义的实例是否都为暂停状态
        boolean suspended = processDefinition.isSuspended();
//        流程定义id
        String processDefinitionId = processDefinition.getId();
//        判断是否为暂停
        if(suspended){
//          如果是暂停，可以执行激活操作 ,参数1 ：流程定义id ，参数2：是否激活，参数3：激活时间

 repositoryService.activateProcessDefinitionById(processDefinitionId,
                    true,
                    null
            );
            System.out.println("流程定义："+processDefinitionId+",已激活");
        }else{
//          如果是激活状态，可以暂停，参数1 ：流程定义id ，参数2：是否暂停，参数3：暂停时间

 repositoryService.suspendProcessDefinitionById(processDefinitionId,
                    true,
                    null);
            System.out.println("流程定义："+processDefinitionId+",已挂起");
        }

    }

```

另一种方式是将某一个具体的流程实例挂起。例如对某一个有问题的请假申请进行挂起操作，数据调整完成后再进行激活。继续执行挂起状态的流程将会抛出异常

```
/**
     * 单个流程实例挂起与激活
     */
```

```
    @Test
    public void SuspendSingleProcessInstance(){
//        获取processEngine
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
//        RuntimeService
        RuntimeService runtimeService = processEngine.getRuntimeService();
//        查询流程定义的对象
        ProcessInstance processInstance = runtimeService.
                createProcessInstanceQuery().
                processInstanceId("15001").
                singleResult();
//        得到当前流程定义的实例是否都为暂停状态
        boolean suspended = processInstance.isSuspended();
//        流程定义id
        String processInstanceId = processInstance.getId();
//        判断是否为暂停
        if(suspended){
//          如果是暂停，可以执行激活操作 ,参数：流程定义id
            runtimeService.activateProcessInstanceById(processInstanceId);
            System.out.println("流程定义："+processDefinitionId+",已激活");
        }else{
//            如果是激活状态，可以暂停，参数：流程定义id
            runtimeService.suspendProcessInstanceById( processInstanceId);
            System.out.println("流程定义："+processDefinitionId+",已挂起");
        }
    }

    /**
     * 测试完成个人任务
     */
    @Test
    public void completTask(){
//        获取引擎
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
//        获取操作任务的服务 TaskService
        TaskService taskService = processEngine.getTaskService();
//        完成任务,参数：流程实例id,完成zhangsan的任务
        Task task = taskService.createTaskQuery()
                .processInstanceId("15001")
                .taskAssignee("rose")
                .singleResult();

        System.out.println("流程实例id="+task.getProcessInstanceId());
        System.out.println("任务Id="+task.getId());
        System.out.println("任务负责人="+task.getAssignee());
        System.out.println("任务名称="+task.getName());
        taskService.complete(task.getId());
```

```
51      }
```

4.2 流程变量

流程变量也是Activiti中非常重要的角色。我们之前定义的请假流程并没有用到流程变量，每个步骤都是非常固定的，但是，当我们需要实现一些复杂的业务流程，比如请假3天以内由部门经理审批，3天以上需要增加总经理审批这样的流程时，就需要用到流程变量了。

> 注：这个流程变量和之前介绍的业务关键字其实是有些相似的，都可以携带业务信息。并且也都可以通过activiti的api查询出来。但是通常在使用过程中，应该尽量减少流程变量中的业务信息，这样能够减少业务代码对activiti工作流的代码侵入。

在上一章节介绍到，流程变量的类型是Map<String,Object>。所以，流程变量比业务关键字要强大很多。变量值不仅仅是字符串，也可以是POJO对象。但是当需要将一个POJO对象放入流程变量时，要注意这个对象必须要实现序列化接口serializable。

4.2.1 流程变量的作用域

变量的作用域可以设置为Global和Local两种。

- Global变量

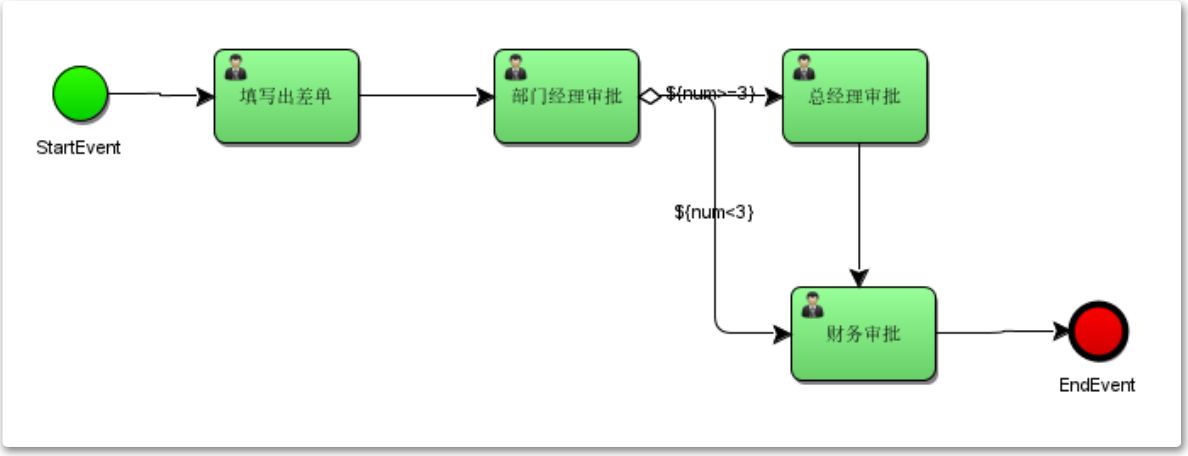
  这个是流程变量的默认作用域，表示是一个完整的流程实例。 Global变量中变量名不能重复。如果设置了相同的变量名，后面设置的值会直接覆盖前面设置的变量值。
- Local 变量

  Local变量的作用域只针对一个任务或一个执行实例的范围，没有流程实例大。Local变量由于作用在不同的任务或不同的执行实例中，所以不同变量的作用域是互不影响的，变量名可以相同。Local变量名也可以和Global变量名相同，不会有影响。

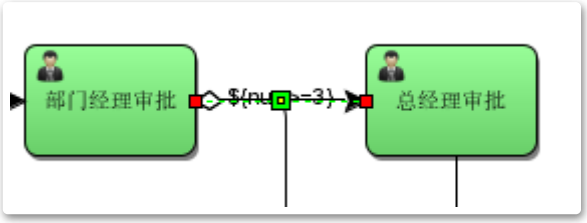
4.2.2 使用流程变量

定义好流程变量后，就可以在整个流程定义中使用这些流程变量了。例如可以在某些任务属性如assignee上使用${assignee}，或者在某些连线上使用${day<3}。

Activiti中可以使用UEL表达式来使用这些流程变量。UEL表达式可以直接获取一个变量的值，可以计算一个Boolean结果的表达式，还可以直接使用某些对象的属性。例如对于之前创建的请假流程，如果要实现3天以内部门经理审核，3天以上增加总经理审核，可以做如下调整：



1）、出差天数大于等于3连线条件

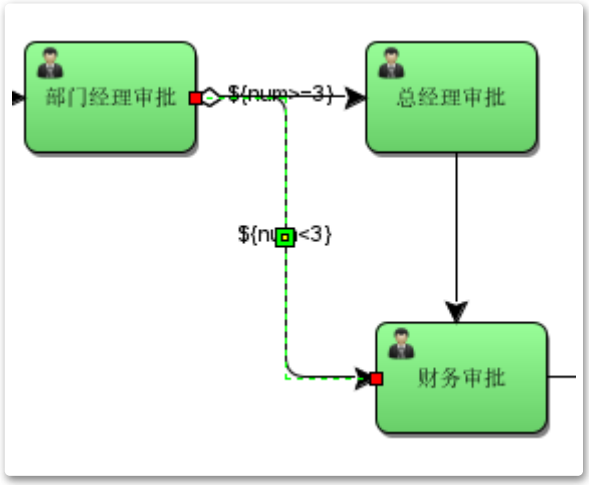


property	value
Name	
Condition	${num>=3}
Documentation	
Execution Listeners	

也可以使用对象参数命名，如evection.num：

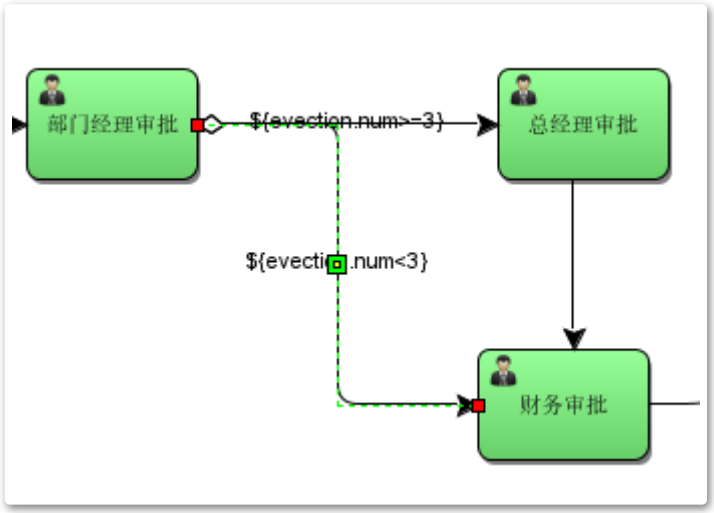
property	value
Name	
Condition	${evection.num>=3}
Documentation	
Execution Listeners	

2）、出差天数小于3连线条件



property	value
Name	
Condition	${num<3}
Documentation	
Execution Listeners	

也可以使用对象参数命名，如：



property	value
Name	
Condition	${evection.num<3}
Documentation	
Execution Listeners	

4.2.3 设置Global流程变量

在流程定义中使用到了流程变量，就需要在后台JAVA代码中设置对应的流程变量。实际上在流程执行的很多过程中都可以设计自流程变量。

1） 启动流程时设置变量

在启动流程实例时设置流程变量，这时流程变量的作用域是整个流程实例。相当于是Global作用域。核心代码：

```
ProcessInstance processInstance =
runtimeService.startProcessInstanceByKey(key, map);
```

2) 任务办理时设置变量

在完成任务时设置流程变量，该流程变量只有在该任务完成后其它结点才可使用该变量，它的作用域是整个流程实例，如果设置的流程变量的key在流程实例中已存在相同的名字则后设置的变量替换前边设置的变量。核心代码：

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

> 注意：这种方式设置流程变量，如果当前执行的任务ID不存在，则会抛出异常，流程变量也会设置失败。

3） 通过当前流程实例设置

通过流程实例id设置全局变量，该流程实例必须未执行完成。

```
  @Test
    public void setGlobalVariableByExecutionId(){
//    当前流程实例执行 id，通常设置为当前执行的流程实例
        String executionId="2601";
//     获取processEngine
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
//        获取RuntimeService
        RuntimeService runtimeService = processEngine.getRuntimeService();
//        创建出差pojo对象
        Evection evection = new Evection();
//        设置天数
        evection.setNum(3d);
//      通过流程实例 id设置流程变量
        runtimeService.setVariable(executionId, "myLeave", evection);
//      一次设置多个值
//      runtimeService.setVariables(executionId, variables)
    }
```

> 注意：ececutionId必须是当前未完成的流程实例的执行ID。通常此ID设置流程实例的ID。流程变量设计完成后，也可以通过runtimeService.getVariable()获取流程变量

4） 通过当前任务设置

```
@Test
    public void setGlobalVariableByTaskId(){

        //当前待办任务id
        String taskId="1404";
//      获取processEngine
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
        TaskService taskService = processEngine.getTaskService();
        Evection evection = new Evection();
        evection.setNum(3);
        //通过任务设置流程变量
        taskService.setVariable(taskId, "evection", evection);
        //一次设置多个值
        //taskService.setVariables(taskId, variables)
    }
```

> 注： 任务id必须是当前待办任务id，act_ru_task中存在。如果该任务已结束，会报错。也可以通过taskService.getVariable()获取流程变量。

注意事项

1、 如果UEL表达式中流程变量名不存在则报错。

2、 如果UEL表达式中流程变量值为空NULL，流程不按UEL表达式去执行，而流程结束 。

3、 如果UEL表达式都不符合条件，流程结束

4、 如果连线不设置条件，会走flow序号小的那条线

5、设置流程变量会在当前执行流程变量表act_ru_variable中插入记录，同时也会在历史流量变量表act_hi_varinst中也插入记录。

4.2.4 设置Local流程变量

local流程变量同样可以有多个设置的地方。

1） 任务办理时设置

任务办理时设置local流程变量，当前运行的流程实例只能在该任务结束前使用，任务结束该变量无法在当前流程实例使用，可以通过查询历史任务查询。关键代码：

```
//  设置local变量，作用域为该任务
   taskService.setVariablesLocal(taskId, variables);
//  完成任务
   taskService.complete(taskId);
```

2) 通过当前任务设置

```
@Test
public void setLocalVariableByTaskId(){
//   当前待办任务id
    String taskId="1404";
//  获取processEngine
    ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
    TaskService taskService = processEngine.getTaskService();
    Evection evection = new Evection ();
    evection.setNum(3d);
//  通过任务设置流程变量
    taskService.setVariableLocal(taskId, "evection", evection);
//  一次设置多个值
    //taskService.setVariablesLocal(taskId, variables)
}
```

> 注： 任务ID必须是当前待办任务id，要在act_ru_task中存在

4.3 网关

网关是用来控制流程流向的重要组件，通常都会要结合流程变量来使用。

4.3.1 排他网关ExclusiveGateway

排他网关，用来在流程中实现决策。 当流程执行到这个网关，所有分支都会判断条件是否为true，如果为true则执行该分支，

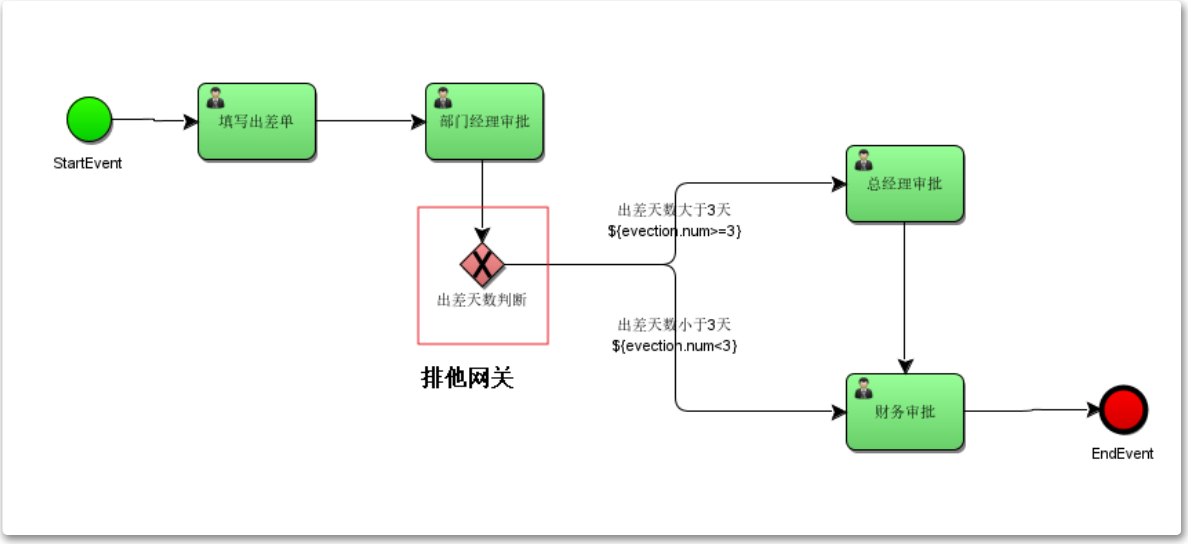
> **注意**：排他网关只会选择一个为true的分支执行。如果有两个分支条件都为true，排他网关会选择id值较小的一条分支去执行。

为什么要用排他网关?

不用排他网关也可以实现分支，如：在连线的condition条件上设置分支条件。

在连线设置condition条件的缺点：如果条件都不满足，流程就结束了(是异常结束)。

如果 使用排他网关决定分支的走向，如下：



如果从网关出去的线所有条件都不满足则系统抛出异常。

```
org.activiti.engine.ActivitiException: No outgoing sequence flow of the exclusive gateway 'exclusivegateway1' could be selected for continuing the process
```

4.3.2 并行网关ParallelGateway

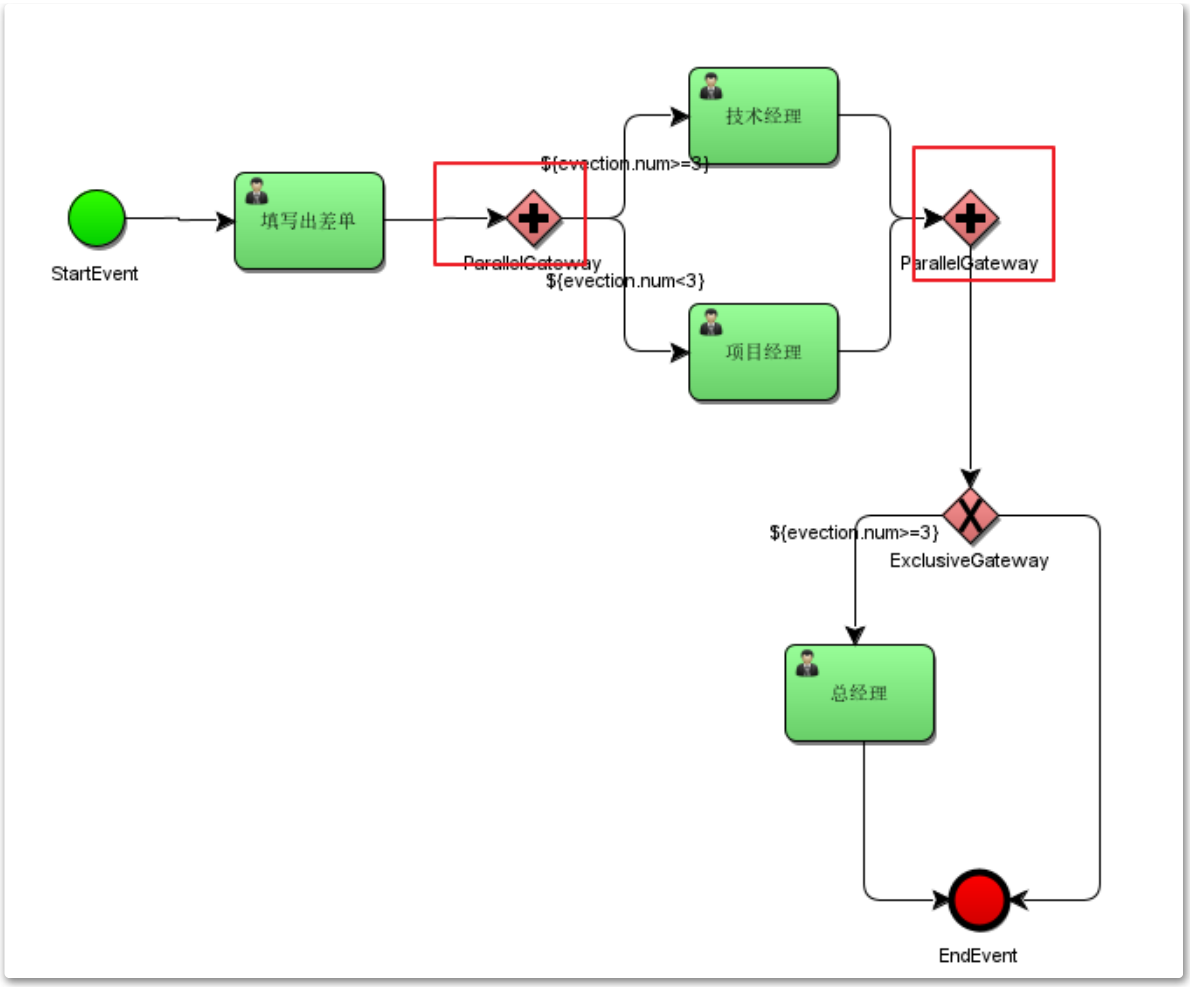
并行网关允许将流程分成多条分支，也可以把多条分支汇聚到一起，并行网关的功能是基于进入和外出顺序流的：

fork分支：并行后的所有外出顺序流，为每个顺序流都创建一个并发分支。

join汇聚： 所有到达并行网关，在此等待的进入分支， 直到所有进入顺序流的分支都到达以后， 流程就会通过汇聚网关。

> 注意，如果同一个并行网关有多个进入和多个外出顺序流， 它就同时具有分支和汇聚功能。 这时，网关会先汇聚所有进入的顺序流，然后再切分成多个并行分支。

与其他网关的主要区别是，并行网关不会解析条件。 即使顺序流中定义了条件，也会被忽略。



> 说明：此时会要求技术经理和项目经理都进行审批。而连线上的条件会被忽略。
>
> 技术经理和项目经理是两个execution分支，在act_ru_execution表有两条记录分别是技术经理和项目经理，act_ru_execution还有一条记录表示该流程实例。
>
> 待技术经理和项目经理任务全部完成，在汇聚点汇聚，通过parallelGateway并行网关。
>
> 并行网关在业务应用中常用于会签任务，会签任务即多个参与者共同办理的任务。

4.3.3 包含网关InclusiveGateway

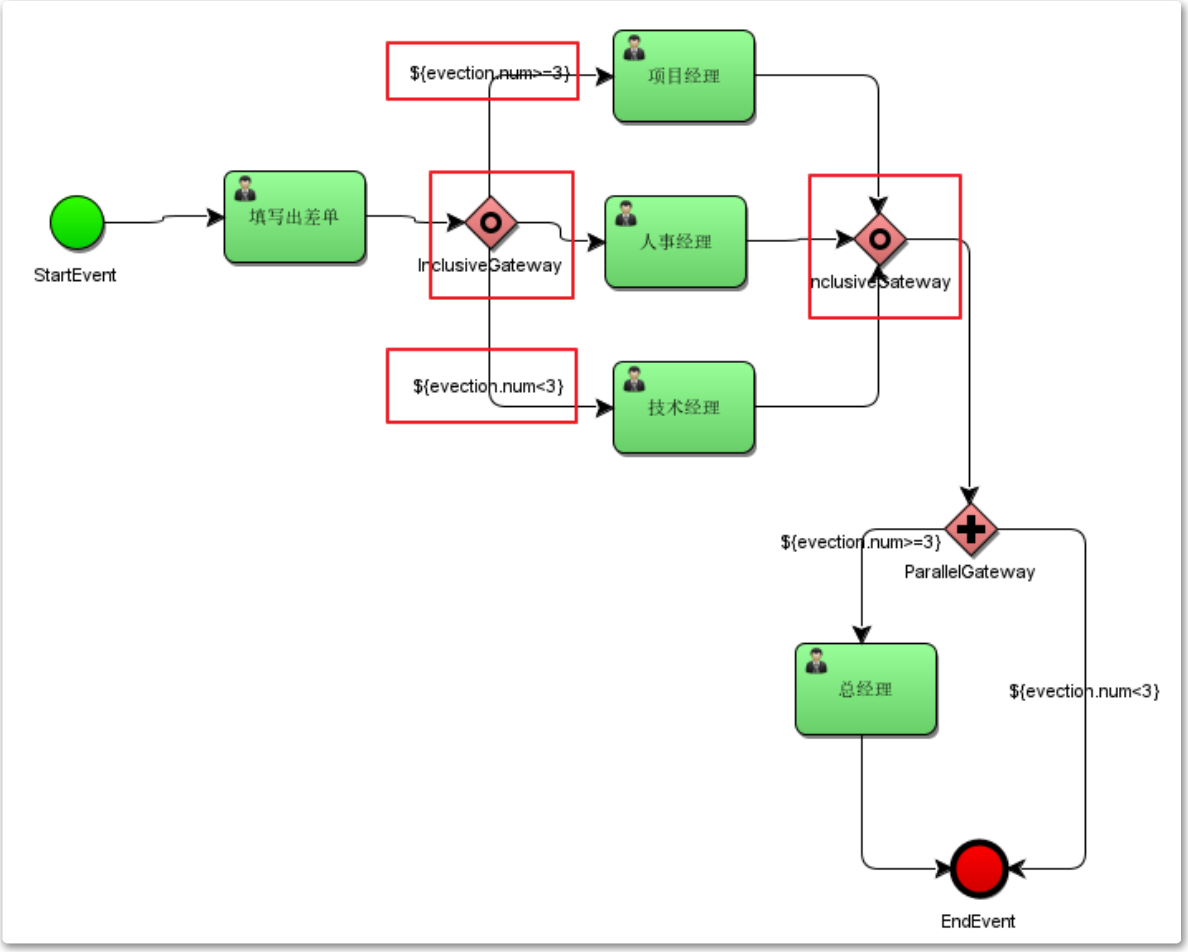
包含网关可以看做是排他网关和并行网关的结合体。

和排他网关一样，你可以在外出顺序流上定义条件，包含网关会解析它们。 但是主要的区别是包含网关可以选择多于一条顺序流，这和并行网关一样。

包含网关的功能是基于进入和外出顺序流的：

分支： 所有外出顺序流的条件都会被解析，结果为true的顺序流会以并行方式继续执行， 会为每个顺序流创建一个分支。

汇聚： 所有并行分支到达包含网关，会进入等待状态， 直到每个包含流程token的进入顺序流的分支都到达。 这是与并行网关的最大不同。换句话说，包含网关只会等待被选中执行了的进入顺序流。 在汇聚之后，流程会穿过包含网关继续执行。



> 说明：这里当请假天数超过3天，需要项目经理和人事经理一起审批。而请假天数不超过3填，需要技术经理和人事经理一起审批。
>
> 所有符合条件的分支也会在后面进行汇聚。

4.3.4 事件网关EventGateway

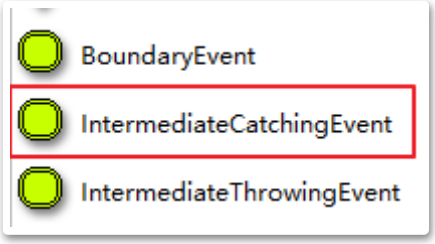
事件网关允许根据事件判断流向。网关的每个外出顺序流都要连接到一个中间捕获事件。 当流程到达一个基于事件网关，网关会进入等待状态：会暂停执行。与此同时，会为每个外出顺序流创建相对的事件订阅。

事件网关的外出顺序流和普通顺序流不同，这些顺序流不会真的"执行"， 相反它们让流程引擎去决定执行到事件网关的流程需要订阅哪些事件。 要考虑以下条件：

1. 事件网关必须有两条或以上外出顺序流；

2. 事件网关后，只能使用intermediateCatchEvent类型（activiti不支持基于事件网关后连接ReceiveTask）
3. 连接到事件网关的中间捕获事件必须只有一个入口顺序流。

与事件网关配合使用的intermediateCatchEvent:

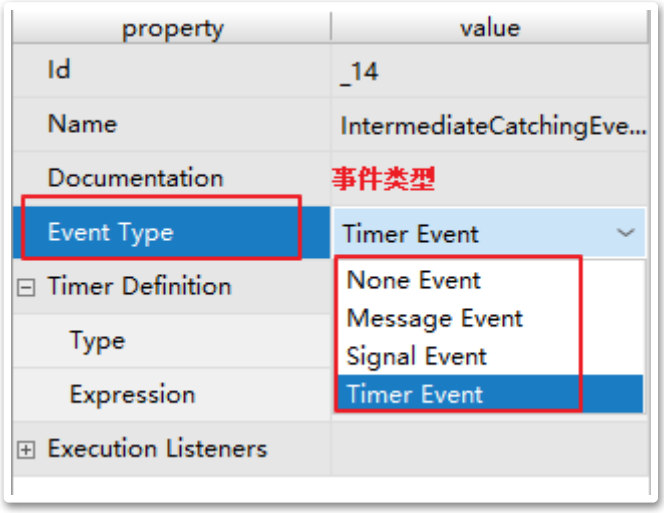


这个事件支持多种事件类型：

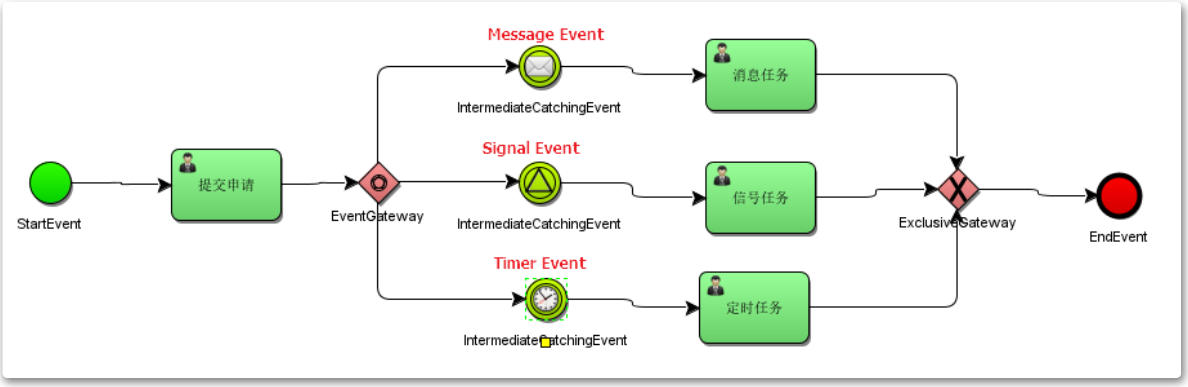
Message Event：消息事件

Singal Event： 信号事件

Timer Event： 定时事件



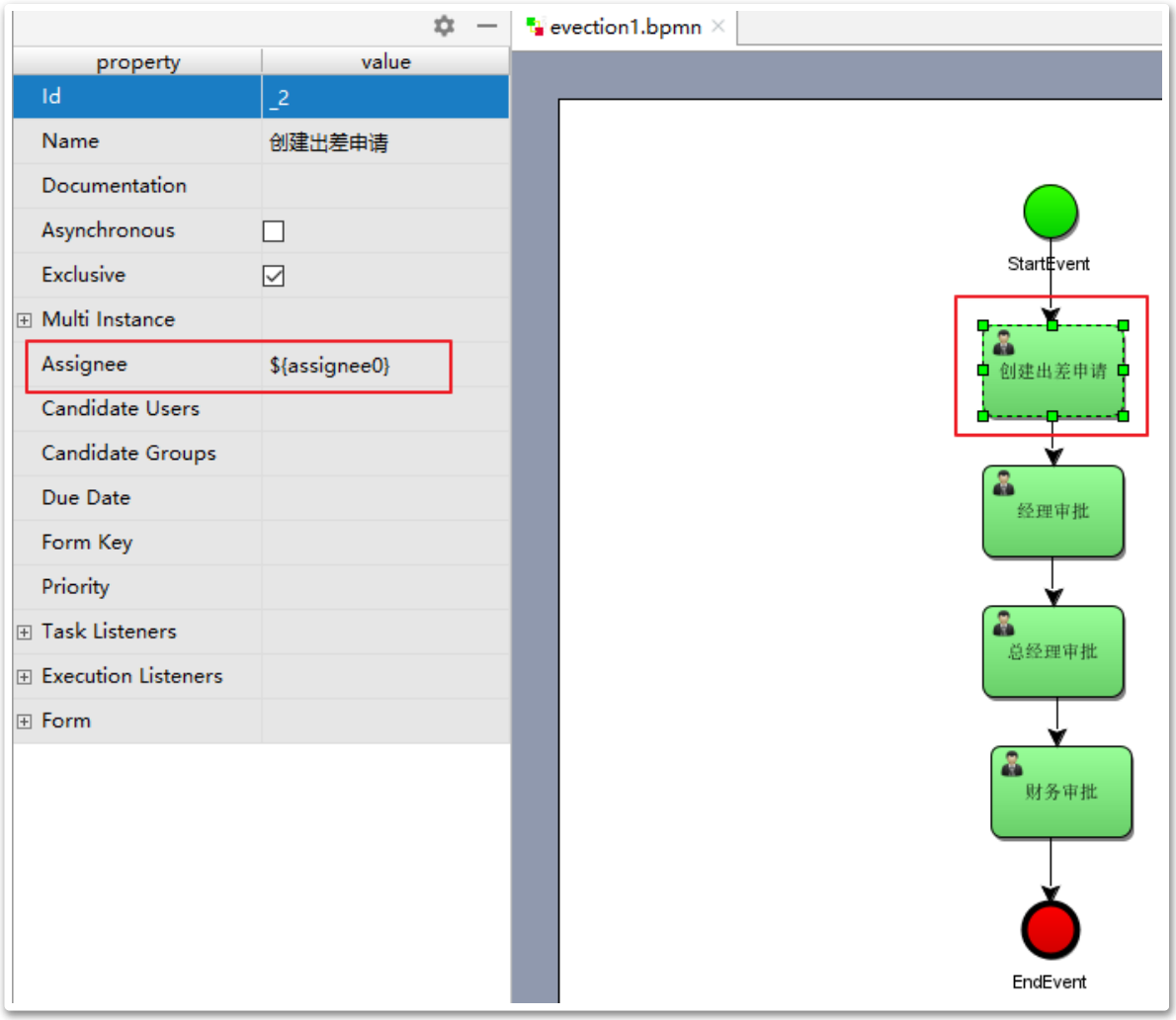
使用事件网关定义流程：



4.4 个人任务管理

4.4.1 分配任务负责人

在之前的简单示例中，我们已经可以通过配置Assignee属性来指定任务的负责人。但是我们之前的示例中，是简单的配置为worker、manager、finacer等这样的固定的任务人。但是在实际工作中，往往不会是这样固定的人。可能是对应某个职位或者某个角色的系统用户。这时，这种固定分配的方式就非常不灵活了。这时，就可以使用UEL表达式配合流程变量来灵活指定。例如这样：



这个assignee0就对应activiti中的一个流程变量。

而如果配置成${user.assignee}表示通过user的getter方法获取属性值

也可以配置成使用具体的方法${user.getUserId()}。

甚至可以结合Spring容易来使用。例如
${ldapService.findManagerForEmployee(emp)}
ldapService 是 spring 容器的一个 bean，findManagerForEmployee 是该 bean 的一个方法，emp 是 activiti
流程变量， emp 作为参数传到 ldapService.findManagerForEmployee 方法中。

配置了这个流程后，可以配合流程变量使用。例如：

```
/**
     * 设置流程负责人
     */
    @Test
    public void assigneeUEL(){
//        获取流程引擎
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
//          获取 RuntimeService
        RuntimeService runtimeService = processEngine.getRuntimeService();
//          设置assignee的取值，用户可以在界面上设置流程的执行
        Map<String,Object> assigneeMap = new HashMap<>();
        assigneeMap.put("assignee0","张三");
        assigneeMap.put("assignee1","李经理");
        assigneeMap.put("assignee2","王总经理");
        assigneeMap.put("assignee3","赵财务");
//          启动流程实例，同时还要设置流程定义的assignee的值

 runtimeService.startProcessInstanceByKey("myEvection1",assigneeMap);
//         输出
        System.out.println(processEngine.getName());
    }
```

执行完成后，可以在act_ru_variable表中看到刚才map中的数据

ID_	REV_	TYPE_	NAME_	EXECUTION_ID_	PROC_INST_ID_	TASK_ID_	BYTEARRAY_ID_	DOUBLE_	LONG_	TEXT_
22503	1	string	assignee0	22501	22501	(NULL)	(NULL)	(NULL)	(NULL)	zhangsan
22505	1	string	assignee1	22501	22501	(NULL)	(NULL)	(NULL)	(NULL)	李经理
22504	1	string	assignee2	22501	22501	(NULL)	(NULL)	(NULL)	(NULL)	王总经理
22502	1	string	assignee3	22501	22501	(NULL)	(NULL)	(NULL)	(NULL)	赵财务
(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)

注意事项

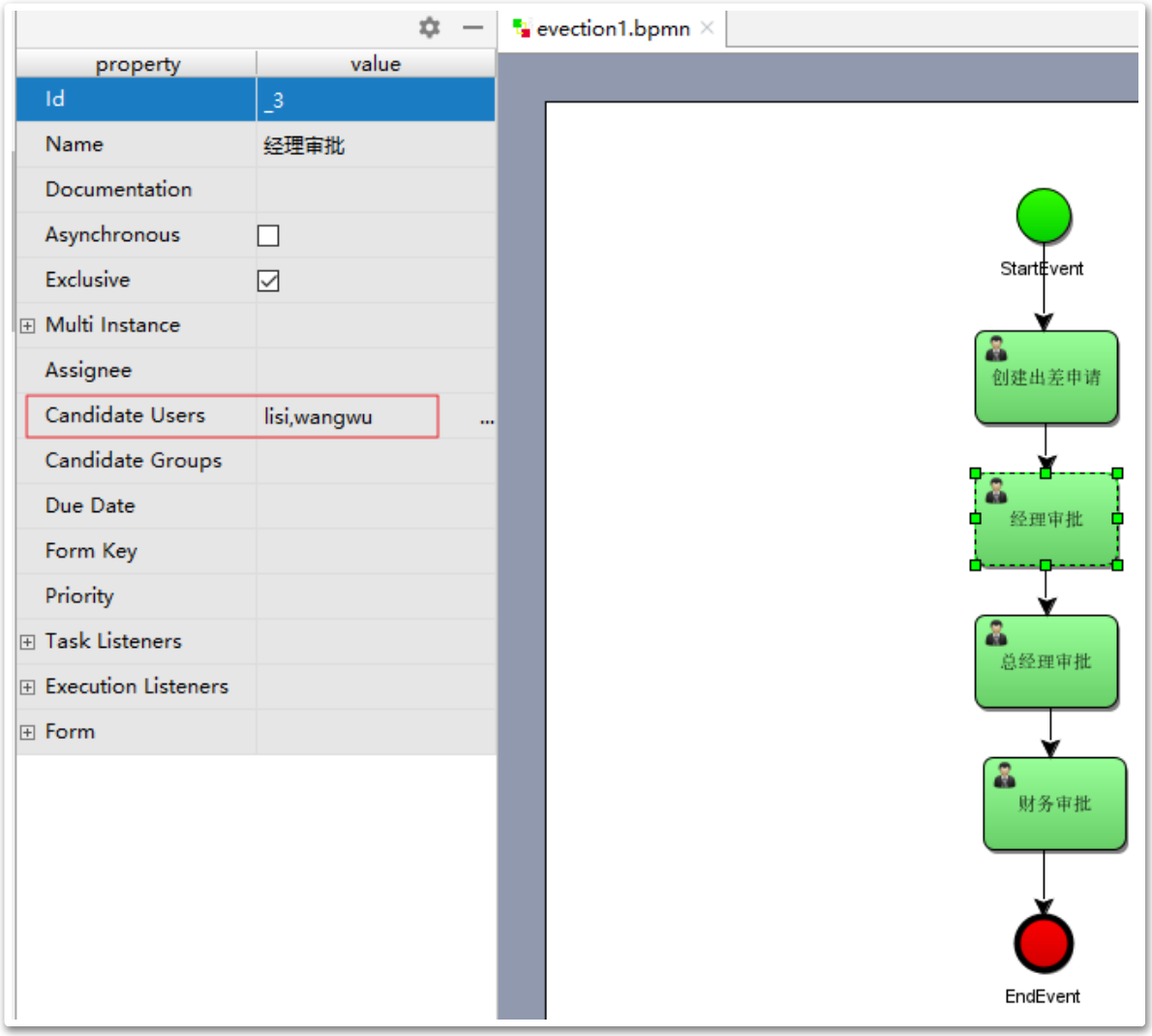
由于使用表达式分配，必须保证在任务执行过程中表达式执行成功。否则会抛出activiti异常。

4.5 组任务分配

4.5.1 设置多个候选责任人

之前我们已经可以给任务灵活的设定负责人。但是在日常工作中，还有一类非常常见的需求无法支持。例如某个订单合同，需要找部门经理级别的负责人签字。而公司中有多个部门经理，业务上只需要找其中任意一个人完成审批就可以了。 这种场景下，我们就无法通过设置流程变量的方式来设置负责人。这时，就需要用到Activiti提供的另一个利器-任务候选人Candidate Users。

这时，可以给任务设置多个候选人 candidate-uses，多个候选人之间用逗号隔开。



在BPMN文件中可以看到

```
<userTask activiti:candidateUsers="lisi,wangwu" activiti:exclusive="true"
id="_3" name="经理审批"/>
```

这样就给这个任务设置了一组候选人。

4.5.2 组任务办理流程

给任务分配了候选人后，后续就需要这些候选人主动认领自己的业务，然后进行处理。

1、查询组任务

指定候选人，查询该候选人当前的待办任务。候选人不能立即办理任务，需要先认领业务。

```
@Test
```

```
    public void findGroupTaskList() {
        // 流程定义key
        String processDefinitionKey = "evection3";
        // 任务候选人
        String candidateUser = "lisi";
         //  获取processEngine
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
        // 创建TaskService
        TaskService taskService = processEngine.getTaskService();
        //查询组任务
        List<Task> list = taskService.createTaskQuery()
                .processDefinitionKey(processDefinitionKey)
                .taskCandidateUser(candidateUser)//根据候选人查询
                .list();
        for (Task task : list) {
            System.out.println("----------------------------");
            System.out.println("流程实例id: " + task.getProcessInstanceId());
            System.out.println("任务id: " + task.getId());
            System.out.println("任务负责人: " + task.getAssignee());
            System.out.println("任务名称: " + task.getName());
        }
    }
```

2、拾取(claim)任务

该组任务的所有候选人都能拾取。将候选人的组任务，变成个人任务。原来候选人就变成了该任务的负责人。如果拾取后不想办理该任务，负责人也可以将已经拾取的个人任务归还到组里边，将个人任务变成了组任务。

候选人认领组任务

```
@Test
    public void claimTask(){
          //  获取processEngine
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
        TaskService taskService = processEngine.getTaskService();
        //要拾取的任务id
        String taskId = "6302";
        //任务候选人id
        String userId = "lisi";
        //拾取任务
        //即使该用户不是候选人也能拾取(建议拾取时校验是否有资格)
        //校验该用户有没有拾取任务的资格
```

```
        Task task = taskService.createTaskQuery()
                .taskId(taskId)
                .taskCandidateUser(userId)//根据候选人查询
                .singleResult();
        if(task!=null){
          //拾取任务
            taskService.claim(taskId, userId);
            System.out.println("任务拾取成功");
        }
    }
```

> 注：Activiti中，即使该用户不是候选人，也能认领热舞。所以建议要在业务中自行校验是否有资格。
>
> 任务被认领后，该任务就有了具体的负责人。其他候选人将查询不到该任务。

3、查询个人任务

与原有的流程相同

4、办理个人任务

与原有的流程相同

5、归还组任务

如果个人不想办理该组任务，可以在认领之后归还组任务，归还后该用户就不再是该任务的负责人了。

```
/*
*归还组任务，由个人任务变为组任务，还可以进行任务交接
*/
@Test
public void setAssigneeToGroupTask() {
    //  获取processEngine
        ProcessEngine processEngine =
ProcessEngines.getDefaultProcessEngine();
   // 查询任务使用TaskService
   TaskService taskService = processEngine.getTaskService();
   // 当前待办任务
   String taskId = "6004";
   // 任务负责人
   String userId = "zhangsan2";
    // 校验userId是否是taskId的负责人，如果是负责人才可以归还组任务
   Task task = taskService
        .createTaskQuery()
```

```
        .taskId(taskId)
        .taskAssignee(userId)
        .singleResult();
    if (task != null) {
        // 如果设置为null，归还组任务,该 任务没有负责人
        taskService.setAssignee(taskId, null);
    }
}
```

> 注：从这个代码中可以看到，实际上是允许直接给任务设定责任人的，即使被委托用户不是候选人，也可以直接指定。
>
> 这就是任务交接的流程。

数据库表操作

查询当前任务执行表

```
SELECT * FROM act_ru_task
```

任务执行表，记录当前执行的任务，由于该任务当前是组任务，所有assignee为空，当拾取任务后该字段就是拾取用户的id

查询任务参与者

```
SELECT * FROM act_ru_identitylink
```

任务参与者，记录当前参考任务用户或组，当前任务如果设置了候选人，会向该表插入候选人记录，有几个候选就插入几个

与act_ru_identitylink对应的还有一张历史表act_hi_identitylink，向act_ru_identitylink插入记录的同时也会向历史表插入记录。任务完成

五、Activiti与Spring整合

Activiti与Spring整合的基本思想是将Activiti最为核心的ProcessEngine类交由Spring容器进行管理。

核心的pom依赖：

```
<groupId>org.activiti</groupId>
        <artifactId>activiti-spring</artifactId>
        <version>7.0.0.Beta1</version>
    </dependency>
    <dependency>
```

然后在classpath下创建activiti-spring.xml文件

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/tx
         http://www.springframework.org/schema/tx/spring-tx.xsd
         http://www.springframework.org/schema/aop
         http://www.springframework.org/schema/aop/spring-aop.xsd">
    <!-- 数据源 -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/activiti"/>
        <property name="username" value="root"/>
        <property name="password" value="123456"/>
        <property name="maxActive" value="3"/>
        <property name="maxIdle" value="1"/>
    </bean>
    <!-- 工作流引擎配置bean -->
    <bean id="processEngineConfiguration"
class="org.activiti.spring.SpringProcessEngineConfiguration">
        <!-- 数据源 -->
        <property name="dataSource" ref="dataSource"/>
        <!-- 使用spring事务管理器 -->
        <property name="transactionManager" ref="transactionManager"/>
        <!-- 数据库策略 -->
        <property name="databaseSchemaUpdate" value="drop-create"/>
    </bean>
    <!-- 流程引擎 -->
    <bean id="processEngine"
class="org.activiti.spring.ProcessEngineFactoryBean">
        <property name="processEngineConfiguration"
ref="processEngineConfiguration"/>
    </bean>
    <!-- 资源服务service -->
    <bean id="repositoryService" factory-bean="processEngine" factory-
method="getRepositoryService"/>
    <!-- 流程运行service -->
```

```
    <bean id="runtimeService" factory-bean="processEngine"  factory-method="getRuntimeService"/>
    <!-- 任务管理service -->
    <bean id="taskService" factory-bean="processEngine" factory-method="getTaskService"/>
    <!-- 历史管理service -->
    <bean id="historyService" factory-bean="processEngine" factory-method="getHistoryService"/>
    <!-- 事务管理器 -->
    <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <!-- 通知 -->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <!-- 传播行为 -->
            <tx:method name="save*" propagation="REQUIRED"/>
            <tx:method name="insert*" propagation="REQUIRED"/>
            <tx:method name="delete*" propagation="REQUIRED"/>
            <tx:method name="update*" propagation="REQUIRED"/>
            <tx:method name="find*" propagation="SUPPORTS" read-only="true"/>
            <tx:method name="get*" propagation="SUPPORTS" read-only="true"/>
        </tx:attributes>
    </tx:advice>
</beans>
```

然后就可以从Spring容器中直接引用对应的service，进行具体的业务操作了。

```
/**
   测试activiti与spring整合是否成功
**/
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:activiti-spring.xml")
 public class ActivitiTest {
     @Autowired
     private RepositoryService repositoryService;

     @Test
     public void test01(){
         System.out.println("部署对象:"+repositoryService);
     }
 }
```

下面我们一起来分析Activiti与Spring整合加载的过程。

1、加载activiti-spring.xml配置文件

2、加载SpringProcessEngineConfiguration对象，这个对象它需要依赖注入dataSource对象和transactionManager对象。

3、加载ProcessEngineFactoryBean工厂来创建ProcessEngine对象，而ProcessEngineFactoryBean工厂又需要依赖注入processEngineConfiguration对象。

4、processEngine对象来负责创建我们的Service对象，从而简化Activiti的开发过程。

六、Activiti7与SpringBoot整合开发

Activiti7正式版发布后，就已经支持与SpringBoot2.X版本完全整合开发了。

1、引入maven依赖

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.0.RELEASE</version>
</parent>
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
    <dependency>
        <groupId>org.activiti</groupId>
```

```
        <artifactId>activiti-spring-boot-starter</artifactId>
        <version>7.0.0.Beta2</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.29</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

核心的依赖就是org.activiti:activiti-spring-boot-starter

2、创建配置文件application.yml

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/activiti?useUnicode=true&characterEncoding=utf8&serverTimezone=GMT
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
  activiti:
    database-schema-update: true
    #检测历史表是否存在 activiti7默认没有开启数据库历史记录 启动数据库历史记录
    db-history-used: true
    #记录历史等级 可配置的历史级别有none, activity, audit, full
    #none：不保存任何的历史数据，因此，在流程执行过程中，这是最高效的。
    #activity：级别高于none，保存流程实例与流程行为，其他数据不保存。
    #audit：除activity级别会保存的数据外，还会保存全部的流程任务及其属性。audit为history的默认值。
    #full：保存历史数据的最高级别，除了会保存audit级别的数据外，还会保存其他全部流程相关的细节数据，包括一些流程参数等。
    history-level: full
    #校验流程文件，默认校验resources下的processes文件夹里的流程文件
    check-process-definitions: false
```

3、编写启动类

```java
@SpringBootApplication
public class ActApplication {
    public static void main(String[] args) {
        SpringApplication.run(ActApplication.class,args);
    }
}
```

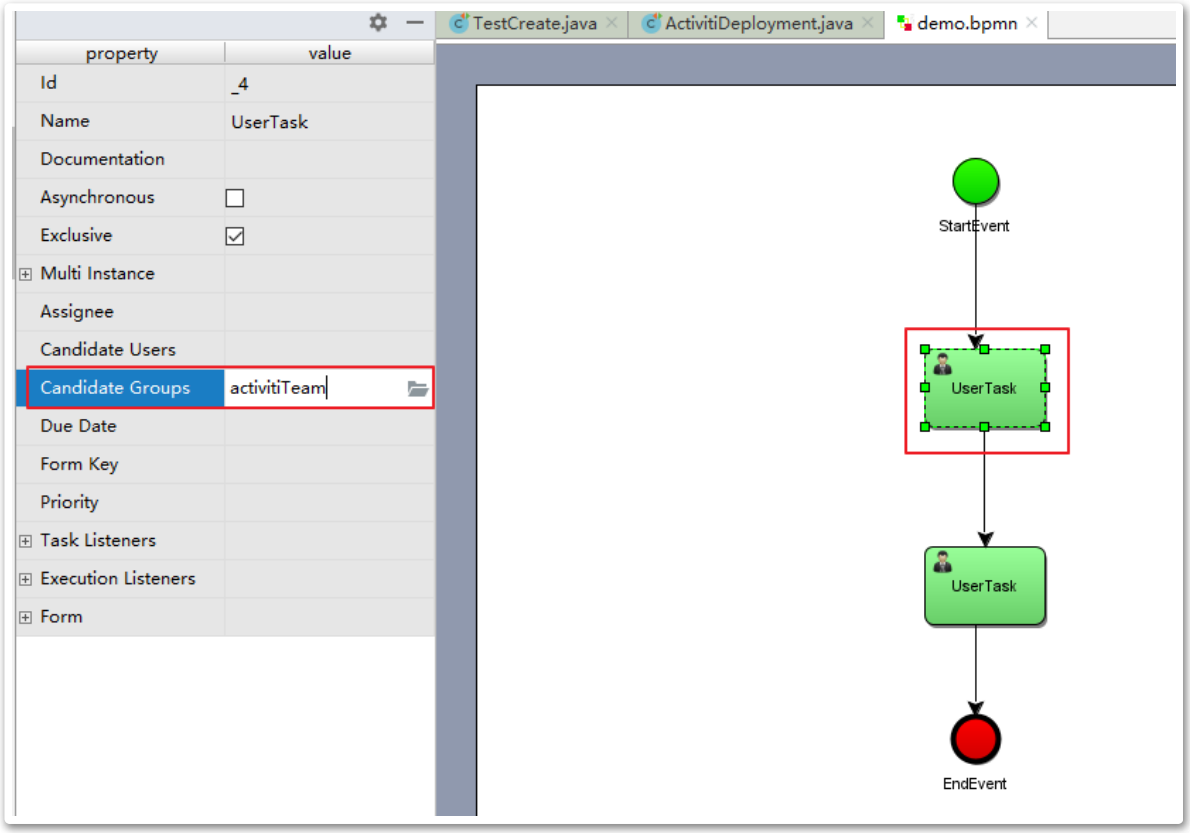
4、创建BPMN文件

Activiti7可以自动部署流程，前提是在resources目录下，创建一个新的目录processes，用来放置bpmn文件。

创建一个简单的Bpmn流程文件，并设置任务的用户组Candidate Groups。

Candidate Groups中的内容与上面DemoApplicationConfiguration类中出现的用户组名称要保持一致，可以填写：activitiTeam 或者 otherTeam。

这样填写的好处：当不确定到底由谁来负责当前任务的时候，只要是Groups内的用户都可以拾取这个任务



并且，将我们之前创建的myLeave.bpmn也放到这个目录下。

5、使用junit方式测试

使用SpringBoot集成Activiti后，每次启动项目时，会自动部署之前放的bpmn文件，并且，会在Spring容器中自动注入Activiti的几个核心Service。

> 这时关注下启动日志，并且到MySQL当中查看下关键的数据表。

这样，我们可以快速的将之前的流程移植过来。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpingBootActivitiDemo {

    @Autowired
    private RuntimeService runtimeService;

    @Autowired
    private TaskService taskService;

    @Autowired
    private RepositoryService repositoryService;
    @Autowired
    private HistoryService historyService;

    /**
     * 启动流程实例
     */
    @Test
    public void testStartProcess(){
        ProcessInstance processInstance = runtimeService
                .startProcessInstanceByKey("myLeave");
//          输出内容
        System.out.println("流程定义id: " +
processInstance.getProcessDefinitionId());
        System.out.println("流程实例id: " + processInstance.getId());
        System.out.println("当前活动Id: " +
processInstance.getActivityId());
    }

    /**
     * 查询当前个人待执行的任务
     */
    @Test
    public void testFindPersonalTaskList() {
//          任务负责人
        String assignee = "worker";
//          根据流程key 和 任务负责人 查询任务
```

```
        List<Task> list = taskService.createTaskQuery()
                .processDefinitionKey("myLeave") //流程Key
                .taskAssignee(assignee)//只查询该任务负责人的任务
                .list();

        for (Task task : list) {
            System.out.println("流程实例id: " +
task.getProcessInstanceId());
            System.out.println("任务id: " + task.getId());
            System.out.println("任务负责人: " + task.getAssignee());
            System.out.println("任务名称: " + task.getName());

        }
    }

    // 完成任务
    @Test
    public void completTask(){
//         根据流程key 和 任务的负责人 查询任务
//         返回一个任务对象
        Task task = taskService.createTaskQuery()
                .processDefinitionKey("myLeave") //流程Key
                .taskAssignee("worker")  //要查询的负责人
                .singleResult();
//         完成任务,参数: 任务id
        taskService.complete(task.getId());
    }

    /**
     * 查询流程定义
     */
    @Test
    public void queryProcessDefinition(){
//         得到ProcessDefinitionQuery 对象
        ProcessDefinitionQuery processDefinitionQuery =
repositoryService.createProcessDefinitionQuery();
//           查询出当前所有的流程定义
//           条件: processDefinitionKey =evection
//           orderByProcessDefinitionVersion 按照版本排序
//         desc倒叙
//         list 返回集合
        List<ProcessDefinition> definitionList =
processDefinitionQuery.processDefinitionKey("myLeave")
                .orderByProcessDefinitionVersion()
                .desc()
                .list();
//      输出流程定义信息
        for (ProcessDefinition processDefinition : definitionList) {
            System.out.println("流程定义 id="+processDefinition.getId());
```

```
            System.out.println("流程定义
name="+processDefinition.getName());
            System.out.println("流程定义 key="+processDefinition.getKey());
            System.out.println("流程定义
Version="+processDefinition.getVersion());
            System.out.println("流程部署ID
="+processDefinition.getDeploymentId());
        }
    }

    /**
     * 查询流程实例
     */
    @Test
    public void queryProcessInstance() {
        // 流程定义key
        String processDefinitionKey = "myLeave";
        List<ProcessInstance> list = runtimeService
                .createProcessInstanceQuery()
                .processDefinitionKey(processDefinitionKey)//
                .list();

        for (ProcessInstance processInstance : list) {
            System.out.println("----------------------------");
            System.out.println("流程实例id："
                    + processInstance.getProcessInstanceId());
            System.out.println("所属流程定义id："
                    + processInstance.getProcessDefinitionId());
            System.out.println("是否执行完成：" +
processInstance.isEnded());
            System.out.println("是否暂停：" +
processInstance.isSuspended());
            System.out.println("当前活动标识：" +
processInstance.getActivityId());
            System.out.println("业务关键
字："+processInstance.getBusinessKey());
        }
    }

    @Test
    public void deleteDeployment() {
        // 流程部署id
        String deploymentId = "1";

        //删除流程定义，如果该流程定义已有流程实例启动则删除时出错
        repositoryService.deleteDeployment(deploymentId);
        //设置true 级联删除流程定义，即使该流程有流程实例启动也可以删除，设置为false
非级别删除方式，如果流程
//        repositoryService.deleteDeployment(deploymentId, true);
```

```
124     }
125
126     @Test
127     public void  queryBpmnFile() throws IOException {
128 //        3、得到查询器：ProcessDefinitionQuery，设置查询条件,得到想要的流程定义
129         ProcessDefinition processDefinition =
repositoryService.createProcessDefinitionQuery()
130                 .processDefinitionKey("myLeave")
131                 .singleResult();
132 //        4、通过流程定义信息，得到部署ID
133         String deploymentId = processDefinition.getDeploymentId();
134 //        5、通过repositoryService的方法，实现读取图片信息和bpmn信息
135 //        png图片的流
136         InputStream pngInput =
repositoryService.getResourceAsStream(deploymentId,
processDefinition.getDiagramResourceName());
137 //        bpmn文件的流
138         InputStream bpmnInput =
repositoryService.getResourceAsStream(deploymentId,
processDefinition.getResourceName());
139 //        6、构造OutputStream流
140         File file_png = new File("d:/myLeave.png");
141         File file_bpmn = new File("d:/myLeave.bpmn");
142         FileOutputStream bpmnOut = new FileOutputStream(file_bpmn);
143         FileOutputStream pngOut = new FileOutputStream(file_png);
144 //        7、输入流，输出流的转换
145         IOUtils.copy(pngInput,pngOut);
146         IOUtils.copy(bpmnInput,bpmnOut);
147 //        8、关闭流
148         pngOut.close();
149         bpmnOut.close();
150         pngInput.close();
151         bpmnInput.close();
152     }
153
154     /**
155      * 查看历史信息
156      */
157     @Test
158     public void findHistoryInfo(){
159 //        获取 actinst表的查询对象
160         HistoricActivityInstanceQuery instanceQuery =
historyService.createHistoricActivityInstanceQuery();
161 //        查询 actinst表，条件：根据 InstanceId 查询，查询一个流程的所有历史信息
162         instanceQuery.processInstanceId("25001");
163 //        查询 actinst表，条件：根据 DefinitionId 查询，查询一种流程的所有历史信息
164 //        instanceQuery.processDefinitionId("myLeave:1:22504");
165 //        增加排序操作,orderByHistoricActivityInstanceStartTime 根据开始时间排
序 asc 升序
```

```
        instanceQuery.orderByHistoricActivityInstanceStartTime().asc();
//          查询所有内容
        List<HistoricActivityInstance> activityInstanceList =
instanceQuery.list();
//          输出
        for (HistoricActivityInstance hi : activityInstanceList) {
            System.out.println(hi.getActivityId());
            System.out.println(hi.getActivityName());
            System.out.println(hi.getProcessDefinitionId());
            System.out.println(hi.getProcessInstanceId());
            System.out.println("<========================>");
        }
    }
}

```

6、快速集成SpringSecurity安全框架

因为Activiti7与SpringBoot整合后，默认情况下，集成了SpringSecurity安全框架，这样便于我们快速与SpringSecurity进行集成。

我们可以去准备SpringSecurity整合进来的相关用户权限配置信息。SpringBoot的依赖包已经将SpringSecurity的依赖包也添加进项目中。

添加SecurityUtil类

```
package com.roy.utils;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.context.SecurityContextImpl;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.stereotype.Component;

import java.util.Collection;


@Component
public class SecurityUtil {
    private Logger logger = LoggerFactory.getLogger(SecurityUtil.class);

```

```
22      @Autowired
23      @Qualifier("myUserDetailsService")
24      private UserDetailsService userDetailsService;
25
26     public void logInAs(String username) {
27      UserDetails user = userDetailsService.loadUserByUsername(username);
28
29      if (user == null) {
30          throw new IllegalStateException("User " + username + " doesn't
exist, please provide a valid user");
31      }
32      logger.info("> Logged in as: " + username);
33
34      SecurityContextHolder.setContext(
35              new SecurityContextImpl(
36                      new Authentication() {
37                          @Override
38                          public Collection<? extends GrantedAuthority>
getAuthorities() {
39                              return user.getAuthorities();
40                          }
41                          @Override
42                          public Object getCredentials() {
43                              return user.getPassword();
44                          }
45                          @Override
46                          public Object getDetails() {
47                              return user;
48                          }
49                          @Override
50                          public Object getPrincipal() {
51                              return user;
52                          }
53                          @Override
54                          public boolean isAuthenticated() {
55                              return true;
56                          }
57                          @Override
58                          public void setAuthenticated(boolean
isAuthenticated) throws IllegalArgumentException { }
59                          @Override
60                          public String getName() {
61                              return user.getUsername();
62                          }
63      }));
64
org.activiti.engine.impl.identity.Authentication.setAuthenticatedUserId(use
rname);
65   }
```

```
66    }
```

> 这个类可以从Activiti7官方提供的example中找到。

添加DemoApplicationConfig配置类

在Activiti7官方下载的Example中找到DemoApplicationConfig类，它的作用是为了实现SpringSecurity框架的用户权限的配置，这样我们就可以在系统中使用用户权限信息。

本次项目中基本是在文件中定义出来的用户信息，当然也可以是数据库中查询的用户权限信息。

后面处理流程时用到的任务负责人，需要添加在这里

```
1  package com.roy.config;
2
3  import org.slf4j.Logger;
4  import org.slf4j.LoggerFactory;
5  import org.springframework.context.annotation.Bean;
6  import org.springframework.context.annotation.Configuration;
7  import org.springframework.security.core.authority.SimpleGrantedAuthority;
8  import org.springframework.security.core.userdetails.User;
9  import org.springframework.security.core.userdetails.UserDetailsService;
10 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
11 import org.springframework.security.crypto.password.PasswordEncoder;
12 import
   org.springframework.security.provisioning.InMemoryUserDetailsManager;
13
14 import java.util.Arrays;
15 import java.util.List;
16 import java.util.stream.Collectors;
17
18 @Configuration
19 public class DemoApplicationConfiguration {
20     private Logger logger =
   LoggerFactory.getLogger(DemoApplicationConfiguration.class);
21      @Bean
22      public UserDetailsService myUserDetailsService() {
23          InMemoryUserDetailsManager inMemoryUserDetailsManager = new
   InMemoryUserDetailsManager();
24          //这里添加用户，后面处理流程时用到的任务负责人，需要添加在这里
25          String[][] usersGroupsAndRoles = {
26                  {"jack", "password", "ROLE_ACTIVITI_USER",
   "GROUP_activitiTeam"},
```

```
                {"rose", "password", "ROLE_ACTIVITI_USER",
"GROUP_activitiTeam"},
                {"tom", "password", "ROLE_ACTIVITI_USER",
"GROUP_activitiTeam"},
                {"other", "password", "ROLE_ACTIVITI_USER",
"GROUP_otherTeam"},
                {"system", "password", "ROLE_ACTIVITI_USER"},
                {"admin", "password", "ROLE_ACTIVITI_ADMIN"},
        };

        for (String[] user : usersGroupsAndRoles) {
            List<String> authoritiesStrings =
Arrays.asList(Arrays.copyOfRange(user, 2, user.length));
            logger.info("> Registering new user: " + user[0] + " with the
following Authorities[" + authoritiesStrings + "]");
            inMemoryUserDetailsManager.createUser(new User(user[0],
passwordEncoder().encode(user[1]),
                    authoritiesStrings.stream().map(s -> new
SimpleGrantedAuthority(s)).collect(Collectors.toList())));
        }

        return inMemoryUserDetailsManager;
    }
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

6、测试与Security的集成

```
package com.roy.test;

import com.roy.utils.SecurityUtil;
import org.activiti.api.process.model.ProcessInstance;
import org.activiti.api.process.model.builders.ProcessPayloadBuilder;
import org.activiti.api.process.runtime.ProcessRuntime;
import org.activiti.api.runtime.shared.query.Page;
import org.activiti.api.runtime.shared.query.Pageable;
import org.activiti.api.task.model.Task;
import org.activiti.api.task.model.builders.TaskPayloadBuilder;
import org.activiti.api.task.runtime.TaskRuntime;
import org.activiti.engine.repository.ProcessDefinition;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
```

```
18
19  @RunWith(SpringRunner.class)
20  @SpringBootTest
21   public class Actviti7DemoApplicationTests {
22       @Autowired
23       private ProcessRuntime processRuntime;
24       @Autowired
25       private TaskRuntime taskRuntime;
26       @Autowired
27       private SecurityUtil securityUtil;
28
29      @Test
30      public void testActBoot(){
31          System.out.println(taskRuntime);
32      }
33
34      /**
35       * 查看流程定义
36       */
37      @Test
38      public void contextLoads() {
39          securityUtil.logInAs("system");
40          Page<org.activiti.api.process.model.ProcessDefinition>
    processDefinitionPage =
41                  processRuntime.processDefinitions(Pageable.of(0, 10));
42          System.out.println("可用的流程定义数量: " +
    processDefinitionPage.getTotalItems());
43          for (org.activiti.api.process.model.ProcessDefinition pd :
    processDefinitionPage.getContent()) {
44              System.out.println("流程定义: " + pd);
45          }
46      }
47
48      /**
49       * 启动流程实例
50       */
51      @Test
52      public void testStartProcess() {
53          securityUtil.logInAs("system");
54          ProcessInstance pi = processRuntime.start(ProcessPayloadBuilder.
55                  start().
56                  withProcessDefinitionKey("myProcess").
57                  build());
58          System.out.println("流程实例ID: " + pi.getId());
59      }
60
61
62      /**
63       **查询任务，并完成自己的任务
```

```
     **/
    @Test
    public void testTask() {
        securityUtil.logInAs("jack");
        Page<Task> taskPage=taskRuntime.tasks(Pageable.of(0,10));
        if (taskPage.getTotalItems()>0){
            for (Task task:taskPage.getContent()){
                taskRuntime.claim(TaskPayloadBuilder.
                        claim().
                        withTaskId(task.getId()).build());
                System.out.println("任务："+task);
                taskRuntime.complete(TaskPayloadBuilder.
                        complete().
                        withTaskId(task.getId()).build());
            }
        }
        Page<Task> taskPage2=taskRuntime.tasks(Pageable.of*(0,10));
        if (taskPage2.getTotalItems()>0){
            System.out.println("任务："+taskPage2.getContent());
        }
    }
}
```