

Activiti

讲师：波波

工作流(Workflow)，就是通过计算机对业务流程自动化执行管理。它主要解决的是“使在多个参与者之间按照某种预定义的规则自动进行传递文档、信息或任务的过程，从而实现某个预期的业务目标，或者促使此目标的实现”。

课程内容的介绍

1. Activiti基础篇
2. Activiti进阶篇
3. Activiti整合篇

一、Activiti基础篇

1. 工作流介绍

1.1 工作流概念介绍

工作流(Workflow)，就是通过计算机对业务流程自动化执行管理。它主要解决的是“使在多个参与者之间按照某种预定义的规则自动进行传递文档、信息或任务的过程，从而实现某个预期的业务目标，或者促使此目标的实现”。

1.2 工作流系统

一个软件系统中具有工作流的功能，我们把它称为工作流系统，一个系统中工作流的功能是什么？就是对系统的业务流程进行自动化管理，所以工作流是建立在业务流程的基础上，所以一个软件的系统核心根本上还是系统的业务流程，工作流只是协助进行业务流程管理。即使没有工作流业务系统也可以开发运行，只不过有了工作流可以更好的管理业务流程，提高系统的可扩展性。

1.3 适用行业

消费品行业，制造业，电信服务业，银证险等金融服务业，物流服务业，物业服务业，物业管理，大中型进出口贸易公司，政府事业机构，研究院所及教育服务业等，特别是大的跨国企业和集团公司。

1.4 具体应用

- 1、关键业务流程：订单、报价处理、合同审核、客户电话处理、供应链管理等
- 2、行政管理类：出差申请、加班申请、请假申请、用车申请、各种办公用品申请、购买申请、日报周报等凡是原来手工流转处理的行政表单。
- 3、人事管理类：员工培训安排、绩效考评、职位变动处理、员工档案信息管理等。
- 4、财务相关类：付款请求、应收款处理、日常报销处理、出差报销、预算和计划申请等。
- 5、客户服务类：客户信息管理、客户投诉、请求处理、售后服务管理等。

6、特殊服务类：ISO系列对应流程、质量管理对应流程、产品数据信息管理、贸易公司报关处理、物流公司货物跟踪处理等各种通过表单逐步手工流转完成的任务均可应用工作流软件自动规范地实施。

1.5 实现方式

在没有专门的工作流引擎之前，我们之前为了实现流程控制，通常的做法就是采用状态字段的值来跟踪流程的变化情况。这样不用角色的用户，通过状态字段的取值来决定记录是否显示。

针对有权限可以查看的记录，当前用户根据自己的角色来决定审批是否合格的操作。如果合格将状态字段设置一个值，来代表合格；当然如果不合也需要设置一个值来代表不合格的情况。

这是一种最为原始的方式。通过状态字段虽然做到了流程控制，但是当我们的流程发生变更的时候，这种方式所编写的代码也要进行调整。

那么有没有专业的方式来实现工作流的管理呢？并且可以做到业务流程变化之后，我们的程序可以不用改变，如果可以实现这样的效果，那么我们的业务系统的适应能力就得到了极大提升。

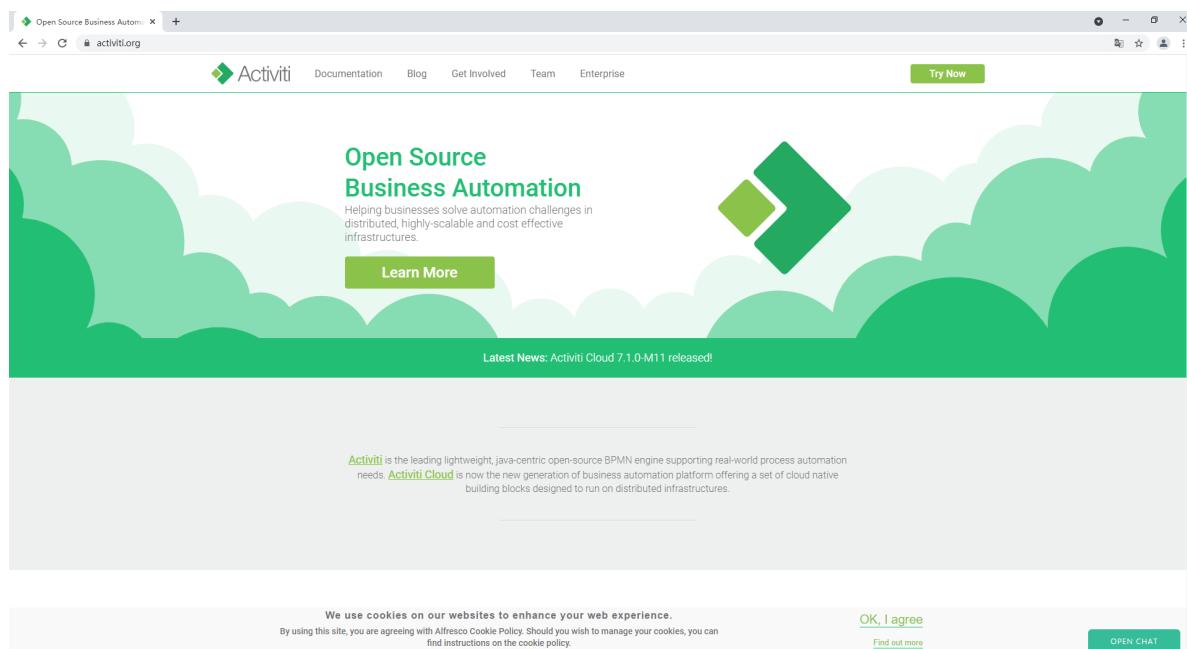
2.Activiti介绍

2.1 介绍

Alfresco软件在2010年5月17日宣布Activiti业务流程管理（BPM）开源项目的正式启动，其首席架构师由业务流程管理BPM的专家 Tom Baeyens担任，Tom Baeyens就是原来jbpm的架构师，而jbpm是一个非常有名的工作流引擎，当然activiti也是一个工作流引擎。

Activiti是一个工作流引擎，activiti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进行执行，实现了系统的流程由activiti进行管理，减少业务系统由于流程变更进行系统升级改造的工作量，从而提高系统的健壮性，同时也减少了系统开发维护成本。

官方网站：<https://www.activiti.org/>



2.1.1 BPM

BPM (Business Process Management) , 即业务流程管理，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。常见商业管理教育如EMBA、MBA等均将BPM包含在内。

2.1.2 BPM软件

BPM软件就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与解决方案的IT工具。

通过BPM软件对企业内部及外部的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，利润得以大幅提升。

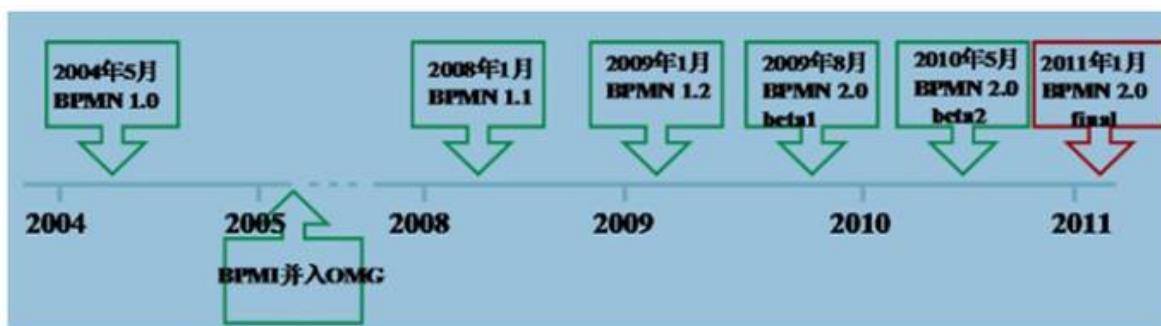
BPM软件在企业中应用领域广泛，凡是业务流程的地方都可以BPM软件进行管理，比如企业人事办公管理、采购流程管理、公文审批流程管理、财务管理等。

2.1.3 BPMN

BPMN (Business Process Model AndNotation) - 业务流程模型和符号 是由BPMI (BusinessProcess Management Initiative) 开发的一套标准的业务流程建模符号，使用BPMN提供的符号可以创建业务流程。

2004年5月发布了BPMN1.0规范.BPMI于2005年9月并入OMG (The Object Management Group对象管理组织)组织。OMG于2011年1月发布BPMN2.0的最终版本。

具体发展历史如下：



BPMN 是目前被各 BPM 厂商广泛接受的 BPM 标准。Activiti 就是使用 BPMN 2.0 进行流程建模、流程执行管理，它包括很多的建模符号，比如：

Event

用一个圆圈表示，它是流程中运行过程中发生的事情。



活动用圆角矩形表示，一个流程由一个活动或多个活动组成



Bpmn图形其实是通过xml表示业务流程，上边的.bpmn文件使用文本编辑器打开：

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:activiti="http://activiti.org/bpmn"
  xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
  xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DC"
  xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI"
  typeLanguage="http://www.w3.org/2001/XMLSchema"
  expressionLanguage="http://www.w3.org/1999/XPath"
  targetNamespace="http://www.activiti.org/test">
  <process id="myProcess" name="My process" isExecutable="true">
    <startEvent id="startevent1" name="Start"></startEvent>
    <userTask id="usertask1" name="创建请假单"></userTask>
    <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="usertask1">
    </sequenceFlow>
    <userTask id="usertask2" name="部门经理审核"></userTask>
    <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="usertask2">
    </sequenceFlow>
    <userTask id="usertask3" name="人事复核"></userTask>
    <sequenceFlow id="flow3" sourceRef="usertask2" targetRef="usertask3">
    </sequenceFlow>
    <endEvent id="endevent1" name="End"></endEvent>
    <sequenceFlow id="flow4" sourceRef="usertask3" targetRef="endevent1">
    </sequenceFlow>
  </process>
  <bpmndi:BPMNDiagram id="BPMNDiagram_myProcess">
    <bpmndi:BPMNPlane bpmnElement="myProcess" id="BPMNPlane_myProcess">

```

```

<bpmndi:BPMSHape bpmnElement="startevent1" id="BPMSHape_startevent1">
    <omgdc:Bounds height="35.0" width="35.0" x="130.0" y="160.0">
</omgdc:Bounds>
</bpmndi:BPMSHape>
<bpmndi:BPMSHape bpmnElement="usertask1" id="BPMSHape_usertask1">
    <omgdc:Bounds height="55.0" width="105.0" x="210.0" y="150.0">
</omgdc:Bounds>
</bpmndi:BPMSHape>
<bpmndi:BPMSHape bpmnElement="usertask2" id="BPMSHape_usertask2">
    <omgdc:Bounds height="55.0" width="105.0" x="360.0" y="150.0">
</omgdc:Bounds>
</bpmndi:BPMSHape>
<bpmndi:BPMSHape bpmnElement="usertask3" id="BPMSHape_usertask3">
    <omgdc:Bounds height="55.0" width="105.0" x="510.0" y="150.0">
</omgdc:Bounds>
</bpmndi:BPMSHape>
<bpmndi:BPMSHape bpmnElement="endevent1" id="BPMSHape_endevent1">
    <omgdc:Bounds height="35.0" width="35.0" x="660.0" y="160.0">
</omgdc:Bounds>
</bpmndi:BPMSHape>
<bpmndi:BPMEEdge bpmnElement="flow1" id="BPMEEdge_flow1">
    <omgdi:waypoint x="165.0" y="177.0"></omgdi:waypoint>
    <omgdi:waypoint x="210.0" y="177.0"></omgdi:waypoint>
</bpmndi:BPMEEdge>
<bpmndi:BPMEEdge bpmnElement="flow2" id="BPMEEdge_flow2">
    <omgdi:waypoint x="315.0" y="177.0"></omgdi:waypoint>
    <omgdi:waypoint x="360.0" y="177.0"></omgdi:waypoint>
</bpmndi:BPMEEdge>
<bpmndi:BPMEEdge bpmnElement="flow3" id="BPMEEdge_flow3">
    <omgdi:waypoint x="465.0" y="177.0"></omgdi:waypoint>
    <omgdi:waypoint x="510.0" y="177.0"></omgdi:waypoint>
</bpmndi:BPMEEdge>
<bpmndi:BPMEEdge bpmnElement="flow4" id="BPMEEdge_flow4">
    <omgdi:waypoint x="615.0" y="177.0"></omgdi:waypoint>
    <omgdi:waypoint x="660.0" y="177.0"></omgdi:waypoint>
</bpmndi:BPMEEdge>
</bpmndi:BPMLane>
</bpmndi:BPMDiagram>
</definitions>

```

2.2 使用步骤

部署activiti

Activiti是一个工作流引擎（其实就是一堆jar包API），业务系统访问(操作)activiti的接口，就可以方便的操作流程相关数据，这样就可以把工作流环境与业务系统的环境集成在一起。

流程定义

使用activiti流程建模工具(activity-designer)定义业务流程(.bpmmn文件)。

.bpmmn文件就是业务流程定义文件，通过xml定义业务流程。

流程定义部署

activiti部署业务流程定义 (.bpnn文件) 。

使用activiti提供的api把流程定义内容存储起来，在Activiti执行过程中可以查询定义的内容

Activiti执行把流程定义内容存储在数据库中

启动一个流程实例

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行。

在员工请假流程定义部署完成后，如果张三要请假就可以启动一个流程实例，如果李四要请假也启动一个流程实例，两个流程的执行互相不影响。

用户查询待办任务(Task)

因为现在系统的业务流程已经交给activiti管理，通过activiti就可以查询当前流程执行到哪了，当前用户需要办理什么任务了，这些activiti帮我们管理了，而不需要开发人员自己编写在sql语句查询。

用户办理任务

用户查询待办任务后，就可以办理某个任务，如果这个任务办理完成还需要其它用户办理，比如采购单创建后由部门经理审核，这个过程也是由activiti帮我们完成了。

流程结束

当任务办理完成没有下一个任务结点了，这个流程实例就完成了。

3.Activiti应用

3.1Activiti的基本使用

3.1.1 创建Maven项目

创建一个普通的Maven项目，并添加相关的依赖

```
<properties>
    <slf4j.version>1.6.6</slf4j.version>
    <log4j.version>1.2.12</log4j.version>
    <activiti.version>7.0.0.Beta1</activiti.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>
    <exclusions>
        <exclusion>
            <groupId>com.github.jgraph</groupId>
            <artifactId>jgraphx</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<!-- activiti 云支持 -->
<dependency>
    <groupId>org.activiti.cloud</groupId>
    <artifactId>activiti-cloud-services-api</artifactId>
    <version>${activiti.version}</version>
</dependency>
<!-- mysql驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.40</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>

```

3.1.2 log4j

添加一个日志文件log4j.properties

```

# 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=d:\log\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

```

3.1.3 添加Activiti配置文件

我们在本案例中使用的数据库是mysql8.0.

Activiti的默认的使用方式是要求我们在resources下创建activiti.cfg.xml文件，默认的方式的名称是不能修改的。

在配置文件中我们有两种配置方式：一种是单独配置数据源，另一种是不单独配置数据源

```

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

```

```

<bean
class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration"
id="processEngineConfiguration">
    <property name="jdbcDriver" value="com.mysql.cj.jdbc.Driver"/>
        <property name="jdbcUrl" value="jdbc:mysql://activiti2?
characterEncoding=utf-8&nullCatalogMeansCurrent=true&serverTimezone=UTC"
/>

    <property name="jdbcUsername" value="root" />
    <property name="jdbcPassword" value="123456" />
    <property name="databaseSchemaUpdate" value="true" />
    <!--<property name="dataSource" ref="dataSource" /-->
</bean>
<bean class="org.apache.commons.dbcp.BasicDataSource" id="dataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://activiti2?
characterEncoding=utf-8&nullCatalogMeansCurrent=true&serverTimezone=UTC"
/>

    <property name="username" value="root"/>
    <property name="password" value="123456"/>
    <property name="maxActive" value="3" />
    <property name="maxIdle" value="2" />
</bean>
</beans>

```

3.1.4 Java程序生成表结构

创建一个工具类，调用Activiti的工具类来生成activiti需要的表结构

```

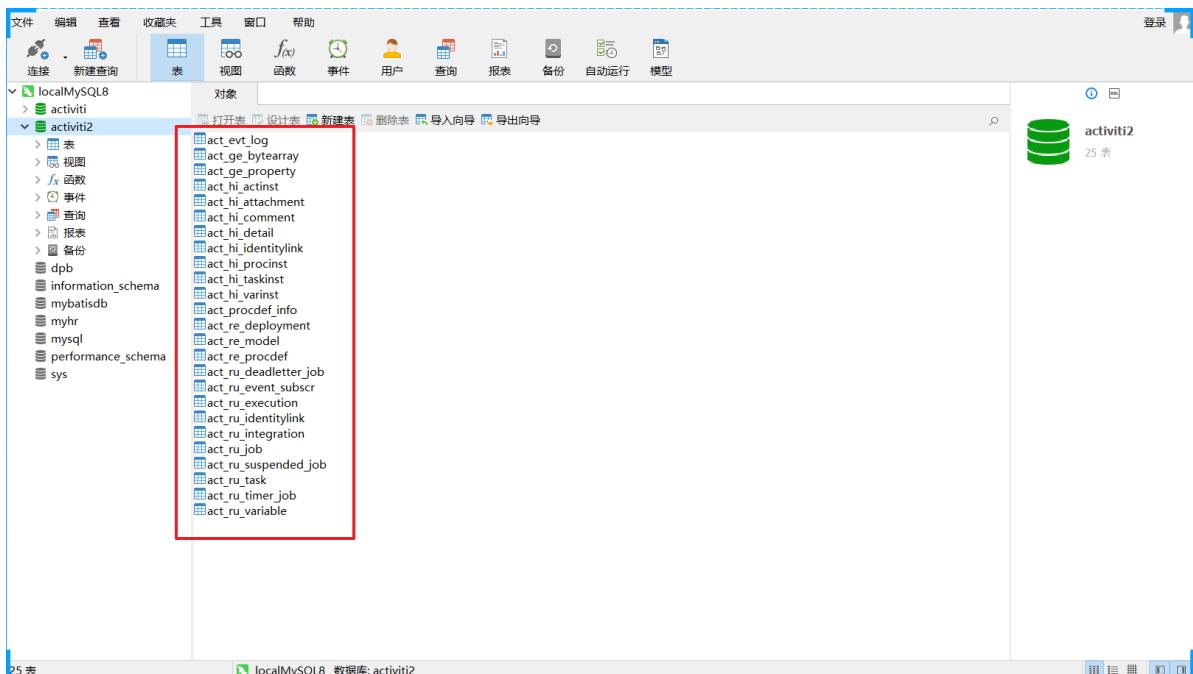
public class Test01 {

    /**
     * 生成Activiti的相关的表结构
     */
    @Test
    public void test01(){
        // 使用classpath下的activiti.cfg.xml中的配置来创建 ProcessEngine对象
        ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
        System.out.println(engine);
    }
}

```

执行的效果

执行完成后我们查看数据库，在其中创建了25张表，结果如下：



3.2 表结构介绍

3.2.1 表的命名规则和作用

看到刚才创建的表，我们发现Activiti 的表都以 ACT_ 开头。

第二部分是表示表的用途的两个字母标识。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

3.2.2 Activiti数据表介绍

表分类	表名	解释
一般数据		
	[ACT_GE_BYTERARRAY]	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
	[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

3.3 ProcessEngine创建方式

前面使用的是getDefalutProcessEngine()方法来加载classpath下的 activiti.cfg.xml文件，有些情况下我们可能没有按照默认的方式来处理，那这时我们应该怎么办呢？

```
/**  
 * 自定义的方式来加载配置文件  
 */  
@Test  
public void test02(){  
    // 首先创建ProcessEngineConfiguration对象  
    ProcessEngineConfiguration configuration =  
  
    ProcessEngineConfiguration.createProcessEngineConfigurationFromResource("activi  
ti.cfg.xml");  
    // 通过ProcessEngineConfiguration对象来创建 ProcessEngine对象  
    ProcessEngine processEngine = configuration.buildProcessEngine();  
}
```

3.4 Service服务接口

Service是工作流引擎提供用于进行工作流部署、执行、管理的服务接口，我们使用这些接口可以就是操作服务对应的数据表

3.4.1 Service创建方式

通过ProcessEngine创建Service

方式如下：

```
RuntimeService runtimeService = processEngine.getRuntimeService();  
RepositoryService repositoryService = processEngine.getRepositoryService();  
TaskService taskService = processEngine.getTaskService();
```

3.4.2 Service总览

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。

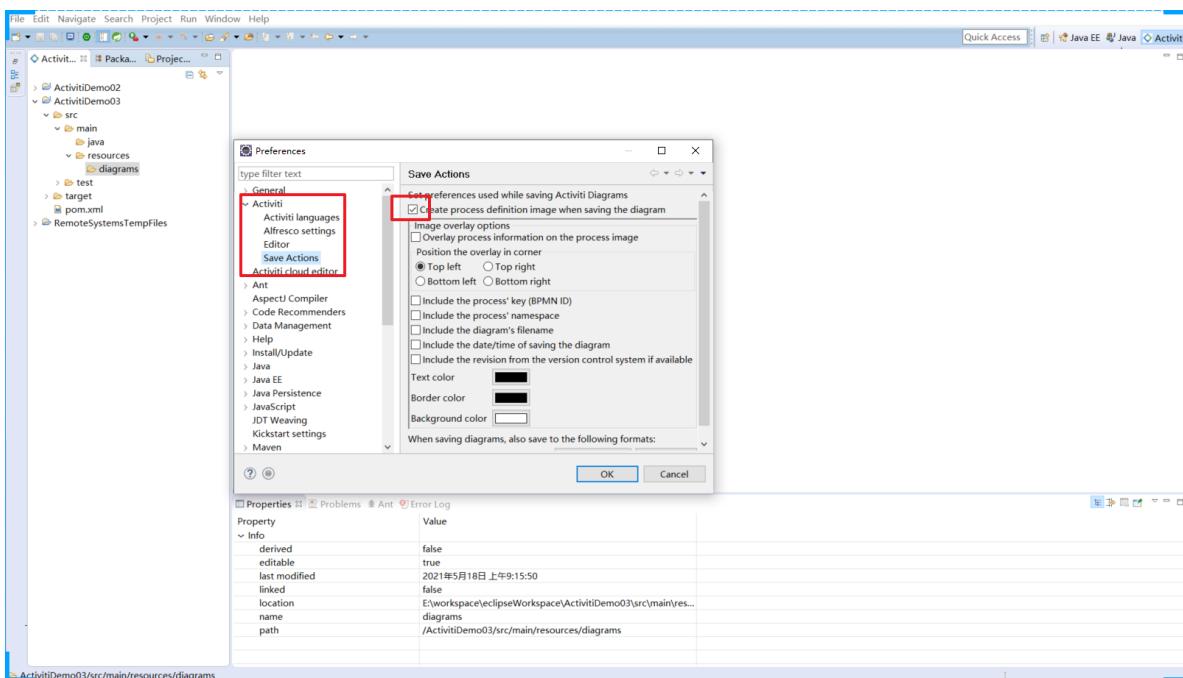
3.5 流程绘制

3.5.1 绘制插件

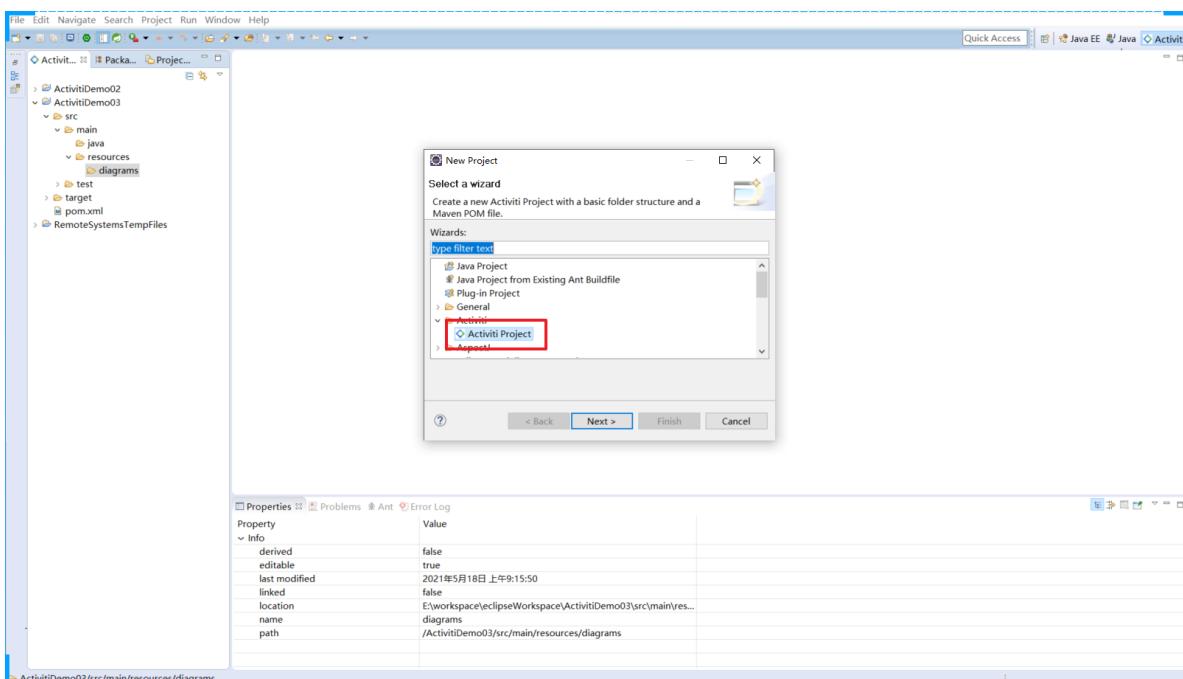
由于Idea 在2019年之后就没有再更新维护Activiti的设计工具了，那么在高版本的IDEA中我们就没法使用actiBPM插件来绘制了，这时可以选择降低版本来使用，或者使用我们给大家提供的Eclipse来实现流程的设计。

Version	Compatibility with IntelliJ IDEA Ultimate	Update Date	Action
3.E-8	12.0 – 2019.1.4	Nov 11, 2014	Download
2.E-8	12.0 – 2019.1.4	Oct 16, 2014	Download
1.E-8	12.0 – 2019.1.4	Oct 01, 2014	Download
1.E-9	12.0 – 2019.1.4	Jul 19, 2014	Download
1.E-10	12.0 – 2019.1.4	Mar 29, 2014	Download

我们提供给大家的Eclipse是已经集成好了Activiti插件的。

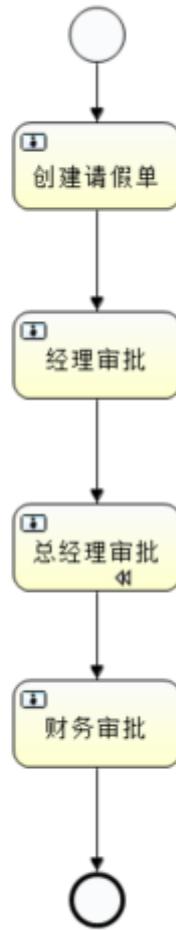


创建一个Activiti项目



3.5.2 绘制流程

使用滑板来绘制流程，通过从右侧把图标拖拽到左侧的面板，最终的效果



指定流程的主键

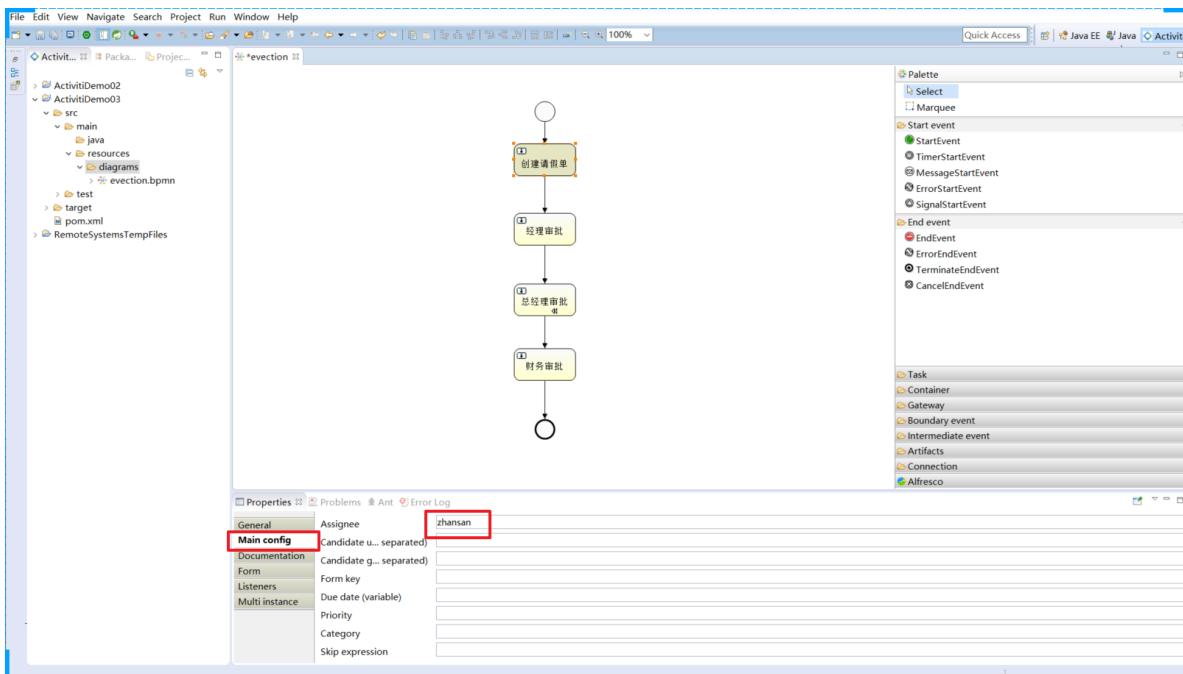
The screenshot shows the Activiti Designer interface with the following details:

- Project Structure:** The left sidebar shows a project named "ActivitiDemo03" containing "ActivitiDemo02", "main", "resources", "diagrams", and "test". A file named "eviction.bpmn" is selected.
- Process View:** The main canvas displays the BPMN process diagram with four tasks: "创建请假单", "经理审批", "总经理审批", and "财务审批", connected by arrows.
- Palette:** On the right, the palette lists various BPMN elements: Start event, End event, Task, Container, Gateway, Boundary event, Intermediate event, Artifacts, Connection, and Alfresco.
- Properties View:** The bottom-left panel shows the process properties:

Process	Id: eviction Name: 出差申请单 Namespace: http://www.activiti.org/test Candidate s... separated) Candidate s... separated) Documentation
---------	--

指定任务的负责人

在Properties视图中指定每个任务节点的负责人：

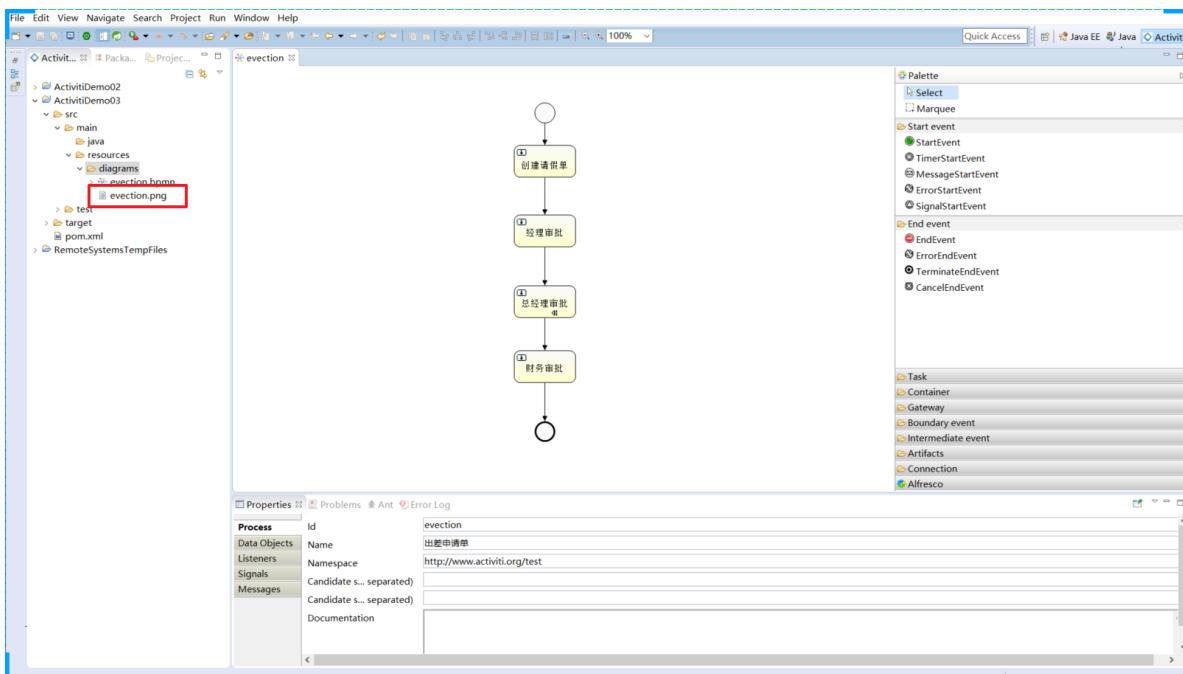


经理审批: lisi

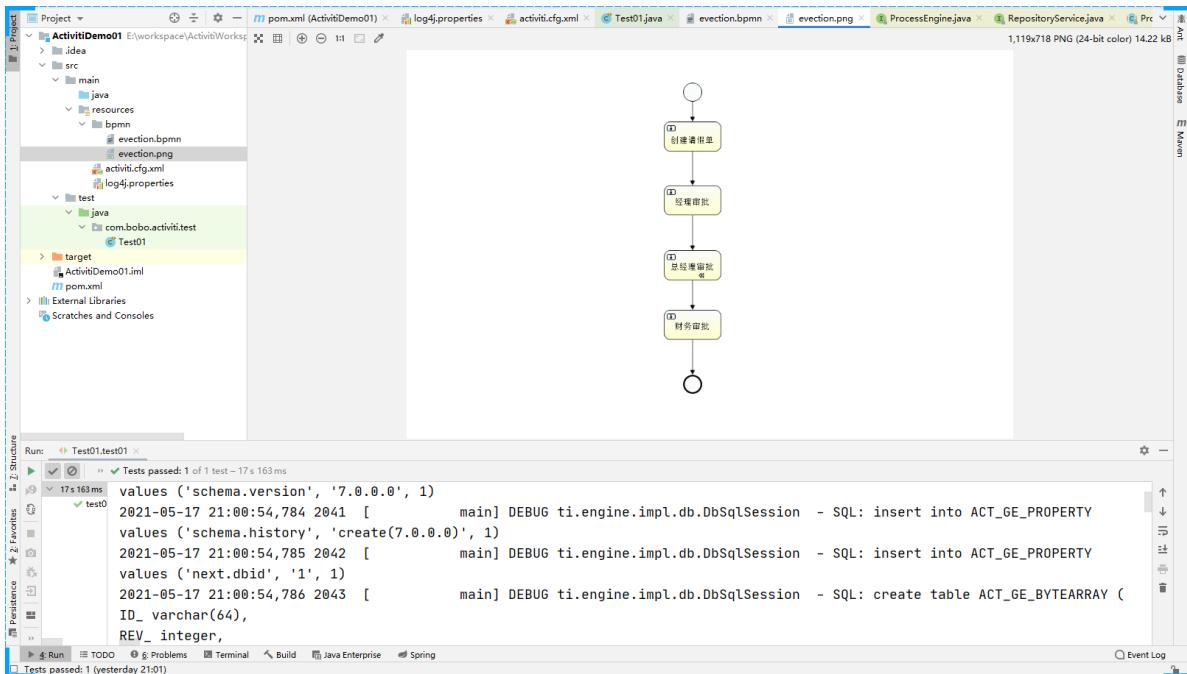
总经理审批: wangwu

财务审批: xiaoming

当我们设置完成后保存文件，会同时生成png图片



然后将这两个文件拷贝到IDEA项目中即可



3.5.3 图标介绍

流程符号

BPMN 2.0是业务流程建模符号2.0的缩写。

它由Business Process Management Initiative这个非营利协会创建并不断发展。作为一种标识，BPMN 2.0是使用一些**符号**来明确业务流程设计流程图的一整套符号规范，它能增进业务建模时的沟通效率。

目前BPMN2.0是最新的版本，它用于在BPM上下文中进行布局和可视化的沟通。

接下来我们先来了解在流程设计中常见的 符号。

BPMN2.0的基本符号主要包含：

事件 Event



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，还可以是一个当前流程的子处理流程；其次，你还可以为活动指定不同的类型。常见活动如下：



网关 GateWay

网关用来处理决策，有几种常用网关需要了解：



排他网关



并行网关



包容网关



综合网关



事件网关

排他网关 (x)

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

如果多条线路计算结果都是 true，则会执行第一个值为 true 的线路。如果所有网关计算结果没有 true，则引擎会抛出异常。

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

拆分 —— 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线路。

合并 —— 所有从并行网关拆分并执行完成的线路均在此等候，直到所有的线路都执行完成才继续向下执行。

包容网关 (+)

—— 可以同时执行多条线路，也可以在网关上设置条件

拆分 —— 计算每条线路上的表达式，当表达式计算结果为 true 时，创建一个并行线路并继续执行

合并 —— 所有从并行网关拆分并执行完成的线路均在此等候，直到所有的线路都执行完成才继续向下执行。

事件网关 (+)

—— 专门为中间捕获事件设置的，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，流程处于等待状态，需要等待抛出事件才能将等待状态转换为活动状态。

流向 Flow

流是连接两个流程节点的连线。常见的流向包含以下几种：



Sequence Flow

顺序流



Message Flow

消息流



Association

关联



Data Association

数据关联

流程设计器使用

Palette (画板)

Connection—连接

Event---事件

Task---任务

Gateway--网关

Container—容器

Boundary event—边界事件

Intermediate event- -中间事件

4. Activiti流程操作

4.1 流程的部署

将上面在设计器中定义的流程部署到activiti数据库中，就是我们讲的流程部署。

通过调用Activiti的api将流程定义的bpmn和png两个文件一个一个添加部署到activiti中，还可以将两个文件打成zip包部署。

4.1.1 单个文件部署

分别将bpmn文件和png图片分别部署

```
/**
 * 实现文件的单个部署
 */
@Test
public void test03(){
    // 1.获取ProcessEngine对象
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 2.获取RepositoryService进行部署操作
    RepositoryService service = engine.getRepositoryService();
    // 3.使用RepositoryService进行部署操作
    Deployment deploy = service.createDeployment()
        .addClasspathResource("bpmn/evection.bpmn") // 添加bpmn资源
        .addClasspathResource("bpmn/evection.png") // 添加png资源
        .name("出差申请流程")
        .deploy(); // 部署流程
    // 4.输出流程部署的信息
    System.out.println("流程部署的id:" + deploy.getId());
    System.out.println("流程部署的名称: " + deploy.getName());
}
```

日志中查看到相关的输出信息

The screenshot shows the Eclipse IDE interface with the following details:

- Project Explorer:** Shows the `ActivitiDemo01` project structure under the `src/main/resources` folder, including `activiti.cfg.xml` and `log4j.properties`.
- Logcat Output:** The `Test01.test03` test case is running, and the log output is as follows:

```
2s 110ms ine.impl.db.DbSqlSession - insert ProcessDefinitionEntity[evection:2:2504]
ine.impl.db.DbSqlSession - insert DeploymentEntity[id=2501, name=出差申请流程]
ine.impl.db.DbSqlSession - insert ResourceEntity[id=2502, name=bpmn/evection.png]
ine.impl.db.DbSqlSession - insert ResourceEntity[id=2503, name=bpmn/evection.bpmn]
ine.impl.db.DbSqlSession - flush summary: 4 insert, 0 update, 0 delete.
ine.impl.db.DbSqlSession - now executing flush...
ine.impl.db.DbSqlSession - inserting: ProcessDefinitionEntity[evection:2:2504]
.insertProcessDefinition - ==> Preparing: insert into ACT_RE_PROCDEF(ID_, REV_, CATEGORY_, NAME_, KEY_, VERSION_, DEPLOYMENT_ID_)
.insertProcessDefinition - ==> Parameters: evection:2:2504(String), http://www.activiti.org/test(String), 出差申请单(String), ev...
.insertProcessDefinition - <== Update: 1
ine.impl.db.DbSqlSession - inserting: DeploymentEntity[id=2501, name=出差申请流程]
ityImpl.insertDeployment - ==> Preparing: insert into ACT_RE_DEPLOYMENT(ID_, NAME_, CATEGORY_, KEY_, TENANT_ID_, DEPLOY_TIME_)
ityImpl.insertDeployment - ==> Parameters: 2501(String), 出差申请流程(String), null, null, (String), 2021-05-18 09:53:32.701(Time)
ityImpl.insertDeployment - <== Update: 1
/Impl.bulkInsertResource - ==> Preparing: INSERT INTO ACT_GE_BYTEARRAY(ID_, REV_, NAME_, BYTES_, DEPLOYMENT_ID_, GENERATED_DATE_)
/Impl.bulkInsertResource - ==> Parameters: 2502(String), bpmn/evection.png(String), java.io.ByteArrayInputStream@361c294e(Byte[])
/Impl.bulkInsertResource - <== Update: 2
/batisTransactionContext - firing event committing...
/batisTransactionContext - committing the ibatis sql session...
ion.jdbc.JdbcTransaction - Committing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@78452606]
/batisTransactionContext - firing event committed...
```
- Bottom Status Bar:** Shows the status bar with "Tests passed: 1 (a minute ago)".
- Bottom Right:** Event Log tab.

4.1.2 部署zip文件

将bpmn文件和png文件两个打包为一个zip文件，统一上传

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Tree:** The project is named "ActivitiDemo01". The file "evection.bpmn" is selected in the tree, and its contents are displayed in the code editor.
- Code Editor:** The code for the test method `test04` is shown. It uses a ZipInputStream to read a file named "evection.zip".
- Run Output:** The bottom pane shows the terminal output of the test execution. The log includes several INFO and DEBUG messages from the Activiti engine and K.core.env.StandardEnvironment.
- Toolbars and Status Bar:** Standard IntelliJ toolbars are visible at the top and bottom. The status bar at the bottom right shows the date as "2021-05-18" and the time as "10:05:04".

```
/**  
 * 通过一个zip文件来部署操作  
 */  
  
@Test  
public void test04(){  
    // 定义zip文件的输入流  
    InputStream inputStream =  
this.getClass().getClassLoader().getResourceAsStream("bpmn/evection.zip");  
    // 对 inputStream 做装饰  
    ZipInputStream zipInputStream = new ZipInputStream(inputStream);  
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
```

```

RepositoryService repositoryService = engine.getRepositoryService();
Deployment deploy = repositoryService.createDeployment()
    .addZipInputStream(zipInputStream)
    .name("出差申请流程")
    .deploy();
// 4.输出流程部署的信息
System.out.println("流程部署的id:" + deploy.getId());
System.out.println("流程部署的名称: " + deploy.getName());
}

```

上传后的数据库中的数据和单个文件上传其实是一样的。

4.1.3 操作数据表

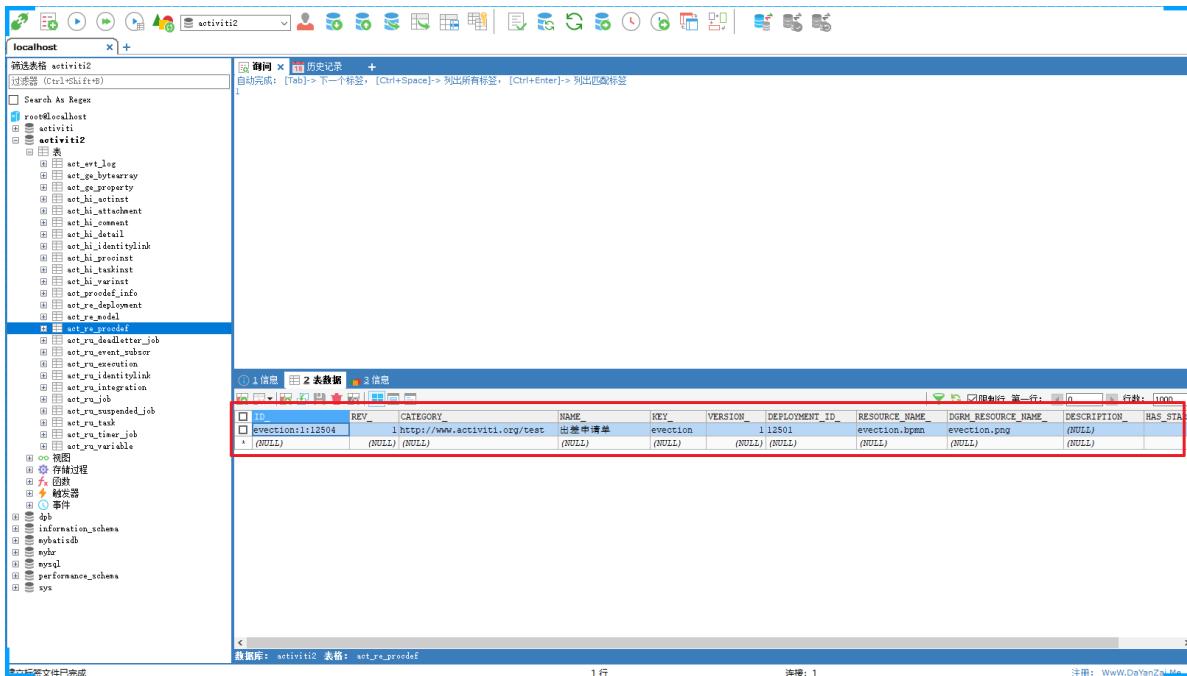
流程定义部署后操作activiti中的三张表

act_re_deployment: 流程定义部署表，每部署一次就增加一条记录

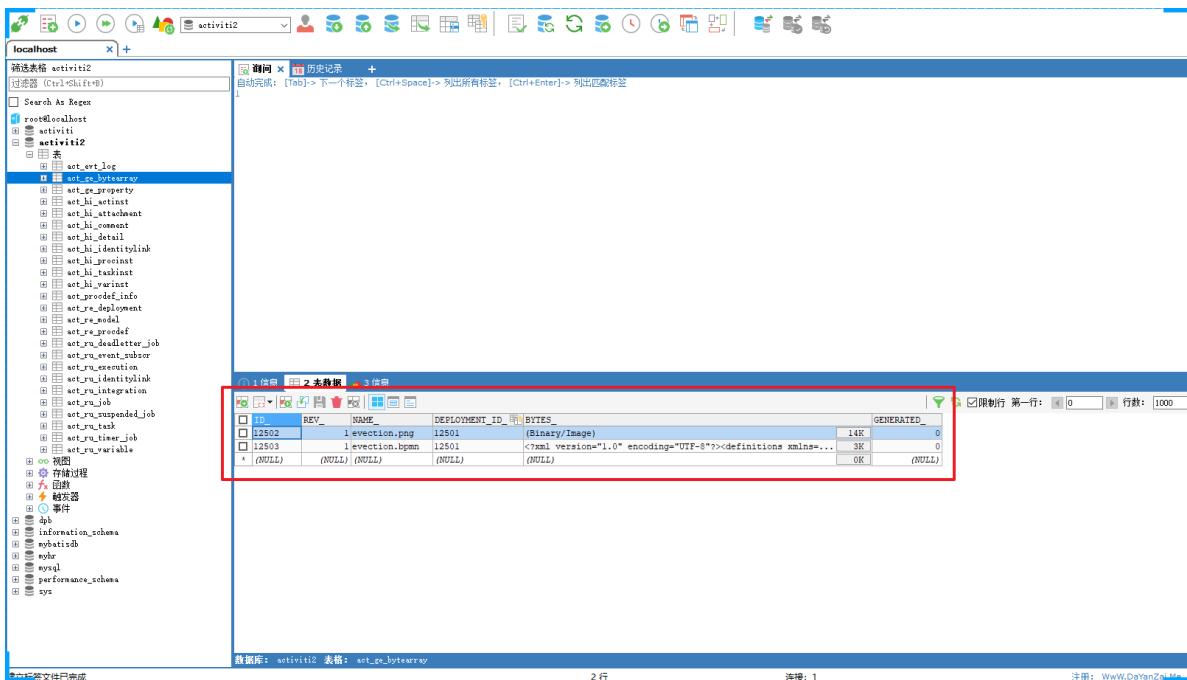
The screenshot shows the MySQL Workbench interface with the 'activiti2' database selected. The left pane displays the schema browser with the 'act_re_deployment' table highlighted. The right pane shows the data for this table, which contains a single row:

ID	NAME	CATEGORY	KEY_	TENANT_ID_	DEPLOY_TIME_	ENGINE_VERSION
12501	出差申请流程	(NULL)	(NULL)	(NULL)	2021-05-18 02:05:06,452	(NULL)

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



act_ge_bytearray：流程资源表，流程部署的 bpmn文件和png图片会保存在该表中



4.2 启动流程实例

流程定义部署在Activiti后就可以通过工作流管理业务流程，也就是说上边部署的出差申请流程可以使用了。

针对该流程，启动一个流程表示发起一个新的出差申请单，这就相当于Java类和Java对象的关系，类定义好了后需要new创建一个对象使用，当然可以new出多个对象来，对于出差申请流程，张三可以发起一个出差申请单需要启动一个流程实例。

```
/*
 * 启动一个流程实例
 */
@Test
```

```

public void test05(){
    // 1. 创建ProcessEngine对象
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 2. 获取RuntimeService对象
    RuntimeService runtimeService = engine.getRuntimeService();
    // 3. 根据流程定义的id启动流程
    String id= "evection";
    ProcessInstance processInstance =
    runtimeService.startProcessInstanceByKey(id);
    // 4. 输出相关的流程实例信息
    System.out.println("流程定义的ID: " +
processInstance.getProcessDefinitionId());
    System.out.println("流程实例的ID: " + processInstance.getId());
    System.out.println("当前活动的ID: " + processInstance.getActivityId());
}

```

输出内容：

```

// 4. 输出流程部署的信息
System.out.println("流程部署的id:" + deploy.getId());
System.out.println("流程部署的名称: " + deploy.getName());

/**
 * 启动一个流程实例
 */
@Test
public void test05(){
    // 1. 创建ProcessEngine对象
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 2. 获取RuntimeService对象
}

2021-05-18 10:25:29,217 2080 [main] DEBUG aloneMybatisTransactionContext - firing event committing...
2021-05-18 10:25:29,217 2080 [main] DEBUG aloneMybatisTransactionContext - committing the ibatis sql session...
2021-05-18 10:25:29,217 2080 [main] DEBUG ansaction.jdbc.JdbcTransaction - Committing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@638f3d]
2021-05-18 10:25:29,229 2092 [main] DEBUG aloneMybatisTransactionContext - firing event committed...
2021-05-18 10:25:29,229 2092 [main] DEBUG ansaction.jdbc.JdbcTransaction - Resetting autocommit to true on JDBC Connection
2021-05-18 10:25:29,229 2092 [main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@638f3d]
2021-05-18 10:25:29,229 2092 [main] DEBUG source.pooled.PooledDataSource - Returned connection 2017797638 to pool.
2021-05-18 10:25:29,230 2093 [main] DEBUG mpl.interceptor.LogInterceptor - --- StartProcessInstanceCmd finished ---
2021-05-18 10:25:29,250 2093 [main] DEBUG mpl.interceptor.LogInterceptor - 

流程定义的ID: evection:1:12504
流程实例的ID: 15001
当前活动的ID: null

```

启动流程实例涉及到的表结构

act_hi_actinst 流程实例执行历史

act_hi_identitylink 流程的参与用户的历史信息

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

4.3 任务查找

流程启动后，任务的负责人就可以查询自己当前能够处理的任务了，查询出来的任务都是当前用户的待办任务

```
/*
 * 任务查询
 */
@Test
public void test06(){
    String assignee ="zhansan";
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 任务查询 需要获取一个 TaskService 对象
    TaskService taskService = engine.getTaskService();
    // 根据流程的key和任务负责人 查询任务
    List<Task> list = taskService.createTaskQuery()
        .processDefinitionKey("evection")
        .taskAssignee(assignee)
        .list();
    // 输出当前用户具有的任务
    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());
    }
}
```

输出结果

The screenshot shows the IntelliJ IDEA interface. On the left is the project structure tree, which includes a 'ActivitiDemo01' project with a 'src' folder containing 'main' and 'test' packages. The 'test' package contains a 'java' folder with a 'Test01' class. The code editor on the right displays the 'test06()' method from the 'Test01' class. The terminal window at the bottom shows the execution results of the test case 'test0'. A red box highlights the terminal output, which shows the following log entries:

```
2021-05-18 10:38:08,614 1967 [main] DEBUG ti.engine.impl.db.DbSqlSession - now executing flush...
2021-05-18 10:38:08,614 1967 [main] DEBUG aloneMybatisTransactionContext - firing event committing...
2021-05-18 10:38:08,614 1967 [main] DEBUG aloneMybatisTransactionContext - committing the ibatis sql session...
2021-05-18 10:38:08,614 1967 [main] DEBUG aloneMybatisTransactionContext - firing event committed...
2021-05-18 10:38:08,614 1967 [main] DEBUG ansaction.jdbc.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@631f14]
2021-05-18 10:38:08,615 1968 [main] DEBUG ansaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@631f14]
2021-05-18 10:38:08,615 1968 [main] DEBUG source.pooled.PooledDataSource - Returned connection 2017797638 to pool.
2021-05-18 10:38:08,615 1968 [main] DEBUG mpl.interceptor.LogInterceptor - --- TaskQueryImpl finished -----
2021-05-18 10:38:08,615 1968 [main] DEBUG mpl.interceptor.LogInterceptor -
```

A red box also highlights the terminal output, showing the results of the task query:

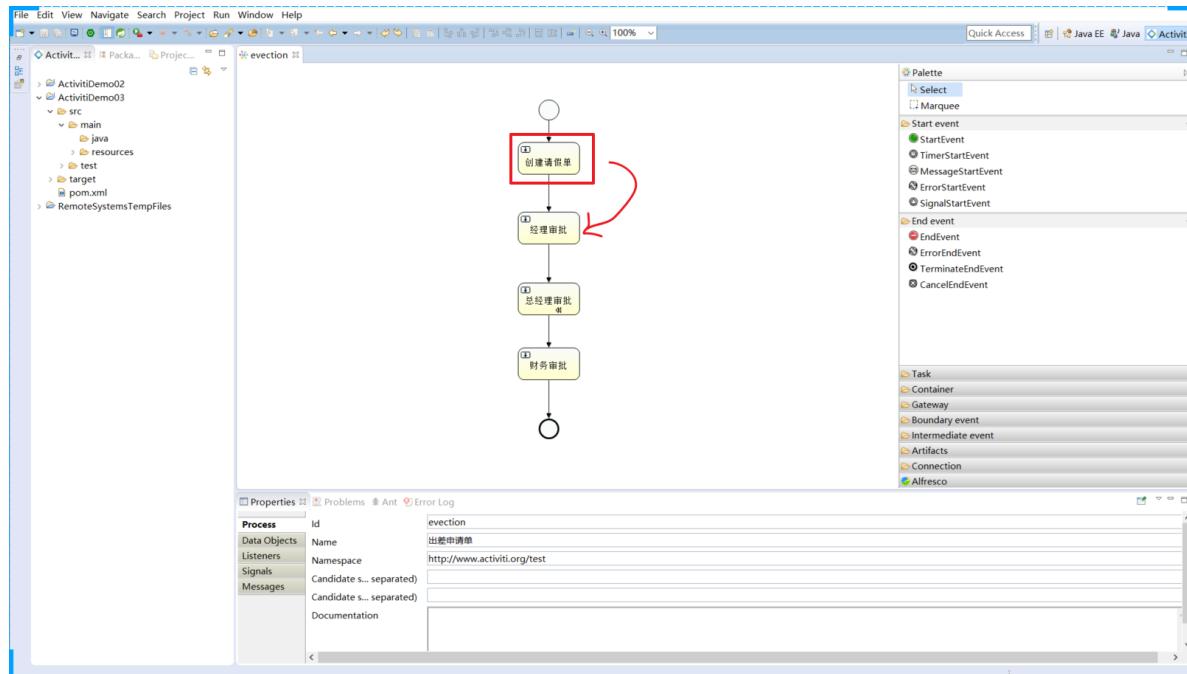
```
流程实例id: 15001
任务id:15005
任务负责人: zhansan
任务名称: 创建请假单
```

4.4 流程任务处理

任务负责人查询出来了待办的人，选择任务进行处理，完成任务

```
/**  
 * 流程任务的处理  
 */  
@Test  
public void test07(){  
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();  
    TaskService taskService = engine.getTaskService();  
    Task task = taskService.createTaskQuery()  
        .processDefinitionKey("evection")  
        .taskAssignee("zhansan")  
        .singleResult();  
    // 完成任务  
    taskService.complete(task.getId());  
}
```

zhangsan处理了这个操作后，流程就流转到了 lisi处



然后就是不同的用户登录，然后查询任务处理任务，直到任务流程走完。

4.5 流程定义的查询

查询流程相关的信息，包括流程的定义，流程的部署，流程定义的版本

```
/**  
 * 查询流程的定义  
 */  
@Test  
public void test08(){  
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();  
    RepositoryService repositoryService = engine.getRepositoryService();  
    // 获取一个 ProcessDefinitionQuery 对象 用来查询操作
```

```

ProcessDefinitionQuery processDefinitionQuery =
repositoryService.createProcessDefinitionQuery();
List<ProcessDefinition> list =
processDefinitionQuery.processDefinitionKey("evection")
    .orderByProcessDefinitionVersion() // 安装版本排序
    .desc() // 倒序
    .list();
// 输出流程定义的信息
for (ProcessDefinition processDefinition : list) {
    System.out.println("流程定义的ID: " + processDefinition.getId());
    System.out.println("流程定义的name: " + processDefinition.getName());
    System.out.println("流程定义的key:" + processDefinition.getKey());
    System.out.println("流程定义的version:" +
processDefinition.getVersion());
    System.out.println("流程部署的id:" +
processDefinition.getDeploymentId());
}
}

```

输出结果

```

流程定义的ID: evection:1:12504
流程定义的name: 出差申请单
流程定义的key:evection
流程定义的version:1
流程部署的id:12501

```

4.6 流程的删除

```

/**
 * 删除流程
 */
@Test
public void test09(){
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    RepositoryService repositoryService = engine.getRepositoryService();
    // 删除流程定义，如果该流程定义已经有了流程实例启动则删除时报错
    repositoryService.deleteDeployment("12501");
    // 设置为TRUE 级联删除流程定义，及时流程有实例启动，也可以删除，设置为false 非级联删除操作。
    //repositoryService.deleteDeployment("12501",true);
}

```

注意：项目开发中级联删除操作的权限一般只开发给超级管理员使用。

4.7 流程资源的下载

现在我们的流程资源文件已经上传到了数据库中，如果其他用户想要查看这些资源，可以从数据库中把这些资源下载到本地。

解决方案：

1. jdbc对blob类型处理clob类型数据读取出来就可以了。
2. 使用activiti的api来实现操作。

使用activiti的api来操作我们需要添加commons-io的依赖

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

实现代码

```
/**
 * 读取数据库中的资源文件
 */
@Test
public void test10() throws Exception{
    // 1.得到ProcessEngine对象
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 2.获取RepositoryService对象
    RepositoryService repositoryService = engine.getRepositoryService();
    // 3.得到查询器
    ProcessDefinition definition =
repositoryService.createProcessDefinitionQuery()
        .processDefinitionKey("evection")
        .singleResult();
    // 4.获取流程部署的id
    String deploymentId = definition.getDeploymentId();
    // 5.通过repositoryService对象的相关方法 来获取图片信息和bpnn信息
    // png图片
    InputStream pngInput = repositoryService
        .getResourceAsStream(deploymentId,
definition.getDiagramResourceName());
    // bpnn 文件的流
    InputStream bpmnInput = repositoryService
        .getResourceAsStream(deploymentId,
definition.getResourceName());
    // 6.文件的保存
    File filePng = new File("d:/evection.png");
    File fileBpmn = new File("d:/evection.bpmn");
    OutputStream pngOut = new FileOutputStream(filePng);
    OutputStream bpmnOut = new FileOutputStream(fileBpmn);

    IOutils.copy(pngInput,pngOut);
    IOutils.copy(bpmnInput,bpmnOut);

    pngInput.close();
```

```
    pngOut.close();
    bpmnInput.close();
    bpmnOut.close();
}
```

4.8 流程历史信息查看

即使流程定义已经被删除了，流程执行的实例信息通过前面的分析，依然保存在Activiti的act_hi_* 的相关表结构中，所以我们还是可以查询流程的执行的历史信息，可以通过HistoryService来查看

```
/**
 * 流程历史信息查看
 */
@Test
public void test11(){
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 查看历史信息我们需要通过 HistoryService来实现
    HistoryService historyService = engine.getHistoryService();
    // 获取 actinst 表的查询对象
    HistoricActivityInstanceQuery instanceQuery =
historyService.createHistoricActivityInstanceQuery();
    instanceQuery.processDefinitionId("evection:1:12504");
    instanceQuery.orderByHistoricActivityStartTime().desc();
    List<HistoricActivityInstance> list = instanceQuery.list();
    // 输出查询的结果
    for (HistoricActivityInstance hi : list) {
        System.out.println(hi.getActivityId());
        System.out.println(hi.getActivityName());
        System.out.println(hi.getActivityType());
        System.out.println(hi.getAssignee());
        System.out.println(hi.getProcessDefinitionId());
        System.out.println(hi.getProcessInstanceId());
        System.out.println("-----");
    }
}
```

输出结果

```
usertask3
总经理审批
userTask
wangwu
evection:1:12504
15001
-----
usertask2
经理审批
userTask
lisi
evection:1:12504
15001
-----
```

```
usertask1  
创建请假单  
userTask  
zhansan  
evection:1:12504  
15001
```

```
startevent1  
Start  
startEvent  
null  
evection:1:12504  
15001
```

二、Activiti进阶篇

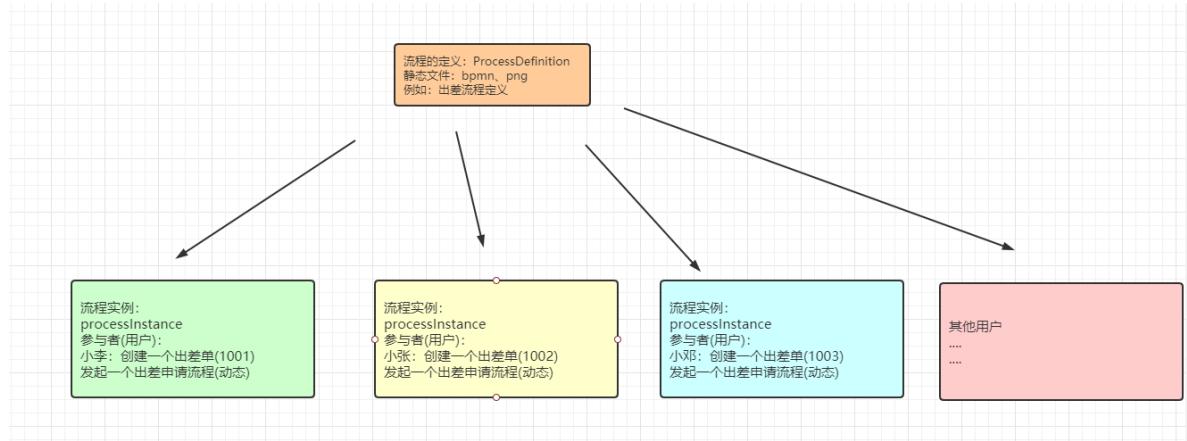
1. 流程实例

1.1 什么是流程实例

流程实例(ProcessInstance)代表流程定义的执行实例

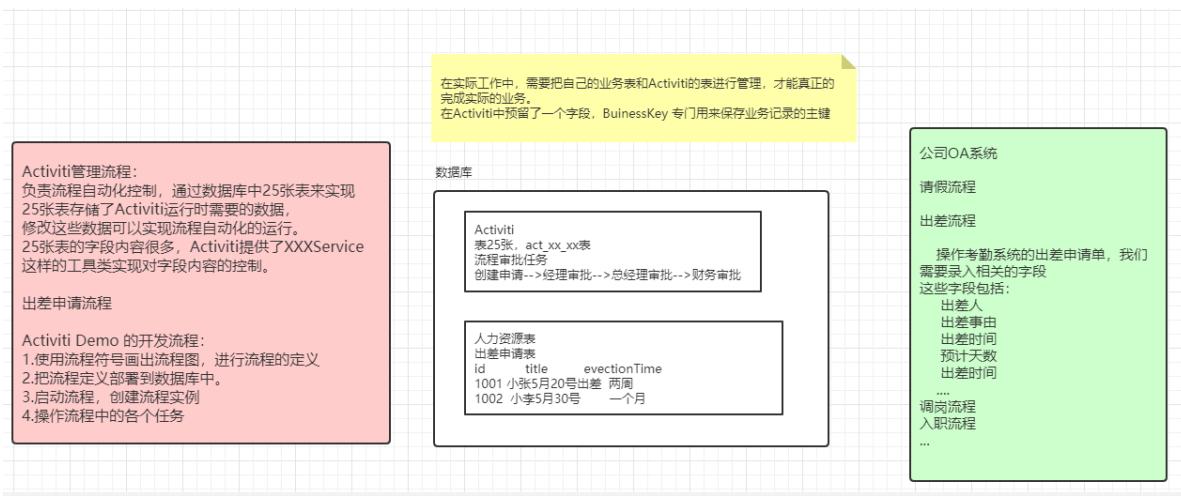
一个流程实例包括了所有的运行节点，我们可以利用这个对象来了解当前流程实例的进度等信息

例如：用户或者程序安装流程定义的内容发起了一个流程，这个就是一个流程实例



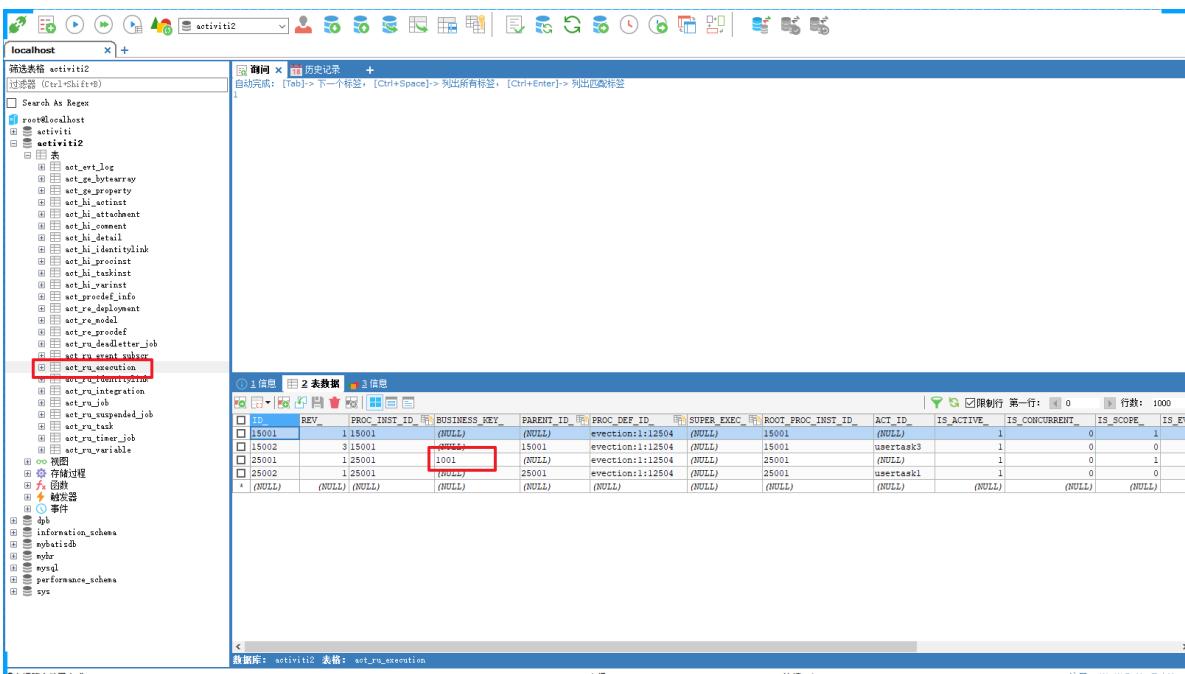
1.2 业务管理

流程定义部署在Activiti后，我们就可以在系统中通过Activiti去管理流程的执行，但是如果我们要将我们的流程实例和业务数据关联，这时我们需要使用到Activiti中预留的BusinessKey(业务标识)来关联



实现代码：

```
/*
 * 启动流程实例，添加businessKey
 */
@Test
public void test01(){
    // 1.获取ProcessEngine对象
    ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
    // 2.获取RuntimeService对象
    RuntimeService runtimeService = processEngine.getRuntimeService();
    // 3.启动流程实例
    ProcessInstance instance = runtimeService
        .startProcessInstanceByKey("evection", "1001");
    // 4.输出processInstance相关属性
    System.out.println("businessKey = "+instance.getBusinessKey());
}
```



1.3 流程实例的挂起和激活

在实际场景中可能由于流程变更需要将当前运行的流程暂停而不是删除，流程暂停后将不能继续执行。

1.3.1 全部流程挂起

操作流程的定义为挂起状态，该流程定义下边所有的流程实例全部暂停。

流程定义为挂起状态，该流程定义将不允许启动新的流程实例，同时该流程定义下的所有的流程实例都将全部挂起暂停执行。

```
/*
 * 全部流程挂起实例与激活
 */
@Test
public void test02(){
    // 1.获取ProcessEngine对象
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 2.获取RepositoryService对象
    RepositoryService repositoryService = engine.getRepositoryService();
    // 3.查询流程定义的对象
    ProcessDefinition processDefinition =
repositoryService.createProcessDefinitionQuery()
    .processDefinitionKey("evection")
    .singleResult();
    // 4.获取当前流程定义的状态
    boolean suspended = processDefinition.isSuspended();
    String id = processDefinition.getId();
    // 5.如果挂起就激活，如果激活就挂起
    if(suspended){
        // 表示当前定义的流程状态是 挂起的
        repositoryService.activateProcessDefinitionById(
            id // 流程定义的id
            ,true // 是否激活
            ,null // 激活时间
        );
        System.out.println("流程定义: " + id + ",已激活");
    }else{
        // 非挂起状态，激活状态 那么需要挂起流程定义
        repositoryService.suspendProcessDefinitionById(
            id // 流程id
            ,true // 是否挂起
            ,null // 挂起时间
        );
        System.out.println("流程定义: " + id + ",已挂起");
    }
}
```

挂起流程定义后，对于的实例对象中的状态会修改为2

The screenshot shows the MySQL Workbench interface. On the left, the database structure for 'activiti2' is displayed, including tables like 'act_ge_bytearray', 'act_hi_attachment', and 'act_ru_task'. In the center, a table named 'act_ru_task' is selected. The first row of the table is highlighted with a red box. The table has columns such as 'TASK_DEF_KEY', 'OWNER', 'ASSIGNEE', 'DELEGATION_', 'PRIORITY', 'CREATE_TIME', 'DUE_DATE', 'CATEGORY', 'SUSPENSION_STATE', 'EXVENT_ID', 'FORM_KEY', and 'CLAIM_TIME'. The 'SUSPENSION_STATE' column for the first row contains the value '2'.

然后再去操作对于的流程实例会抛异常信息

The screenshot shows an IDE environment (IntelliJ IDEA). On the left, a project structure is visible with files like 'Test01.java', 'Test02.java', 'pom.xml', 'ProcessInstance.java', 'Deployment.java', 'evection.bpmn', 'evection.png', 'evection.zip', 'activiti.cfg.xml', and 'log4j.properties'. In the center, a Java code editor shows a snippet of code. The code includes a conditional block that calls `repositoryService.suspendProcessDefinitionById()`. The terminal output at the bottom shows a stack trace for an `org.activiti.engine.ActivitiException: Cannot complete a suspended task`.

```

else{
    // 非挂起状态，激活状态 那么需要挂起流程定义
    repositoryService.suspendProcessDefinitionById(
        id // 流程id
        , suspendProcessInstances: true // 是否挂起
        , suspensionDate: null // 挂起时间
    );
    System.out.println("流程定义: " + id + ",已挂起");
}

```

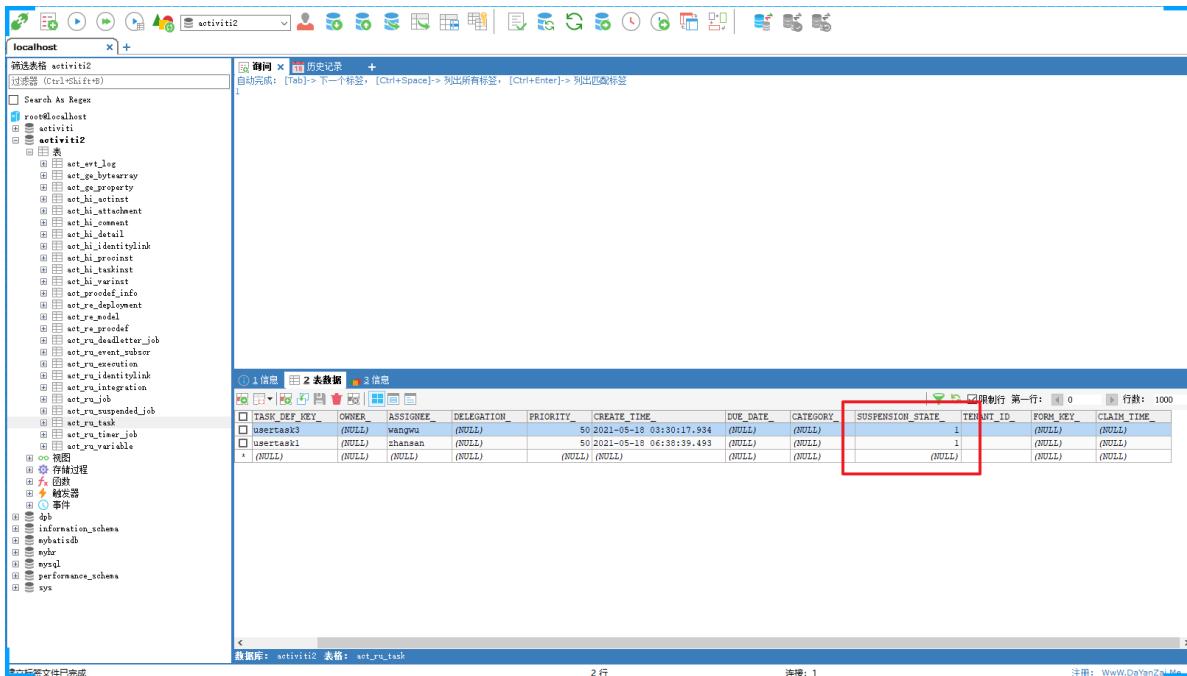
```

2021-05-18 15:18:29,920 1947 [main] DEBUG source.pooled.PooledDataSource - Returned connection 1150058854 to pool.
2021-05-18 15:18:29,920 1947 [main] DEBUG mpl.interceptor.LogInterceptor - --- CompleteTaskCmd finished -----
2021-05-18 15:18:29,920 1947 [main] DEBUG mpl.interceptor.LogInterceptor -
|  

org.activiti.engine.ActivitiException: Cannot complete a suspended task
    at org.activiti.engine.impl.cmd.NeedsActiveTaskCmd.execute(NeedsActiveTaskCmd.java:53)
    at org.activiti.engine.impl.interceptor.CommandInvoker$1.run(CommandInvoker.java:37)
    at org.activiti.engine.impl.interceptor.CommandInvoker.executeOperation(CommandInvoker.java:78)
    at org.activiti.engine.impl.interceptor.CommandInvoker.executeOperations(CommandInvoker.java:57)
    at org.activiti.engine.impl.interceptor.CommandInvoker.execute(CommandInvoker.java:42)
    at org.activiti.engine.impl.interceptor.TransactionContextInterceptor.execute(TransactionContextInterceptor.java:48)
    at org.activiti.engine.impl.interceptor.CommandContextInterceptor.execute(CommandContextInterceptor.java:63)
    at org.activiti.engine.impl.interceptor.LogInterceptor.execute(LogInterceptor.java:35)

```

我们再将挂起的流程转变为激活状态，对于的状态值会从2更新为1



然后就是业务流程可以正常处理了

1.3.2 单个实例挂起

操作流程实例对象，针对单个流程执行挂起操作，某个流程实例挂起则此流程不再继续执行，当前流程定义的其他流程实例是不受干扰的。完成该流程实例的当前任务会抛异常

```

/*
 * 单个流程实例挂起与激活
 */
@Test
public void test03(){
    // 1.获取ProcessEngine对象
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 2.获取RuntimeService
    RuntimeService runtimeService = engine.getRuntimeService();
    // 3.获取流程实例对象
    ProcessInstance processInstance =
        runtimeService.createProcessInstanceQuery()
            .processInstanceId("25001")
            .singleResult();
    // 4.获取相关状态操作
    boolean suspended = processInstance.isSuspended();
    String id = processInstance.getId();
    if(suspended){
        // 挂起-->激活
        runtimeService.activateProcessInstanceById(id);
        System.out.println("流程定义: " + id + ", 已激活");
    }else{
        // 激活-->挂起
        runtimeService.suspendProcessInstanceById(id);
        System.out.println("流程定义: " + id + ", 已挂起");
    }
}

```

然后我们可以在数据库中查看到状态的更新

The screenshot shows the MySQL Workbench interface. On the left, the database structure for 'activiti2' is displayed, including tables like act_hi_attachment, act_hi_log, and act_ru_task. On the right, a query results window titled '历史记录' (History) is open, showing a table with columns such as TASK_DEF_KEY, OWNER, ASSIGNEE, PRIORITY, CREATE_TIME, DUE_DATE, CATEGORY, SUSPENSION_STATE, TENANT_ID, FORM_KEY, and CLAIM_TIME. The table contains three rows of data, with the third row's 'ASSIGNEE' field highlighted with a red border.

2. 个人任务

2.1 分配任务责任人

2.1.1 固定分配

在进行业务流程建模的时候指定固定的任务负责人：

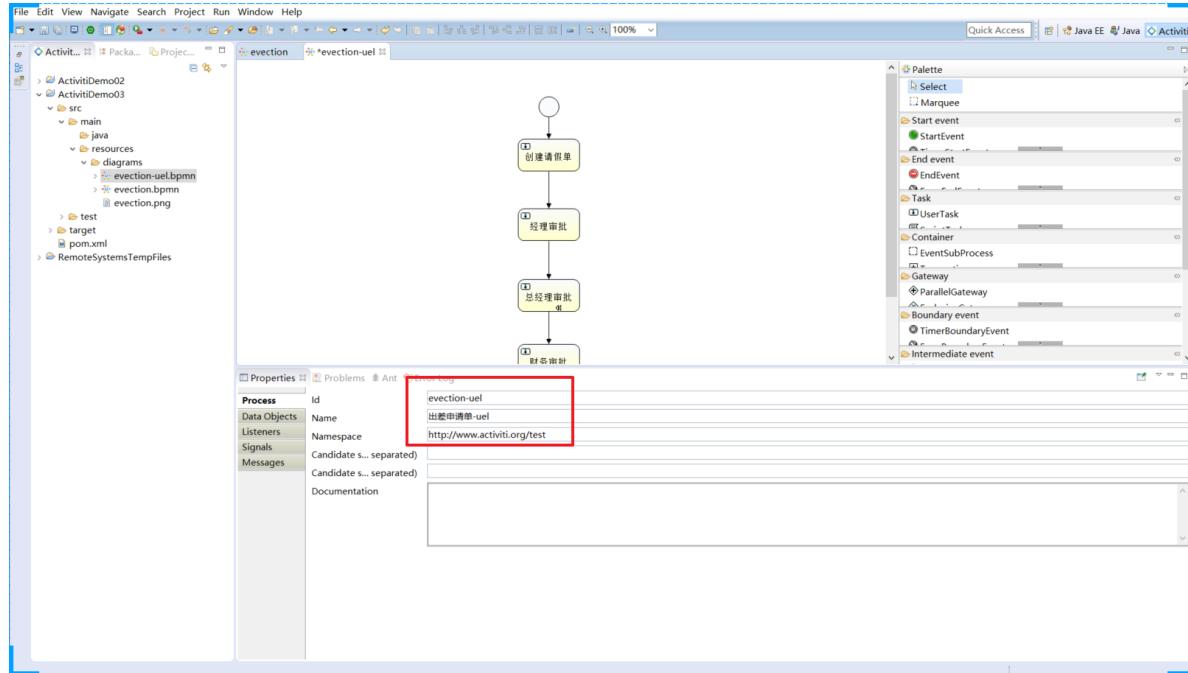
The screenshot shows the Activiti Designer interface. On the left, the project structure for 'ActivitiDemo3' is shown. In the center, a process flow diagram is displayed with four tasks: '创建请单' (Create Application), '经理审批' (Manager Approval), '总经理审批' (General Manager Approval), and '财务审批' (Finance Approval). The 'Assignee' field for the first task is highlighted with a red border and set to 'zhansan'. On the right, the 'Properties' view is open for the first task, showing the 'Assignee' field populated with 'zhansan'. The palette on the right side of the interface lists various BPMN elements like Start event, Task, and End event.

在Properties视图中，填写Assignee项为任务负责人

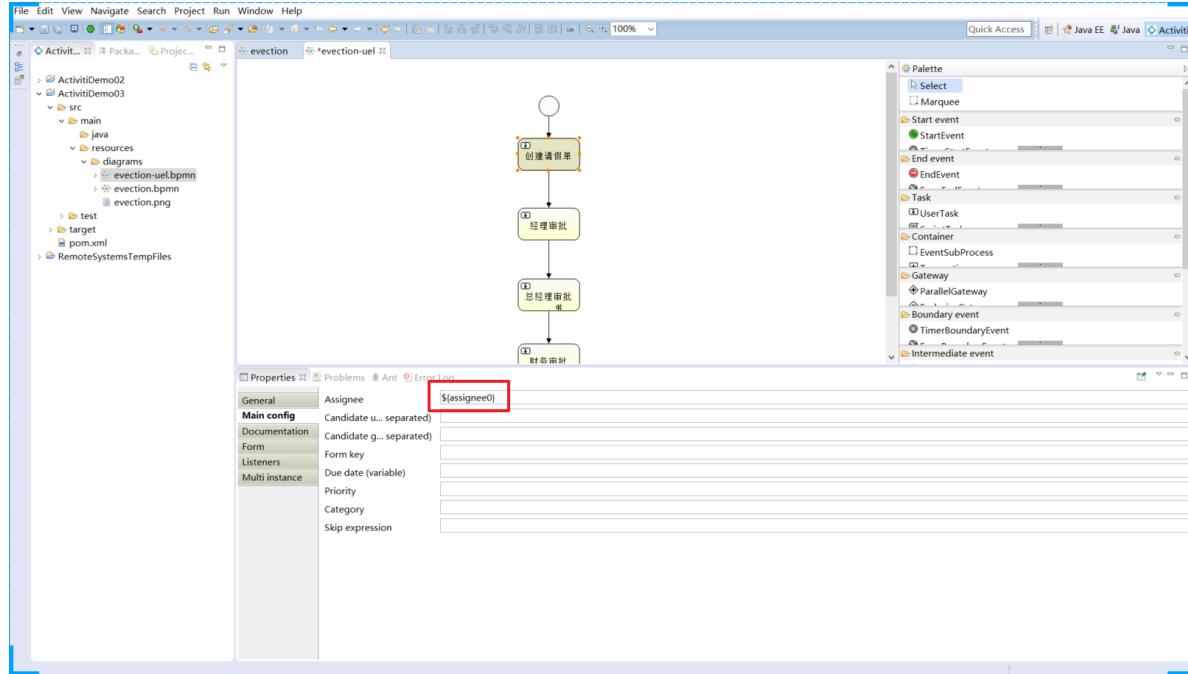
2.1.2 表达式分配

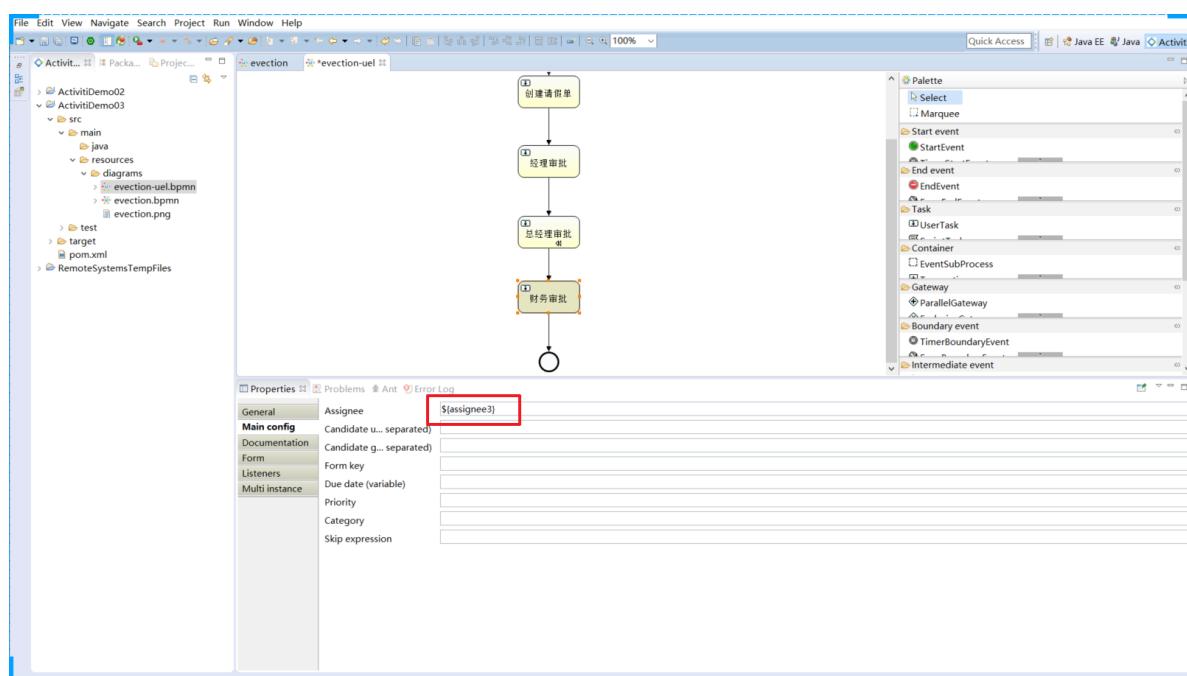
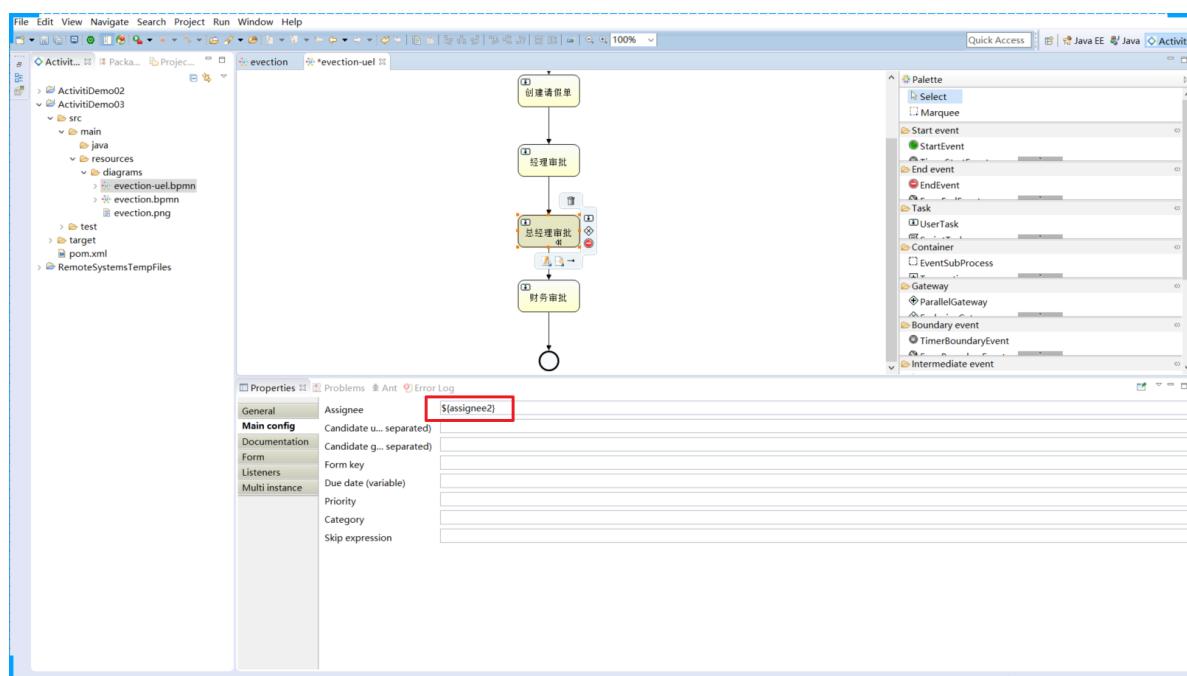
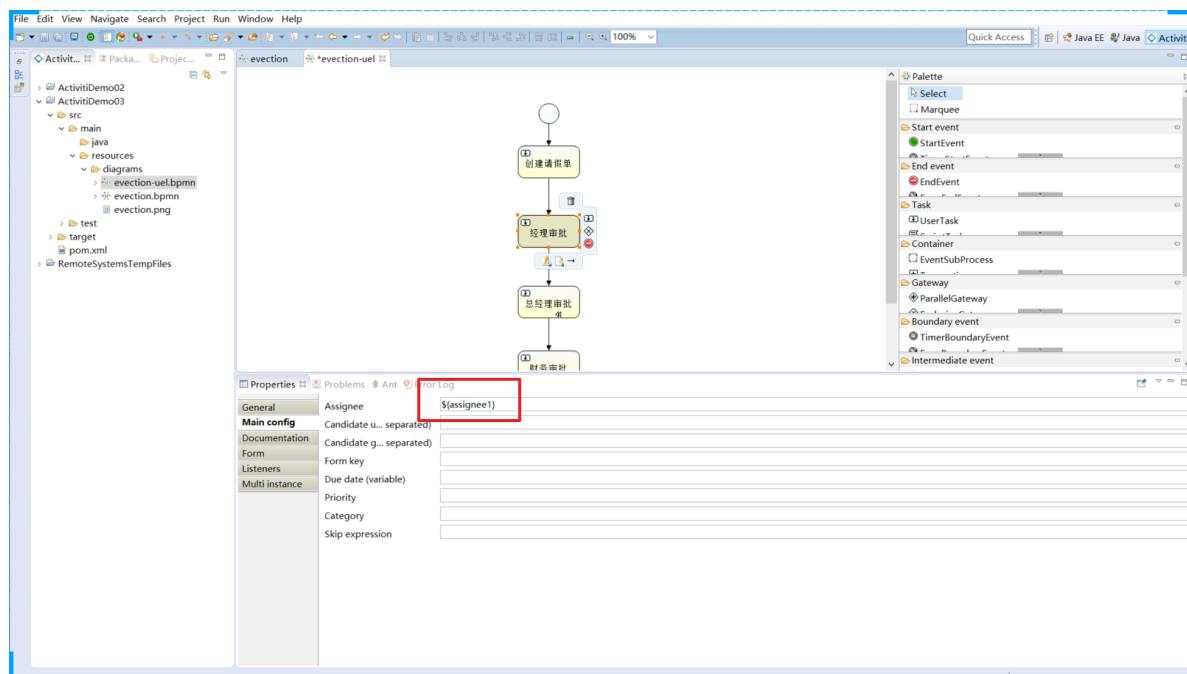
在Activiti中支持使用UEL表达式，UEL表达式是Java EE6 规范的一部分，UEL(Unified Expression Language) 即 统一表达式语言，Activiti支持两种UEL表达式：UEL-value 和UEL-method

UEL-value



在assignee中使用流程变量处理





然后我们可以来操作

首先我们需要将定义的流程部署到Activiti数据库中

```
/*
 * 先将新定义的流程部署到Activiti中数据库中
 */
@Test
public void test01(){
    // 1.获取ProcessEngine对象
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 2.获取RepositoryService进行部署操作
    RepositoryService service = engine.getRepositoryService();
    // 3.使用RepositoryService进行部署操作
    Deployment deploy = service.createDeployment()
        .addClasspathResource("bpmn/evection-uel.bpmn") // 添加bpmn资源
        .addClasspathResource("bpmn/evection-uel.png") // 添加png资源
        .name("出差申请流程-UEL")
        .deploy(); // 部署流程
    // 4.输出流程部署的信息
    System.out.println("流程部署的id:" + deploy.getId());
    System.out.println("流程部署的名称: " + deploy.getName());
}
```

部署成功后我们需要启动一个新的流程实例，然后在流程实例创建的其实关联UEL表达式

```
/*
 * 创建一个流程实例
 *      给流程定义中的 UEL表达式赋值
 */
@Test
public void test02(){
    // 获取流程引擎
    ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
    // 获取RuntimeService对象
    RuntimeService runtimeService = processEngine.getRuntimeService();
    // 设置 assignee 的取值,
    Map<String, Object> map = new HashMap<>();
    map.put("assignee0", "张三");
    map.put("assignee1", "李四");
    map.put("assignee2", "王五");
    map.put("assignee3", "赵财务");
    // 创建流程实例
    runtimeService.startProcessInstanceByKey("evection-uel", map);
}
```

启动成功后我们在 act_ru_variable 中可以看到 UEL 表达式对应的赋值信息

MySQL Workbench Screenshot:

- 左侧树状结构:** 显示了activiti2数据库中的所有表，包括act_ru_variable表。
- 右侧查询结果:** 显示了act_ru_variable表的数据，共有5行记录。每行代表一个任务的变量赋值。红色框标注了部分列：NAME、EXECUTION_ID_、PROC_INST_ID_、TASK_ID_、BYTETARRAY_ID_、DOUBLE_、LONG_、TEXT_和TEXT2_。

ID	REV_	TYPE_	NAME	EXECUTION_ID_	PROC_INST_ID_	TASK_ID_	BYTETARRAY_ID_	DOUBLE_	LONG_	TEXT_	TEXT2_
37502	1	string	assignee3	37501	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	财务	(NULL)
37503	1	string	assignee0	37501	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	经理三	(NULL)
37504	1	string	assignee2	37501	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	经理五	(NULL)
37505	1	string	assignee1	37501	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	经理四	(NULL)
(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)

UEL-method

Eclipse IDE Screenshot:

- 左侧文件树:** 显示了Activiti项目的文件结构，包括ActivitiDemo02和ActivitiDemo03目录下的src、test、target和pom.xml文件。
- 中心工作区:** 显示了一个名为'evection-uel'的BPMN图。图中包含四个任务节点：'创建请报单'、'经理审批'、'总经理审批'、'财务审批'，以及一个结束事件。箭头表示任务的执行顺序。
- 右侧Palette:** 显示了Activiti元素库，包括Start event、End event、Task、UserTask、EventSubProcess、ParallelGateway、Boundary event和TimerBoundaryEvent等。
- 下方Properties面板:** 显示了任务 '创建请报单' 的属性配置。在 'Main config' 选项卡下，'Assignee' 字段被设置为表达式 \${UserBean.getUserId()}，并用红色框标注。

userBean 是 spring 容器中的一个 bean，表示调用该 bean 的 getUserId()方法。

UEL-method 与 UEL-value 结合

再比如：

``${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。

其它

表达式支持解析基础类型、 bean、 list、 array 和 map，也可作为条件判断。

如下：

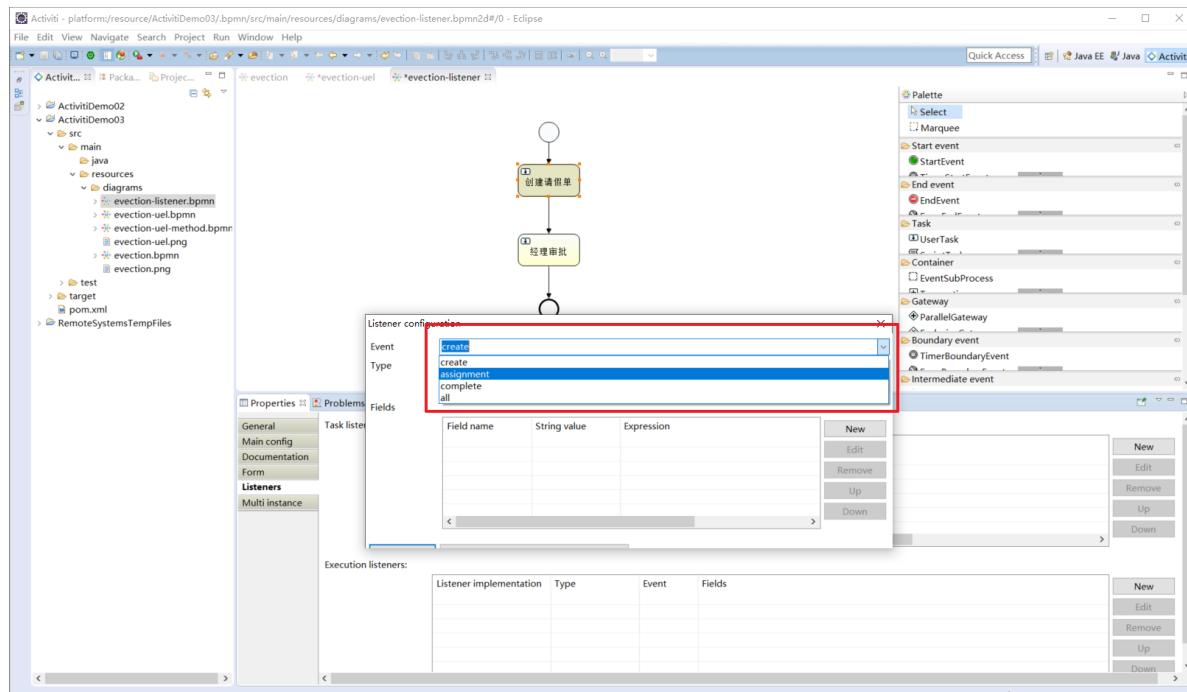
`${order.price > 100 && order.price < 250}`

2.1.3 监听器分配

可以使用监听器来完成很多Activiti的流程业务。

我们在此处使用监听器来完成负责人的指定，那么我们在流程设计的时候就不需要指定assignee

Event选项



create:任务创建后触发
assignment:任务分配后触发
Delete:任务完成后触发
All: 所有事件都触发

自定义的监听器

```
import org.activiti.engine.delegate.DelegateTask;
import org.activiti.engine.delegate.TaskListener;

public class MyTaskListener implements TaskListener {
    @Override
    public void notify(DelegateTask delegateTask) {
        if("创建请假单".equals(delegateTask.getName())
        && "create".equals(delegateTask.getEventName())){
            // 指定任务的负责人
            delegateTask.setAssignee("张三-Listener");
        }
    }
}
```

```
}
```

测试代码

```
/**  
 * 先将新定义的流程部署到Activiti中数据库中  
 */  
@Test  
public void test01(){  
    // 1.获取ProcessEngine对象  
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();  
    // 2.获取RepositoryService进行部署操作  
    RepositoryService service = engine.getRepositoryService();  
    // 3.使用RepositoryService进行部署操作  
    Deployment deploy = service.createDeployment()  
        .addClasspathResource("bpnn/evection-listener.bpmn") // 添加bpnn资源  
        .addClasspathResource("bpnn/evection-listener.png") // 添加png资源  
        .name("出差申请流程-UEL")  
        .deploy() // 部署流程  
    // 4.输出流程部署的信息  
    System.out.println("流程部署的id:" + deploy.getId());  
    System.out.println("流程部署的名称: " + deploy.getName());  
}  
  
/**  
 * 创建一个流程实例  
 *      给流程定义中的 UEL表达式赋值  
 */  
@Test  
public void test02(){  
    // 获取流程引擎  
    ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();  
    // 获取RuntimeService对象  
    RuntimeService runtimeService = processEngine.getRuntimeService();  
  
    // 创建流程实例  
    runtimeService.startProcessInstanceByKey("evection-listener");  
}
```

The screenshot shows the Eclipse IDE interface with the following details:

- Project Explorer:** Shows the project structure for "ActivitiDemo01".
- Code Editor:** Displays the file "MyTaskListener.java" containing Java code for a task listener.
- Variables View:** Shows the state of variables during the execution of the code, including the current task and its assignee.
- Console:** Shows the message "Build completed successfully in 1 s 147 ms (moments ago)".
- Event Log:** Shows the message "9:1 CRLF UTF-8 4 spaces".

The screenshot shows the SQLyog Ultimate database management tool interface with the following details:

- Database:** activiti2
- Schemas:** activiti2
- Tables:** act_ru_task
- Data View:** A grid view of the 'act_ru_task' table with columns: ID_, REV_, EXECUTION_ID_, PROC_INST_ID_, PROC_DEF_ID_, NAME_, PARENT_TASK_ID_, DESCRIPTION_, TASK_DEF_KEY_, OWNER_, ASSIGNEE_, and DELEGATE_. The table contains several rows of task data.

2.2 查询任务

查询任务负责人的待办任务

代码如下：

```
// 查询当前个人待执行的任务
@Test
public void findPersonalTaskList() {
    // 流程定义key
    String processDefinitionKey = "myEvection1";
    // 任务负责人
    String assignee = "张三";
```

```

// 获取TaskService
TaskService taskService = processEngine.getTaskService();
List<Task> taskList = taskService.createTaskQuery()
    .processDefinitionKey(processDefinitionKey)
    .includeProcessVariables()
    .taskAssignee(assignee)
    .list();
for (Task task : taskList) {
    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());
}
}

```

关联 businessKey

需求：

在 activiti 实际应用时，查询待办任务可能要显示出业务系统的一些相关信息。

比如：查询待审批出差任务列表需要将出差单的日期、 出差天数等信息显示出来。

出差天数等信息在业务系统中存在，而并没有在 activiti 数据库中存在，所以是无法通过 activiti 的 api 查询到出差天数等信息。

实现：

在查询待办任务时，通过 businessKey（业务标识）关联查询业务系统的出差单表，查询出出差天数等信息。

```

@Test
public void findProcessInstance(){
    // 获取processEngine
    ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
    // 获取TaskService
    TaskService taskService = processEngine.getTaskService();
    // 获取RuntimeService
    RuntimeService runtimeService = processEngine.getRuntimeService();
    // 查询流程定义的对象
    Task task = taskService.createTaskQuery()
        .processDefinitionKey("myEvection1")
        .taskAssignee("张三")
        .singleResult();
    // 使用task对象获取实例id
    String processInstanceId = task.getProcessInstanceId();
    // 使用实例id，获取流程实例对象
    ProcessInstance processInstance =
        runtimeService.createProcessInstanceQuery()
            .processInstanceId(processInstanceId)
            .singleResult();
    // 使用processInstance，得到 businessKey
    String businessKey = processInstance.getBusinessKey();

    System.out.println("businessKey==" + businessKey);
}

```

2.3 办理任务

注意：在实际应用中，完成任务前需要校验任务的负责人是否具有该任务的办理权限。

```
/**  
 * 完成任务，判断当前用户是否有权限  
 */  
@Test  
public void compleTask() {  
    //任务id  
    String taskId = "15005";  
    // 任务负责人  
    String assingee = "张三";  
    //获取processEngine  
    ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();  
    // 创建TaskService  
    TaskService taskService = processEngine.getTaskService();  
    // 完成任务前，需要校验该负责人可以完成当前任务  
    // 校验方法：  
    // 根据任务id和任务负责人查询当前任务，如果查到该用户有权限，就完成  
    Task task = taskService.createTaskQuery()  
        .taskId(taskId)  
        .taskAssignee(assingee)  
        .singleResult();  
    if(task != null){  
        taskService.complete(taskId);  
        System.out.println("完成任务");  
    }  
}
```

3.流程变量

3.1、什么是流程变量

流程变量在 activiti 中是一个非常重要的角色，流程运转有时需要靠流程变量，业务系统和 activiti 结合时少不了流程变量，流程变量就是 activiti 在管理工作流时根据管理需要而设置的变量。

比如：在出差申请流程流转时如果出差天数大于 3 天则由总经理审核，否则由人事直接审核，出差天数就可以设置为流程变量，在流程流转时使用。

注意：虽然流程变量中可以存储业务数据可以通过 activiti 的 api 查询流程变量从而实现 查询业务数据，但是不建议这样使用，因为业务数据查询由业务系统负责， activiti 设置流程变量是为了流程执行需要而创建。

3.2、流程变量类型

如果将 pojo 存储到流程变量中，必须实现序列化接口 serializable，为了防止由于新增字段无法反序列化，需要生成 serialVersionUID。

Table 15.9. Variable Types

Type name	Description
string	Value is threaded as a <code>java.lang.String</code> . Raw JSON-type.
integer	Value is threaded as a <code>java.lang.Integer</code> . When writing, the value is converted to a string.
short	Value is threaded as a <code>java.lang.Short</code> . When writing, the value is converted to a string.
long	Value is threaded as a <code>java.lang.Long</code> . When writing, the value is converted to a string.
double	Value is threaded as a <code>java.lang.Double</code> . When writing, the value is converted to a string.
boolean	Value is threaded as a <code>java.lang.Boolean</code> . When writing, the value is converted to a string.
date	Value is treated as a <code>java.util.Date</code> . When writing, the value is converted to a string.
binary	Binary variable, treated as an array of bytes. The value is a <code>byte[]</code> .
serializable	Serialized representation of a Serializable Java-object. URL pointing to the raw binary stream. All serializable variables of this type. https://blog.csdn.net/cqq_20042935

3.3、流程变量作用域

流程变量的作用域可以是一个流程实例(processInstance)，或一个任务(task)，或一个执行实例(execution)

3.3.1、global变量

流程变量的默认作用域是流程实例。当一个流程变量的作用域为流程实例时，可以称为 global 变量

注意：

如： Global变量：userId (变量名) 、 zhangsan (变量值)

global 变量中变量名不允许重复，设置相同名称的变量，后设置的值会覆盖前设置的变量值。

3.3.2、local变量

任务和执行实例仅仅是针对一个任务和一个执行实例范围，范围没有流程实例大，称为 local 变量。

Local 变量由于在不同的任务或不同的执行实例中，作用域互不影响，变量名可以相同没有影响。Local 变量名也可以和 global 变量名相同，没有影响。

3.4、流程变量的使用方法

3.4.1、在属性上使用UEL表达式

可以在 assignee 处设置 UEL 表达式，表达式的值为任务的负责人，比如： \${assignee}， assignee 就是一个流程变量名称。

Activiti获取UEL表达式的值，即流程变量assignee的值，将assignee的值作为任务的负责人进行任务分配

3.4.2、在线上使用UEL表达式

可以在连线上设置UEL表达式，决定流程走向。

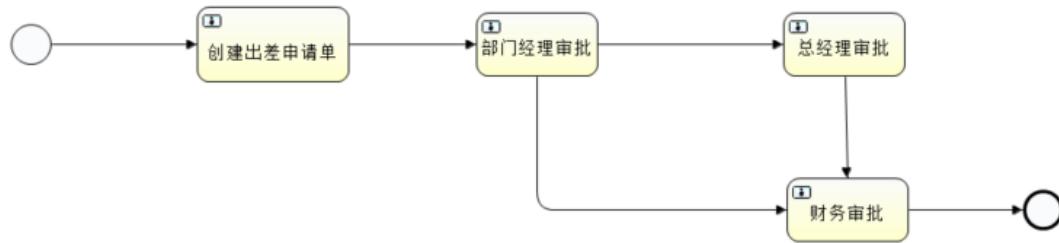
比如：\${price<10000}。price就是一个流程变量名称，uel表达式结果类型为布尔类型。

如果UEL表达式是true，要决定流程执行走向。

3.5 流程变量使用

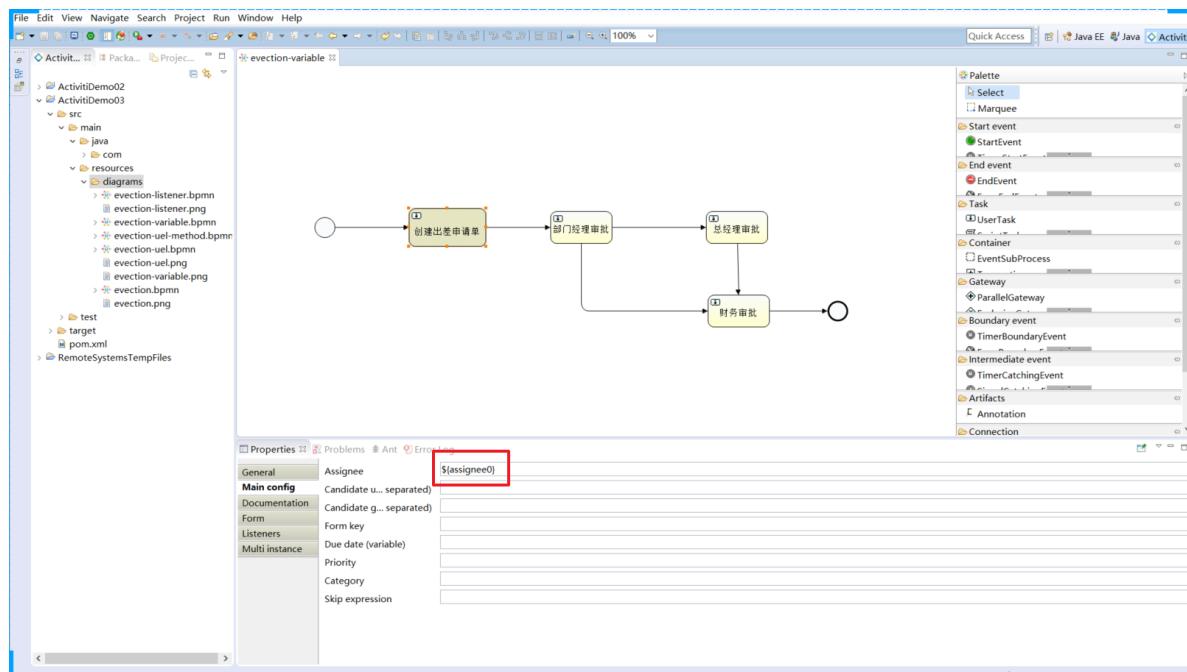
3.5.1 需求

员工创建出差申请单，由部门经理审核，部门经理申请通过后3天以下由财务直接审批，3天以上先由总经理审批，总经理审批通过后再由财务审批。

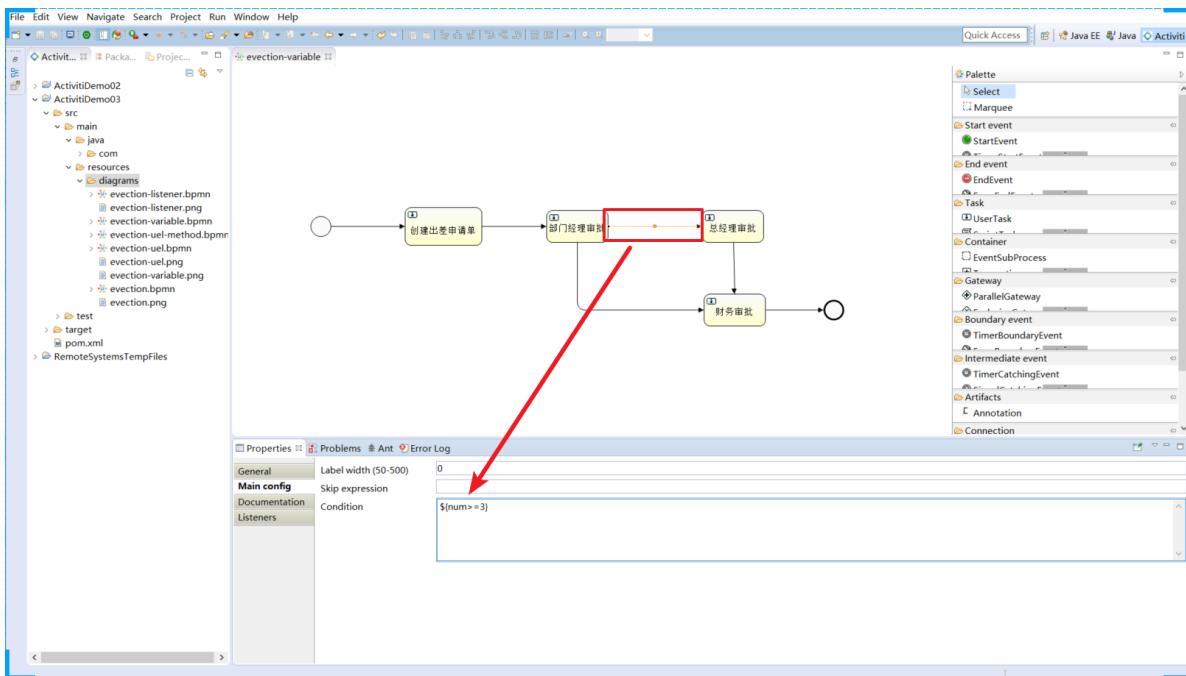


3.5.2 流程定义

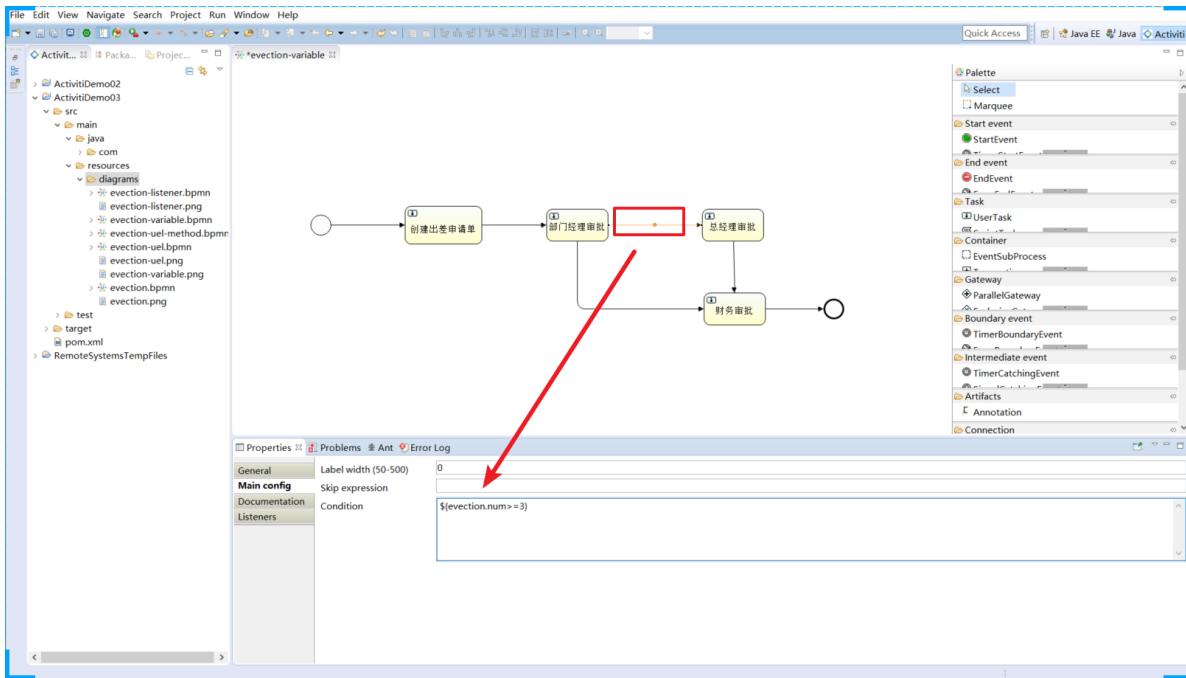
先通过UEL-value来设置负责人



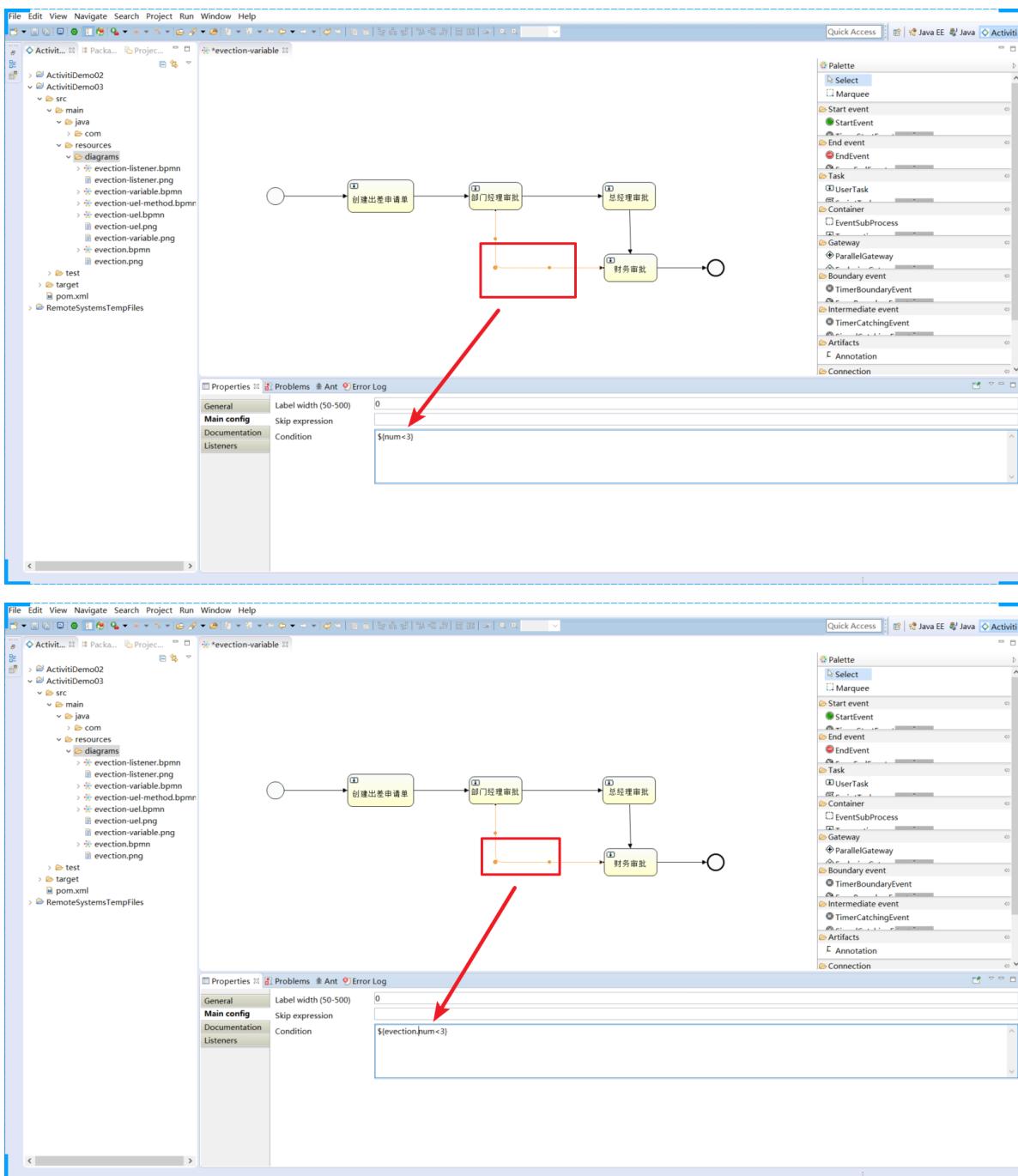
然后在分支线上来设置条件



那么还可以通过对象参数命名，比如 `evection.num`:



另一根线对应的设置



然后可以将相关的资源文件拷贝到项目中，

3.5.3 使用Global变量

接下来使用Global变量控制流程

3.5.3.1 POJO创建

首先创建POJO对象

```
/**
 * 出差申请的POJO对象
 */
@Data
public class Evection {

    private long id;
}
```

```

private String evetionName;

/**
 * 出差的天数
 */
private double num;

private Date beginDate;

private Date endDate;

private String destination;

private String reson;
}

```

3.5.3.2 流程的部署

```

/**
 * 部署流程
 */
@Test
public void test01(){
    // 1.获取ProcessEngine对象
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    // 2.获取RepositoryService进行部署操作
    RepositoryService service = engine.getRepositoryService();
    // 3.使用RepositoryService进行部署操作
    Deployment deploy = service.createDeployment()
        .addClasspathResource("bpnn/evection-variable.bpmn") // 添加bpnn资源
        .addClasspathResource("bpnn/evection-variable.png") // 添加png资源
        .name("出差申请流程-流程变量")
        .deploy(); // 部署流程
    // 4.输出流程部署的信息
    System.out.println("流程部署的id:" + deploy.getId());
    System.out.println("流程部署的名称: " + deploy.getName());
}

```

3.5.3.3 设置流程变量

a.启动时设置流程变量

在启动流程时设置流程变量，变量的作用域是整个流程实例。

```

/**
 * 启动流程实例，设置流程变量
 */
@Test

```

```

public void test02(){
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    RuntimeService runtimeService = engine.getRuntimeService();
    // 流程定义key
    String key = "evection-variable";
    // 创建变量集合
    Map<String, Object> variables = new HashMap<>();
    // 创建出差对象 POJO
    Evection evection = new Evection();
    // 设置出差天数
    evection.setNum(4d);
    // 定义流程变量到集合中
    variables.put("evection", evection);
    // 设置assignee的取值
    variables.put("assignee0", "张三1");
    variables.put("assignee1", "李四1");
    variables.put("assignee2", "王五1");
    variables.put("assignee3", "赵财务1");
    ProcessInstance processInstance =
        runtimeService.startProcessInstanceByKey(key, variables);
    // 输出信息
    System.out.println("获取流程实例名称: "+processInstance.getName());
    System.out.println("流程定义ID: " +
        processInstance.getProcessDefinitionId());
}

```

完成任务

```

/**
 * 完成任务
 */
@Test
public void test03(){
    String key = "evection-variable";
    String assignee = "李四1";
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    TaskService taskService = engine.getTaskService();
    Task task = taskService.createTaskQuery()
        .processDefinitionKey(key)
        .taskAssignee(assignee)
        .singleResult();
    if(task != null){
        taskService.complete(task.getId());
        System.out.println("任务执行完成... ");
    }
}

```

通过startProcessInstanceByKey方法设置流程变量的作用域是一个流程实例，流程变量使用Map存储，同一个流程实例map中的key相同，后者会覆盖前者

b.任务办理时设置

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。

这里需要在创建出差单任务完成时设置流程变量

```
/*
 * 启动流程实例，设置流程变量
 */
@Test
public void test02(){
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    RuntimeService runtimeService = engine.getRuntimeService();
    // 流程定义key
    String key = "evection-variable";
    // 创建变量集合
    Map<String, Object> variables = new HashMap<>();

    // 设置assignee的取值
    variables.put("assignee0", "张三1");
    variables.put("assignee1", "李四1");
    variables.put("assignee2", "王五1");
    variables.put("assignee3", "赵财务1");
    ProcessInstance processInstance =
        runtimeService.startProcessInstanceByKey(key, variables);
    // 输出信息
    System.out.println("获取流程实例名称: "+processInstance.getName());
    System.out.println("流程定义ID: " +
processInstance.getProcessDefinitionId());
}

/*
 * 完成任务
 */
@Test
public void test03(){
    String key = "evection-variable";
    String assignee = "李四1";
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    TaskService taskService = engine.getTaskService();
    Task task = taskService.createTaskQuery()
        .processDefinitionKey(key)
        .taskAssignee(assignee)
        .singleResult();

    Map<String, Object> variables = new HashMap<>();
    // 创建出差对象 POJO
    Evection evection = new Evection();
    // 设置出差天数
    evection.setNum(4d);
    // 定义流程变量到集合中
    variables.put("evection", evection);

    if(task != null){
        taskService.complete(task.getId(), variables);
        System.out.println("任务执行完成...");
    }
}
```

```
    }  
}
```

说明：

通过当前任务设置流程变量，需要指定当前任务id，如果当前执行的任务id不存在则抛出异常。

任务办理时也是通过map<key,value>设置流程变量，一次可以设置多个变量。

c.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, "evection", evection);  
    // 一次设置多个值  
    // runtimeService.setVariables(executionId, variables)  
}
```

注意：

executionId必须当前未结束 流程实例的执行id，通常此id设置流程实例 的id。也可以通过 runtimeService.getVariable()获取流程变量。

d.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)  
}
```

注意：

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。

3.5.4 设置local流程变量

3.5.4.1、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。

```
/*
*处理任务时设置local流程变量
*/
@Test
public void compleTask() {
    //任务id
    String taskId = "1404";
    // 获取processEngine
    ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
    TaskService taskService = processEngine.getTaskService();
    // 定义流程变量
    Map<String, Object> variables = new HashMap<String, Object>();
    Evection evection = new Evection();
    evection.setNum(3d);
    // 定义流程变量
    Map<String, Object> variables = new HashMap<String, Object>();
    // 变量名是holiday，变量值是holiday对象
    variables.put("evection", evection);
    // 设置local变量，作用域为该任务
    taskService.setVariablesLocal(taskId, variables);
    // 完成任务
    taskService.complete(taskId);
}
```

说明：

设置作用域为任务的local变量，每个任务可以设置同名的变量，互不影响。

3.5.4.2、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)
}
```

注意：

任务id必须是当前待办任务id，act_ru_task中存在。

3.5.4.3、Local变量测试1

如果上边例子中设置global变量改为设置local变量是否可行？为什么？

Local变量在任务结束后无法在当前流程实例执行中使用，如果后续的流程执行需要用到此变量则会报错。

3.5.4.4、Local变量测试2

在部门经理审核、总经理审核、财务审核时设置local变量，可通过historyService查询每个历史任务时将流程变量的值也查询出来。

代码如下：

```
// 创建历史任务查询对象
HistoricTaskInstanceQuery historicTaskInstanceQuery =
historyService.createHistoricTaskInstanceQuery();
// 查询结果包括 local变量
historicTaskInstanceQuery.includeTaskLocalVariables();
for (HistoricTaskInstance historicTaskInstance : list) {
    System.out.println("=====");
    System.out.println("任务id: " + historicTaskInstance.getId());
    System.out.println("任务名称: " + historicTaskInstance.getName());
    System.out.println("任务负责人: " + historicTaskInstance.getAssignee());
    System.out.println("任务local变量: "+
historicTaskInstance.getTaskLocalVariables());
}
```

注意：查询历史流程变量，特别是查询pojo变量需要经过反序列化，不推荐使用。

4.组任务

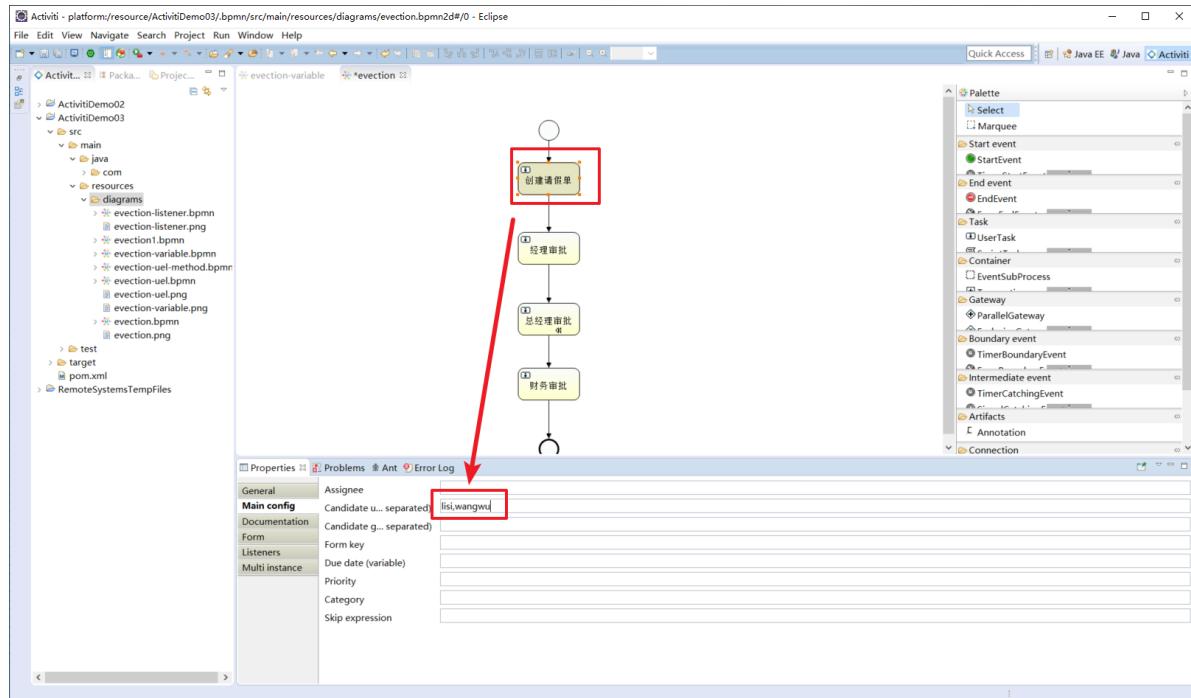
4.1、需求

在流程定义中在任务结点的 assignee 固定设置任务负责人，在流程定义时将参与者固定设置在.bpmn 文件中，如果临时任务负责人变更则需要修改流程定义，系统可扩展性差。

针对这种情况可以给任务设置多个候选人，可以从候选人中选择参与者来完成任务。

4.2、设置任务候选人

在流程图中任务节点的配置中设置 candidate-users(候选人)，多个候选人之间用逗号分开。



查看bpmn文件

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

我们可以看到部门经理的审核人已经设置为 lisi,wangwu 这样的一组候选人，可以使用 activiti:candidateUsers="用户 1,用户 2,用户 3" 的这种方式来实现设置一组候选人

4.3、组任务

4.3.1、组任务办理流程

a、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。

b、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？

需要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。

c、查询个人任务

查询方式同个人任务部分，根据assignee查询用户负责的个人任务。

d、办理个人任务

4.3.2、查询组任务

根据候选人查询组任务

```
/**  
 * 查询组任务  
 */  
@Test  
public void test03(){  
    String key = "evection1";  
    String candidateUser = "lisi";  
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();  
    TaskService taskService = engine.getTaskService();  
    List<Task> list = taskService.createTaskQuery()  
        .processDefinitionKey(key)  
        .taskCandidateUser(candidateUser)  
        .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("负责人: " + task.getAssignee());  
        System.out.println("任务名称: " + task.getName());  
    }  
}
```

4.3.3、拾取组任务

候选人员拾取组任务后该任务变为自己的个人任务。

```
/**  
 * 候选人 拾取任务  
 */  
@Test  
public void test04(){  
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();  
    TaskService taskService = engine.getTaskService();  
    String taskId = "72505";  
    // 候选人  
    String userId = "lisi";  
    // 拾取任务  
    Task task = taskService.createTaskQuery()  
        .taskId(taskId)  
        .taskCandidateUser(userId) // 根据候选人查询  
        .singleResult();  
    if(task != null){  
        // 可以拾取任务  
        taskService.claim(taskId,userId);  
        System.out.println("拾取成功");  
    }  
}
```

4.3.4、查询个人待办任务

查询方式同个人任务查询

```
@Test
public void test03(){
    String key = "eviction1";
    String candidateUser = "lisi";
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    TaskService taskService = engine.getTaskService();
    List<Task> list = taskService.createTaskQuery()
        .processDefinitionKey(key)
        //.taskCandidateUser(candidateUser)
        //.taskCandidateOrAssigned(candidateUser)
        .taskAssignee(candidateUser)
        .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("负责人:" + task.getAssignee());
        System.out.println("任务名称: " + task.getName());
    }
}
```

4.3.5、办理个人任务

同个人任务办理

```
/**
 * 完成个人任务
 */
@Test
public void test05(){
    String taskId = "72505";
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    TaskService taskService = engine.getTaskService();
    taskService.complete(taskId);
    System.out.println("完成任务: " + taskId);
}
```

4.3.6、归还组任务

如果个人不想办理该组任务，可以归还组任务，归还后该用户不再是该任务的负责人

```
/**
 * 归还任务
 */
@Test
public void test06(){
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    TaskService taskService = engine.getTaskService();
```

```

String taskId = "75002";
String userId= "zhangsan";
Task task = taskService.createTaskQuery()
    .taskId(taskId)
    .taskAssignee(userId)
    .singleResult();
if(task != null){
    // 如果设置为null, 归还组任务, 任务没有负责人
    taskService.setAssignee(taskId,null);
}
}

```

4.3.7、任务交接

任务负责人将任务交给其他负责人来处理

```

/**
 * 任务交接
 */
@Test
public void test07(){
    ProcessEngine engine = ProcessEngines.getDefaultProcessEngine();
    TaskService taskService = engine.getTaskService();
    String taskId = "75002";
    String userId= "zhangsan";
    Task task = taskService.createTaskQuery()
        .taskId(taskId)
        .taskAssignee(userId)
        .singleResult();
    if(task != null){
        // 设置该任务的新的负责人
        taskService.setAssignee(taskId,"赵六");
    }
}

```

4.3.8、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，有几个候选人就插入几个

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

5.网关

网关用来控制流程的流向

5.1 排他网关ExclusiveGateway

5.1.1 什么是排他网关：

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。

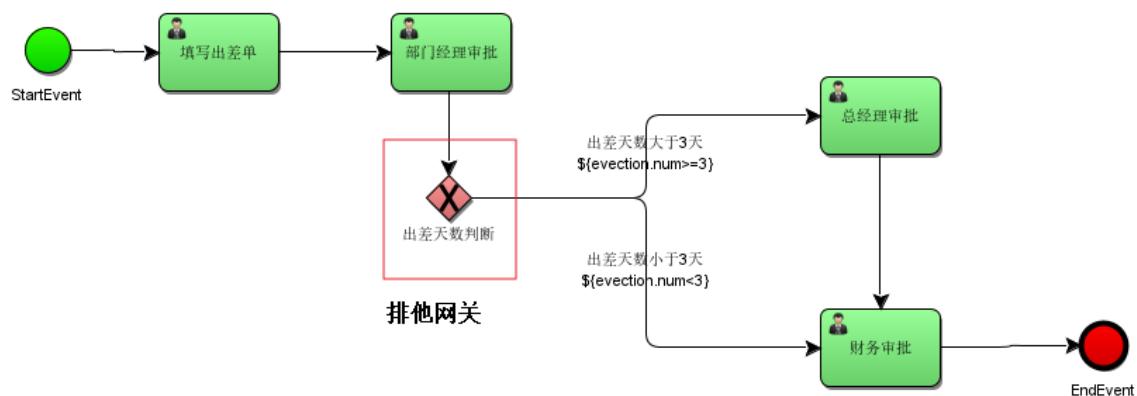
注意：排他网关只会选择一个为true的分支执行。如果有两个分支条件都为true，排他网关会选择id值较小的一条分支去执行。

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条件下设置分支条件。

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
at
org.activiti.engine.impl.bpmn.behavior.ExclusiveGatewayActivityBehavior.leave(ExclusiveGatewayActivityBehavior.java:85)
```

5.1.2 流程定义

排他网关图标，红框内：



5.1.3 测试

在部门经理审核后，走排他网关，从排他网关出来的分支有两条，一条是判断出差天数是否大于3天，另一条是判断出差天数是否小于等于3天。

设置分支条件时，如果所有分支条件都不是true，报错：

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

at
org.activiti.engine.impl.bpmn.behavior.ExclusiveGatewayActivityBehavior.leave(ExclusiveGatewayActivityBehavior.java:85)
```

5.2 并行网关ParallelGateway

5.2.1 什么是并行网关

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。

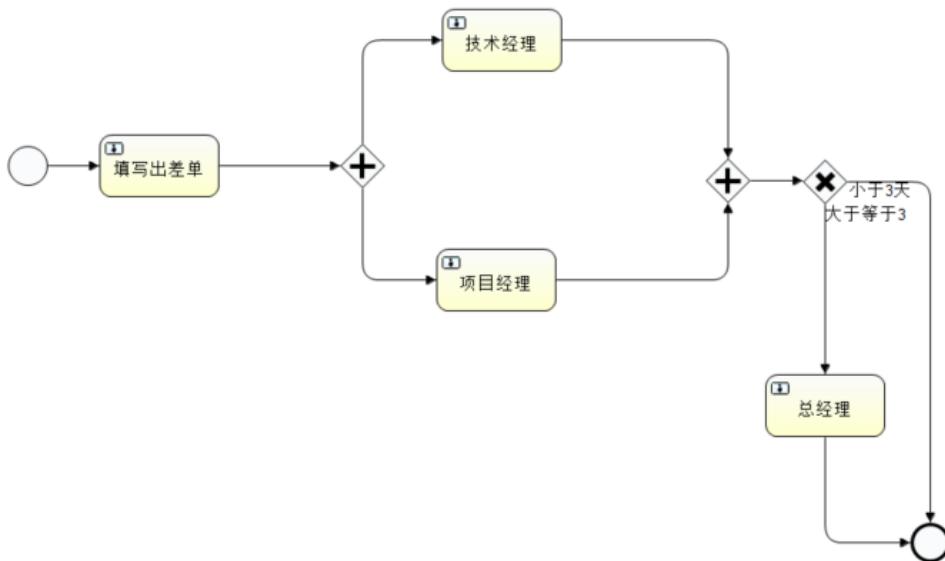
| join汇聚：

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，平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。

例子：



说明：

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和项目经理，act_ru_execution还有一条记录表示该流程实例。

待技术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。

5.2.2 流程定义

并行网关图标，红框内：



5.2.3 测试

当执行到并行网关数据库跟踪如下：

当前任务表：SELECT * FROM act_ru_task

ID	REV_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	NAME_	PARENT_TASK_ID_	DESCRIPTION_	TASK_DEF_KEY_
7504	1	5005	5001	parallel:1:2503	技术经理	(NULL)	(NULL)	_15
7507	1	7502	5001	parallel:1:2503	项目经理	(NULL)	(NULL)	_16

上图中：有两个任务当前执行。

查询流程实例执行表：SELECT * FROM act_ru_execution

ID_	REV_	PROC_INST_ID_	BUSINESS_KEY_	PARENT_ID_	PROC_DEF_ID_	SUPER_EXEC_	ROOT_PROC_INST_ID_	ACT_ID_
5001	2	5001	(NULL)	(NULL)	parallel:1:2503	(NULL)	5001	(NULL)
5005	2	5001	(NULL)	5001	parallel:1:2503	(NULL)	5001	_15
7502	1	5001	(NULL)	5001	parallel:1:2503	(NULL)	5001	_16

上图中，说明当前流程实例有多个分支(两个)在运行。

对并行任务的执行：

并行任务执行不分前后，由任务的负责人去执行即可。

执行技术经理任务后，查询当前任务表 SELECT * FROM act_ru_task

ID_	REV_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	NAME_
7507	1	7502	5001	parallel:1:2503	项目经理

已完成的技术经理任务在当前任务表act_ru_task已被删除。

在流程实例执行表：SELECT * FROM act_ru_execution有中多个分支存在且有并行网关的汇聚结点。

ID_	REV_	PROC_INST_ID_	BUSINESS_KEY_	PARENT_ID_	PROC_DEF_ID_	SUPER_EXEC_	ROOT_PROC_INST_ID_	ACT_ID_
5001	3	5001	(NULL)	(NULL)	parallel:1:2503	(NULL)	5001	(NULL)
5005	3	5001	(NULL)	5001	parallel:1:2503	(NULL)	5001	19
7502	1	5001	(NULL)	5001	parallel:1:2503	(NULL)	5001	16

有并行网关的汇聚结点：说明有一个分支已经到汇聚，等待其它的分支到达。

当所有分支任务都完成，都到达汇聚结点后：

流程实例执行表：SELECT * FROM act_ru_execution，执行流程实例已经变为总经理审批，说明流程执行已经通过并行网关

ID_	REV_	PROC_INST_ID_	BUSINESS_KEY_	PARENT_ID_	PROC_DEF_ID_	SUPER_EXEC_	ROOT_PROC_INST_ID_	ACT_ID_
5001	4	5001	(NULL)	(NULL)	parallel:1:2503	(NULL)	5001	(NULL)
7502	2	5001	(NULL)	5001	parallel:1:2503	(NULL)	5001	34

总结：所有分支到达汇聚结点，并行网关执行完成。

5.3 包含网关InclusiveGateway

5.3.1 什么是包含网关

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。

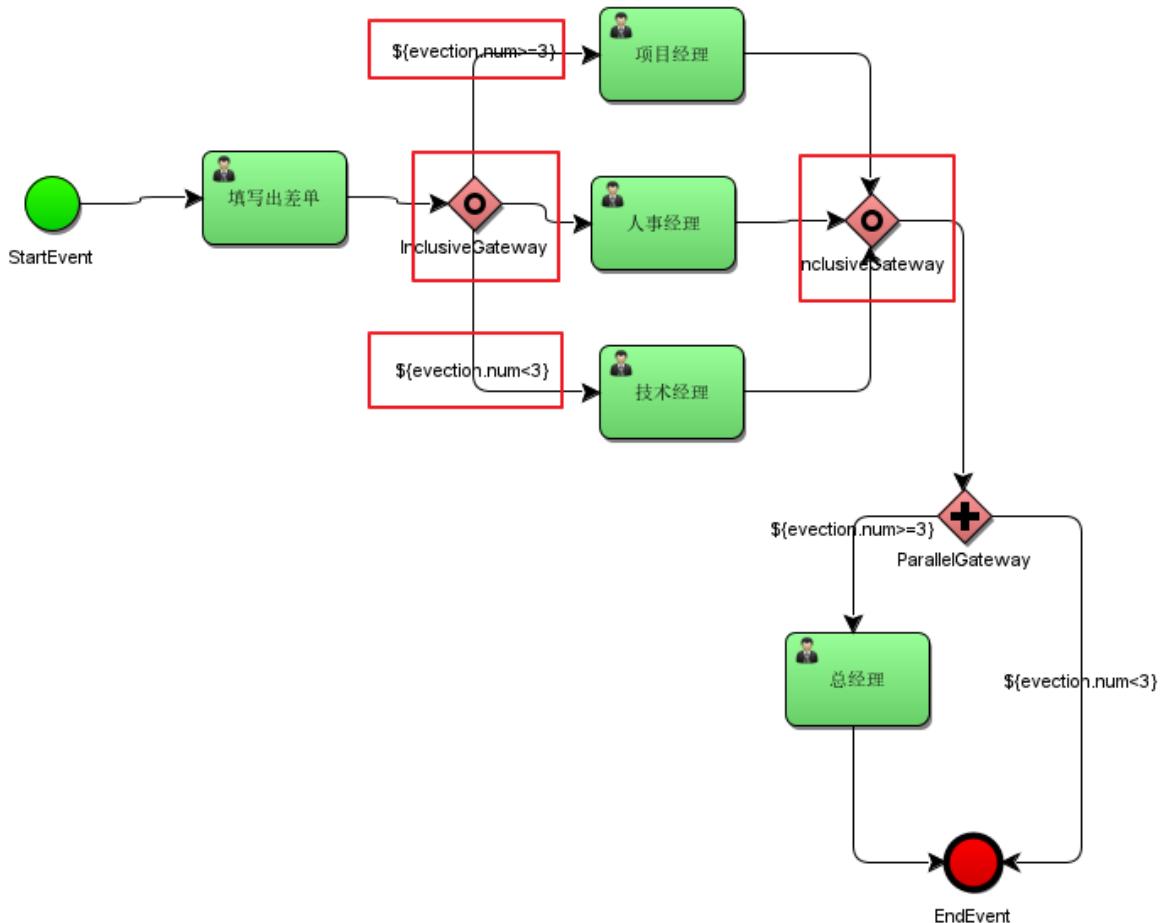
5.3.2 流程定义：

出差申请大于等于3天需要由项目经理审批，小于3天由技术经理审批，出差申请必须经过人事经理审批。

包含网关图标，红框内：



定义流程：



注意：通过包含网关的每个分支的连线上设置condition条件。

5.3.3 测试

如果包含网关设置的条件中，流程变量不存在，报错；

```
org.activiti.engine.ActivitiException: Unknown property used in expression:  
 ${evection.num>=3}
```

需要在流程启动时设置流程变量evection.num。

1)、当流程执行到第一个包含网关后，会根据条件判断，当前要走哪几个分支：

流程实例执行表：SELECT * FROM act_ru_execution

ID_	REV_	PROC_INST_ID_	BUSINESS_KEY_	PARENT_ID_	PROC_DEF_ID_	SUPER_EXEC_	ROOT_PROC_INST_ID_	ACT_ID_
2501	2	2501	(NULL)	(NULL)	inclusive:1:3	(NULL)	2501	(NULL)
2505	2	2501	(NULL)	2501	inclusive:1:3	(NULL)	2501	_13
5002	1	2501	(NULL)	2501	inclusive:1:3	(NULL)	2501	5
(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)

第一条记录：包含网关分支。

后两条记录代表两个要执行的分支：

ACT_ID = "_13" 代表项目经理神品

ACT_ID = "_5" 代表 人事经理审批

当前任务表： ACT_RU_TASK

ID	REV	EXECUTION_ID	PROC_INST_ID	PROC_DEF_ID	NAME	PARENT_TASK_ID	DESCRIPTION	TASK_DEF_KEY
5004	1	2505	2501	inclusive:1:3	项目经理	(NULL)	(NULL)	_13
5007	1	5002	2501	inclusive:1:3	人事经理	(NULL)	(NULL)	_5
(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)

上图中，项目经理审批、人事经理审批 都是当前的任务，在并行执行。

如果有一个分支执行先走到汇聚结点的分支，要等待其它执行分支走到汇聚。

2) 、先执行项目经理审批，然后查询当前任务表： ACT_RU_TASK

ID	REV	EXECUTION_ID	PROC_INST_ID	PROC_DEF_ID	NAME	PARENT_TASK_ID	DESCRIPTION	TASK_DEF_KEY
5007	1	5002	2501	inclusive:1:3	人事经理	(NULL)	(NULL)	_5
(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)

当前任务还有人事经理审批需要处理。

流程实例执行表：SELECT * FROM act_ru_execution

ID	REV	PROC_INST_ID	BUSINESS_KEY	PARENT_ID	PROC_DEF_ID	SUPER_EXEC	ROOT_PROC_INST_ID	ACT_ID
2501	3	2501	(NULL)	(NULL)	inclusive:1:3	(NULL)	2501	(NULL)
2505	3	2501	(NULL)	2501	inclusive:1:3	(NULL)	2501	18
5002	1	2501	(NULL)	2501	inclusive:1:3	(NULL)	2501	5

发现人事经理的分支还存在，而项目经理分支已经走到ACT_ID = _18的节点。而ACT_ID=_18就是第二个包含网关

这时，因为有2个分支要执行，包含网关会等所有分支走到汇聚才能执行完成。

3) 、执行人事经理审批

然后查询当前任务表： ACT_RU_TASK

ID	REV	EXECUTION_ID	PROC_INST_ID	PROC_DEF_ID	NAME	PARENT_TASK_ID	DESCRIPTION	TASK_DEF_KEY
10005	1	5002	2501	inclusive:1:3	总经理	(NULL)	(NULL)	23

当前任务表已经不是人事经理审批了，说明人事经理审批已经完成。

流程实例执行表：SELECT * FROM act_ru_execution

ID	REV	PROC_INST_ID	BUSINESS_KEY	PARENT_ID	PROC_DEF_ID	SUPER_EXEC	ROOT_PROC_INST_ID	ACT_ID
2501	4	2501	(NULL)	(NULL)	inclusive:1:3	(NULL)	2501	(NULL)
5002	2	2501	(NULL)	2501	inclusive:1:3	(NULL)	2501	23

包含网关执行完成，分支和汇聚就从act_ru_execution删除。

小结：在分支时，需要判断条件，**符合条件的分支，将会执行**，符合条件的分支最终才进行汇聚。

5.4 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：

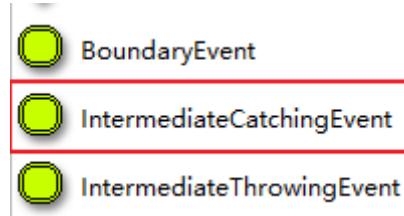
1. 事件网关必须有两条或以上外出顺序流；
2. 事件网关后，只能使用intermediateCatchEvent类型（activiti不支持基于事件网关后连接ReceiveTask）
3. 连接到事件网关的中间捕获事件必须只有一个入口顺序流。

5.4.1流程定义

事件网关图标，红框内



intermediateCatchEvent:



intermediateCatchEvent支持的事件类型：

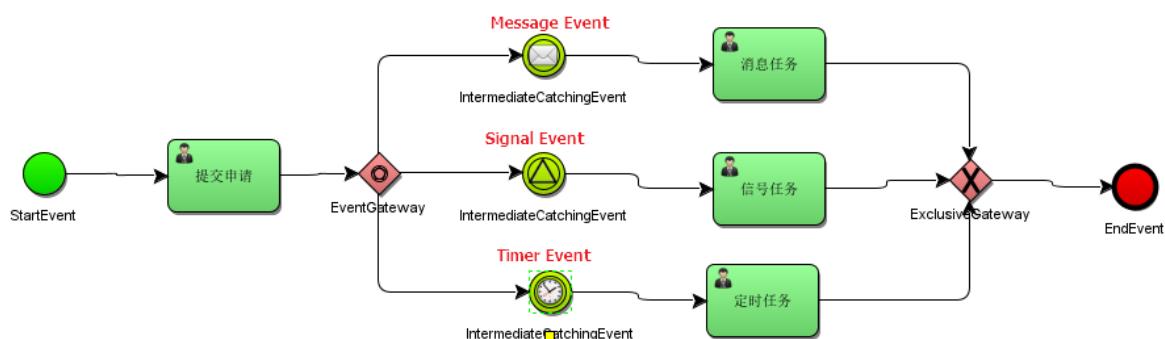
Message Event: 消息事件

Signal Event: 信号事件

Timer Event: 定时事件

property	value
Id	_14
Name	IntermediateCatchingEve...
Documentation	事件类型
Event Type	Timer Event
Timer Definition	
Type	None Event Message Event Signal Event
Expression	Timer Event
Execution Listeners	

使用事件网关定义流程：



三、Activiti整合篇

1. 和Spring整合

1.1 添加相关的依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.bobo</groupId>
    <artifactId>ActivitiDemo02Spring</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <slf4j.version>1.6.6</slf4j.version>
        <log4j.version>1.2.12</log4j.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.activiti</groupId>
            <artifactId>activiti-engine</artifactId>
            <version>7.0.0.Beta1</version>
        </dependency>
        <dependency>
            <groupId>org.activiti</groupId>
            <artifactId>activiti-spring</artifactId>
            <version>7.0.0.Beta1</version>
        </dependency>
        <dependency>
            <groupId>org.activiti</groupId>
            <artifactId>activiti-bpmn-model</artifactId>
            <version>7.0.0.Beta1</version>
        </dependency>
        <dependency>
            <groupId>org.activiti</groupId>
            <artifactId>activiti-bpmn-converter</artifactId>
            <version>7.0.0.Beta1</version>
        </dependency>
        <dependency>
            <groupId>org.activiti</groupId>
            <artifactId>activiti-json-converter</artifactId>
            <version>7.0.0.Beta1</version>
        </dependency>
        <dependency>
            <groupId>org.activiti</groupId>
            <artifactId>activiti-bpmn-layout</artifactId>
            <version>7.0.0.Beta1</version>
        </dependency>
        <exclusions>
            <exclusion>
                <groupId>com.github.jgraph</groupId>
                <artifactId>jgraphx</artifactId>
            </exclusion>
        </exclusions>
    </dependencies>

```

```
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.activiti.cloud</groupId>
    <artifactId>activiti-cloud-services-api</artifactId>
    <version>7.0.0.Beta1</version>
</dependency>
<dependency>
    <groupId>aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.5.4</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.11</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.0.7.RELEASE</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>${slf4j.version}</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-nop</artifactId>
    <version>${slf4j.version}</version>
</dependency>
<!-- log end -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.5</version>
</dependency>
<dependency>
    <groupId>commons-dbc</groupId>
    <artifactId>commons-dbc</artifactId>
```

```

        <version>1.4</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.12</version>
    </dependency>
</dependencies>
<repositories>
    <repository>
        <id>alfresco</id>
        <name>Activiti Releases</name>

        <url>https://artifacts.alfresco.com/nexus/content/repositories/activiti-
releases/</url>
        <releases>
            <enabled>true</enabled>
        </releases>
    </repository>
</repositories>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.18.1</version>
            <configuration>
                <skipTests>true</skipTests>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>

```

1.2 添加整合的配置文件

添加一个Spring的配置文件，并在其中完成Activiti的整合操作

```

<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.cj.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/activiti?
characterEncoding=utf-
&nullCatalogMeansCurrent=true&serverTimezone=UTC"/>

```

```

<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
        flase:      默认值。activiti在启动时，会对比数据库表中保存的版本，如果没有表
或者版本不匹配，将抛出异常。（生产环境常用）
        true:       activiti会对数据库中所有表进行更新操作。如果表不存在，则自动创
建。（开发时常用）
        create_drop: 在activiti启动时创建表，在关闭时删除表（必须手动关闭引擎，才能删
除表）。（单元测试常用）
        drop-create: 在activiti启动时删除原来的旧表，然后在创建新表（不需要手动关闭引
擎）。
    -->
    <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>

```

```

</tx:attributes>
</tx:advice>
<!-- 切面，根据具体项目修改切点配置
<aop:config proxy-target-class="true">
    <aop:advisor advice-ref="txAdvice"
        pointcut="execution(*com.bobo.service.impl..(..))"/>
</aop:config>-->
</beans>

```

databaseSchemaUpdate的取值注意：

- flase： 默认值。activiti在启动时，会对比数据库表中保存的版本，如果没有表或者版本不匹配，将抛出异常。（生产环境常用）
- true： activiti会对数据库中所有表进行更新操作。如果表不存在，则自动创建。（开发时常用）
- create_drop： 在activiti启动时创建表，在关闭时删除表（必须手动关闭引擎，才能删除表）。（单元测试常用）
- drop-create： 在activiti启动时删除原来的旧表，然后在创建新表（不需要手动关闭引擎）。

1.3 创建测试类测试

```

package com.bobo.test;

import org.activiti.engine.RepositoryService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:activiti-spring.xml"})
public class ActivitiTest {

    @Autowired
    private RepositoryService repositoryService;

    @Test
    public void test01(){
        System.out.println(repositoryService);
    }

}

```

通过方法的执行我们能够发现相关的表结构在数据库中完成了创建，说明Activiti和Spring的整合成功。

2. 和SpringBoot的整合

Activiti7发布正式版本之后，它和SpringBoot2.x已经完全整合开发了

2.1 添加相关的依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</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>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

2.2 修改配置文件

```
# 配置Spring的数据源
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://activiti?characterEncoding=utf-
&nullCatalogMeansCurrent=true&serverTimezone=UTC
spring.datasource.name=root
spring.datasource.password=123456

# activiti的配置
#1.flase: 默认值。activiti在启动时，对比数据库表中保存的版本，如果没有表或者版本不匹配，将抛出
异常
#2.true: activiti会对数据库中所有表进行更新操作。如果表不存在，则自动创建
#3.create_drop: 在activiti启动时创建表，在关闭时删除表（必须手动关闭引擎，才能删除表）
#4.drop-create: 在activiti启动时删除原来的旧表，然后在创建新表（不需要手动关闭引擎）
spring.activiti.database-schema-update=true

# 检测历史表是否存在， Activiti7中默认是没有开启数据库历史记录的，启动数据库历史记录
spring.activiti.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。
spring.activiti.history-level=full
# 校验流程文件，默认校验resources下的 process 文件夹里的流程文件
spring.activiti.check-process-definitions=false

```

2.3 整合SpringSecurity

因为Activiti7和SpringBoot整合后，默认情况下，集成了SpringSecurity安全框架，这样我们就要准备SpringSecurity的相关配置信息

添加一个SpringSecurity的工具类

```

package com.bobo.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.context.annotation.ComponentScan;
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);

    @Autowired
    @Qualifier("myUserDetailsService")
    private UserDetailsService userDetailsService;

    public void logInAs(String username) {
        UserDetails user = userDetailsService.loadUserByUsername(username);

        if (user == null) {
            throw new IllegalStateException("User " + username + " doesn't exist, please provide a valid user");
        }
        logger.info("> Logged in as: " + username);

        SecurityContextHolder.setContext(
                new SecurityContextImpl(
                        new Authentication() {
                            @Override

```

```

        public Collection<? extends GrantedAuthority>
getAuthorities() {
            return user.getAuthorities();
        }

        @Override
        public Object getCredentials() {
            return user.getPassword();
        }

        @Override
        public Object getDetails() {
            return user;
        }

        @Override
        public Object getPrincipal() {
            return user;
        }

        @Override
        public boolean isAuthenticated() {
            return true;
        }

        @Override
        public void setAuthenticated(boolean
isAuthenticated) throws IllegalArgumentException {
        }

        @Override
        public String getName() {
            return user.getUsername();
        }
    });

    org.activiti.engine.impl.identity.Authentication.setAuthenticatedUserId(username);
}
}

```

这个类可以从Activiti7官方提供的Example中找到。

添加一个SpringSecurity的配置文件

```

package com.bobo.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

```

```

import org.springframework.security.provisioning.InMemoryUserDetailsManager;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
@Configuration
public class SpringSecurityConfiguration {
    private Logger logger =
LoggerFactory.getLogger(SpringSecurityConfiguration.class);
    @Bean
    public UserDetailsService myUserDetailsService() {
        InMemoryUserDetailsManager inMemoryUserDetailsManager = new
InMemoryUserDetailsManager();
        //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
        String[][] usersGroupsAndRoles = {
            {"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"},

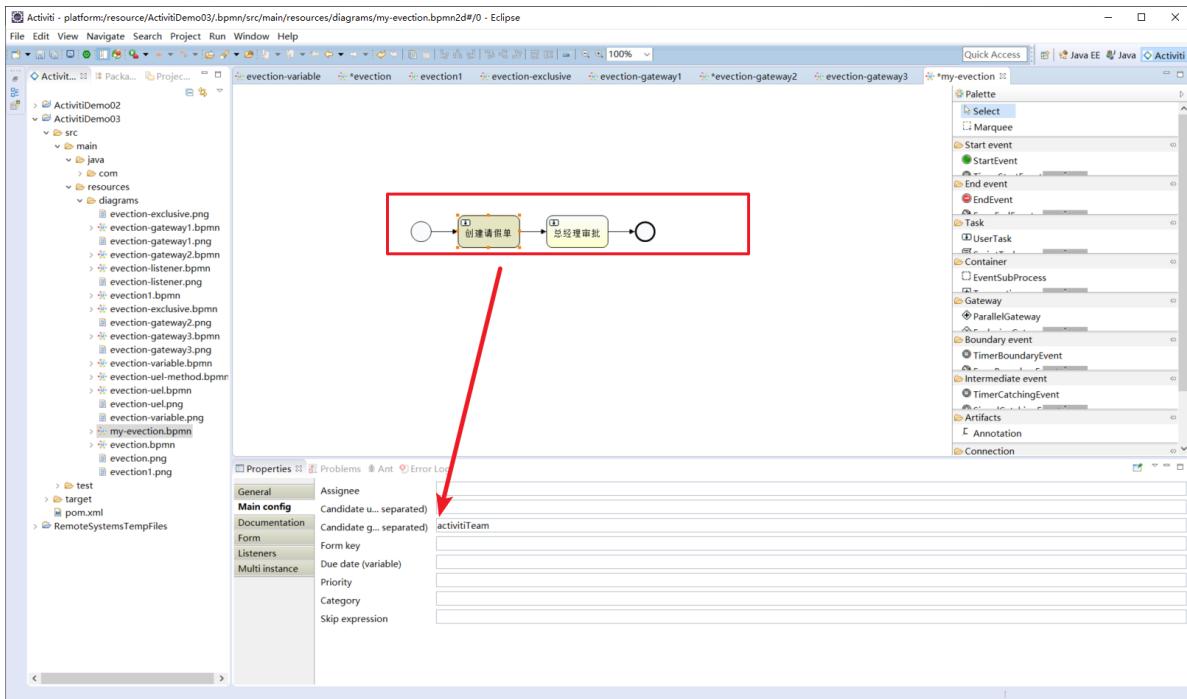
            {"admin", "password", "ROLE_ACTIVITI_ADMIN"},

        };
        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();
    }
}

```

2.4 创建bpmn文件

创建一个简单的bpmn文件，并设置任务的用户组，CandidateGroups，CandidateGroups中的内容要与在SpringSecurity的配置文件中配置的用户组的名称要保持一致，可以填写activitiTeam或者otherTeam。这样填写的好处是，当不确定到底由谁来负责当前的任务的时候，只要是Groups内的用户都可以拾取这个任务。



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

The screenshot shows an IDE (IntelliJ IDEA) with a Java Spring Boot project named 'Activiti03SpringBootDemo'. The project structure on the left includes a 'processes' folder containing 'my-evection.bpmn' and 'my-evection.png'. The main code editor on the right shows a Spring Security configuration class 'SpringSecurityConfiguration'. A red box highlights the 'processes' folder in the project structure. A red arrow points from the 'processes' folder in the project structure to the 'my-evection.bpmn' file in the code editor.

```

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
@Configuration
public class SpringSecurityConfiguration {
    private Logger logger = LoggerFactory.getLogger(SpringSecurityConfiguration.class);
    @Bean
    public UserDetailsService myUserDetailsService() {
        InMemoryUserDetailsManager inMemoryUserDetailsManager = new InMemoryUserDetailsManager();
        //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
        String[][] usersGroupsAndRoles = {
            {"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;
}

```

2.5 单元测试

```

package com.bobo;

import com.bobo.utils.SecurityUtil;
import org.activiti.api.process.model.ProcessDefinition;
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.ClaimTaskPayloadBuilder;

```

```
import org.activiti.api.task.model.builders.TaskPayloadBuilder;
import org.activiti.api.task.model.payloads.ClaimTaskPayload;
import org.activiti.api.task.runtime.TaskRuntime;
import org.activiti.engine.RepositoryService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class ActSpringbootApplicationTests {

    @Autowired
    private ProcessRuntime processRuntime;

    @Autowired
    private TaskRuntime taskRuntime;

    @Autowired
    private SecurityUtil securityUtil;

    @Autowired
    private RepositoryService repositoryService;

    @Test
    void contextLoads() {
        System.out.println(taskRuntime);
    }

    /**
     * 查询流程的定义
     */
    @Test
    public void test02(){
        securityUtil.logInAs("system");
        Page<ProcessDefinition> processDefinitionPage =
            processRuntime.processDefinitions(Pageable.of(0, 10));
        System.out.println("可用的流程定义数量：" +
processDefinitionPage.getTotalItems());
        for (ProcessDefinition processDefinition :
processDefinitionPage.getContent()) {
            System.out.println("流程定义：" + processDefinition);
        }
    }

    /**
     * 部署流程
     */
    @Test
    public void test03(){
        repositoryService.createDeployment()
            .addClasspathResource("processes/my-evection.bpmn")
            .addClasspathResource("processes/my-evection.png")
            .name("出差申请单")
            .deploy();
    }

    /**
     * 启动流程实例
     */
}
```

```
 */
@Test
public void test04(){
    securityUtil.logInAs("system");
    ProcessInstance processInstance =
processRuntime.start(ProcessPayloadBuilder
        .start()
        .withProcessDefinitionKey("my-evection")
        .build()
);
    System.out.println("流程实例id:" + processInstance.getId());
}

/**
 * 任务查询、拾取及完成操作
 */
@Test
public void test05(){
    securityUtil.logInAs("jack");
    Page<Task> tasks = taskRuntime.tasks(Pageable.of(0, 10));
    if(tasks != null && tasks.getTotalItems() > 0){
        for (Task task : tasks.getContent()) {
            // 拾取任务
            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());
    }
}
}
```